Pascal Nasahl

Memory Safety and Fault Security Through Cryptography

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. Jean-Max Dutertre
École Nationale Supérieure des Mines de Saint-Étienne, France

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

The number of computing systems deployed and their complexity steadily increases, resulting in more frequent and intense attack waves. These attacks comprise software attacks, such as memory safety vulnerabilities, as well as physical attacks, such as fault attacks. To counteract memory safety vulnerabilities and fault attacks, a wide variety of different protection mechanisms have emerged over the past decades.

Contrary to this trend, we, in this thesis, showcase that mitigating these threats can be achieved by utilizing a single primitive: cryptography. Based on this building block, we design novel memory safety and control-flow integrity schemes and introduce verification techniques to ensure the correctness of fault countermeasures.

The first part of the thesis focuses on mitigating one of the most common and severe software vulnerability: memory safety corruptions. We showcase that by utilizing cryptography as a primitive, and pursuing a hardware/software co-design approach, it is possible to efficiently and securely protect software against the exploitation of memory safety bugs, hindering adversaries, for example, from manipulating the control-flow of programs.

In the second part of this thesis, we focus on protecting smaller, embedded RISC-V processors as well as fully-fledged Intel and ARM cores against physical fault attacks. We first demonstrate that fault attackers can still manipulate the control-flow of programs, even in the absence of software bugs. Based on this threat model, we then introduce control-flow integrity schemes either requiring minimal or no hardware changes. By using cryptography again as a main building block, we showcase that the researched control-flow integrity schemes provide strong security guarantees with reasonable runtime overheads. We further design a compiler extension utilizing a cryptography-based hardware extension to efficiently close an attack vector of state-of-the-art control-flow integrity schemes. Furthermore, within the scope of this work, we demonstrate that control-flow integrity, realized by using concepts used in encryption primitives, can also be utilized to protect hardware blocks, i.e., finite-state machines, against fault-induced control-flow attacks. Finally, we highlight the importance of leveraging pre-silicon verification techniques to design fault countermeasures providing tangible security guarantees. Within this context, we present a fault injection framework allowing us to verify security-critical parts of a secure element.
Acknowledgements

This thesis would not have been possible without the contributions and support of colleagues, friends, and family who accompanied me throughout my PhD journey. Their guidance, insights, and encouragement have been substantial in shaping this work.

I would like to express my deepest gratitude to my supervisor Stefan Mangard for introducing me to the field of information security and continuously supporting and mentoring me starting from my bachelor studies until the end of my doctoral research. Furthermore, I would like to thank Jean-Max Duterre for joining my doctoral examination committee and for his valuable feedback helping to improve this thesis.

Furthermore, I would like to thank all my co-authors for the exciting papers we did together. In particular, I want to thank David M. Durham, Karanvir Grewal, Jan Hoogerbrugge, Lukas Lamster, Michael LeMay, Hans Liljestrand, Lukas Maar, Marcel Medwed, Rishub Nagpal, Miguel Osorio, Dominic Rizzo, Michael Schaffner, Robert Schilling, David Schrammel, Stefan Steinegger, Salmin Sultana, Nick Timmers, Timothy Trippel, Martin Unterguggenberger, Pirmin Vogel, Stefan Weighlhofer, Samuel Weiser, and Mario Werner for either helping me with my papers or involving me in their work. Moreover, I am really thankful for all the external research stays in the industry, allowing me to gain experience outside the academic world. I want to thank all current and past members of the SESYS group for all the fun we had during or after work. Furthermore, I would also like to express gratitude to Robert Schilling and Mario Werner for making the start of my PhD journey smooth and seamless and for all the lunch discussions we had. Robert, thank you for sharing the office with me and all the numerous papers we have worked on together.

Finally, I am really grateful to my parents, Annelies and Jacques, for always encouraging me to follow my ideas and supporting me in pursuing my goals. Thanks to my soon-to-be wife, Theresa, for being my constant companion through life and always cheering me up when I needed it most. Your endless patience and support made this thesis possible.

Pascal
Table of Contents

Abstract iii
Acknowledgements v
Table of Contents vii
List of Tables xiii
List of Figures xv
Glossary xvii

1 Introduction 1
1.1 Objectives and Contributions 3
1.2 Outline 3

2 Background 7
2.1 Memory Safety Vulnerabilities 7
2.1.1 Techniques to Provide Memory Safety 10
2.2 Fault Attacks 10
2.3 Control-Flow Integrity 11
2.3.1 Control-Flow Integrity in the Presence of a Fault Attacker 11
2.3.2 Control-Flow Integrity in the Presence of a Software Attacker 12
2.4 Trusted Execution Environments 13
2.5 Memory Encryption 15
2.5.1 Memory Encryption Modes 16
2.5.2 MEMSEC 16
2.5.3 Intel TME and TME-MK 17
2.5.4 OpenTitan Scrambling Unit 18

I Mitigating Memory Safety Vulnerabilities 19

3 Cryptographic Memory Safety 23
3.1 Background 25
3.1.1 Tagged Memory 25
3.2 Threat Model 26
# Table of Contents

## 3.3 CrypTag Design Goals

3.4 Design

3.4.1 Hardware

3.4.2 Software

3.5 Implementation

3.5.1 Hardware Extensions

3.5.2 Software Extensions

3.6 Performance Evaluation

3.6.1 Hardware Overhead

3.6.2 Runtime Overhead

3.6.3 Prototype Limitations

3.7 Security Evaluation

3.8 Related Work

3.8.1 Overhead Comparison

3.8.2 Security Comparison

3.9 Future Work

3.10 Conclusion

II Counteracting Fault Attacks

4 Fault-Induced Control-Flow Manipulations on Automotive Systems

4.1 Background

4.1.1 Electronic Control Units

4.1.2 AUTOSAR

4.2 Threat Model & Attacker Description

4.3 Attack Setup

4.3.1 Fault Parameter Characterization

4.4 Malicious Code Execution on AUTOSAR

4.4.1 Fault Model

4.4.2 Fault Target

4.4.3 Attack

4.5 Countermeasures

4.6 Conclusion

5 Cryptographic Control-Flow Integrity on OpenTitan

5.1 Threat Model

5.2 Analysis

5.2.1 Attack Vectors

5.2.2 OpenTitan Countermeasures

5.3 SCRAMBLE-CFI

5.3.1 Overview

5.3.2 Program Instrumentation

5.3.3 Program Deployment

5.3.4 Hardware Changes
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>5.4 Security Analysis &amp; Comparison</td>
<td>5.4.1 Security Comparison</td>
</tr>
<tr>
<td>5.5</td>
<td>Performance Overhead</td>
<td>5.5.1 Security Comparison</td>
</tr>
<tr>
<td>5.6</td>
<td>Area Overhead</td>
<td>5.5.2 Security Comparison</td>
</tr>
<tr>
<td>5.7</td>
<td>Related Work</td>
<td>5.5.3 Security Comparison</td>
</tr>
<tr>
<td>5.8</td>
<td>Conclusion</td>
<td>5.5.4 Security Comparison</td>
</tr>
<tr>
<td>6</td>
<td>Cryptographic Control-Flow Integrity on Commodity Hardware</td>
<td>6.1</td>
</tr>
<tr>
<td>6.1</td>
<td>Background</td>
<td>6.1.1 Intel Virtualization Technology</td>
</tr>
<tr>
<td>6.2</td>
<td>Threat Model</td>
<td>6.1.2 ARM Pointer Authentication</td>
</tr>
<tr>
<td>6.3</td>
<td>Design</td>
<td>6.1.3 ARM Pointer Authentication</td>
</tr>
<tr>
<td>6.4</td>
<td>Implementation</td>
<td>6.1.4 ARM Pointer Authentication</td>
</tr>
<tr>
<td>6.5</td>
<td>Security Discussion</td>
<td>6.1.5 ARM Pointer Authentication</td>
</tr>
<tr>
<td>6.6</td>
<td>Performance Evaluation</td>
<td>6.1.6 ARM Pointer Authentication</td>
</tr>
<tr>
<td>6.7</td>
<td>TME-MK Hardware Change</td>
<td>6.1.7 ARM Pointer Authentication</td>
</tr>
<tr>
<td>6.8</td>
<td>Related Work</td>
<td>6.1.8 ARM Pointer Authentication</td>
</tr>
<tr>
<td>6.9</td>
<td>Future Work and Limitations</td>
<td>6.1.9 ARM Pointer Authentication</td>
</tr>
<tr>
<td>6.10</td>
<td>Conclusion</td>
<td>6.1.10 ARM Pointer Authentication</td>
</tr>
<tr>
<td>7</td>
<td>Protecting Indirect Branches against Faults</td>
<td>7.1</td>
</tr>
<tr>
<td>7.1</td>
<td>Background</td>
<td>7.1.1 Data Redundancy</td>
</tr>
<tr>
<td>7.2</td>
<td>Problem Definition</td>
<td>7.1.2 ARM Pointer Authentication</td>
</tr>
<tr>
<td>7.3</td>
<td>Design</td>
<td>7.1.3 ARM Pointer Authentication</td>
</tr>
<tr>
<td>7.4</td>
<td>Implementation</td>
<td>7.1.4 ARM Pointer Authentication</td>
</tr>
</tbody>
</table>
7.4.1 FIPAC .................................................. 103
7.4.2 Address Protection ................................. 103
7.4.3 Linking the Branch ............................... 105
7.4.4 Combination ......................................... 106
7.4.5 Key Management ................................. 107
7.4.6 Compatibility with other CFI schemes .......... 107
7.5 Evaluation............................................ 107
7.5.1 Performance Evaluation ....................... 107
7.5.2 Code Size Overhead Evaluation ............... 108
7.5.3 Functional Evaluation ........................... 108
7.5.4 Security Evaluation ............................ 109
7.6 Related Work ........................................ 110
7.7 Conclusion .......................................... 111

8 Control-Flow Integrity for Finite-State Machines 113
8.1 Background ........................................... 115
8.1.1 Finite-State Machines .......................... 115
8.2 Threat Model ........................................ 116
8.2.1 Attacker Description ............................ 116
8.2.2 Goal - Fault Secure Finite-State Machine (FSM) 117
8.3 Design ................................................. 117
8.3.1 Selection of the Hardened Next-State Function 119
8.4 Implementation ....................................... 120
8.4.1 Next-State Logic .................................. 120
8.5 Evaluation ............................................. 121
8.5.1 Area Overhead .................................... 122
8.5.2 Timing Overhead .................................. 122
8.5.3 Security Evaluation ............................. 123
8.5.4 Formal Security Analysis ....................... 124
8.6 Limitation & Future Work .......................... 124
8.7 Conclusion ............................................ 125

9 Pre-Silicon Fault Countermeasure Analysis 127
9.1 Design and Implementation ......................... 130
9.1.1 Overview ......................................... 130
9.1.2 Phase 0 - Cell Library & Netlist Converter .. 132
9.1.3 Phase 1 - Target Graph Extraction ............ 133
9.1.4 Phase 1 - Fault Injection ....................... 135
9.1.5 Phase 1 - Differential Graph Creation ......... 136
9.1.6 Phase 1 - Transformation & Evaluation ....... 138
9.1.7 SYNFI Guarantees ............................... 138
9.2 Analysis of OpenTitan .............................. 140
9.2.1 AES ............................................. 140
9.2.2 Life Cycle Controller ............................ 152
9.2.3 Ibex .............................................. 156
9.2.4 Generic Primitives .............................. 157
# Table of Contents

9.3 Related Work ........................................... 159
9.4 Limitations and Future Work ............................ 160
9.5 Conclusion ............................................ 161

10 Conclusion and Outlook ............................... 163
  10.1 Outlook ............................................. 165

List of Contributions ........................................ 167
Bibliography .................................................. 171
Affidavit ...................................................... 193
List of Tables

3.1 Hardware overhead for the $TS=25$ and $TG=16$ memory coloring configuration. ........................................ 36
3.2 Number of additional cache bits and total cache overhead for storing colors. ........................................ 37
3.3 Security comparison of different memory vulnerability mitigation schemes. ........................................ 43

5.1 Protection guarantees of different countermeasure when targeting different attack surfaces. .................. 70
5.2 Code size overhead for the Embench-IoT benchmarks. ......... 71
6.1 Code size overhead for SPEC CPU2017. ....................... 88
8.1 Area overhead for protecting different FSMs using redundancy or SCFI. ............................................... 122
9.1 Verification results for the AES round counter performed on a 16-core machine. ................................. 141
9.2 Verification results for the AES handshake signal on a 16-core machine. .............................................. 146
9.3 Verification results for the AES multi-rail FSM on a 16- or 72-core* machine. ....................................... 149
9.4 Verification results for the AES FSM state encoding on a 16-core machine. ........................................... 151
9.5 Verification results for entering the Return Material Authorization (RMA) state on a 16-core machine. .......... 152
9.6 Verification results for glitching the locking mechanism performed on a 16- or 72-core* machine. .............. 154
9.7 Verification results for the Ibex processor on a 16-core machine. ......................................................... 156
9.8 Verification results for the prim_double_lfsr and prim_count modules. ............................................... 157
List of Figures

2.1 Different forms of buffer overflows. .......................... 9
2.2 Code snippet with the corresponding control-flow graph. .... 12
2.3 Different trusted execution environment implementation strategies. 14
2.4 High-level overview of the OpenTitan chip. .................... 15
2.5 Intel’s TME-MK memory encryption engine. .................... 17

3.1 Memory coloring enforcing spatial memory safety. ............... 26
3.2 Overall CrypTag architecture. The memory encryption unit, placed between the memory subsystem and the memory controller, uses the color as a tweak. .......................... 28
3.3 Set-associative cache architecture extended to support a color for each TG-bytes. On a cache hit, the data cache also checks the color. On a tag mismatch, the cache line is fetched from the memory. .......................... 29
3.4 Runtime overhead for SPEC CPU2017 with MEMSEC. .......... 37
3.5 Runtime overhead for MiBench with MEMSEC. ................. 38
3.6 Runtime overhead for Embench-IoT with CrypTag on a system already featuring a memory encryption engine. ............... 38
3.7 Memory latency measured with LMBench. ...................... 39

4.1 Attack setup consisting of the target ECU and the glitching equipment. .................................................. 56
4.2 Diagram of the attack setup. .................................... 59
4.3 Timing diagram of the attack. .................................... 59
4.4 Different glitch length and voltage combinations. The fault parameters for successful attacks are highlighted in red. ............... 60

5.1 Encrypted call graph. ............................................ 66
5.2 Instrumentation of direct and indirect calls. The different colors highlight code blocks encrypted with different tweaks. ............... 66
5.3 Deployment of protected programs. Code blocks in flash memory encrypted with different SCRAMBLE-CFI tweaks are highlighted with different colors. .......................... 68
5.4 Runtime overhead for the Embench-IoT benchmarks. ........... 71

6.1 Call graph with manipulated control-flow. ...................... 76
6.2 EPT aliasing combined with memory encryption for fine-grained memory encryption. 77
6.3 Signature init, update, and key switch for a direct call. 79
6.4 Security implications when the call sites $A$ and $C$ derive an identical key for the multi-call target $B$. 80
6.5 Secure handling of multi-call targets. 80
6.6 Overview of our EC-CFI prototype implementation. 81
6.7 Runtime overhead for SPEC CPU2017. 89
6.8 Runtime overhead for Embench-IoT. 89
6.9 TLB misses for Embench-IoT. 90
6.10 Emulated runtime overhead for SPEC CPU2017. 91
6.11 Emulated runtime overhead for Embench-IoT. 91
7.1 Control-flow graph with state updates, patches, and checks. 99
7.2 Pointer signed with ARM pointer authentication. 102
7.3 SPECspeed2017 performance overhead for different protection configurations. 108
8.1 General structure of a finite-state machine. 115
8.2 Control-flow graph of an FSM. 116
8.3 Mapping of valid and invalid input tuples to a valid or invalid next state. 118
8.4 Unprotected and protected next-state logic of an example FSM. 118
8.5 SCFI hardened next-state function. 119
8.6 Internal structure of the MDS matrix multiplication [DL18]. All elements operate on 1-bytes each. 121
8.7 The next-state logic hardening pass. 121
8.8 Area-time product for the adc_ctrl_fsm module in different configurations. 123
9.1 Block diagram of the SYNFI framework. 131
9.2 Example circuit with assigned input (blue) and output (green) values. 135
9.3 Extracted and preprocessed target graph. 135
9.4 Differential graph. 136
9.5 Multi-rail FSM approach. 147
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADI</td>
<td>Application Data Integrity</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASLR</td>
<td>Address Space Layout Randomization</td>
</tr>
<tr>
<td>AUTOSAR</td>
<td>Automotive Open System Architecture</td>
</tr>
<tr>
<td>AXI</td>
<td>Advanced eXtensible Interface</td>
</tr>
<tr>
<td>BSW</td>
<td>Basic Software</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>CBC</td>
<td>Cipher Block Chaining</td>
</tr>
<tr>
<td>CET</td>
<td>Control-Flow Enforcement Technology</td>
</tr>
<tr>
<td>CFG</td>
<td>Control-Flow Graph</td>
</tr>
<tr>
<td>CFI</td>
<td>Control-Flow Integrity</td>
</tr>
<tr>
<td>CNF</td>
<td>Conjunctive Normal Form</td>
</tr>
<tr>
<td>CPI</td>
<td>Code Pointer Integrity</td>
</tr>
<tr>
<td>CTR</td>
<td>Counter</td>
</tr>
<tr>
<td>DEP</td>
<td>Data Execution Prevention</td>
</tr>
<tr>
<td>DOP</td>
<td>Data-Oriented Programming</td>
</tr>
<tr>
<td>DRM</td>
<td>Digital Rights Management</td>
</tr>
<tr>
<td>ECC</td>
<td>Error Correction Code</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>FF</td>
<td>Flip-Flop</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite-State Machine</td>
</tr>
<tr>
<td>GE</td>
<td>Gate Equivalent</td>
</tr>
<tr>
<td>GPIO</td>
<td>General Purpose Input Output</td>
</tr>
<tr>
<td>HD</td>
<td>Hamming Distance</td>
</tr>
<tr>
<td>IBT</td>
<td>Indirect Branch Tracking</td>
</tr>
<tr>
<td>ICT</td>
<td>Information and Communication Technology</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>JOP</td>
<td>Jump-Oriented Programming</td>
</tr>
<tr>
<td>LFSR</td>
<td>Linear-Feedback Shift Register</td>
</tr>
<tr>
<td>LUT</td>
<td>Lookup-Table</td>
</tr>
<tr>
<td>MAC</td>
<td>Message Authentication Code</td>
</tr>
<tr>
<td>MCU</td>
<td>Microcontroller Unit</td>
</tr>
<tr>
<td>MDS</td>
<td>Maximum Distance Separable</td>
</tr>
<tr>
<td>MEE</td>
<td>Memory Encryption Engine</td>
</tr>
<tr>
<td>MISR</td>
<td>Multi-Input Signature Register</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>MPU</td>
<td>Memory Protection Unit</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>MTE</td>
<td>Memory Tagging Extension</td>
</tr>
<tr>
<td>NS</td>
<td>Non-Secure</td>
</tr>
<tr>
<td>OTP</td>
<td>One Time Programmable</td>
</tr>
<tr>
<td>PA</td>
<td>Pointer Authentication</td>
</tr>
<tr>
<td>PAC</td>
<td>Pointer Authentication Code</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>PCB</td>
<td>Printed Circuit Board</td>
</tr>
<tr>
<td>PTE</td>
<td>Page Table Entry</td>
</tr>
<tr>
<td>REE</td>
<td>Rich Execution Environment</td>
</tr>
<tr>
<td>RMA</td>
<td>Return Material Authorization</td>
</tr>
<tr>
<td>ROP</td>
<td>Return-Oriented Programming</td>
</tr>
<tr>
<td>RoT</td>
<td>Root-of-Trust</td>
</tr>
<tr>
<td>RTE</td>
<td>Runtime Environment</td>
</tr>
<tr>
<td>RTL</td>
<td>Register-Transfer Level</td>
</tr>
<tr>
<td>RTLIL</td>
<td>Register-Transfer Level Intermediate Language</td>
</tr>
<tr>
<td>SE</td>
<td>Secure Element</td>
</tr>
<tr>
<td>SEP</td>
<td>Secure Enclave Processor</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-Chip</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td>SSA</td>
<td>Static Single Assignment</td>
</tr>
<tr>
<td>TCB</td>
<td>Trusted Computing Base</td>
</tr>
<tr>
<td>TDX</td>
<td>Trusted Domain Extension</td>
</tr>
<tr>
<td>TEC</td>
<td>Tamper-Evident Counter</td>
</tr>
<tr>
<td>TEE</td>
<td>Trusted Execution Environment</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>TME</td>
<td>Total Memory Encryption</td>
</tr>
<tr>
<td>TME-MK</td>
<td>Total Memory Encryption - Multi Key</td>
</tr>
<tr>
<td>TOCTOU</td>
<td>Time-of-Check to Time-of-Use</td>
</tr>
<tr>
<td>UAF</td>
<td>Use-After-Free</td>
</tr>
<tr>
<td>VT</td>
<td>Virtualization Technology</td>
</tr>
<tr>
<td>XEX</td>
<td>Xor–Encrypt–Xor</td>
</tr>
</tbody>
</table>
Introduction

With the ongoing digital transformation process, the number of computing systems deployed in industrial settings and people’s direct environments is massively growing. Due to the immense technological progress, e.g., the shrinking of the CMOS feature size, driven by the Information and Communication Technology (ICT) sector, these systems are not only getting more powerful but also more complex. Prominent examples of this rising complexity are the codebase of the Linux kernel that has increased from 11.3 to 25.6 million lines of code or the number of transistors on an Intel server processor that has skyrocketed from 5 to more than 44 billion in just ten years [Wik].

The continuous improvement in this sector paves the way for a rich set of new applications. For example, the computing power offered by nowadays personal computers or cloud services fuels research and engineering by helping to process more sophisticated computing models. Moreover, due to the miniaturization of technology, compact handhelds and wearables, such as smartphones and smartwatches, provide new forms of usage models, starting from communication up to medical applications.

Although the progress in technology enables new applications, the growing complexity is also increasingly challenging, especially when considering the security and privacy of data handled by these computing systems. In particular, due to the rising number of architectural hardware features deployed into chips as well as the corresponding growing software stacks, the attack surface of computing systems is immensely broadening, resulting in more frequent and intense attack waves. For example, the Heartbleed vulnerability affected more than 24% of the HTTPS servers deployed worldwide enabling adversaries to leak secret data [Dur+14]. The GHOST [MITc] and Shellshock [MITb] vulnerabilities affected a similar number of targets and even enabled malicious entities to
remotely execute code on remote systems.

A malicious entity can perform these attacks by targeting various attack vectors. For this work, at a high-level, we classify attacks into classical software attacks as well as physical attacks:

Software attacks. In these attacks, the adversary exploits logical weaknesses in software deployed on a target device. One common software attack is the class of memory safety vulnerabilities. In these vulnerabilities, a malicious entity exploits a memory safety bug to leak or modify security-sensitive data. For example, by corrupting control-flow-related data, such as return addresses or code-pointers, the adversary can manipulate the control-flow to, eventually, gain full remote code execution on the target device. Although memory-safe languages exist, such as Rust, preventing this class of attacks, legacy codebases, low-level code, and high-performance applications still are mostly written in memory-unsafe languages, such as C. Hence, dedicated countermeasures mitigating these attacks are required. In the course of this work, we elaborate that existing memory safety countermeasures either are incomplete or induce tremendous runtime overheads.

Physical attacks. Contrary to software attacks, in physical attacks, the device itself is targeted by an adversary. Here, a malicious entity aims to leak or modify data by exploiting physical properties of the underlying system. Physical attacks consist of two attack phases: the interaction and exploitation phase [KS04]. During the interaction phase, the adversary interacts with the device under attack in an unintended way. For example, the attacker could measure the power consumption or the emitted electromagnetic radiation or could briefly interrupt the power supply or apply extreme heat to the chip. Depending on the interaction phase, two major forms of physical attacks exist. In side-channel attacks, secret data, e.g., an encryption key, is leaked by measuring and analyzing physical parameters, e.g., the power supply or electromagnetic radiation, of the target device. Common side-channel attacks include differential or simple power analysis attacks on cryptographic primitives [KJJ99]. In fault attacks, the malicious entity tampers with the environment of the chip to trigger an unexpected behavior during the computation. This fault can be induced with different fault injection techniques, for example, electromagnetic glitching [KSV13]. During the exploitation phase, the effect of the fault enables the adversary to achieve different attack goals. While in the early days of these attacks, the attack goal mostly was to leak keys of cryptographic primitives [BS97], more recent attacks also focus on, for example, gaining remote code execution on target systems [Zer] by manipulating the control-flow. To protect systems, such as secure elements, against faults, dedicated hardware- and software-based countermeasures are frequently deployed. This thesis is motivated by the fact that current fault countermeasures are either inadequate, require intrusive hardware changes, or do not provide the expected security.
1.1 Objectives and Contributions

The goal of this thesis is to research novel protection mechanisms that enhance the resilience of computer systems against software and physical attacks. Within our work, we focus on mitigating *memory safety vulnerabilities* as well as *fault attacks*. Current countermeasures mitigating memory safety vulnerabilities or fault attacks utilize a wide variety of different protection mechanisms. In terms of memory safety, this includes shadow stacks [Eva+15], bounds registers [Vah+19], tagged memory systems [Ser19], as well as other customized architectures [Wat+15]. To mitigate fault attacks, commonly some form of redundancy is used. For example, this can be spatial or temporal redundancy [OTK17], encoding [SWM18], or control-flow integrity [VHM03]. In this thesis, we demonstrate that protection against memory corruptions as well as faults can be achieved by a single, fundamental primitive: cryptography. Concretely, our main contributions in this context are as follows:

**Memory Safety.** We illustrate that memory encryption engines not only can be used to provide protection against physical attacks, e.g., cold boot attacks. We demonstrate that memory encryption also can be leveraged as a lightweight solution against memory safety corruptions. To that end, we show that enforcing memory safety with cryptography enables us to provide strong security guarantees with small runtime overheads.

**Control-Flow Integrity.** We showcase that control-flow integrity is a versatile protection mechanism suitable to protect the control-flow of software as well as hardware against fault-induced control-flow attacks. We demonstrate that enforcing control-flow integrity with encryption enables us to achieve strong protection guarantees with minimal detection latency and hardware modifications.

**Verification.** Moreover, in this thesis, we highlight the importance of checking the functionality of fault countermeasures at the pre-silicon level. In this context, we demonstrate that verification is a powerful concept to ensure that hardware-based fault protection mechanisms provide tangible security guarantees.

1.2 Outline

This thesis consists of two parts. In the first part, we made progress towards designing novel memory safety countermeasures using cryptography as a main building block. In the second part, we researched fault mitigation strategies protecting systems at different levels and explored new methodologies for verifying fault countermeasures’ correctness during the chip development phase. We provide an outline and a summary of the corresponding contribution points of our work in the next paragraphs:
Chapter 2 provides background on software and physical attacks as well as on countermeasures addressing these attack vectors. For the attacks, we provide an overview on software-exploitable memory bugs and physical fault injection attacks. Furthermore, we discuss common countermeasures, such as control-flow integrity, trusted execution environments, and memory encryption, protecting systems against logical and physical attacks.

Part I - Mitigating Memory Safety Vulnerabilities

Chapter 3 introduces CrypTag, our hardware/software co-design mitigating a broad range of memory safety violations. In this chapter, we showcase that a memory encryption engine, in addition to mitigating physical attacks, can also be utilized to provide protection against logical attacks, i.e., memory safety vulnerabilities. By encrypting memory objects with different encryption tweaks when storing them in memory, data only can be successfully fetched when knowing the correct decryption tweak. We demonstrate that this mitigates the exploitation of memory safety corruptions, e.g., control-flow attacks, as the decryption tweak on a memory safety violation mismatches. This contribution was published at the AsiaCCS 2021 conference [Nas+21a].

Part II - Counteracting Fault Attacks

Chapter 4 serves as a motivation for the subsequent chapters. In this work, we demonstrate that an adversary can manipulate the control-flow of software even in the absence of software bugs, e.g., memory safety vulnerabilities. Here, we focus on targeting an automotive platform running a common operating system used in vehicles. We show that an adversary having physical access to a device can gain full malicious code execution by injecting messages into the automotive bus system and performing fault attacks. This contribution was published at escar USA 2019 briefly before the start of my doctoral studies [NT19].

Chapter 5 first analyzes existing hardware- and software-based fault countermeasures integrated into the open-source OpenTitan secure element. Then, to increase the resilience of the system against fault-induced control-flow attacks, we introduce a cryptographically enforced control-flow integrity scheme. By utilizing existing hardware features of the platform, our approach provides strong security guarantees with inducing only a small performance overhead and requiring minimal hardware changes. We presented this contribution at GLSVLSI 2023 (best paper award - third place) [NM23].

Chapter 6 shows that fully-fledged Intel processors also require protection against fault-induced control-flow manipulations, as these attacks can be performed remotely in software. To that end, we demonstrate that Intel’s memory encryption engine, which is already broadly available, can be utilized beyond its intended usage to provide fine-grained memory encryption. Based on the introduced approach, we design a cryptographic control-flow integrity scheme
protecting Intel CPUs against fault-induced control-flow manipulations. Furthermore, we discuss and analyze potential minimal hardware changes modifying the behavior of the memory encryption engine to minimize runtime overheads. This work was accepted at the HOST 2023 conference [Nas+23a].

**Chapter 7** showcases that faults into addresses used by indirect branches enable an adversary to manipulate the control-flow within the borders of the control-flow graph, even in the presence of a control-flow integrity scheme. To mitigate this threat, we introduce a software-based scheme that protects these addresses using a cryptographic message authentication code. We demonstrate that by utilizing the pointer authentication extension offered by recent ARM systems, we can improve the resilience of control-flow integrity schemes against faults with a small runtime overhead. We published this contribution at HOST 2021 [NSM21].

**Chapter 8** demonstrates that control-flow integrity is a generic concept also suitable to protect hardware primitives against fault-induced control-flow attacks. In this chapter, we introduce a hardware-based control-flow integrity scheme utilizing an MDS-based diffusion layer also used in cryptographic primitives to protect the control-flow of finite-state machines against fault attacks. Our analysis of protecting security-sensitive finite-state machines of the OpenTitan secure element reveals that our approach provides strong probabilistic security guarantees with a smaller area-time product than comparable protection techniques. We presented this work at DATE 2023 [Nas+23b].

**Chapter 9** first highlights the importance of conducting a pre-silicon fault countermeasure analysis. Then, in this context, we present an open-source fault injection verification framework allowing the user to analyze the resilience of a chip against fault attacks. To demonstrate the capabilities of our tool, we verify security-critical parts of OpenTitan and, based on the analysis results, provide an improved hardware design mitigating the identified weaknesses. This contribution was published at CHES 2022 [Nas+22].

**Chapter 10** concludes this thesis.
This chapter provides background necessary for the remainder of this work. We first introduce the topic of memory safety vulnerabilities and discuss potential countermeasures. Then, we focus on fault attacks, where we highlight different fault injection techniques and potential attack targets. Furthermore, we discuss control-flow integrity, a countermeasure mechanism capable of protecting the control-flow against software attacks as well as fault attacks. Moreover, we introduce the concept of trusted execution environments and summarize different implementation approaches. Finally, we conclude this chapter by giving an overview of memory encryption and highlight solutions offered by academia and industry.

2.1 Memory Safety Vulnerabilities

Memory safety vulnerabilities are a prevalent security issue negatively impacting systems already for decades. Microsoft [Mil19] and Google [Chr] stated that 70% of security bug fixes in Windows and Chrome are due to memory safety vulnerabilities. Similar, 3 out of 10 exploits in MITRE’s [MITa] most dangerous software weaknesses list are based on memory bugs, again highlighting the severity of these vulnerabilities.

These vulnerabilities are often the entry point for more sophisticated attacks, such as Return-Oriented Programming (ROP) [Sha07], Jump-Oriented Programming (JOP) [Ble+11], or Data-Oriented Programming (DOP) [Hu+16]. By exploiting memory errors and subsequently performing these attacks, the adversary can arbitrarily manipulate the control-flow of the program. For example, when performing a ROP attack, the adversary first exploits a memory bug to
Chapter 2. Background

overwrite the return address of a function. When the attacker then redirects the control-flow to code blocks containing attacker controllable return instructions, \textit{i.e.}, ROP gadgets, Turing Complete attacks can be performed [Sha07]. More specifically, by chaining together ROP gadgets, arbitrary malicious programs can be crafted. In addition to these control-flow attacks, adversaries also can exploit memory errors to threaten confidentiality and integrity of arbitrary data. One prominent example was the Heartbleed [Dur+14] bug that enabled malicious entities to read secret data processed and stored on servers.

The root cause of memory safety vulnerabilities are memory bugs in programs written in unsafe programming languages, such as C or C++. In general, these memory errors are categorized into spatial and temporal bugs [Sze+13].

**Spatial Memory Safety Errors.** A spatial memory safety bug enables the adversary to read or write data beyond the intended boundaries of a memory object.

```
#include <stdio.h>
int main(int argc, char **argv)
{
    char buf[8];
    gets(buf);
    printf("%s\n", buf);
    return 0;
}
```

Listing 2.1: Example of a buffer overflow attack from OWASP [OWAa].

Listing 2.1 shows a code snippet containing a spatial memory bug. The program first allocates a memory object of size 8 B (cf. Line 4) and then retrieves and stores user input into this object (cf. Line 5). However, as \texttt{gets} does not check the size of the provided input and C does not conduct a bounds check, data of arbitrary length can be provided by a malicious party. If this is the case, a buffer overflow happens, which allows the attacker to write data over the bounds of the object. Hence, data stored on the stack or heap can be manipulated when exploiting this vulnerability and, e.g., a ROP attack can be conducted.

Figure 2.1 shows different categories of buffer overflows. Whereas linear overflows target adjacent memory objects \( \varnothing \), non-linear overflows affect non-adjacent objects \( \natural \). Intra-object overflows are spatial errors within a memory object \( \natural \), e.g., a C structure (struct).

**Temporal Memory Safety Errors.** Temporal memory bugs enable the attacker to use a pointer referencing a deallocated memory object. These pointers, which are also called \textit{dangling} pointers, can be used to access memory beyond its intended life cycle.
Listing 2.2: Example of a use-after-free attack from OWASP [OWAb].

```c
char * ptr = (char *)malloc (SIZE);
...
if (err) {
    abrt = 1;
    free(ptr);
}
if (abrt) {
    logError("operation aborted before commit", ptr);
}
```

Listing 2.2 illustrates the problematic of temporal errors in terms of a Use-After-Free (UAF) vulnerability. Although the program deallocates the memory using `free` (cf. Line 5), the pointer referencing this memory object does not get cleared. Hence, in Line 9, the program is still able to access the freed memory. This can be problematic as the memory location could be reassigned to a new memory object. Then, the attacker controlling this dangling pointer can leak or modify data of the new memory object stored on the same location.

In addition to UAF, temporal memory safety violations also comprise double-free attacks as well as uninitialized memory accesses [Xu+22]. Similar to UAF, these attacks enable the adversary to manipulate or leak data stored in memory.

As indicated in Section 2.1, the root cause of temporal errors is the usage
of unsafe programming languages. While these languages, such as C and C++, require that the programmer correctly deallocates memory and clears the corresponding pointers, type-safe languages provide automatic garbage collectors preventing these memory errors [Sze+13].

2.1.1 Techniques to Provide Memory Safety

To limit the impact of exploitable memory safety bugs, several attack mitigations like W⊕X, Address Space Layout Randomization (ASLR), or Data Execution Prevention (DEP) are deployed in modern computer architectures. However, these countermeasures typically only raise the bar for a successful attack. Although simpler attacks, like the execution of attacker-injected code, can be mitigated, more advanced techniques, such as ROP, still can bypass these protection mechanisms [Sha07]. Even more sophisticated countermeasures, like ensuring the integrity of the control-flow [Mas+15], can be defeated by techniques like DOP [Hu+16], where an attacker can craft Turing Complete exploits. To successfully defeat memory vulnerabilities, memory safety is required [Sze+13]. Memory safety can be achieved by preventing all spatial and temporal memory vulnerabilities in the system.

To mitigate spatial memory issues, state-of-the-art countermeasures [Wat+15; NMW02; Jim+02; Nag+09; Dev+08; Ked+21; KLK20; Li+22] ensure that memory accesses stay within the intended object bounds by performing bounds checks. SoftBound [Nag+09], for example, assures spatial memory safety by storing the memory bounds, i.e., base and upper bound, of objects in a shadow memory. On a memory access, programs instrumented with SoftBound use this metadata to verify that the used pointer does not read or write data beyond these boundaries. Because of the expensive monitoring of the object bounds by software checks, SoftBound adds an average runtime overhead of around 67%.

Similar, for temporal memory safety, countermeasures [NS07; Ven+07; Nag+10; NMZ12; Ser+12; AJ20] use metadata to track the liveness of memory objects. CETS [Nag+10], for example, assigns each object in a program a unique identifier. By using a table stored in shadow memory, CETS links the object’s pointer to metadata. This metadata contains information whether the object is already deallocated or not. CETS inserts code before each memory access to check whether the object is still alive. Mitigating dangling pointers with this method induces a performance overhead of 48% on average.

By combining SoftBound with CETS, for memory safety, i.e., spatial and temporal memory safety, can be guaranteed. However, the large performance overhead of 116% on average makes the deployment on a larger scale hard.

2.2 Fault Attacks

In a fault attack, the adversary tampers with the physical parameters of the device’s environment to trigger a fault in the circuit, e.g., in the processor’s memory or instruction pipeline or in other Intellectual Property (IP) blocks.
of the hardware. These faults cause several effects at the physical level, e.g., transient voltage and current changes as well as timing violations [RSG21]. However, typically, the effect of a fault is modeled as a transient, i.e., a bit-flip, or permanent, i.e., stuck-at, effect [RU96].

More specifically, a fault $f \in F$ is described using the set $K = \{e, s, t\}$ where $e$ is the effect of a fault, i.e., transient or permanent effects, $s$ the spatial, and $t$ the temporal dimension of the fault. The spatial $s$ and temporal $t$ dimensions of a fault describe where (which gate or wire) and when (which clock cycle) a fault is induced. The set $F$ consists of all possible fault combinations and an adversary typically can inject up to a certain number of faults into the circuit.

Originally, fault attacks were pure local attacks requiring an adversary to have physical access to the target device. To induce a fault, attackers interrupt the supply voltage or the clock signal, decapsulate the chip and shoot with a laser directly into the die, or use electromagnetic pulses [KSV13]. However, recent publications, such as Plundervolt [Mur+20], CLKSCREW [TSS17], or VoltJockey [Qiu+19a], demonstrated that faults also could be induced remotely in software, increasing the attack surface of fault attacks even more.

While faults in the past were mainly used to break cryptographic schemes [BS97; BDL97; BS03; Dob+18b; Dob+18a], recent work [Zer; NT19; Vas+20; Her+21; Fre20; TS16; Mil+18; OFl20; WWM11; WP17; TM17; TSW16] demonstrates that fault attacks can lever out security assumptions of the device under attack and bypass secure boot [Vas+20; Her+21; Fre20; Vas+17] or escalate privileges on Linux [TM17; TSW16].

## 2.3 Control-Flow Integrity

Control-Flow Integrity (CFI) [Aba+09] is a well-established countermeasure aiming to protect programs against control-flow manipulations. Depending on the presumed threat model, i.e., a fault adversary or a software attacker, different control-flow integrity policies and implementations exist.

### 2.3.1 Control-Flow Integrity in the Presence of a Fault Attacker

The goal of these schemes is to detect or prevent fault-induced control-flow manipulations at different granularities. Most of the introduced countermeasures [VHM03; WS88; Gol+03; OSM02; LHB14; HLB19; Abe+16; SNM22a; Rei+05] use a signature-based CFI approach.

In these signature-based approaches, the countermeasure first needs to analyze the intended control-flow of a program. Usually, this is analysis is statically performed during the compilation by the compiler [SNM22a; WWM15; Wer+18]. Here, the modified compiler extracts the Control-Flow Graph (CFG) of the program to protect. As shown in Figure 2.2, the CFG comprises all control-flow transfers between basic-blocks, i.e., direct or indirect calls and conditional or
unconditional branches. Basic-blocks are linear code sequences with a control-flow transfers instruction at the end.

Afterwards, these CFI schemes assign each node in the CFG a random signature. Note that this signature typically is not a signature in the cryptographic sense, instead it is a unique fingerprint of the corresponding node. Finally, the program to protect is instrumented in such a way that it automatically derives this signature. Here, in each node, instructions are inserted that accumulatively update the signature with a compile-time defined constant for each node. By inserting checks, i.e., comparisons between the derived and the assigned signature, at certain points in the program, control-flow redirections can be detected. More specifically, any control-flow manipulation that redirects the control-flow outside of the CFG is detected by these schemes.

However, as the target of a indirect call cannot be exactly determined at compile-time, the adversary still can manipulate the control-flow within the bounds of the CFG. Moreover, as the signature checks induce runtime overheads, schemes, such as CFCSS [OSM02], only conduct these checks at a coarse granularity, e.g., at the end of each basic-block, function, or even at the end of the program. In addition, most of these CFI schemes cannot withstand attacks from software adversaries as the signature can be predicted and altered when exploiting software vulnerabilities.

### 2.3.2 Control-Flow Integrity in the Presence of a Software Attacker

Monitoring all control-flow transfers using software checks is expensive [Aba+05]. Hence, CFI schemes aiming to mitigate software-induced control-flow manipulations focus on protecting control-flow related data an attacker can alter. Here, the adversary exploits a memory vulnerability (cf. Section 2.1) to overwrite return addresses [Sha07] or addresses used by indirect branches [Ble+11]. One of the schemes preventing some control-flow attacks are stack cookies [Cow98], which are placed between return addresses and data on the stack. When these stack...
2.4. Trusted Execution Environments

cookies are manipulated by a linear buffer overflow targeting a return address, an error is triggered. Although the overhead of this countermeasure is small, non-linear buffer overflows circumvent the protection and indirect branches are still vulnerable.

Hence, more recent control-flow integrity schemes aim to directly maintain the integrity of vulnerable pointers. Code Pointer Integrity (CPI) \cite{Kuz+18} performs a static code analysis to find pointers allowing an adversary to perform control-flow attacks. These pointers and corresponding metadata are then securely stored in an attacker inaccessible shadow memory. On control-flow transfers, CPI checks the validity of the pointers using the protected shadow memory. Similar, CCFI \cite{Mas+15} identifies all return addresses and addresses used by indirect branches. When storing these addresses into memory, CCFI computes a Message Authentication Code (MAC) of these pointers and stores them into memory. When a memory vulnerability is exploited to change the addresses in memory, the MAC check later fails allowing CCFI to detect these attacks. For the MAC generation, the scheme uses the AES-NI instructions for hardware acceleration. PARTS \cite{Li+19} utilizes ARM’s Pointer Authentication (PA) \cite{Qua} instructions to create a Pointer Authentication Code (PAC), i.e., a tweakable MAC, for each sensitive code and data pointer. When loading these pointers from memory, the PAC is verified enabling PARTS to detect manipulated addresses. Recently, Intel introduced the Control-Flow Enforcement Technology (CET) \cite{Int16} that is available since the Tiger Lake mobile processors. This CPU extension consists of a shadow memory and the Indirect Branch Tracking (IBT) feature. In CET, return addresses are stored in the shadow memory and before a return, the current return address is compared to the address stored in this protected memory region. The IBT feature ensures that an indirect branch only can transfer the control-flow to CET-specific \texttt{endbranch} instructions. Hence, control-flow manipulations to any other instruction cause an exception.

Since these control-flow integrity schemes do not consider a fault adversary in their threat model, fault-induced control-flow attacks are still possible. More specifically, manipulating the execution of a direct branch is still feasible as the protection of these approaches does not comprise these branches.

2.4 Trusted Execution Environments

Due to the rising complexity of modern CPUs that is fueled by the demand for speed and efficiency, vendors struggle in providing a secure computing foundation \cite{Lip+20; Koc+20; Bor+22; Lip+16}. However, security-sensitive applications, such as key storages, authentication services, as well as Digital Rights Management (DRM), rely on the fact that the hardware provides strong protection against adversaries. One common approach of creating a safe area in a potentially unsafe system is to minimize the Trusted Computing Base (TCB) by introducing a dedicated secure and isolated execution environment, \textit{i.e.}, an enclave or a trusted execution environment \cite{KDC20}. These safe regions are inaccessible for adversaries and communication is only possible over a well-defined interface, \textit{e.g.}, the
GlobalPlatform Application Programming Interface (API) [Glo].

As shown in Figure 2.3, different implementation strategies are available to realize a secure execution environment. In the following, we differentiate between the Rich Execution Environment (REE), \textit{i.e.}, the non-secure domain, and the Trusted Execution Environment (TEE), \textit{i.e.}, the secure domain.

\textbf{Virtual Processor TEE.} Trusted execution environments, such as ARM TrustZone [ARM09], Intel SGX [XSL16], and Intel TDX [Int23a], are implemented as virtual processors that are part of the main processor. As shown in Figure 2.3a, hardware and software extensions are provided to realize the separation between the secure and non-secure domain. In TrustZone, for example, the processor maintains a security identifier, \textit{i.e.}, the Non-Secure (NS) bit. Since the hardware is aware of this identifier, TrustZone ensures that accessing resources, e.g., cache lines, over domains is not possible. SERVAS [Ste+21a], a recently introduced TEE approach, achieves the isolation between security domains using memory encryption.

\textbf{On-SoC Processor TEE.} In this approach (cf. Figure 2.3b), the TEE is realized by embedding a dedicated security core into the main System-on-Chip (SoC). By establishing a communication interface between the application processor, \textit{i.e.}, the REE, and this secure core, data can be exchanged. Compared to the virtual processor approach, stronger isolation properties can be achieved. More specifically, as security-sensitive resources, such as caches, are not shared among the secure and non-secure domain, cache-based side-channel attacks [Lip+16; Zha+16; Gua+16; Bra+17; Sch+17; Göt+17] or transient attacks [Bul+18; Bul+20; Che+20] cannot be performed. With the Secure Enclave Processor (SEP) [Inc20] from Apple and the Pluton from Microsoft, dedicated solutions are already available. HECTOR-V [Nas+21b] is a heterogeneous RISC-V CPU consisting of a 64-bit application processor and a fault-hardened 32-bit secure processor.

\textbf{External Processor TEE.} The idea of this approach (cf. Figure 2.3c) is to install a dedicated secure chip, \textit{i.e.}, a Secure Element (SE), next to the main application processor on the Printed Circuit Board (PCB). Hence, as such a secure element is entirely independent of the REE, these secure execution
environments provide strong protection against software adversaries. In addition, Google’s Titan M2 [Goo23] or Apple’s T2 [App20] security chips also implement countermeasures against physical adversaries, such as fault attackers. To enable communication between secure and non-secure domain, protocols, such as Serial Peripheral Interface (SPI), are deployed.

Another example of an external processor based secure execution environment is Google’s OpenTitan processor [Joh+18]. The chip is intended to be used as a secure anchor point, i.e., Root-of-Trust (RoT), in smartphones and data centers. Hence, OpenTitan offers various services, such as cryptographic functions, key storage, and support for secure boot protocols. As a security breach could be fatal, these integrated circuits typically offer a certain level of protection [Roc+21] against fault attacks. RoT chips introduced so far are closed, proprietary designs making it necessary for the system integrator to trust the manufacturer of these devices. The OpenTitan [Joh+18] project aims to obviate this requirement by providing the first open-source RoT chip. The hardware design as well as the firmware of the secure element is open-source\(^1\). OpenTitan contains a rich set of hardware- and software IP, including a key storage and an AES accelerator.

Figure 2.4 highlights the main architectural building blocks of the OpenTitan secure element. The Ibex 32-bit RISC-V processor is connected over the TileLink bus to the program memory (flash), the data memory (SRAM), and several other IP blocks.

2.5 Memory Encryption

Memory encryption is a security primitive that is gaining ground in recent computer systems. The basic idea of this concept is to encrypt security sensitive or all data leaving the processor to the external memory using a Memory Encryption Engine (MEE). Vendors, such as Intel and AMD, motivate the integration of memory encryption into their processor with the protection against adversaries performing physical attacks, e.g., cold boot attacks [Hal+08], and cryptographic separation of virtual machines [AMD23; Int23b]. In the following, we first discuss different modes of memory encryption and then highlight three different MEE frameworks introduced by industry and academia that we utilize as a base for our countermeasures in the upcoming chapters.

\(^1\)https://github.com/lowRISC/opentitan
2.5.1 Memory Encryption Modes

Depending on the used memory encryption mode, i.e., encryption only or encryption and authentication, different security guarantees are provided.

**Data Confidentiality.** In this mode, the encryption engine ensures that an attacker, without knowing the encryption key, cannot reveal data stored in external memory. This prevents physical attacks, where the adversary aims to directly probe data in memory, e.g., cold boot attacks. Memory encryption engines achieve this level of protection by using block ciphers in different encryption modes. Examples are AES [DR02] in the XTS mode or QARMA [Ava17] in the Cipher Block Chaining (CBC) mode.

**Data Confidentiality & Authenticity.** Data confidentiality alone cannot mitigate other physical attacks, such as spoofing, splicing, or replay attacks [Elb+09]. In these attacks, the adversary either replaces a memory block with an arbitrary one (spoofing), with one from another position (splicing), or with an old one from the same location (replay). To protect against these attacks, data authenticity is required. Memory encryption engines offering this protection generate, during the encryption of a block, an authentication tag that is stored together with the encrypted data block into memory. When encrypted data is manipulated by an adversary (spoofing), the verification of the tag at the decryption fails with a high probability. As the address of the data block is incorporated into the authenticated encryption, splicing attacks also can be detected. To mitigate replay attacks, a nonce is used to differentiate between blocks stored at different points in time at the same memory location. Commonly Tamper-Evident Counter (TEC) trees [Elb+07] are used to efficiently and securely store these nonces in memory.

2.5.2 MEMSEC

MEMSEC [Wer+17] is an open-source memory encryption framework. By offering an Advanced eXtensible Interface (AXI) [ARM11] at the input as well as the output, MEMSEC can be directly integrated into the bus infrastructure of a SoC supporting this protocol. Then, all data sent from cache to the memory controller are transparently encrypted at memory writes and decrypted at memory reads. MEMSEC can be configured to use different encryption primitives offering different memory encryption modes for the protection.

For example, when using the QARMA cipher in the CBC mode, MEMSEC offers data confidentiality. QARMA, which is a lightweight tweakable block cipher with a block size of 64 bit or 128 bit and a key size of 128 bit or 256 bit, is also used in the ARM pointer authentication extension [Qua] to derive a cryptographic MAC of security-sensitive pointers (cf. Section 2.3.2). MEMSEC combines authenticated encryption offered by ASCON [Dob+21] with TEC trees

^https://github.com/IAIK/memsec
2.5. Memory Encryption

To provide data confidentiality and to mitigate spoofing, splicing, and replay attacks.

2.5.3 Intel TME and TME-MK

With the Ice Lake platform, Intel released in 2021 Total Memory Encryption (TME) [Int21] for their processors. At boot time, a random number generator in the CPU creates a key that is used by TME to encrypt all data leaving the processor to the external memory. By using AES in the XTS mode for the encryption, TME aims to provide data confidentiality protecting against physical attacks on the memory.

Intel’s Total Memory Encryption - Multi Key (TME-MK) is an extension introduced with the Alder Lake platform allowing the system to utilize multiple different keys for the encryption [Int22]. As shown in Figure 2.5, the memory encryption engine is located between the cache subsystem and the memory controller. On each memory request, the key identifier embedded into the upper bits of the physical address is used by the TME-MK engine to select the corresponding encryption key and mode. For the actual memory access, after the encryption, the key identifier is stripped from the physical address. To select an encryption key for a memory page, the key identifier field in the Page Table Entry (PTE) needs to be set. Currently, TME-MK supports AES-XTS with 128 bit or 256 bit keys and features up to $2^{15}$ different keys. However, the number of actually available keys is limited by the number of free bits in the physical address and the implementation of TME-MK. By default, the key linked to the key identifier 0 is randomly generated by the CPU and is used for system-wide encryption.

Currently, neither TME nor TME-MK support authenticated memory encryption. However, with the introduction of Intel’s Trusted Domain Extension (TDX) [Int23a] data integrity is offered. Here, the memory encryption engine uses a 28 bit SHA-3 based MAC at cache line granularity to detect data integrity violations.

![Figure 2.5: Intel’s TME-MK memory encryption engine.](image-url)
2.5.4 OpenTitan Scrambling Unit

To protect off-chip data, the OpenTitan secure element integrates a memory encryption engine into the SoC infrastructure. Here, OpenTitan can be configured to encrypt the data (flash) and program memory (SRAM) with the PRINCE [Bor+12] cipher. To reduce memory latency, the memory encryption engine uses a round-reduced version (5 instead of 11 rounds) of the cipher in the Xor–Encrypt–Xor (XEX) or Counter (CTR) mode. Hence, as therefore only limited cryptographic security for data confidentiality can be provided, OpenTitan calls the protection memory scrambling.
Part I

Mitigating Memory Safety Vulnerabilities
Memory safety vulnerabilities enable adversaries to leak or modify data and, for example, manipulate the control-flow of software. Hence, countermeasures addressing this threat are needed. However, current software- and hardware-based countermeasures either induce tremendous runtime overheads or provide insufficient protection guarantees.

To that end, we, in this part of the thesis, present a new memory safety primitive that is based on memory encryption. We showcase that leveraging a memory encryption engine enables the introduced architecture to efficiently and securely thwart memory vulnerabilities.


Together with the co-authors, I designed and implemented the CrypTag prototype. Moreover, I was in charge of conducting the performance and area evaluation. In addition, I led the writing of the paper text.
Cryptographic Memory Safety

Memory safety vulnerabilities enable the adversary to leak [Dur+14] or modify [Sha07; Ble+11; Hu+16] arbitrary data on a target system. To protect against these attacks, full memory safety, i.e., mitigating all spatial and temporal memory safety issues, is required. However, as indicated in Section 2.1, pure software-based solutions induce tremendous runtime overheads.

To reduce high performance penalties, hardware assistance can be used. A promising attempt to detect memory safety violations with hardware support are tagged memory architectures [Ser19]. Tagged memory assigns additional metadata to the memory, enforcing different security policies [Zel+08; Woo+14]. One policy allowing to detect memory safety vulnerabilities is memory coloring, which is implemented on top of tagged memory. The basic idea of memory coloring is to lock each memory allocation through a key. A later memory access is only permitted when using the correct key. This lock-and-key approach is implemented in the ARM Memory Tagging Extension (MTE) [Lima] and provides tagged memory in hardware. Google announced to work on deploying memory coloring based on MTE in Android on a larger scale [SH19], through the MemTagSanitizer [Pro20] project integrated into LLVM [LA04]. While this concept is a step in the right direction, the memory overhead for storing the tags is still problematic for large-scale applications. To reduce the memory overhead of memory coloring, ARM decided to limit their concept to small tags. In ARM MTE, a 16 byte memory block is tagged with a 4 bits tag, resulting in a memory overhead of 3.125%. While this memory overhead might be feasible for most applications, a tag size of 4 bits only leads to 16 individual colors, thus limiting the use of MTE as a security mechanism and making debugging the main application possible. Increasing the tag size from 4 to 16 bits not only increases the available color space and, therefore, also the security guarantees, but also
raises the memory overhead to 12.5%.

**Contribution**

To overcome the limitations of memory coloring approaches realized so far, we, in this chapter, combine memory coloring with memory encryption. Towards that end, we introduce CrypTag [Nas+21a], a hardware/software co-design mitigating a broad range of memory safety issues by providing cryptographically enforced memory safety. We demonstrate that realizing memory coloring on top of an already implemented memory encryption unit (cf. Section 2.5) only adds reasonable runtime overhead. In exploiting properties of the memory encryption scheme, we overcome limitations of traditional memory coloring schemes. While previous tagged memory architectures [Zel+08; Lima; Ain+15; SBM15; Joa+17; Son+16] store the tag in memory and trade security against lower memory overhead, CrypTag completely avoids storing tags in memory and thus allows using larger tag sizes. CrypTag uses the memory color as additional input for the memory encryption scheme to encrypt every allocation differently. Inside the processor, we store the color information directly in the upper bits of the pointer, avoiding any additional storage overhead there. The tag is propagated through the system, stored in the cache, and finally used to encrypt the data when being stored in memory. Based on the capabilities of the underlying memory encryption engine, we derive two security policies for CrypTag.

We further present a software concept utilizing the hardware architecture to mitigate memory safety violations. Our approach assigns each allocated memory object on the heap, stack, and global data a random color. When accessing a memory object with the wrong color, e.g., due to a spatial or temporal memory bug, the CrypTag architecture, in its strongest security policy, identifies the color mismatch. This strategy allows us to successfully detect most spatial and temporal memory vulnerabilities, providing memory safety.

To evaluate our concept, we implemented an FPGA prototype based on the RISC-V CVA6 core. Furthermore, we extended the LLVM compiler to automatically instrument the code and protect all memory allocations without the need for user annotations. We evaluate the performance of CrypTag by executing different programs, from microbenchmarks to application code on our FPGA-based prototyping platform with Linux as host operating system. The evaluation shows that the geometric mean performance penalty introduced by CrypTag in the strongest configuration is 27.16% on a system already featuring a memory encryption engine.

Summarized, our contributions are:

- We efficiently combine memory encryption with memory coloring and show that the overhead for storing tags in memory can be entirely eliminated. This allows us to scale the tag size without additional memory cost.

- We develop a hardware-assisted memory safety concept based on our memory coloring architecture. We demonstrate that the increased tag size of
CrypTag achieves stronger security guarantees than comparable hardware-assisted memory safety designs, such as ARM MTE.

- We extend the RISC-V CVA6 core with a memory encryption engine and our memory coloring approach. We further provide a modified LLVM-based toolchain enforcing hardware-assisted memory safety by automatically instrumenting the application code. We show that the hardware overhead, for a system already using a memory encryption scheme, is less than 1% and the software geometric mean overhead is 27.16% for the strongest CrypTag configuration. While highly optimized commercial memory encryption systems typically induce an overhead between 5% and 26% [Rob20], our evaluation of an open-source memory encryption unit with two different encryption primitives shows a geometric mean runtime overhead of 52.46% and 102.12% for SPEC CPU2017.

Outline

Section 3.1 provides background information on tagged memory. Then, in Section 3.2, we discuss the threat model presumed in this work and in Section 3.3 we highlight design goals of CrypTag. Section 3.4 presents CrypTag, our efficient memory coloring architecture utilizing memory encryption and Section 3.5 details the hardware and software implementation for the prototype. Section 3.6 evaluates the hardware overhead and software performance, and Section 3.7 discusses the security guarantees of CrypTag. Finally, Section 3.9 highlights potential future work and Section 3.10 concludes this chapter.

3.1 Background

In this section, we, in detail, introduce tagged memory and memory coloring.

3.1.1 Tagged Memory

The concept of tagged memory [Feu72; May82; CC89] is long-established and describes the idea of associating blocks of memory with additional metadata, i.e., tags. Particularly, $TG$-bytes of memory are linked with a $TS$-bits wide tag, where $TG$ denotes the tag granularity and $TS$ the tag size. In these early computer architectures, tags were primarily used for debugging and for dynamically tracking the numeric type of data words. However, since tag bits are only memory, somewhat arbitrary policies can be implemented [Ven+08; Dha+14]. Many recent designs utilize tags primarily for memory coloring, as shown in Figure 3.1. In such a coloring scheme, specific tag values, denoted as colors, are assigned to larger memory regions. When accessing the memory, these colors are used to determine if a particular read or write operation is genuine. A mismatch between the color of the accessed memory and the expected color results in a memory error. Memory coloring is used, e.g., for debugging [Ser+12], isolation [Woo+14], access control [Son+16; Wei+19], and for enforcing memory...
Chapter 3. Cryptographic Memory Safety

3. Cryptographic Memory Safety

With ARM’s new Armv8.5-A instruction set, the MTE [Lima] was announced, which embeds a tagged memory architecture into consumer hardware, such as mobile phones. A first attempt using the tagged memory approach on a larger scale is already integrated into the MemTagSanitizer [Pro20] project of LLVM. Similar to the address sanitizer ASan [Tea20a] and the hardware-assisted address sanitizer HWASAN [Tea20b], Google’s MemTagSanitizer intends to detect several spatial and temporal memory bugs. As the MemTagSanitizer benefits from hardware features, the high performance overhead of comparable software-based address sanitizer solutions is reduced to a minimum. Nevertheless, MTE requires the architecture to store the tags in memory. To avoid large memory overheads, MTE uses a small tag size of 4 bits, resulting in only 16 distinct memory colors. However, the security of the memory coloring scheme directly depends on the number of unique colors. Since colors are assigned randomly for each memory object, two adjacent objects can have the same color. For security critical systems, a detection probability of only 93.7%, when having a tag size of 4 bits, is insufficient. Increasing the tag size from 4 to 16 bits would already result in a detection probability of 99.998%, but also increases the memory overhead for tag storage from 3.125% to 12.5%.

While the Armv8.5-A architecture with the MTE feature has not yet been released in hardware, SPARC already implements a hardware-based memory tagging scheme with the Application Data Integrity (ADI) [Ain+15] feature embedded into Oracle’s SPARC M7 processor. Similar to ARM MTE, the SPARC ADI feature also only supports a tag size of 4 bits.

3.2 Threat Model

Similar to other threat models [Unt+23] in the context of memory safety, we are considering an adversary using an exploitable memory bug to craft a memory vulnerability. This exploitable memory bug could be a spatial or temporal memory safety error (cf. Section 2.1). The goal of the adversary is to exploit this bug to leak or modify security critical data, such as return addresses or secrets.

3.3 CrypTag Design Goals

Based on the capabilities of the underlying memory encryption engine (encryption only or with authentication), we design our hardware-assisted memory coloring
3.4 Design

design CrypTag in such a way that it provides two different levels of security guarantees.

**S1 Encryption & Authentication:** When detecting a spatial memory safety violation, CrypTag immediately triggers a system exception via the inbuilt authentication mechanism of the transparent memory encryption scheme. Here, CrypTag is capable of detecting out-of-bound reads or writes, i.e., a spatial memory bugs. Furthermore, CrypTag also is capable of reporting the exploitation of temporal memory bugs, e.g., use-after-free vulnerabilities.

**S2 Encryption:** Compared to S1, this security policy limits the exploitation of spatial and temporal memory bugs. For out-of-bound memory reads, CrypTag guarantees the confidentiality of the data stored in the target buffer. Since the underlying memory encryption engine does not provide data integrity, CrypTag cannot maintain the integrity of data in the target buffer in an out-of-bound memory write. However, CrypTag with S2 aggravates the exploitation of temporal bugs and spatial out-of-bound writes.

3.4 Design

In our architecture, which is depicted in Figure 3.2, memory is allocated in software and a dedicated instruction assigns a random color to the memory object and stores it in the upper bits of the pointer. When writing data to the memory, the color information is propagated through the Memory Management Unit (MMU), the cache, and then finally is used as a tweak in the memory encryption unit. On a memory access, the hardware transparently performs a cryptographic check without any further instrumentation. This hardware architecture allows CrypTag to protect dynamic, local, and global data.

3.4.1 Hardware

CrypTag utilizes a built-in memory encryption unit to implement an efficient memory coloring scheme. Initiated by a custom instruction, a random color is assigned for each memory object, which is stored in the upper unused bits of the pointer. When accessing the colored memory object, it requires the correct color to be in place for the memory request. CrypTag implements this lock-and-key approach by using the color of the memory object as an additional input for the transparent memory encryption scheme. Due to this strategy, each memory object colored with a random color is encrypted differently.

**Memory Coloring**

In CrypTag, the color of the memory object is assigned to the pointer. Since memory allocations are a frequent task and assigning and generating a color in software is costly, a custom instruction using a hardware-based random number
Chapter 3. Cryptographic Memory Safety

Figure 3.2: Overall CrypTag architecture. The memory encryption unit, placed between the memory subsystem and the memory controller, uses the color as a tweak.

generator is used. Similar to other designs [Qua; Kwo+13; Sch+18a], CrypTag uses the upper bits of the pointer to store the color information. This approach causes zero costs in terms of storing the color information and also minimizes any overhead to use pointers in software. Since the address information and the corresponding color are already stored in the same register, there is no need to extensively modify the instruction set. However, storing the color directly inside the pointer results in two disadvantages. First, the number of colors, which influences the security of memory coloring, directly corresponds to the number of free bits in the pointer. Second, using the upper bits of the pointer reduces the virtual address space of the system. Nevertheless, in practice, a trade-off between the available address space and security guarantees can be found. In most 64-bit platforms, already a reduced address space with free upper bits is used. For example, the AArch64 Linux port [Mar20] limits, by default, the virtual address space to 39 bits and, therefore, supports colors up to 25 bits. While this address space might be sufficient for, e.g., mobile devices, a 512 GB address space is not acceptable for high-performance servers. By using the larger 48 bits addressing model, the address space can be extended to address 256 TB and supporting colors up to 16 bits. Security limitations of different color sizes are discussed in Section 3.7. When using a colored pointer, the color needs to be propagated throughout the system up to the memory encryption. Since the MMU of the processor only considers the lower bits of a pointer to translate the virtual to the physical address, CrypTag needs to bypass the MMU translation and directly forwards the color information to the cache (see Figure 3.2).

Cache Architecture. Figure 3.3 shows our extension to the cache architecture. In CrypTag, each $TG$-bytes of memory $W$ are tagged with a $TS$-bits color $C$. For cache management, the color $C$ is also stored in the cache for each memory object $W$. A cache hit is only valid if the color stored in the cache matches the color stored in the pointer. The design of CrypTag also supports sub-cache line tag granularities, e.g., one color for two words, which can be configured.
In Section 3.5.1, we present a concrete cache implementation supporting the proposed color management.

Memory Encryption

When writing to memory, the color is used as a tweak to encrypt the data using the transparent memory encryption unit. To decrypt this data on a memory access, the read operation needs to have the correct color stored in the upper bits of the pointer. Depending on whether the memory encryption unit provides encryption and authentication or solely encryption, CrypTag either implements detection strategy S1 or S2.

**S1: Exception-based Notification.** This strategy is possible for Memory Encryption Engines (MEEs) providing encryption and authentication. The exception-based notification policy immediately triggers an exception if the system performs a wrong memory access on a color mismatch. The encryption operation \( C, T = Enc_{AE}(k, t, P) \) takes the encryption key \( k \), the color as the cipher tweak \( t \), and the plaintext data \( P \) as an input to compute the ciphertext \( C \) and the authentication tag \( T \) as the output. Both the ciphertext and the authentication tag are stored inside the memory, while the color is not. When reading data back from memory, the MEE verifies the integrity of the ciphertext and decrypts the data, i.e., \( P \parallel \bot = Dec_{AE}(k, t, C, T) \). On a successful ciphertext verification using the authentication tag, the cipher returns the correct plaintext data \( P \). If the integrity verification fails, e.g., owing to a wrong color, the MEE returns an error \( \bot \) and the system automatically triggers an exception indicating an invalid memory access.

**S2: Detection-based Notification.** The detection-based notification policy corrupts the data when performing a wrong memory access. Here, the architecture
Chapter 3. Cryptographic Memory Safety

uses a tweakable block cipher \( C = Enc(k, t, P) \) without authentication. The ciphertext \( C \) is computed using the encryption key \( k \), the color as the tweak \( t \), and the plaintext data \( P \). When reading data from memory, the MEE automatically decrypts the ciphertext under the encryption key and the tweak given by the memory color stored in the address. On a correct memory access, this also returns the correct plaintext data. However, using the wrong color on a malicious memory access decrypts the ciphertext with the wrong tweak leading to an invalid plaintext.

3.4.2 Software

The CrypTag hardware architecture alone does not thwart memory safety errors. It requires software support and protecting all memory allocations to detect most spatial and temporal memory vulnerabilities. The principle idea of the memory protection is that all memory allocations are colored, meaning that every associated pointer to a memory allocation stores a color in the upper bits of the pointer. Only when using the pointer with the correct color, the memory access is successful. Otherwise, depending on the detection strategy, either an error is raised or the payload data is destroyed. In this section, we describe how to use the hardware design in software to thwart memory safety vulnerabilities.

Heap Data. On each dynamic memory allocation, e.g., via a call to `malloc`, the returned pointer is assigned a dedicated random color. Furthermore, the memory is properly aligned to match the tag granularity \( TG \). As discussed previously, our design uses a dedicated hardware instruction to perform this operation and, therefore, only adds a small overhead to manage the colors. When accessing heap data later on, every access encrypts or decrypts the data automatically using the assigned tag information. When releasing dynamic memory through a `free` operation, the color information of the pointer is removed and the memory is released to the OS.

Local Data. Local allocations on the stack are aligned to match the tag granularity \( TG \). The corresponding pointer is colored using the custom instruction. Further accesses then encrypt or decrypt the data when accessing the memory.

Global Data. Protecting global data requires more effort. There are two possibilities to deal with global data. First, the protection of global data can be realized during compile time, where the compiler assigns each global variable a dedicated color. During the compilation, initialized global data is then encrypted using the pre-assigned color so that memory accesses in the program yield the correct data. However, this approach requires additional overhead to manage the colored pointers in software. Furthermore, access to global data always uses the same color for encryption, enabling e.g., replay attacks. To avoid the problem of replay attacks and unnecessary color management in software, we aim for a second approach. Our design replaces all references to global variables with a new
pointer. Additionally, the compiler adds a dedicated startup hook function for each global variable. During the startup of the program, this hook function first colors the new pointer. Second, it reads the unencrypted global data from the executable and then writes this data to memory using the new colored pointer. Thereby, the global data automatically gets encrypted using the colored pointer. This approach allows us to use the same instruction to randomly color the new pointer. By using a random color in the pointer, we also mitigate replay attacks since the global data is encrypted differently at every program start. While this approach enhances security, it also simplifies the software support.

3.5 Implementation

In this section, we introduce the base platform where we integrate CrypTag and show the necessary hardware extensions. We discuss the color generation and propagation and further explain how the memory encryption framework is used to implement the coloring scheme. Finally, we introduce the compiler extension utilizing the CrypTag architecture to protect data.

**Base Platform.** We build the prototype for CrypTag on top of the CVA6 platform [ZB19], a System-on-Chip (SoC) using a 64-bit 6-stage RISC-V processor supporting to run Linux when mapped to an FPGA. The CVA6 is extended with the open-source memory encryption scheme MEMSEC [Wer+17]. MEMSEC is placed between the data cache and the DDR3 memory controller and automatically encrypts all data leaving the processor.

3.5.1 Hardware Extensions

The necessary hardware extensions to implement CrypTag are minimal and only require two adaptions. First, the system requires a mechanism to create a color and to propagate it through the system. Second, the MEE needs to be extended to handle the additional color input for the cipher.

**Color Generation**

Tagging a memory region with a dedicated memory color is initiated in software. Thus, we extend the RISC-V instruction set with a dedicated instruction to allow performing this operation efficiently in software.

**mstp rd,rs.** To color a pointer, the custom instruction **mstp** is added to the RISC-V instruction set. This instruction takes the value from the source register `rs` (typically the pointer), colors it, and stores the result to the destination register `rd`. Our architecture uses the SV39 addressing model [Wat+16] of RISC-V, where the lower 39 bits of the virtual address space are used. The remaining upper bits of the pointer are set to the sign bit of the pointer value (either all-zero or all-one). The **mstp** instruction colors the pointer and replaces the upper 25 bits (T.S-bits)
with a random color value. To differentiate between a colored and a non-colored pointer, the color bits cannot be set to all-zero or all-one. The random color value is generated using a hardware-internal pseudo-random number generator, which is initialized during processor startup with a software inaccessible seed value.

**Color Propagation**

After instrumenting a pointer with the color bits, the MMU translates the virtual to a physical address. In SV39 of RISC-V, the MMU only uses the lower 39 bits of the address for its translation. The upper 25 bits containing the color information bypass the MMU’s address translation. Both, the physical address and the color bits, get processed by the L1 data cache and are then propagated to the MEE via the processor’s bus architecture.

**Cache Design**

As data in CrypTag is tagged with the color of the corresponding pointer, the cache also needs to be aware of these colors.

**Colors.** The prototype implementation of CrypTag uses a tag granularity (TG) of 16 bytes and a color size (TS) of 25 bits. As the default cache line size of the CVA6 processor is 16 bytes, each cache line stores the TS-bit color. Internally, the cache differentiates between three values for a color: no color, valid color, and invalid color. When accessing the cache with an address where there is no color stored inside (the upper bits are all-zero or all-one), the cache-internal color is set to no color. A invalid color is stored in the cache when prefetching a cache line with the wrong color triggers a decryption exception. When accessing the cache with an instrumented pointer, the valid color information is stored in the cache.

**Cache Hit.** A cache hit is triggered when having the correct data and correct color in the cache. If there is a color mismatch, a cache miss is triggered.

**Cache Miss.** On a cache miss, the cache architecture issues a memory read request to the main memory. Here, the colors of the cache line are used as a cipher tweak to decrypt data from the memory. After fetching the decrypted data from memory, the colors are stored in the metadata structure of the cache. In detection strategy S1, accessing the cache with a wrong color leads to a decryption and verification error and the error is forwarded to the processor as an exception.

**Cache Prefetching.** Cache prefetching is used to reduce the latency for memory accesses by precautionary fetching an entire cache line from memory. This technique speeds up memory accesses but also challenges our colored cache.
3.5. Implementation

Architecture. When performing a cache prefetch, only the color of the first $TG$ bytes of the cache line is known. To decrypt the remaining cache line, the system assumes the latter colors of the cache line are the same and uses the first color to decrypt the whole cache line. Due to the memory fragmentation, this assumption is correct with high probability and the MEE decrypts the cache line correctly. Furthermore, the used color is copied to all color entries of the cache line. In case of a wrong decryption operation due to prefetching, detection strategy $S1$ detects a wrong decryption and thus invalidates the color entry in the cache but does not raise an exception. When the tag granularity is identical to the cache line size, the objects in the entire cache line are tagged with the same color, i.e., the entire cache line can be successfully decrypted with the correct color.

Cache Eviction. During cache eviction, a dirty cache line is written back to memory and is encrypted using the colors stored inside the cache. In case of having invalid colors in the cache, i.e., due to prefetching, the cache only issues memory updates for entries with valid colors. Invalid cache entries are filtered and not written back to memory.

Memory Encryption Engine MEMSEC, which is directly placed between the cache and the memory controller, transparently encrypts all data leaving the processor on bus level. Similar to other MEEs, the encryption key is randomly generated during the device startup. For the CrypTag architecture, we extended the MEE to support the additional color input. To implement detection strategy $S1$, we use the authenticated encryption cipher ASCON [Dob+16], which provides data confidentiality and integrity. We tweak the cipher by using the size-extended color value for the nonce input of the cipher. To re-initialize already encrypted memory, MEMSEC also allows suppressing authentication errors using a defined memory pattern. For implementing strategy $S2$, we use the low-latency tweakable block cipher QARMA [Ava16], which is also used in ARM’s pointer authentication scheme [Qua]. Since QARMA natively supports an additional input, we use the size-extended color value as input for the tweak.

3.5.2 Software Extensions

To detect memory safety violations and to protect every memory allocation, we need software support. We extended an LLVM-based C compiler [LA04] with an LLVM IR pass and a tiny runtime support library. The compiler needs to protect three storage classes: heap, local, and global data.

Protection of Heap Data. Protection of heap data is accomplished by using the GNU linker functionality to create wrappers around heap functions such as $malloc$, $free$, and $realloc$. For policy $S2$, the $malloc$ wrapper aligns the size argument to $TG$ and then calls $malloc$ itself. Because heap allocation for 64-bit RISC-V systems is already 16B aligned, we only have to apply the
mstp instruction on the pointer before returning to the application. Hence, the overhead therefore is negligible. In Listing 3.1 we show the implementation of the wrapped malloc function for S2 of the runtime library. When utilizing CrypTag for detection policy S1, malloc additionally initializes the memory with its color. Since this memory object could already be encrypted with a different color, naively re-coloring would trigger an authentication error. Thus, CrypTag uses the nullification mechanism of MEMSEC to initialize the memory.

Listing 3.1: S2 malloc wrapper tagging the returned pointer.

```c
void* __wrap_malloc(size_t size) {
    size = roundup(size);
    void* ptr = __real_malloc(size);
    if (ptr == NULL)
        return NULL;
    return mstp(ptr);
}
```

For free, the wrapper removes the color from the pointer argument and then calls the free function. This operation is done purely in software by using two shift operations. Notice that the heap administration data is stored in plain in between the encrypted heap data. Writing via a heap data-pointer out of bounds into the heap administration overwrites plain data with encrypted data making it hard for an attacker to modify the heap administration in a controlled way.

**Protection of Local Data.** Local variables on the stack are protected by scanning for AllocaInst instructions through a custom LLVM IR compiler pass. For each AllocaInst, we align the size argument to TG bytes, we align its address alignment to TG, and we insert an mstp instruction between the AllocaInst instructions and all its users. Additionally, for S1, we re-initialize the allocated memory with the assigned color. We perform a simple analysis on AllocaInst instructions to exclude protection of cases where incorrect usage will not be possible. For example, cases where the result of AllocaInst is not used by a GetElementPtrInst instruction with non-constant indices and the result is not stored in memory or leaves the function as argument of a function call.

**Protection of Global Data.** For each global definition/declaration of a variable named foo, we create a new global definition/declaration of a pointer called __foo_mst that points to foo. Furthermore, the LLVM IR pass replaces all references to foo with __foo_mst to get a colored pointer to foo. The runtime support library is informed about definitions foo, the size of foo, and the new pointer __foo_mst via a constructor function. During the startup, the constructor function in the runtime library will insert a color on __foo_mst by means of an mstp instruction. It will also encrypt foo using the color that has been put on __foo_mst. This happens by simply reading the data in plain using the original all-zero pointer and then writing it back to memory using
The runtime overhead of referencing a global variable is therefore one-load instruction. Furthermore, notice that on every execution of a protected application, its global variables will get different colors.

As with local data, we perform an analysis to exclude protection of (static) global data where we can prove that incorrect usage is not possible. Protecting global variables that have initializers with pointers to global variables complicates the protection. The runtime support library is informed via a constructor function about the positions of these pointers in global the initializers and patches those pointers with the color of the global variable it is pointing to.

**Backward Compatibility.** Application code that is protected by CrypTag can be combined with unprotected code. For example, the unprotected pointer results of `fopen()` or `mmap()` can be used in protected code without problems. Furthermore, protected pointers of heap, local, or global data can arbitrarily be passed to unprotected library functions without problems. The only compatibility issue that we are aware of is sharing global variables between protected and unprotected code, i.e., `stdout`. Unprotected code will expect it unencrypted, while protected code will expect the data to be encrypted. Due to indirection via `__foo_mst` this issue will result in linking errors rather than runtime errors. The user should then manually place these global variables on a blacklist of global variables that are not suitable for protection.

**Pointer Arithmetic.** Incrementing a pointer to reach data within a colored object is natively supported in the CrypTag architecture because the color information in the upper bits of the pointer is not altered. However, subtracting or adding two pointers or performing shifts or multiplications on such pointers can modify the color and is therefore dangerous. To also support these operations and enable safe arithmetic operations avoiding integer overflows on colored pointers, dedicated instructions could be added. In the Armv8.5-A instruction set [Lima] supporting MTE, dedicated add and subtract instructions, ignoring the upper bits, are used for pointer arithmetic.

### 3.6 Performance Evaluation

To quantify the hardware and software overhead, we synthesize the CrypTag architecture for a Xilinx Kintex-7 series FPGA and run a recent Linux operating system on our platform. We report the performance and hardware overhead introduced by CrypTag for the $TS=25$ and $TG=16$ memory coloring configuration using different applications, from microbenchmarks to application code.

#### 3.6.1 Hardware Overhead

In Table 3.1, we show the hardware overhead for the FPGA design in terms of Lookup-Tables (LUTs) and Flip-Flops (FFs). The overhead numbers include the changes required for the new instruction, the color propagation, and the extended
cache. Clearly, the hardware overhead of less than 1\% is very attractive and negligible in practice.

**Cache Architecture.** For CrypTag, the cache architecture is extended to also store colors along with the data. Furthermore, the decision logic to detect cache hits and misses is extended to also consider the colors in the cache. The hardware overhead for this comparison logic is relatively small compared to the overhead for storing the colors. The required hardware overhead for storing the colors in the cache is a function of the used color size $TS$ and tag granularity $TG$. Note, our design only needs to store colors in the cache and there is no need for a separate, large cache for colors as it is required in other architectures [SBM15; Joa+17] to speed up accessing tags in memory. Thus, the design not only has less hardware overhead but also improves the runtime latency and bandwidth, as there are no memory accesses for colors needed. For an $m$-way $n$-set associative cache with a cache line size of $C$ bytes, a tag granularity of $TG$ bytes, and a color size of $TS$ bits, we can compute the required number of color bits $T$ as defined in Equation (3.1).

$$ T = n\text{Sets} \cdot m\text{Ways} \cdot TS \cdot \frac{C}{TG} \quad (3.1) $$

The 16 kB data cache of the CVA6 core is organized as a 8-way 128-set associative cache with a 16 B cache line. For a tag granularity of $TG=16$ bytes and a color size $TS=8$ bits, the overhead for storing the colors is 8192 bits. For a configuration with $TG=16$ bytes and $TS=25$ bits the overhead for storing the colors is 25,600 bits. Table 3.2 shows the total number of color bits for different configurations including the corresponding overhead.

For our Xilinx-based FPGA, the cache is mapped to multiple block RAM instances. For a small memory color configuration (course tag granularity and small color size), the color bits even fit in the already instantiated block RAM resources of the cache and thus have no impact on the block RAM utilization. Only for the worst-case memory color configuration ($TG=16$ byte and $TS=25$ bits), an additional block RAM module needs to be instantiated. For other hardware technologies, e.g., ASIC designs, the cache overhead directly results from the color size and tag granularity. For, e.g., a 16 B cache line and a memory coloring configuration of $TG=16$ byte and $TS=25$ bits, a cache line is extended by 25 bits

| Table 3.1: Hardware overhead for the $TS=25$ and $TG=16$ memory coloring configuration. |
|---------------------------------|-----------------|-----------------|
| Config. | LUTs [LUTs] | Baseline [\%] | Overhead [\%] | FFs [FFs] | Baseline [\%] | Overhead [\%] |
| ASCON  | 57386 | 0.53 | 33885 | 0.14 |
| QARMA  | 55804 | 0.67 | 32173 | 0.18 |
3.6. Performance Evaluation

for storing colors, resulting in an overhead for the cache of 19.5%.

3.6.2 Runtime Overhead

To evaluate the impact of CrypTag on the system performance, we split our evaluation into two parts: First, we measure the runtime overhead of the memory encryption engine by executing and comparing a benchmark suite with and without the MEMSEC MEE. Then, we analyze the relative runtime overhead of CrypTag utilizing the MEE for memory safety.

**Memory Encryption Runtime Overhead.** To measure the runtime impact of integrating a memory encryption engine into our architecture, we embed the MEMSEC MEE into the CVA6 platform and execute the SPEC CPU2017 and MiBench benchmarks on the protected and unprotected system.

Figure 3.4 depicts the percentual runtime overhead of different SPEC CPU2017 benchmarks in the MEMSEC ASCON and QARMA configuration. Here, we observed a runtime overhead between 57.64% and 186.25% for ASCON with a geometric mean of 102.12%. For QARMA, we measured a runtime overhead between 23.53% and 106.24% and a geometric mean of 52.46%. Similar, as shown in Figure 3.5, we measured a runtime overhead for MiBench between 42.43% and 193.47% (geometric mean 112.75%) for ASCON and 31.02% and 117.18% (geometric mean 68.15%) for QARMA.

<table>
<thead>
<tr>
<th>Cache Configuration</th>
<th>Color bits T [bit]</th>
<th>Cache Overhead [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS=8, TG=16, C=16</td>
<td>8192</td>
<td>6.25</td>
</tr>
<tr>
<td>TS=25, TG=16, C=16</td>
<td>25600</td>
<td>19.53</td>
</tr>
<tr>
<td>TS=8, TG=64, C=64</td>
<td>8192</td>
<td>6.25</td>
</tr>
<tr>
<td>TS=8, TG=16, C=64</td>
<td>32768</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 3.2: Number of additional cache bits and total cache overhead for storing colors.
Figure 3.5: Runtime overhead for MiBench with MEMSEC.

Figure 3.6: Runtime overhead for Embench-IoT with CrypTag on a system already featuring a memory encryption engine.

**Embench-IoT.** To measure the software overhead of CrypTag, we compiled the Embench-IoT benchmark with our custom LLVM-based toolchain protecting all dynamic memory, all locals, and all globals. We evaluate the performance overhead by running the different benchmark applications in user mode on the Linux environment running on our FPGA hardware implementation. Note that the entire system, including the Linux operating system, is executed in the encrypted memory domain, but only user applications are additionally instrumented and use memory coloring.

As shown in Figure 3.6, the relative geometric mean overhead for thwarting logical memory safety vulnerabilities on a system already featuring memory encryption is 27.16% for $S_1$ and 25.21% for $S_2$. More specifically, we measured a runtime overhead between 1.90% and 77.81% for $S_1$ and 1.64% and 76.93% for $S_2$ with the Embench-IoT benchmark. This overhead is induced by the instrumentation of pointers with the `mstp` instruction, the alignment of memory objects to the tag granularity, and the color handling in the cache.

**Code Size Overhead.** As we are linking the 244 bytes runtime library while building the binary, the code size overhead to protect dynamic memory is constant and negligible compared to the overall binary size. Furthermore, to also
3.6. Performance Evaluation

3.6.3 Prototype Limitations

As seen in Section 3.6.2, the main factor of the runtime overhead is the MEE and not CrypTag. Hence, our performance evaluation largely is affected by the performance of the underlying memory encryption unit. However, MEMSEC, the only, to the best of our knowledge, open-source MEE available, is not optimized for throughput and latency. Figure 3.7 depicts the significant impact of MEMSEC with ASCON and QARMA on the latency measured by the lat_mem_rd 64M 512 benchmark of LMBench [MS96]. The memory throughput, measured with bw_mem 4M rdwr, also dropped from 52 MB/s to 14 MB/s for QARMA and 10.9 MB/s for ASCON. In comparison, state-of-the-art encryption engines typically yield a performance penalty between 5% and 26%, as reported by ARM [Rob20]. Although optimizing MEMSEC or designing a high-speed MEE is not part of our contribution, we point out different optimization strategies in Section 3.10. Finally, we want to emphasize that we envision CrypTag to be an extension of systems already featuring a transparent memory encryption scheme. With major vendors, such as Intel with SGX, TME, and MKTME [SAL20] and AMD with SME and TSME [KPW16], highlighting the importance of memory encryption, we expect an increasing number of such systems in the near future. Here, CrypTag proposes an efficient solution to realize memory coloring on top of such systems with performance overheads of 27.16% ($S_1$) and 25.21% ($S_2$) for the geometric mean.
3.7 Security Evaluation

CrypTag enhances security guarantees of applications by mitigating the exploitation of most temporal or spatial memory bugs. Based on the underlying MEE, i.e., encryption and authentication or encryption only, CrypTag either enforces security policy $S_1$ or $S_2$.

Spatial Memory Safety in $S_1$. Spatial memory bugs allow an adversary to access data outside of the objects bound. To detect these bugs, CrypTag utilizes the architecture to color the pointer and to initialize the memory object with a random color on a memory allocation. Any subsequent access to this colored object requires that the access pointer is colored with the identical color, or an authentication error is triggered by the MEE. Hence, out-of-bound read or write accesses to memory objects with a wrongly colored pointer are detected by CrypTag in $S_1$. Similar to other tagged memory schemes, CrypTag cannot detect intra-object overflows. Since the vulnerable buffer, as well as the target, are stored in the same memory object, both objects have the same color.

Spatial Memory Safety in $S_2$. Compared to $S_1$, this security policy limits the exploitation of spatial memory bugs. Usually, the attacker either uses spatial memory vulnerabilities to leak sensitive data or to modify control or non-control related data to craft ROP, DOP, or other attacks. In out-of-bound read accesses, data encrypted with the original color is decrypted using the wrong color of this pointer. Hence, CrypTag with $S_2$ maintains the confidentiality of data in spatial out-of-bound reads and provides protection from attacks such as Heartbleed [Dur+14]. Since the underlying MEE does not provide data integrity, CrypTag cannot prevent an attacker from overwriting data in a target buffer using an out-of-bound write. However, when reading this data using the corresponding pointer, pseudorandom values are retrieved. Using this pseudorandom value as control-flow related data, e.g., as a return address, most likely will cause an exception. As the attacker cannot overwrite data in the target buffer in a controlled way, CrypTag raises the complexity for performing data-oriented attacks, such as DOP.

Temporal Memory Safety in $S_1$. In a temporal memory safety violation, a memory object is accessed after it is deallocated. Temporal memory safety violations are mostly exploited by use-after-free vulnerabilities, which CrypTag with $S_1$ can detect. In this attack, a memory object gets deallocated and the space, later on, is used by a new object. The attacker then can use the dangling pointer either to leak sensitive data or to tamper data, e.g., a vtable pointer. A similar concept is used by a double-free attack, where the adversary calls the memory deallocation functionality twice. CrypTag with $S_1$ mitigates such attacks by assigning a new color on each memory allocation and initialization, reading or writing by using the dangling pointer colored with the previous color will trigger an exception. Since the current implementation of CrypTag does not re-color the
memory object on deallocation, a memory read or write to this memory region using the dangling pointer cannot be detected by CrypTag. However, as soon as a new memory object is allocated and initialized on this region, it is tagged with a new color and accesses using the dangling pointer can be detected. To prevent this behavior, CrypTag could be, similar to ARM MTE, extended to colorize memory objects with a new color on each deallocation.

**Temporal Memory Safety in S2.** Although CrypTag with S2 cannot detect temporal memory bugs, it prevents the adversary from leaking data, i.e., CrypTag maintains the confidentiality of data. When using this vulnerability to overwrite sensitive data, such as vtable entries, the attacker cannot insert targeted data because the wrong color of the dangling pointer for the encryption is used.

**Null Pointers.** Pointers created in external libraries, which are not compiled with CrypTag, are not colored and thus have the all-null color. Unlike CHERI [Woo+14], where only the pointer is tagged with additional metadata and not the memory itself, CrypTag explicitly tags memory objects with its color. Hence, a read or write vulnerability on a null-colored object only allows the attacker to access other null-colored memory objects and not the whole memory. Colored data that is allocated by protected code can be passed to unprotected code, e.g., external libraries, and is also protected there.

**Stack Coloring.** By coloring the stack pointer with \texttt{mstp} on program initialization, all objects on the stack, which are not explicitly colored, i.e., stack spills, are assigned a random color. This strategy allows CrypTag to separate the stack from null-colored objects, e.g., objects created in unprotected code.

**Entropy.** Similar to countermeasures like PARTS [Lil+19], MTE [Lima], and ASLR CrypTag is a probabilistic mitigation technique. A memory safety violation, such as a linear or non-linear buffer overflow, cannot be detected in S1 by the CrypTag architecture if the color of the target memory object matches the color of the exploited memory object. However, since CrypTag already detects a memory safety violation at the first mismatch and the attacker cannot influence the color assignment of a memory object, the attacker requires a color collision at the first try. The probability of having a color collision of two memory objects directly corresponds to the number of used color bits. A memory safety vulnerability, such as Heartbleed, can be detected with S1 at the first violation with a probability of 93.7\% for a tag size of 4 bits, with a probability of 99.998\% for a tag size of 16 bits, and for a tag size of 25 bits even with a higher probability. Additionally, since attacks like ROP or JOP require the adversary to build an attack chain, multiple color collisions are required increasing the detection probability. Although larger color sizes also increase the security guarantees, schemes like ARM MTE do not utilize the full available space in the free upper bits of the pointer because storing the color in memory is required, resulting in significant memory overheads. CrypTag prevents this security-overhead trade-off by completely avoiding storing
the color in the memory, allowing the scheme to fully utilize the unused bits in the pointer and maximize security guarantees.

**Tag Granularity.** Due to its nature, memory coloring is an imprecise protection mechanism. For example, when allocating a 30 B memory object in CrypTag with a tag granularity of 16 byte, the full 32 bytes are colored with the same color. When accessing byte 31 using a linear buffer overflow, the memory safety violation cannot be detected by any memory coloring scheme. However, in practice, this issue can be circumvented by choosing an appropriate tag granularity. On 64-bit RISC-V systems, objects on the heap are 16 byte aligned. Here, by choosing a tag granularity of also 16 byte, the adjacent target memory object is tagged with a different color and cannot be reached by the attacker in \( S_1 \). Objects on the stack are also aligned to \( TG \) and the size is increased to a multiple of \( TG \). Now, the victim and target buffer, e.g., a return address, are in different color domains allowing CrypTag in \( S_1 \) to detect a memory safety violation. When the memory object size would not have been resized to a multiple of \( TG \) and the tag granularity would be larger than the memory alignment, e.g., \( TG = 64 \), the same color is assigned to two, e.g., 32 B, adjacent memory objects making it impossible to detect an overflow. Although a smaller \( TG \) allows a more fine granular detection mechanism, it also increases the overhead for storing the colors in the cache architecture. Similar to other research [Ser19], we suggest using a tag granularity of 16 byte on our reference platform.

**Color Checking.** Schemes like PARTS, which uses ARM’s pointer authentication feature [Qua], or CCFI [Mas+15] use dedicated authentication instructions to verify the integrity of the pointer. Since verification and usage is, except for dedicated instructions like the \texttt{bira} instruction in ARM, not atomic, these countermeasures are vulnerable against Time-of-Check to Time-of-Use (TOCTOU) attacks. CrypTag in \( S_1 \) circumvents this problem by enforcing a color check automatically in hardware for each memory access.

**Color Management.** Coloring a memory object with a color either can be done using a randomized or a deterministic color assignment strategy. When using a deterministic coloring scheme, a color management mechanism needs to track the color assignment to assure that two adjacent memory objects have a different color. An example of a system deterministically assigning tags is ARM’s pointer authentication scheme, which is integrated into Apple smartphones. However, past research [Aza19] showed that an attacker can forge arbitrary signed pointers by using signing gadgets. To prevent color management security issues and avoid additional overhead introduced by the mechanism, CrypTag uses a randomized coloring approach.
3.8 Related Work

In Table 3.3, this section summarizes and compares security guarantees and runtime overheads of different memory vulnerability schemes to CrypTag.

### 3.8.1 Overhead Comparison

As shown in Table 3.3, the performance overhead of a geometric mean of 27.16% for the strongest CrypTag configuration on a system already featuring a memory encryption unit is similar or lower than the overhead induced by comparable systems. These numbers show that extending a system with an already integrated memory encryption scheme with CrypTag is reasonable, as memory safety can be implemented relatively cheaply. We argue that with the increasing amount of systems providing memory encryption, such as Intel SGX, TDX, TME-MK, or AMD TSME, also the number of platforms potentially supporting CrypTag increases. Typically, highly optimized memory encryption units can be implemented with an overhead between 5% and 26% [Rob20]. As we do not have access to these commercial MEEs for our prototype, we used the open-source MEMSEC framework, where we measured a geometric mean performance overhead between 52.46% and 102.12% for SPEC CPU2017. These numbers show that the dominating performance factor is MEMSEC and not CrypTag. Furthermore, we contend that a naïve combination of logical and physical memory safety, such as combining PARTS with memory encryption, accumulates both overheads.

### 3.8.2 Security Comparison

In general, logical memory safety strategies can be categorized into schemes that are limiting the attacker’s capabilities when exploiting a memory bug (e.g., PARTS, CPI, CCFI) and those detecting the exploitation of a memory safety

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Code-Pointer Integrity</th>
<th>Data-Pointer Integrity</th>
<th>Temporal Safety</th>
<th>Spatial Safety</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCFI</td>
<td>✔</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
<td>52%</td>
</tr>
<tr>
<td>CPI</td>
<td>✔</td>
<td>✔</td>
<td>✖</td>
<td>✖</td>
<td>8.4%</td>
</tr>
<tr>
<td>PARTS</td>
<td>✔</td>
<td>✔</td>
<td>✖</td>
<td>✖</td>
<td>19.5%</td>
</tr>
<tr>
<td>SoftBound+CETS</td>
<td>▲</td>
<td>▲</td>
<td>✔</td>
<td>✔</td>
<td>116%</td>
</tr>
<tr>
<td>MemTagSanitizer</td>
<td>☐</td>
<td>☐</td>
<td>✖</td>
<td>✔</td>
<td>-</td>
</tr>
<tr>
<td>CrypTag</td>
<td>▲</td>
<td>✖</td>
<td>✔</td>
<td>✖</td>
<td>27.16%*</td>
</tr>
</tbody>
</table>

* Without Memory Encryption Overhead
bug. CrypTag with S1 uses the latter approach to thwart logical memory safety attacks by detecting a broad range of spatial and temporal memory bugs.

**Control-flow integrity.** Control-Flow Integrity (CFI) minimizes the attacker’s capability when exploiting a memory bug by limiting the control-flow of a program to only valid paths through the Control-Flow Graph (CFG) [Car+15]. The security of CFI schemes depends on the precision of the CFG, which is typically determined using static analysis, and the reliability of the security enforcement. Cryptographic CFI (CCFI) [Mas+15] improves the precision of commodity CFI schemes by dynamically performing pointer classification at runtime. Similar to CrypTag, CCFI utilizes cryptography to enforce runtime security. Each object that influences the control-flow of a program is tagged with the MAC over the pointer and its dynamically determined class. The MAC is then checked before using the object. Since computing and verifying a MAC is costly, CCFI increases the overhead by 52%. Due to the nature of CFI schemes, CCFI cannot provide spatial and temporal memory safety, as shown in Table 3.3 (✖).

**Code-pointer integrity.** Similar to CFI, Code Pointer Integrity (CPI) [Kuz+18] claims to prevent all control-flow hijack attacks, while simultaneously decreasing the performance overhead. CPI protects sensitive code-pointers by storing them and metadata in a safe region. While the overhead introduced by CPI is negligible, its security completely relies on the isolation of this region. On systems without segmentation protection support, like for x86-64 systems, CPI uses information hiding to protect its safe region making it vulnerable to attacks leaking this location [Eva+15]. As shown in Table 3.3, similar to CFI, CPI cannot provide spatial and temporal memory safety (✖).

**Code- and data-pointer Integrity.** Advanced attack scenarios, like ROP or DOP, show that providing data- or control-flow integrity exclusively is not sufficient. It requires a combination of defense strategies to mitigate against a powerful attacker. One promising attempt utilizing hardware features offered by the underlying architecture is PARTS [Lil+19]. PARTS implements a compiler instrumentation, which automatically adds pointer integrity checks to protect all code- and data-pointers. Here, dedicated pointer authentication instructions are used to perform pointer signing and verification. PARTS protects all backward-edge and forward-edge code-pointers, as well as all data-pointers. However, the data-pointer integrity scheme does not provide temporal or spatial memory safety, as shown in Table 3.3 (✖). Therefore, PARTS is vulnerable against attacks targeting the data plane, like Heartbleed [Dur+14], or other security-critical attacks on non-control data [CXS05].

**Memory Safety.** Memory safety prevents the exploitation of memory bugs. As such, it is considered to be a stronger concept than mitigating the effects of an exploited memory bug [Sze+13]. However, software-based solutions, like the combination of SoftBound and CETS, typically yield a high performance penalty,
3.9 Future Work

As mentioned in Section 3.6.3, the performance of CrypTag largely depends on the performance of the MEE. Hence, a possible future work would be to optimize the performance of MEMSEC. Currently, MEMSEC operates at the same clock frequency as the processor core. To increase the memory bandwidth and further decrease the latency of memory accesses, MEMSEC could be placed next to the memory to operate on a much higher clock frequency. However, this requires to optimize the inner logic of MEMSEC to avoid any timing violations. Currently, MEMSEC is a highly flexible framework allowing several corner cases, such as AXI bursts and strobes. Here, one strategy to maximize the performance of the MEE could be to adapt MEMSEC to the target architecture and remove functionalities not supported by this architecture. If providing physical memory safety is not needed, a final optimization step could be to only encrypt colored memory objects and bypass the MEE for non-colored objects.

3.10 Conclusion

We have shown that current memory security schemes either are incomplete [Mas+15; Lil+19], do not provide enough security [Kuz+18; Lima], or add non-negligible overhead [Nag+09; Nag+10], especially when combined with physical memory safety, to the system. CrypTag closes these gaps by introducing a memory safety concept based on a hardware-assisted memory coloring scheme. CrypTag combines memory encryption with memory coloring to thwart a broad range of physical and logical memory safety vulnerabilities. By combining these two mechanisms, we have demonstrated that memory coloring almost comes for free and memory safety vulnerabilities can efficiently be protected. The design uses a color, stored inside the related pointer, and propagates this value up to the data cache of the system. This color value is used to tweak the memory encryption system, thus avoiding storing the color in memory. Our approach shows that the performance overhead for CrypTag is reasonable and, therefore, can be used for large scale deployment. In this chapter, we have provided an end-to-end solution from the concept to the prototype implementation of our design. We have integrated CrypTag to a RISC-V based processing platform and adapted
an LLVM toolchain and developed a runtime library to automatically instrument programs and protect all memory allocations of the application without the need for user annotations. Our evaluation shows that the hardware overhead for these changes is less than 1% and the geometric mean software overhead compared to a system already featuring a memory encryption unit is 27.16% in the strongest CrypTag configuration, which makes this design practical for real-life applications.
Part II

Counteracting Fault Attacks
In this part of the thesis, we first show that fault attacks can be utilized to manipulate the control-flow of a system, even in the absence of logical software bugs. Then, based on this attacker model, we introduce cryptographically enforced control-flow integrity schemes that can be deployed with minimal hardware changes on a secure element or without hardware modifications on recent Intel processors. Based on our work on control-flow integrity schemes, we identify weaknesses in the protection of indirect branches and introduce a mitigation strategy based on the ARM pointer authentication extension. Within this thesis, we demonstrate that control-flow integrity can also be utilized to protect hardware primitives, i.e., finite-state machines, against control-flow redirections. Finally, we emphasize the importance of conducting a pre-silicon analysis of fault countermeasures and introduce a framework capable of verifying protection guarantees of hardware-based fault mitigations.

Summarized, this part of the thesis consists of the following contributions:

- Chapter 4 is based on the following publication that was presented at escar USA 2019 in Ypsilanti, Michigan, USA: Pascal Nasahl and Niek Timmers. “Attacking AUTOSAR using Software and Hardware Attacks.” In: Embedded Security in Cars USA (escar). 2019 In this work, I was responsible for porting AUTOSAR to the platform, setting up the attack gear, as well as developing the attack idea and conducting the end-to-end exploit. Moreover, the text in this chapter was written by myself. I worked on this paper before the start of my doctoral studies while being employed at Riscure.

- Chapter 5 discusses our paper on cryptographic control-flow integrity for the OpenTitan secure element that was presented at GLSVLSI 2023 in Knoxville, TN, USA (best paper award - third place): Pascal Nasahl and Stefan Mangard. “SCRAMBLE-CFI: Mitigating Fault-Induced Control-Flow Attacks on OpenTitan.” In: Proceedings of the Great Lakes Symposium on VLSI 2023. Association for Computing Machinery, 2023, pp. 45–50. DOI: 10.1145/3583781.3590221 I am the main author of this work. Here, I was responsible for designing and implementing SCRAMBLE-CFI as well as evaluating the area and runtime overhead. Moreover, I wrote the entirety of the paper.

In collaboration with the co-authors, I derived the idea for the cryptographically enforced control-flow integrity scheme. The toolchain was developed by myself and I contributed to the modified hypervisor developed by the co-authors. Finally, I was in charge of the performance evaluation and wrote most of the paper text. This work was completed while I was employed at Intel Labs.


Together with the co-authors, I designed the fault protection mechanism for indirect branches based on ARM’s pointer authentication hardware extension. I implemented the custom LLVM-toolchain automatically protecting programs with our approach and performed the entire evaluation. Most sections of the paper were written by myself.


In this paper, I derived the original idea for the fault protection approach and designed, together with the co-authors, the used MDS-based diffusion layer. Moreover, I led the implementation of the research prototype and conducted the area and timing overhead analysis. The analyzed hardware designs were provided by the co-authors. Finally, the main portion of the paper was written by myself.


In this publication, I was responsible for investigating approaches to analyze the resilience of fault countermeasures at the gate-level netlist. In this context, I designed and implemented the SYNFI tool and conducted the
entire verification of the secure element. The countermeasures introduced in the paper were contributed by the co-authors. Finally, most of the paper text was written by me. This work was conducted while I was employed at Google.
Fault-Induced Control-Flow Manipulations on Automotive Systems

Manipulating the control-flow of software executed on a processor enables an adversary to execute arbitrary code on the device under attack. As discussed in Section 2.1, a software attacker can achieve full code execution by exploiting memory safety vulnerabilities and using Return-Oriented Programming (ROP) [Sha07], Jump-Oriented Programming (JOP) [Ble+11], or Data-Oriented Programming (DOP) [Hu+16] attack techniques. However, when strong memory safety countermeasures (cf. Chapter 3) are in place, the software is written in a memory-safe language, such as Rust [MI14], or a formally verified software stack, such as seL4 [Kle+09], is used, performing control-flow manipulations with software attacks is challenging.

Contribution

This chapter shows that gaining full code execution on a device under attack can also be achieved with fault attacks, even in the absence of exploitable software bugs [NT19]. For the attack, we focus on a fully-fledged automotive system consisting of an ARM Cortex processor connected to an intra-vehicle communication network. Our system runs an automotive real-time operating system that implements the AUTOSAR specification. In our exploit, we assume a physical adversary capable of injecting messages into the bus and inducing faults into the electronic control unit. We show that by targeting the communication stack with faults, the adversary can achieve full malicious code execution on the device.
This chapter is structured as follows. In Section 4.1 we provide background on
automotive systems. Section 4.2 describes the attacker model and Section 4.3 the
attack setup. In Section 4.3.1 we then perform a fault characterization on the
device. Section 4.4 highlights the end-to-end exploit on the automotive platform.
Finally, Section 4.5 discusses countermeasures and Section 4.6 concludes this
chapter.

4.1 Background

In this section, we provide background on Electronic Control Units (ECUs) and
AUTOSAR.

4.1.1 Electronic Control Units

A modern car contains more than 100 ECUs that are responsible for managing the
gine, controlling the brakes, and offering telematic services [Dij]. These ECUs
utilize a set of sensors and actuators to influence the vehicle’s behavior. Internally,
a control unit consists of a Microcontroller Unit (MCU) that, for example, is
based on an ARM Cortex processor [Bab]. Communication between the different
ECUs is realized by using robust automotive bus systems, such as Controller Area
Network (CAN), FlexRay, or Automotive Ethernet. Usually, the bus interconnect
does not offer authentication, i.e., an adversary with physical access to the bus
infrastructure can arbitrarily intercept or inject messages [NR16].

4.1.2 AUTOSAR

In an automotive environment, the tasks and responsibilities of an operating
system differ from classic computer systems. Here, the focus is on real-time
sensor information processing and heavily using bus protocols for communication
with other ECUs. Hence, tailored automotive operating systems are deployed on
ECUs. To provide intercompatibility among OEMs and suppliers, most players
in the car industry agreed on using a common standard, i.e., the Automotive
Open System Architecture (AUTOSAR) [AUT]. Internally, the AUTOSAR spec-
ification uses a three-layer abstraction approach to simplify software development.
Microcontroller-specific drivers, the kernel, as well as the communication stack
are implemented in the Basic Software (BSW) layer. User applications can be
developed independently of the underlying hardware by using the Runtime Envi-
ronment (RTE), which provides access to features of the BSW. The AUTOSAR
standard is available in two different versions: the AUTOSAR Classic platform
and the AUTOSAR Adaptive platform. While AUTOSAR Classic is targeted for
core ECUs deeply integrated into the car, the Adaptive platform is mainly used
in high-performance applications requiring high network connectivity.
4.2 Threat Model & Attacker Description

The goal of the attack is to gain full malicious code execution on an ECU where there are no exploitable software bugs. For the attack, we assume an adversary with physical access to the device and who is capable of injecting messages into the CAN bus interconnect. Moreover, we assume that the adversary can obtain the firmware deployed on the targeted ECU. Previous research has demonstrated that this assumption is viable [Mil+18]. Finally, similar to related work [Her+21], we assume that the attacker has access to an identical clone of the device under attack. Note that this is a soft requirement reducing the attack complexity.

4.3 Attack Setup

Figure 4.1 depicts the setup used for the attack on the AUTOSAR platform. For the MCU, we use a platform based on an STM32F4 ARM Cortex processor that is running Arctic Core’s AUTOSAR Classic v3.1. This platform also features a CAN controller to enable intra-vehicle communication. To inject messages into the bus, we use Riscure’s Huracan [Risa] system acting as a malicious participant in the CAN network. Furthermore, to control the fault injection attack, we utilize Riscure’s Spider [Risb] device, which is connected to Huracan. To perform voltage glitching, we remove some of the buffer capacitors supporting the supply voltage of the MCU. Finally, we directly power the ECU with the Spider device, allowing us to inject power glitches.

4.3.1 Fault Parameter Characterization

Before performing the actual attack, we first conduct initial experiments on the clone device to test the behavior of the platform when injecting faults. More specifically, we aim to determine whether the device under attack is, in general, susceptible to faults and to find the suitable fault parameters. Therefore, for the characterization, we deploy a code snippet consisting of an unrolled loop incrementing the value stored in a register by 1 to the device. Before the loop starts, we set a General Purpose Input Output (GPIO) pin to high and connect this pin to the trigger input of the Spider glitching device. After the loop, we send the value of the incremented register over a communication interface to the control PC. For this experiment, we consider a fault to be effective if an instruction in the loop was manipulated, i.e., the incremented value does not match the expected value.

The glitch power, which is characterized by the glitch length and the glitch voltage drop, needs to be within a certain range. If the glitch is either too weak or too strong, there is either no observable fault effect or the fault crashes the device. To find the correct fault parameter window, we instrument the Spider device to conduct a random parameter sweep, i.e., we induce faults with different glitch lengths and glitch voltages when the trigger signal is received. When an
Figure 4.1: Attack setup consisting of the target ECU and the glitching equipment.

effective fault is observed, i.e., the device did not crash and the value in the register changed, a correct parameter set is determined.
4.4 Malicious Code Execution on AUTOSAR

In this section, we first introduce our fault model and the fault target and then highlight the end-to-end exploit allowing us to execute arbitrary code on the ECU.

4.4.1 Fault Model

Based on the initial experiments conducted in Section 4.3.1, we assume that an adversary can manipulate the behavior of instructions executed on the MCU by injecting glitches into the power supply.

Listing 4.1: Bit flips in load instruction.

```
1  ldr  r1, [r2, #4] ; 1111 1000 1101 0010 0001 0000 0000 0100
2  ldr  pc, [r2, #4] ; 1111 1000 1101 0010 1111 0000 0000 0100
```

Listing 4.1 shows one possible effect of an injected fault on a load (\texttt{ldr}) instruction. By corrupting bits in the instruction encoding, the CPU could, instead of loading data into register \texttt{r1}, load a value into the Program Counter (PC) register. Hence, with this fault effect, the adversary can manipulate the control-flow of the MCU. Related work [TSW16] showed that a similar behavior also can be achieved by targeting the load multiple (\texttt{ldm}) instruction. Then, due to the instruction encoding, already a single bit flip is sufficient to manipulate the PC. Note that most other architectures, e.g., x86, do not allow to directly load a value into the program counter register. Nonetheless, variants of this attack, \textit{i.e.}, corrupting the stack pointer, are applicable to other architectures as well.

Although MCUs used in the automotive context are certified to meet functional safety requirements [BM10], e.g., with the ASIL scheme, previous research [WP17] has shown that these processors are still vulnerable to fault attacks. Hence, to induce a fault manipulating load instructions to hijack the control-flow, different parts of the MCU can be targeted. For example, a fault induced into the instruction storage, e.g., the cache or program memory, or the instruction bus could manipulate the instruction encoding. In addition, a fault directly into the processor’s pipeline, e.g., the instruction decoder, could also manipulate the instruction’s behavior.

4.4.2 Fault Target

As indicated in Section 4.1.1, communication between different ECUs is essential for the functionality of a vehicle. Hence, one major task of an operating system running on an ECU is to manage the bus in the network software stack. Therefore, in this chapter, we focus on attacking this stack with faults to gain arbitrary code execution on the target device.
In AUTOSAR, an incoming message received on the network interface is transferred through the different layers of the operating system and, in the Arctic Core AUTOSAR implementation, then is copied into a message buffer using `memcpy`. Listing 4.2 shows the disassembly of `memcpy` utilized in the network stack of the AUTOSAR version used for the target platform. The goal, as depicted in Listing 4.3, is to manipulate the load instruction encoding in such a way that the value referenced by the instruction operand is written into the program counter register. As this load instruction is executed multiple times in a loop, the likelihood that the attacker can manipulate this instruction with a fault is high. Since the value loaded into the register is directly retrieved from the CAN interface and the attacker can inject messages into the bus, the value is attacker controllable. Therefore, the adversary can, with a precise fault, redirect the control-flow of the device to arbitrary positions in the program.

### 4.4.3 Attack

To achieve arbitrary code execution on the target platform running AUTOSAR, the adversary first injects a malicious payload into the CAN bus and then induces a fault into the ECU when the payload is copied into a buffer using `memcpy`.

**Payload.** The payload consists of a stager, a malicious task the adversary wants to execute, and the address of the stager code. As AUTOSAR Classic is a static operating system, a new task cannot be deployed during runtime. However, the OSEK/VDX specification, which is the base of the AUTOSAR kernel, requires that the operating system implements an IDLE task that gets scheduled when no other task requests CPU time [Ose]. Since the IDLE task usually does not contain critical code, we replace this task with the malicious task. Therefore, the stager code first copies the malicious task from the buffer to memory and then manipulates the OS IDLE task pointer to point to this injected code. At the end of the payload, the address of the stager code is repeated several times. With
4.4. Malicious Code Execution on AUTOSAR

the ISO-TP protocol deployed on top of CAN, the attacker can inject up to 4095 bytes into the bus [ZS06].

**Fault Injection.** After sending the payload over the CAN bus, the attacker injects a fault into the ECU. The goal of this fault is to manipulate the load instruction when the address of the stager code is copied into the buffer (cf. Section 4.4.2). Then, on a successful fault, the control-flow is redirected to that stager code.

Figure 4.2 depicts the diagram of the fault setup. The computer instruments Riscure’s Huracan system to send the payload over CAN to the target ECU. After a configurable delay, Huracan triggers the Spider fault injector to glitch the power supply of the ECU.

Finding the correct fault parameters, *i.e.*, the glitch delay, length, and voltage, as shown in Figure 4.3, is crucial for the success of the attack. The parameters for the glitch strength are determined during the fault characterization phase (cf. Section 4.3.1). The glitch delay can be estimated by analyzing the firmware, *i.e.*, the number of cycles from an incoming message received on the CAN interface until the payload is copied into the buffer. However, as finding the exact fault parameters is hard [Car+13], the Spider fault injector automatically varies the parameter within a configurable range.

Figure 4.4 shows the plot generated by the fault injection setup. Depending
on the strength of the glitch, \textit{i.e.}, the length and voltage, there could be no measurable fault effect (green), an expected effect (red), or an unexpected effect (yellow). An unexpected effect could be, for example, a crash of the ECU. When the malicious task gets scheduled by the operating system and this task reports back to the computer over UART, an expected effect is registered by the setup. On average, after fine-tuning the setup, we were able to perform 7 successful attacks per hour.

4.5 Countermeasures

To prevent that an adversary is capable of executing injected code on the ECU, the $W \oplus X$ security policy could be deployed on the MCU. However, as the MCU used in the attack target, by default, does not enable this feature in the Memory Protection Unit (MPU), misconfigured ECUs could be deployed in the wild. Nevertheless, even when enabling $W \oplus X$, the attacker can still hijack the control-flow with faults and perform code reuse attacks [Sha07; Ble+11]. Hence, to mitigate this attack vector, we introduce novel countermeasures that prevent fault-induced control-flow attacks in the following chapters.

4.6 Conclusion

In this chapter, we have demonstrated that an adversary can achieve full malicious code execution on a target device, even in the absence of exploitable software bugs, by performing fault attacks. In our attack setup, we have analyzed an electronic control unit consisting of an ARM Cortex MCU running an AUTOSAR Classic compatible operating system. We have shown that injecting faults into a load instruction enables the adversary to manipulate the control-flow. As the data processed by the manipulated instruction is attacker controllable over the CAN bus, the control-flow can be redirected to particular positions in the code. We have demonstrated an end-to-end exploit allowing an adversary to gain arbitrary code execution on the target device.
Cryptographic Control-Flow Integrity on OpenTitan

As we have shown in the previous chapter (cf. Chapter 4), an adversary with physical access to the device can manipulate the control-flow with devastating consequences, even in the absence of exploitable software bugs. These attacks are especially critical for secure elements, such as OpenTitan [Joh+18], as they run security-critical programs, e.g., key storages and authentication services, and are physically accessible to fault attacks as they are deployed in the wild. Hence, secure elements need to provide dedicated hardware- and software-based countermeasures protecting the execution of software against faults.

To detect or prevent fault-induced control-flow manipulations, dedicated Control-Flow Integrity (CFI) schemes [OSM02; SNM22a; Rei+05; WS90] restrict the control-flow during execution to a narrow set of control-flow transfers that are statically determined at compile time. These signature-based approaches maintain a global signature during runtime and compare this signature with the compile time precalculated signature value. On control-flow deviations, the signature check fails and a control-flow attack is detected. This mechanism allows these CFI schemes to verify that the control-flow of a program follows the intended control-flow. However, as the signature checks are only conducted at certain points in the program, control-flow violations are detected with some latency. For example, a fault into the instruction pointer redirecting the control-flow of the program could enable the adversary to still execute security-sensitive code before the signature check detects the violation. Hence, this detection latency can have severe security implications, limiting the practicability of these schemes.
Contribution

In this chapter, we first identify threat vectors enabling the adversary to hijack the control-flow of software by inducing faults into OpenTitan. Subsequently, we thoroughly analyze existing countermeasures aiming to mitigate these attacks.

Our analysis shows that existing countermeasures either induce large area overheads or inadequately reduce the attack surface to address control-flow manipulations. To that end, in this chapter, we introduce SCRAMBLE-CFI [NM23], a cryptographically enforced control-flow integrity scheme that significantly confines the effect of these attacks with no detection latency and a minimal hardware overhead. In SCRAMBLE-CFI, each function is encrypted with a different encryption tweak before the execution of the program. During runtime, the instrumented program automatically updates the decryption tweak in a CPU register. Only when the current execution context and the decryption tweak in the register match, the code can be successfully decrypted and executed. On control-flow redirections to functions outside of the call graph, the tweak mismatches and garbled instructions are decoded, triggering an alert. As SCRAMBLE-CFI utilizes the already existing scrambling unit of OpenTitan for the encryption, our countermeasure only requires minimal hardware changes, yielding an area overhead of less than 3.97%. Furthermore, our performance analysis shows a small runtime overhead of 7.02% for the Embench-IoT benchmarks.

In summary, our contributions are:

• We provide a systematic analysis of threat vectors on OpenTitan allowing an adversary to perform fault-induced control-flow manipulations.

• We discuss existing hardware- and software-based fault countermeasures integrated into OpenTitan.

• We introduce and open-source\footnote{https://extgit.iaik.tugraz.at/sesys/otcfi} SCRAMBLE-CFI, an encryption-based control-flow integrity scheme utilizing hardware features of OpenTitan.

• Finally, we discuss how SCRAMBLE-CFI, complementarily to existing countermeasures, enhances the resilience of OpenTitan against faults with a small runtime and area overhead.

Outline

This chapter is organized as follows. Section 5.1 introduces the threat model used in this work. In Section 5.2, we analyze the resilience of OpenTitan against fault attacks and give an overview of already existing countermeasures. Then, in Section 5.3, we provide the design rational for SCRAMBLE-CFI and highlight implementation details. Section 5.4 summarizes security guarantees and compares SCRAMBLE-CFI with the existing countermeasures. In Section 5.5 we evaluate the performance overhead and in Section 5.6 we provide area overhead numbers. Finally, Section 5.7 compares this chapter to prior work and Section 5.8 concludes this chapter.
5.1 Threat Model

Our threat model comprises an attacker with physical access to the OpenTitan chip performing fault attacks. This attacker is capable of injecting single or multiple faults into the secure element by using clock, voltage, or EM glitching techniques or by performing laser fault injection [Bar+06]. We assume that these faults cause single or multiple bit-flips [VKS11] in the system. The goal of the attacker is to redirect the control-flow of software executed on the chip.

5.2 Analysis

In this section, we discuss potential attack vectors within the presumed threat model (cf. section 5.1) and systematically analyze existing hardware- and software-based countermeasures of OpenTitan aiming to address the identified threats.

5.2.1 Attack Vectors

The control-flow of software can be redirected at different control-flow manipulation (CFM) granularities:

CFM1 The attacker redirects the control-flow to a function that cannot be reached from the current execution context, i.e., escaping the call graph of the program.

CFM2 The control-flow is redirected from one branch target to the other when manipulating conditional branches.

CFM3 The attacker arbitrarily hijacks the control-flow within a function, i.e., to any basic-block.

The attack surface (A) comprises all elements of OpenTitan (cf. Figure 2.4). More specifically, a fault can be injected into CPU internal registers (AR), the instruction cache (AIC), the bus infrastructure (AB), or the memory (AM). Furthermore, the attacker can also target the core (AC), i.e., the instruction fetch, decode, and execute pipeline stages. For example, when targeting AC, the attacker can change the behavior of executed instructions when inducing bit flips into the instruction decoder or influence the comparison for a conditional branch in the ALU.

To manipulate the control-flow, we define three potential data targets (DT):

DT1 Control-flow related data comprises relative (DT1.1) and absolute (DT1.2) addresses as well as the program counter (DT1.3). To flip bits in relative addresses (DT1.1) used by unconditional and conditional branches, the attacker can inject faults into the immediate field of instructions stored in the program memory (AM) or the instruction cache (AIC) or transferred by the instruction bus (AB) [KSV13]. Indirect calls can be manipulated by targeting absolute addresses (DT1.2a) stored
in registers (AR) or the data memory (AM) or transferred by the data bus (AR). Moreover, returns can be redirected by flipping bits in return addresses (DT1.2b). Finally, the attacker also can directly inject faults into the program counter (DT1.3)(AR). Targeting DT1 enables the adversary to arbitrarily manipulate the control-flow, i.e., CFM1, CFM2, and CFM3.

**DT2** Non-control-flow related data, i.e., general purpose data, can be faulted by targeting the SRAM (AM) or the data bus (AB). When the faulted data is used by conditional branches [Vas+20], the attacker can influence their execution (CFM2).

**DT3** Instructions, stored in the flash (AM) or iCache (AIC), transferred by the instruction bus (AB), and processed by the core (AC), can be manipulated by inducing bit flips into the opcode or the operands [Nas+23a; TSW16]. Here, one possible attack would be to skip an instruction by flipping the opcode from a jump (jalr) to a nop. Similarly, by manipulating the operand, e.g., flipping jalr 0(x5) to jalr 0(x6), the control-flow can also be arbitrarily redirected.

### 5.2.2 OpenTitan Countermeasures

In the following, we highlight hardware- and software-based countermeasures already integrated into OpenTitan protecting against fault attacks.

**Hardware-based Countermeasures** OpenTitan protects security-critical data throughout most of its life cycle using an Error Correction Code (ECC). More specifically, the integrity of data is protected in the data memory (AM), the data bus (AB), as well as in the register file (AR) of Ibex. An integrity error allows OpenTitan to detect bit-flips in DT2 and DT1.2a. Additionally, ECC is used in the instruction cache (AIC), the program memory (AM), as well as the instruction bus (AB) to detect fault-induced bit-flips, protecting DT3 and DT1.1. To prevent manipulations of the program counter (DT1.3), Ibex recalculates the derived program counter and triggers an error on a mismatch.

Finally, OpenTitan also provides the possibility of instantiating the Ibex core twice in a lockstep mode. Here, the execution of the second core is delayed by some cycles and the outputs of the core, i.e., the data and instruction interface outputs, are compared. Although this strategy provides strong protection against faults induced into the pipeline (AC), it also more than doubles the area of the CPU.

**Software-based Countermeasures** In addition to the hardware-based countermeasures, the OpenTitan project also provides software-based fault protection mechanisms that are integrated into a modified LLVM toolchain. Programs compiled with this toolchain automatically store the return address into a shadow stack. In the function epilogue, before the return, the current return address is
compared to the return address stored in the shadow stack, mitigating bit-flips in return addresses (DT1.2b). Furthermore, after each indirect branch and return, the compiler inserts an illegal instruction that triggers an exception. This strategy hinders the adversary from redirecting the control-flow by skipping these instructions (DT1.3, DT3, and AC).

5.3 SCRAMBLE-CFI

As shown in the previous section, current OpenTitan countermeasures either induce large area overheads, \textit{i.e.}, the dual-core lockstep approach, or only provide limited protection against fault-induced control-flow manipulations when targeting the core (AC). In this section, we introduce SCRAMBLE-CFI, our control-flow integrity scheme that enhances the resilience of OpenTitan against these attacks with a minimal area and runtime overhead. Afterwards, in Section 5.4, we then highlight security guarantees and compare SCRAMBLE-CFI to existing countermeasures.

5.3.1 Overview

In SCRAMBLE-CFI, each function is assigned an encryption tweak during program compilation. Before execution, when loading the program into flash, the code blocks are encrypted with the corresponding tweak. At runtime, before each function call, the decryption tweak for the call target is placed into a CPU register. As the modified OpenTitan scramble engine incorporates the content of this register into the decryption, the instructions only can be decrypted when the active tweak matches the tweak determined at compile time. Figure 5.1 shows the SCRAMBLE-CFI’s encrypted call graph, \textit{i.e.}, a graph comprising all valid transfers from one function to another. In SCRAMBLE-CFI, each of these functions are encrypted with a different encryption tweak. When redirecting the control-flow from the current execution context to another function encrypted with a different tweak, garbled instructions are fetched and the decoding fails with a high probability. More specifically, as SCRAMBLE-CFI assigns each function, for programs without indirect branches, a unique tweak, any cross-function control-flow manipulation fails. For programs containing indirect branches, SCRAMBLE-CFI guarantees that the attacker cannot redirect the control-flow outside of the call graph. Summarized, the processor only can execute code blocks when the corresponding decryption tweak is active.

5.3.2 Program Instrumentation

In order to decrypt the code of a called function, the corresponding decryption tweak needs to be loaded into the tweak register before the control-flow edge, \textit{i.e.}, direct and indirect branches. This program instrumentation is done fully automatically in SCRAMBLE-CFI by a modified LLVM [LA04] RISC-V compiler.
Chapter 5. Cryptographic Control-Flow Integrity on OpenTitan

Our custom compiler consists of an analysis and instrumentation pass operating in the backend of the toolchain. This pass first performs a program analysis to construct the call graph of the program to protect. Here, we scan each function for direct and indirect calls and determine the call target. While LLVM already provides this information for direct calls, indirect calls require a points-to analysis [NSM21] to reveal the set of potential called functions.

After the extraction of the call graph, we assign each function a unique encryption tweak. Depending on the number of functions, this tweak is either a 5 bit or 20 bit random number.

![Figure 5.1: Encrypted call graph.](image)

**Figure 5.1:** Encrypted call graph.

Figure 5.2 shows the instrumentation of direct and indirect calls conducted by the SCRAMBLE-CFI compiler. For the direct call from function A to B, we set the tweak to the tweak of function B, i.e., $T_B$. When returning from this function, the tweak is set back to the tweak used by function A, i.e., $T_A$. Similarly, for indirect branches, e.g., from C to B, we also set and reset the tweak before and after the call. However, instead of using the same tweak as for the direct call, i.e., $T_B$, we add an additional entry point to the function, which is encrypted with a different tweak, i.e., $T_{BE}$. In this entry point, we again update the tweak to the tweak for the function to $T_B$. Moreover, we add a second exit point which is taken when the function was called by an indirect branch. In this exit point, we set back the tweak to the tweak of the entry point, i.e., $T_{BE}$. The compiler rewrites all addresses used by direct branches to point to the instruction after the added entry point. Furthermore, in the entry point, we set a flag indicating whether the function returns with the default or the added exit point.

Adding entry points to the program is necessary because indirect calls can have
multiple call targets that need to be encrypted with the same tweak. Without this additional entry point, also direct calls would need to be encrypted with this tweak, allowing the adversary to redirect a direct branch to another function which is called by an indirect branch and is outside of the call graph.

Listing 5.1: Tweak update instruction sequence.

```
1 #5 bit tweak:
2 csrrwi x0, csr_tweak, tweak5bit
3 #20 bit tweak:
4 lui x28, tweak20bit
5 csrrw x0, csr_tweak, x28
```

Listing 5.1 shows the instruction sequence used to set the 5 bit or 20 bit tweak into the tweak control and status register (CSR). For programs with 32 or fewer functions, a single `csrrwi` instruction loading the 5 bit tweak from the immediate field into the CSR `csr_tweak` is sufficient.

Alignment to Encryption Granular

As indicated in fig. 5.2, the next instruction after updating the tweak is already decrypted with this tweak. Since the decryption granularity of the underlying cipher, i.e., 64 bit for PRINCE, does not match the natural RISC-V instruction alignment of 16 or 32 bit, we need to align the set tweak instruction sequence to the decryption granularity. We conduct this alignment by padding these instructions with `nops` to 64 bit.

Metadata Section

The tweak for each code block is stored in a custom ELF section generated by our toolchain in the `AsmPrinter` stage. Here, the compiler emits the start address of each function and the corresponding number of basic-blocks. Moreover, the offset of each basic-block and the corresponding decryption tweak is emitted into the metadata section by the compiler. This section is then processed during program deployment.

5.3.3 Program Deployment

Figure 5.3 shows the deployment of a program protected with SCRAMBLE-CFI on OpenTitan. Our toolchain first compiles the C source code of the program to an instrumented ELF binary. Then, the flash utility program converts this binary into a `VMEM` file, which is loaded into flash memory.

As at the time of writing this thesis, the OpenTitan project only provides hardware support for flash scrambling but not the necessary software support and, therefore, disables the scrambling, we extended the flash utility program to encrypt code with the PRINCE cipher. Here, we encrypt each 64 bit word with the PRINCE cipher using the flash scramble key and the SCRAMBLE-CFI tweak.
Chapter 5. Cryptographic Control-Flow Integrity on OpenTitan

Figure 5.3: Deployment of protected programs. Code blocks in flash memory encrypted with different SCRAMBLE-CFI tweaks are highlighted with different colors.

Note that the flash scramble key is stored inside the One Time Programmable (OTP) memory. To retrieve the SCRAMBLE-CFI tweak for each code block, the flash utility tool parses the custom metadata section emitted by our toolchain. Afterwards, the flash is initialized with the encrypted VMEM file and OpenTitan starts the execution of the encrypted code.

5.3.4 Hardware Changes

To realize SCRAMBLE-CFI on OpenTitan, minimal-intrusive hardware changes are required: First (i), additional control and status registers (CSRs) need to be added to the Ibex processor. We implement these registers by using the shadow register primitives provided by the OpenTitan project. As these shadow registers duplicate the registers and compare the stored values, the content of these CSRs are protected from faults. By writing to the CSRs, software, i.e., binaries compiled with our custom toolchain, can set the current active decryption tweak as well as the lower and upper address bound. The address range registers comprise the lowest and highest address of program code, which needs to be protected by SCRAMBLE-CFI. Enabling the tweak only for a certain address range is required to access data, such as globals, also stored in the flash memory. Second (ii), the tweak needs to be incorporated into the encryption primitive. One possibility would be to extend the PRINCE cipher to a tweakable block cipher with the TWEAKEY [JNP14] framework. However, the required TWEAKEY key schedule logic would increase the complexity and area consumption of the cipher. Moreover, as SCRAMBLE-CFI does not need cryptographic strength for control-flow integrity, we inject the tweak into the key using a XOR operation. This exclusive or is conducted when the address sent to the flash controller is between the lower and upper address stored in the added CSRs. Otherwise, the tweak is set to 0 and the PRINCE cipher uses the default key provided by the OTP controller. Third (iii), when the decryption tweak is changed by writing to the CSR, we flush the instruction cache to avoid that scrambled cached instructions are executed.
5.4 Security Analysis & Comparison

When the tweak register contains the tweak of the current function, the execution of code fails when the control-flow is redirected to any other function encrypted with a different encryption tweak. Before a function call, SCRAMBLE-CFI updates the tweak register with the tweak of the called function. Then, the control-flow can only be redirected to this function or functions encrypted with the same tweak. In SCRAMBLE-CFI, the entry points (cf. section 5.3.2) of functions that an indirect branch can call share the same encryption tweak. This is inevitable as, at compile time, the toolchain cannot determine which function gets called by an indirect call at runtime. However, indirect calls are rare in typical programs and the attacker only can redirect the control-flow to functions that can be reached with this indirect call, i.e., are within the call graph. For programs without indirect calls, SCRAMBLE-CFI assigns all functions a unique encryption tweak. Summarized, for any control-flow redirection outside of the call graph (CFM1) the wrong decryption tweak is deterministically used for programs that contain less than $2^{20}$ functions. For programs that contain more functions than the available tweak space, tweak collisions can occur.

When fetching instructions from program memory with a wrong decryption tweak, garbled instructions are retrieved. With a high probability, the decoding of these instructions in the instruction decoder pipeline stage fail and an exception is triggered. However, it could be possible that a garbled instruction again forms a valid instruction. Nevertheless, (i) the probability that the subsequent instructions are also valid is low and (ii) usually, the attacker aims to execute a certain instruction in the function and not any that does not trigger an exception.

Note that SCRAMBLE-CFI cannot prevent an adversary from manipulating conditional branches (CFM2) or from redirecting the control-flow within a function from one basic-block to another (CFM3). However, SCRAMBLE-CFI could be extended to provide fine-granular protection for highly security-critical code blocks by encrypting code blocks within a function with different encryption keys.

In the current prototype of SCRAMBLE-CFI, we incorporate the tweak into the encryption by XORing it to the encryption key. This XOR creates a dependency of key and tweak, allowing an attacker to potentially learn about the key when the tweak is known. However, the binary (including the tweaks) is inaccessible to an adversary as it is stored in the encrypted flash.

5.4.1 Security Comparison

Table 5.1 highlights protection guarantees of different hardware- and software-based OpenTitan countermeasures and SCRAMBLE-CFI against faults into different attack targets.

The ECC-based countermeasures provide full protection when inducing faults into their protection domain. For example, bit-flips induced into instructions (DT3) stored in the instruction cache (AIC) or the program memory (AM) can be detected reliably. As the Program Counter (PC) protection recalculates
Table 5.1: Protection guarantees of different countermeasure when targeting different attack surfaces.

<table>
<thead>
<tr>
<th>Countermeasure</th>
<th>AR</th>
<th>AIC</th>
<th>AB</th>
<th>AM</th>
<th>AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data memory ECC</td>
<td>-</td>
<td>-</td>
<td>✔</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Program memory ECC</td>
<td>-</td>
<td>-</td>
<td>✔</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Data bus ECC</td>
<td>-</td>
<td>-</td>
<td>✔</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Instruction bus ECC</td>
<td>-</td>
<td>-</td>
<td>✔</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Register file ECC</td>
<td>✔</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>iCache ECC</td>
<td>-</td>
<td>✔</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PC protection</td>
<td>✔</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✔</td>
</tr>
<tr>
<td>SW-based defense</td>
<td>☐</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>☐</td>
</tr>
<tr>
<td>dual-core lockstep</td>
<td>✔</td>
<td>✔</td>
<td>-</td>
<td>-</td>
<td>✔</td>
</tr>
<tr>
<td>SCRAMBLE-CFI</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

✔ Full ☐ Partial - No Protection

and compares the PC to the current PC, also faults into the core (AC) can be detected. The software-based defense integrated into the custom LLVM toolchain only provides partial protection, i.e., only control-flow manipulations aiming to manipulate return addresses (DT1.2b) or skipping certain instructions (DT3) can be detected. As shown in table 5.1, from the existing countermeasures, only the lockstep approach can provide strong protection against faults induced into the core (AC). However, this strategy also induces a high area overhead as the Ibex core needs to be instantiated twice and an error detection logic needs to be added. In comparison, SCRAMBLE-CFI provides strong protection against control-flow manipulations for all attack targets, even when targeting the core, with minimal hardware overhead. Especially when combined with the other existing countermeasures, SCRAMBLE-CFI significantly minimizes the attack surface and allows OpenTitan to withstand most of the control-flow attacks. Combined, OpenTitan is protected against arbitrary control-flow manipulations (CFM1, CFM2, and CFM3) when targeting all attack targets except the core (AC). When injecting faults into AC, SCRAMBLE-CFI fills the protection gap of hindering an adversary from redirecting the control-flow outside of the call graph (CFM1). However, when also CFM2 and CFM3 need to be mitigated, SCRAMBLE-CFI needs to be deployed on a finer granularity (cf. section 5.4) or the lockstep approach needs to be installed. Summarized, we argue that SCRAMBLE-CFI provides a good area-security tradeoff allowing OpenTitan to withstand a multitude of different fault attacks.

5.5 Performance Overhead

To evaluate the performance and code size overhead, we compiled the Embench-IoT [Pat+] benchmarks with our custom toolchain. We excluded benchmarks requiring libraries, such as math and string, currently not provided by the
Table 5.2 highlights the percentual code size overhead measured with the GNU size utility when compared to the unprotected baseline. Our analysis shows a code size overhead between 0.74% and 8.88% and a geometric mean of 1.69%. This overhead comprises the (i) inserted tweak switch instructions, the (ii) alignment of these instructions to the encryption granular, and the (iii) metadata binary section. As this section only is needed by the flash utility program to encrypt code blocks with different encryption tweaks (cf. section 5.3.3), this metadata is not stored in the flash memory.

In order to analyze the performance impact of binaries instrumented with SCRAMBLE-CFI, we measured the CPU cycles by reading the mcycle CSR of Ibex. Here, we executed the protected and unprotected binaries on a cycle-accurate Verilator model of OpenTitan. As shown in fig. 5.4, we measured a runtime overhead between 0.22% and 143.35% and a geometric mean of 7.02%.

Table 5.2: Code size overhead for the Embench-IoT benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Overhead [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>aha-mont64</td>
<td>1.7%</td>
</tr>
<tr>
<td>crc32</td>
<td>1.31%</td>
</tr>
<tr>
<td>edn</td>
<td>1.7%</td>
</tr>
<tr>
<td>huffbench</td>
<td>1.53%</td>
</tr>
<tr>
<td>matmult</td>
<td>1.32%</td>
</tr>
<tr>
<td>md5</td>
<td>1.59%</td>
</tr>
<tr>
<td>nettle-aes</td>
<td>1.6%</td>
</tr>
<tr>
<td>nettle-sha256</td>
<td>1.75%</td>
</tr>
<tr>
<td>nsichneu</td>
<td>0.74%</td>
</tr>
<tr>
<td>picojpeg</td>
<td>8.88%</td>
</tr>
<tr>
<td>primecount</td>
<td>1.09%</td>
</tr>
<tr>
<td>sglib</td>
<td>3.09%</td>
</tr>
<tr>
<td>tarfind</td>
<td>1.26%</td>
</tr>
<tr>
<td><strong>Geometric mean</strong></td>
<td><strong>1.69%</strong></td>
</tr>
</tbody>
</table>
Benchmarks, such as \texttt{crc32} or \texttt{tarfind}, frequently calling small functions induce larger runtime overheads than benchmarks only performing a small number of function calls.

5.6 Area Overhead

The public OpenTitan hardware design flow currently only allows to synthesize the Ibex core with open-source synthesis tools. Therefore, we synthesized the Ibex processor with the Yosys open synthesis suite and the Nangate 45 nm standard cell library to analyze the area overhead introduced by our hardware changes. Here, we measured an area increase from 26.48\,kGE to 27.53\,kGE (3.97\%). These hardware changes comprise the additional CSRs as well as the address range comparison. Note that the XOR of the tweak with the key is conducted in the flash controller and is currently not reflected in the hardware overhead number. According to the synthesis logs created with the proprietary design flow published by the OpenTitan project [Ope], the Ibex occupies 3.4\% of the overall chip area. Hence, the hardware overhead induced by SCRAMBLE-CFI to the overall area is negligible.

5.7 Related Work

Currently, encryption-based control-flow integrity schemes, such as SOFIA [Cle+17] or SCFP [Wer+18], require intrusive hardware changes in the processor’s pipeline to realize their protection. Hence, when integrating these changes into a chip, an extensive re-verification of the entire processor is needed.

5.8 Conclusion

In this chapter, we have thoroughly analyzed fault threat vectors allowing an adversary to manipulate the control-flow of software executed on OpenTitan. We have provided an overview of current OpenTitan countermeasures and discussed their protection capabilities. Furthermore, we have introduced SCRAMBLE-CFI, which mitigates fault attacks aiming to redirect the control-flow outside of the call graph. We have shown that SCRAMBLE-CFI is a strong security addition to existing countermeasures inducing minimal runtime and area overheads.
Cryptographic Control-Flow Integrity on Commodity Hardware

Originally, fault attacks were pure local attacks where an adversary needs to have physical access to the target device to inject faults with a laser or to glitch the clock or power supply [Bar+06]. However, recent publications, such as Plundervolt [Mur+20], CLKSCREW [TSS17], or VoltJockey [Qiu+19a; Qiu+19b], demonstrated that faults also can be induced in software over the network. Therefore, the attack surface increased from mostly embedded devices, such as automotive systems (cf. Chapter 4) or secure elements (cf. Chapter 5), to fully-fledged processors deployed in consumer hardware as well as in cloud environments. Hence, to protect security-sensitive programs, dedicated countermeasures on commodity hardware mitigating fault-induced control-flow attacks are needed.

Although control-flow integrity [Aba+09] is already available on these systems, these schemes [Mas+15; Lil+19; Kuz+14] only aim to mitigate control-flow attacks triggered by software vulnerabilities, e.g., memory safety vulnerabilities. As software CFI assumes a software adversary in its threat model, they only protect control-flow edges, such as indirect branches and returns, from control-flow manipulations. However, as the fault attack threat model comprises a broader attack surface, i.e., any control-flow edge including direct branches, these attacks can bypass state-of-the-art CFI countermeasures. In addition to hijacking control-flow edges, faults also enable the attacker to redirect the control-flow at any execution point, e.g., by manipulating the instruction pointer [NT19; TM17; TSW16].

Although there already exist strong CFI schemes mitigating fault attacks [NM23; Wer+18; Cle+16], these schemes require hardware changes, which makes it hard to deploy them on a larger scale. Hence, to protect the control-flow of software
against fault attacks, new countermeasures without requiring intrusive hardware changes on commodity systems are needed.

**Contribution**

This chapter introduces EC-CFI [Nas+23a], a cryptographically enforced control-flow integrity scheme designed to counteract fault attacks aiming to redirect the control-flow of programs outside of their call graph. Similar to SCRAMBLE-CFI (cf. Chapter 5), each function in EC-CFI is encrypted with a different encryption key before the program’s execution. At runtime, EC-CFI-instrumented programs dynamically derive the active decryption key before each control-flow edge, i.e., direct or indirect function calls. This derivation produces the correct decryption key only if the control-flow matches the statically determined call graph that was used to derive the encryption keys at load-time. When a fault redirects a control-flow edge to another function outside of the call graph, the code is decrypted with the wrong key, which can be immediately detected with a high probability. Moreover, the protection of EC-CFI comprises not only control-flow edges, any redirection to other functions, e.g., by instruction pointer manipulations, can be mitigated.

To enable this level of protection on recent commodity Intel platforms without hardware modifications, we utilize the total memory encryption - multi key (TME-MK) feature for the function encryption. However, as Intel’s TME-MK so far is only used for page-granular memory encryption, which is too coarse-grain for function encryption, we introduce a new concept based on extended page table (EPT) aliasing. This mechanism allows us to leverage TME-MK for fine-granular, in the case of EC-CFI, function-granular, memory encryption. Moreover, our approach based on EPT aliasing, which is a combination of Intel’s virtualization technology (VT) and TME-MK, enables us to frequently switch the key used for encryption and decryption.

We showcase how to implement EC-CFI using the generic EPT aliasing approach and introduce a prototype implementation. We open-source our custom LLVM toolchain, which is responsible for automatically instrumenting programs with the key derivation mechanism without any user interaction. Furthermore, we measure the performance impact of EPT aliasing on a recent Intel CPU using the SPEC CPU2017 and Embench-IoT benchmarks. Finally, we discuss potential minimal-invasive hardware changes decreasing the runtime overhead of EC-CFI.

In summary, our contributions are:

- We present a CFI scheme that is designed to hinder a fault adversary from escaping the call graph of a protected program by encrypting each function with a different encryption key. By dynamically deriving the decryption key at runtime, the code of a function can only be successfully decrypted if it is reached by following the static call graph used to encrypt the function.

- We introduce a fine-granular encryption approach for recent Intel platforms based on EPT aliasing consisting of a novel combination of TME-MK and VT. This approach enables us to achieve function-granular encryption and
to use different encryption keys for different functions without hardware changes.

- We showcase how to implement EC-CFI with EPT aliasing on recent Intel platforms. Here, we open-source our LLVM-based toolchain capable of automatically protecting programs with EC-CFI.

- We evaluate the performance impact of our EPT aliasing approach and analyze security benefits of EC-CFI.

- Finally, we discuss minimal TME-MK hardware changes and showcase that these changes minimize the runtime overhead of EC-CFI.

Outline

This chapter is structured as follows: The background for EC-CFI is summarized in Section 6.1 and Section 6.2 defines the threat model. In Section 6.3 and Section 6.4 we present our EC-CFI concept and the prototype implementation. Section 6.5 discusses security benefits and Section 6.6 evaluates the runtime and code size overhead of our EPT aliasing approach. Furthermore, we discuss potential hardware changes improving the runtime overhead of EC-CFI in Section 6.7. Finally, Section 6.8 compares EC-CFI with other CFI schemes and Section 6.9 and Section 6.10 outline future work and summarize our chapter.

6.1 Background

This section gives an overview on the Intel VT feature.

6.1.1 Intel Virtualization Technology

Intel Virtualization Technology (VT) [22] is a set of features allowing the processor to efficiently and securely share computing resources among different workloads. One key feature is the hardware-based second level address translation mechanism allowing each guest to have its own virtual address space. Here, the guest system is responsible for the first level address translation, i.e., guest linear addresses (GLA) to guest physical addresses (GPA), by using page tables. For each guest, the host then provides a mapping from guest physical addresses to host physical addresses (HPA) using extended page tables (EPTs). The \texttt{vmfunc} instruction allows the guest to set the current active EPT from an extended page table pointer (EPTP) list stored in the virtual machine control structure (VMCS).

6.2 Threat Model

In our threat model, we consider an adversary capable of injecting a targeted fault into the processor or the external memory. We assume that this fault is either injected remotely, e.g., by using Plundervolt [Mur+20] or CLKSCREW [TSS17],
A fault attacker can redirect the control-flow outside of the call graph by either targeting the control-flow edges between functions, flipping bits in any other instruction, or manipulating the instruction pointer of the CPU. For the control-flow edges between functions, the attacker can target direct or indirect branches. To manipulate the execution of indirect calls, a fault attacker can flip bits in addresses stored in registers used by these calls. Furthermore, the adversary also can manipulate the address used by direct calls by injecting a fault into the address generation unit (AGU) of the CPU. Moreover, by flipping bits in the program memory of the application, addresses of direct calls or the registers used by indirect calls [Zer] can be manipulated.

In addition, the attacker can also flip bits in any instruction of the program in such a way that the control-flow is redirected, e.g., the opcode is changed to a branch [NT19]. Finally, a redirection of the control-flow also can be performed by injecting faults directly into the instruction pointer of the CPU [TSW16; Gra+15]. In summary, this attacker model is stronger than threat models used by traditional CFI targeting a software-only adversary, where only indirect branches and returns are considered to be vulnerable.

For our work, we exclude side-channel and microarchitectural attacks and assume that the operating system and the hypervisor are trusted by the system.

6.3 Design

EC-CFI aims to hinder an adversary from redirecting the control-flow to arbitrary points in the program by encrypting each function with a different encryption key at load-time. At runtime, EC-CFI restricts the set of callable functions for the current execution context to the set of call targets defined in the call graph by dynamically deriving the decryption key. When the attacker redirects the control-flow to a function outside of the call graph, the encrypted code is decrypted with a wrong key. As this decryption yields garbled code, the instruction decoding
fails with a high probability. Although it could be possible that decrypting an instruction with an invalid key could produce a valid instruction, the likelihood of decrypting multiple instructions correctly is low [AMD16]. Hence, EC-CFI is capable of detecting control-flow manipulations with no or minimal detection latency. EC-CFI achieves this level of protection on recent Intel commodity hardware by combining a signature-based control-flow integrity scheme with fine-granular memory encryption.

### 6.3.1 Fine-Granular Memory Encryption

EC-CFI encrypts each function $F$ with a different encryption key $K_F$ using Intel’s TME-MK memory encryption engine. However, as highlighted in Section 2.5.3, in the intended usage mode, TME-MK only provides the possibility to encrypt entire memory pages (e.g., 4 kB pages) with different encryption keys. Although increasing the code sizes of functions to page sizes would enable the processor to encrypt each function with a different key, this approach would also significantly increase the memory overhead.

To overcome this limitation, we introduce a novel fine-granular memory encryption approach based on a combination of TME-MK with the extended page table (EPT) feature of Intel VT. Hereby, EC-CFI achieves sub-page granular memory encryption by combing EPT aliasing with memory encryption. With this approach, the encryption granularity is only limited by the encryption primitive, e.g., 128 bit block size for AES. Such a small encryption granularity was previously only possible using custom CPU designs [Nas+21a; Ste+21a].

Figure 6.2 illustrates the core idea of EPT aliasing combined with memory encryption based on an example with three functions A, B, and C located inside
the 4 kB virtual page. The first level address translation mechanism translates the guest linear addresses (GLA) of functions A, B, and C to the guest physical addresses (GPA) using the page frame number (PFN) of the page table (PT).

In our example, a PFN of 0x10 is used to translate the addresses. Now, our approach based on EPT aliasing establishes separate extended page tables (EPT1, EPT2, and EPT3) for each encryption domain using a different key, i.e., key 1 for function A, key 2 for function B, and key 3 for function C. In the EPT entries of these EPTs, the guest physical to host physical address (HPA) mapping is identical, i.e., EPT1, EPT2, and EPT3 use the PFN 0x100 for functions A, B, and C. However, the key identifier fields in the EPT entries are different, i.e., key 1 for EPT1, key 2 for EPT2, and key 3 for EPT3.

This approach allows us to have different views ( Opr ) on the memory by switching the current, active extended page table. For example, when EPT2 is active ( Opr ), the GPA of function B is translated by the second level address translation mechanism to the HPA with the address translation information stored in the entries of EPT2. As the key identifier key 2 is embedded into the upper bits of the HPA during the address translation, TME-MK now encrypts or decrypts function B with the key assigned to this key identifier. Note that for the actual physical memory access, the key identifier bits are stripped from the physical address. When accessing function C with EPT2, which was encrypted with key identifier key 3 in the EPT3 memory view, only garbled code is retrieved as the wrong decryption key 2 is used for the access.

To switch between these EPTs, the extended page table pointer (EPTP) that specifies the active EPT can be changed. Such an EPTP switch is initialized with the vmfunc instruction. By passing the EPTP index, e.g., 0, 1, or 2, as shown in Figure 6.2, to this instruction, the CPU switches the EPTP to the corresponding EPTP in the configured EPTP list.

As shown in Figure 6.2, EC-CFI does not restrict the locations of functions in memory, i.e., multiple functions inside of a page or functions occupying multiple pages are supported.

6.3.2 Signature-Based Control-Flow Integrity Scheme

EC-CFI uses a signature-based control-flow integrity approach to automatically derive the decryption keys for each encrypted function at runtime. In our scheme, a random signature $S_F$ is assigned to each function $F$ and the current, active signature is stored in the global signature register $S$.

EC-CFI uses the approach based on EPT aliasing (cf. Section 6.3.1) for fine-grained encryption of code blocks. For each encryption domain, EC-CFI initiates a separate EPT with a different encryption key embedded into the extended page table entry. As each encryption key only is used in one EPT, we have a bijective mapping $EPTP \mapsto K$. EC-CFI now passes the signature $S$ to the vmfunc instruction to select the active EPT and, therefore, the current encryption key, i.e., $S \mapsto EPTP \mapsto K$. Note that the signature $S$ is not a signature in the cryptographic sense, instead, it is the index (cf. Figure 6.2) to the extended page table pointer (EPTP), which points to an extend page table.
EC-CFI consists of three major runtime primitives: (i) signature init, (ii) switch key, and (iii) signature update. At the start of the program, the signature register is initialized (i) with the signature of the entry function. Then, EC-CFI activates the key for decrypting the entry function by switching (ii) the EPTP to the EPT containing the corresponding key. Due to the bijective mapping from the signature to the key over the EPTP, the signature $S$ automatically selects the correct key and the function can be decrypted. During the program’s execution, the current signature is updated (iii) before each control-flow transfer to a different function, i.e., on direct or indirect calls.

$$S = S \oplus C$$

Equation (6.1) shows the used accumulative update function. The compiler selects the position-dependent constant $C$ for the call in such a way that the resulting signature matches the signature of the called function. After updating the signature, the key for the called function is activated by switching (ii) the EPTP using the current $S$. When the derived signature matches the signature of the call target, the function can be successfully decrypted. After returning from the callee, the signature is again updated and the key is switched such that the code of the caller can be decrypted.

Figure 6.3 depicts the signature init (Line 1), switch key (Lines 3 and 7), and signature update (Lines 2 and 6) required by EC-CFI to correctly derive and switch the key for both functions. Hereby, the color highlights the corresponding code encrypted with the different keys $K_{Main}$ and $K_A$. Due to the mapping $S \mapsto K$, correctly deriving $S$ and calling the intended function allows the CPU to successfully decrypt the code with key $K$. Note that the key immediately becomes active after executing the key switching routine. Therefore, the instruction calling function A (Line 4) already needs to be encrypted with the key for this function.

For indirect calls, accurately determining the caller target at compile time is not possible. Hence, EC-CFI determines the possible set of call targets, which then share an encryption key. To ensure that the same key is derived, EC-CFI induces signature collisions, i.e., $C$ is accordingly chosen to derive the same signature $S$ for different indirect calls.
Multi-Call Targets

Assigning multi-call targets, i.e., functions that can be called from multiple other functions, an identical encryption key enables the adversary to escape the call graph. Figure 6.4 describes the security implications of deploying a shared key for the multi-call target B. By inducing a fault during the execution of this function, the adversary can redirect the control-flow either to A or C, independently of the original call site. EC-CFI mitigates this security weakness by adapting the concept of call headers introduced in [SNM22a].

Figure 6.5 shows our approach of securely handling multi-call targets using call headers. Each function is assigned the corresponding signature, i.e., \(S_A, S_B,\) and \(S_C\) and these functions are encrypted with the corresponding key. Furthermore, a call header encrypted with a distinct key, i.e., \(S_{AH} \rightarrow K_{AH}\) and \(S_{CH} \rightarrow K_{CH},\) is added to the multi-call target function B. Before calling function B, the key is switched to this call header key. Then, the function is called and the execution flow is redirected to the corresponding call header. Inside this header, the key is updated to the key of the called function, i.e., \(S_B \rightarrow K_B.\) Additionally, a return constant \(C_{Ret}\) for each header is set. When returning from the function, this
constant is used to switch the key back to the call header key. This ensures that the program only can return to the original call site. After the call instruction, the key is switched back to the signature $S_A$ or $S_C$ of the corresponding function.

For indirect calls, the headers of the possible set of call targets share a common signature, i.e., they are encrypted with the same key.

6.4 Implementation

The prototype implementation of EC-CFI consists of three major building blocks (cf. Figure 6.6). The (i) compiler is responsible for instrumenting binaries, the (ii) hypervisor provides multiple EPTs, and the (iii) loader uses the hypervisor and metadata provided in the instrumented binary to encrypt each code block with a different key before execution.

![Diagram of Compiler, Loader, Hypervisor, Instr. Binary, Metadata, Memory]

Figure 6.6: Overview of our EC-CFI prototype implementation.

6.4.1 Compiler

To automatically protect programs without user interaction, we integrate EC-CFI into a custom LLVM-based toolchain [LA04]. Our backend pass of the custom toolchain is responsible for 1) assigning signatures to functions, 2) instrumenting calls, 3) inserting call headers, and 4) aligning the code blocks to the cache line size.

Signature Assignment

The first step the compiler conducts is the assignment of the signatures $S_F$ to all functions in the program. Here, the compiler chooses a random ID between $S_{LOW}$ and $S_{HIGH}$ for each function and stores this information into a compiler-internal structure. The signature range is configured by the user compiling a program and needs to reflect the number of available TME-MK key identifiers and EPTs of the targeted processor.
Listing 6.1: Key switch instruction sequence.

1 push %rax  ; Save rax & rcx to stack.
2 push %rcx
3 xor %rax, %rax  ; Set rax to 0.
4 mov %r13, %rcx  ; Move signature to rcx.
5 vmfunc  ; Switch EPTP.
6 pop %rcx  ; Restore rax & rcx from
7 pop %rax  ; stack.

Afterwards, the compiler defines a constant used to initialize the signature register with the chosen $S_F$ in the program’s entry point. Hereby, the signature is moved into the signature register and the key_switch routine instructions, which are shown in Listing 6.1, are inserted. This routine first preserves the content of registers rax and rcx by pushing them on the stack, sets up the arguments and invokes the vmfunc instruction, and restores rax and rcx from the stack. The argument rax = 0 for vmfunc instructs the CPU to switch the EPTP to the EPTP specified in rcx = $S$. Note that the compiler reserves the callee-saved register r13 exclusively for the EC-CFI signature.

Call Instrumentation

As functions are encrypted with different encryption keys, the correct decryption key needs to be in place when calling functions. Our toolchain finds all direct and indirect calls and calculates the constant $C = S_{\text{Current}} \oplus S_{\text{Target}}$. For direct calls, the target signature $S_{\text{Target}}$ is the signature of the call header of the corresponding function. As an indirect call can have multiple possible call targets, a points-to analysis is needed to reveal these targets. For external function calls into unprotected programs, e.g., shared libraries, a default target signature using the TME-MK default encryption key is used.

Listing 6.2: Call prologue and epilogue.

1 xor $C, %r13  ; Update signature $S=S \oplus C$.
2 key_switch_routine  ; Switch the key.

Then, right before the call instruction, the compiler inserts the call prologue. As shown in Listing 6.2, this prologue consists of the signature update, i.e., XORing the constant $C$ to the current signature $S$, and the key_switch routine (cf. Listing 6.1). After the call instruction, the identical instruction sequence, i.e., the call epilogue, is inserted to switch back to the key of the caller function.

Call Headers and Footers

To handle multi-call targets (cf. Section 6.3.2), EC-CFI inserts call headers in front of each function.
As illustrated in Listing 6.3, the call header first updates the signature with a
constant to match the signature of the function body. Then, the return constant
$C_{ret}$ is loaded into the reserved r14 register and the key for the function body
is activated. A jump to the function body jumps over the call headers of other
callees. In the callee, the compiler rewrites the addresses of calls to point to the
corresponding call header.

Before each return into the function body, the modified compiler adds, for each
call header, the call footer instructions shown in Listing 6.4. These instructions
update the signature with the return constant such that the signature is identical
to the signature in the call header.

### Code Block Alignment

The **key_switch** routine (cf. Listing 6.1) switches the EPTP and, therefore,
the current, active decryption key. Hence, as the key is immediately switched
after the **vmfunc** instruction, the next fetched instruction is already decrypted
with this key. To avoid that a cache line contains data encrypted with different
encryption keys, which would trigger a cache miss and require a costly additional
memory fetch, our toolchain ensures that **vmfunc** instructions are aligned to
the end of a cache line. Note that this alignment needs to be done in the call
prologues and epilogues as well as in the call headers and footers.

#### 6.4.2 Hypervisor

The hypervisor is responsible for setting up the EPT aliasing functionality and
providing an interface for the binary loader to run protected programs.
System Setup

When booting the system, the hypervisor puts the operating system into the guest mode and creates a virtual machine control structure (VMCS). Furthermore, the hypervisor creates three default and $NUM_{\text{PROT\_EPTS}}$ EPTs and stores the pointer to them into the EPTP list of the VMCS. Note that $NUM_{\text{PROT\_EPTS}}$ is limited by the number of available TME-MK key identifiers and the number of EPTPs which can be stored in the EPTP list. In our prototype implementation, the hypervisor exclusively uses EPT0, the kernel EPT1, and the user mode EPT2. The remaining $NUM_{\text{PROT\_EPTS}}$ EPTs are utilized by protected programs. In the initialization phase, i.e., before starting a protected program, all of these EPTs are identical and use the default 0 TME-MK key identifier in the EPT entries.

Setup of Protected Programs

By using the vmcall instruction, the binary loader communicates with the hypervisor to configure EPT aliasing before starting a program. Here, the loader uses this interface to register the program and the used code pages to the hypervisor. The hypervisor uses this information, i.e., page address and size, to set the TME-MK key identifiers in the entries of the $NUM_{\text{PROT\_EPTS}}$ EPTs. To enable data sharing between functions, only encryption keys for code pages are set. For data pages, the key identifier field in the EPT entries for all EPTs contains the default 0 key. When calling external functions, the compiler ensures that the EPTP is switched to the default user mode EPT2.

Termination of Protected Programs

After the execution of a protected program, a call to the hypervisor is used to deregister the program. Hereby, the hypervisor resets the TME-MK key identifier field in the EPT entries to the default 0 key.

User and Kernel Mode Switches

When switching from user mode to the kernel, the hypervisor needs to save the current, active EPTP, i.e., EPT2 for unprotected programs and EPT2 to EPT2 + $NUM_{\text{PROT\_EPTS}}$ for protected programs, and switch to the kernel EPT1. This saved EPTP is restored when switching back from kernel to user mode. To provide the hypervisor with an opportunity to perform these EPTP switches, the current prototype implementation triggers an EPT violation on each switch between user and kernel using Mode-Based Execution Control (MBEC) by marking pages in user views as only user-executable and pages in the kernel view as only supervisor-executable.
6.4.3 Binary Loader

The modified binary loader is responsible for loading code blocks of an instrumented binary into memory and starting the program. An instrumented binary generated with our custom compiler contains metadata, i.e., address, size, and key identifier for each code block. The loader first allocates a page for the code using this metadata and registers this code page to the hypervisor. Now, the EPT entries of all EPTs for this code page contain the key identifiers specified in the program metadata. To encrypt code blocks with their corresponding key identifier, the loader first switches the EPTP with the `vfmmunc` instruction to the EPT tagged with the key identifier. Then, the code is copied from the binary to the memory encrypting the code block with the key identifier of the current, active EPT. This procedure is repeated for each code block but with a different key. Finally, the loader activates the EPT of the program’s entry point and passes execution to the application.

6.5 Security Discussion

This section discusses security benefits of EC-CFI in respect of the threat model introduced in Section 6.2.

6.5.1 Flipping Address Bits

To redirect the control-flow of a program outside of the call graph, the fault attacker can induce bit-flips into control-flow related addresses. These addresses comprise the instruction pointer `rip` and addresses stored in memory or registers and used by indirect calls. For direct calls, the adversary can induce a fault into the relative address encoded into the instruction, which is then translated to an address by the address generation unit. Here, in EC-CFI, the fault can affect guest linear, guest physical, and host physical addresses. The attacker could aim to redirect the control-flow to any point in the program by injecting faults into guest linear or guest physical addresses. However, when the current active decryption key does not match the encryption key for this point in the program, the execution of these instructions fails. By manipulating both the address and the key identifier in the HPA, the attacker could redirect the control-flow. Nevertheless, this attack vector is hard to exploit, i.e., precisely manipulating both fields is challenging, and the effect is limited. More specifically, executing a single instruction could be possible when redirecting the control-flow by manipulating the address and the key identifier in the HPA. However, as the bit-flip in the key identifier field of the HPA is not permanent for transient faults, the key identifier of the subsequent instruction again is determined by the current EPT. Hence, the decryption of this instruction then fails.
6.5.2 Manipulating EPT Entries

The attacker could try to permanently change the key identifier for an address region by manipulating these bits in the corresponding EPT entries. However, as TME-MK always encrypts the entire external memory using the default key identifier, also the EPTs stored in memory are encrypted. Hence, deterministically flipping key identifier bits in EPT entries without knowing the secret key is not possible. By targeting the translation lookaside buffer (TLB), the attacker could forge the key identifier used for addresses as long as the TLB entry is valid. Here, additional countermeasures, e.g., error detection or correction checks [Sch+21a], could be added.

6.5.3 Leaking Key Identifiers

When the attacker is capable of leaking key identifiers, control-flow manipulations with two precise faults can be possible. Here, the attacker would need to manipulate the key identifier in the EPT entry to the leaked identifier of the target function and redirect the control-flow to this function. However, as controlling a fault, i.e., timing and location, is extremely challenging on a complex Intel CPU, the probability of successfully inducing two subsequent faults is low. Moreover, as the control-flow signature was not changed, it no longer matches the predefined signature. Therefore, the wrong decryption key is used at the next call instruction.

6.5.4 Key Space

Ideally, each function in EC-CFI is encrypted with its own encryption key. Then, redirecting the control-flow to any other function outside of the call graph deterministically fails. However, the encryption key space is limited by the available key identifiers as well as the number of available extended page tables. According to the Intel manual [Int22], in total, TME-MK supports up to $2^{15}$ different key identifiers. However, as our EPT aliasing approach requires us to have multiple extended page tables, the encryption key space is also determined by the number of available EPTs. Currently, the `vmfunc` instruction allows the system to switch between 512 different EPTPs [22]. Hence, when there are more functions in a program than available EPTPs, TME-MK key identifier collisions can occur.

Note that the actual TME-MK key identifier space implemented by the platform could be smaller than the technical upper limit of $2^{15}$ different identifiers in the TME-MK engine. When the key identifier space is smaller than the limit of available entries in the EPTP list, i.e., 512, the following key assignment strategy could be used: The hypervisor assigns each EPT a random key identifier. As some EPTs share the same key identifier, the attacker could redirect the control-flow to other functions and successfully execute code. However, as the signature is accumulatively updated independently from the EPT key identifier, the next derived signature does not match the signature of the next called function.
Hence, with a high probability, at this point, the control-flow manipulation can be detected by EC-CFI.

### 6.5.5 Decrypting Instructions with an Invalid Key

The instruction length in x86-64 is between 1 and 15 B and the opcode can utilize 1 to 3 B in an instruction. To form a valid instruction, both the opcode as well as the other bytes in the instruction need to be valid. Depending on the density of the x86-64 instruction set, which is hard to determine [Eas20], it is possible to retrieve a valid instruction when using an invalid decryption key. Nevertheless, the security impact of decrypting an instruction with a wrong key is minimal due to two reasons. First, the attacker’s goal is to execute a specific instruction and not just a random one. Although some instructions could have multiple opcodes, e.g., 0x00 and 0x01 for an `add`, the remaining decrypted bytes of the instruction are either invalid, causing an instruction fetch failure, or change the behavior of the program. Second, while it might be possible that a single instruction was correctly decrypted, the probability that the subsequent instruction also is valid, is very low. Note that an encryption engine also providing integrity, such as used by Intel TDX [Int23a], could immediately detect decryption attempts with the wrong key.

### 6.5.6 Intra-Function Control-Flow Attacks

Control-flow hijacks within a function, e.g., skipping instructions, cannot be mitigated with the current protection granularity used by EC-CFI. This is in line with our threat model defined in Section 6.2. However, as EC-CFI is a generic concept and not bound to the function-level protection granularity, future work could aim to encrypt code blocks at a finer granularity.

### 6.5.7 Control-Flow Attacks within the Call Graph

Similar to related work [OSM02; SNM22a; Rei+05; WWM15; Cle+16], EC-CFI aims to prevent control-flow manipulations outside of the call graph and not within the borders of the call graph. This is an inherent characteristic of CFI schemes as the compiler cannot exactly determine the targets of indirect calls [Aba+05]. When targeting conditional branches or data used by these instructions, EC-CFI, prevents control-flow redirections outside of the call graph. To mitigate redirections within the call graph, i.e., from one branch target to the other, orthogonal countermeasures [SWM18; SNM22a] are needed.

### 6.5.8 Shared Libraries

As shared libraries need to be accessible for unprotected programs, they are encrypted with the systemwide default 0 key identifier. To avoid that a fault attacker manipulates calls to external functions in libraries, programs can be statically linked, i.e., the libraries are then part of the binary and are, therefore, also...
protected. This protection behavior is in-line with related CFI schemes [OSM02; SNM22a; Rei+05; WWM15].

6.6 Performance Evaluation

In this section, we first evaluate the code size overhead of protecting the Embench-IoT and SPEC CPU2017 benchmarks against fault-based control-flow manipulations using EC-CFI. Then, we analyze the runtime overheads of these benchmarks when using our extended page table aliasing approach. Here, our focus is on evaluating the impact of switching the extended page tables on the translation lookaside buffers (TLBs). We conduct our experiments without enabled TME-MK as the expected performance impact of the memory encryption is small and as we currently do not have access to a system supporting TME-MK for the performance evaluation. According to Intel [Cora], TME-MK induces a performance impact of less than or equal to 2.2% for certain workloads.

6.6.1 Code Size Overhead

To measure the code size overhead of EC-CFI, we compiled the C-based SPEC CPU2017 [Corb] benchmarks without OpenMP support using our custom LLVM-based toolchain. We compiled all benchmarks twice, i.e., in the protected and unprotected mode, with identical compilation flags and enabled the -O3 optimizations. Similarly, we compiled the Embench-IoT [Pat+] benchmark with our custom toolchain. Then, we compared the code sizes of the protected binaries to the unprotected binaries with the GNU size utility.

Table 6.1 highlights the percentual code size overhead of protected SPEC binaries compared to the unprotected baseline. Here, we measured a geometric mean of 82.87% for the code size overhead for all analyzed SPEC CPU2017 benchmarks. For Embench-IoT, we measured a geometric mean of 22.93% for the code size overhead.

In general, the code size overhead consists of three parts: The (i) call headers and footers increase the code size for each function in the program. Similarly, EC-CFI adds (ii) a call epilogue and prologue responsible for switching the key before and after each direct and indirect call. Finally, the (iii) alignment of the

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Overhead [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>600.perlbench_s</td>
<td>143.89</td>
</tr>
<tr>
<td>605.mcf_s</td>
<td>67.29</td>
</tr>
<tr>
<td>619.lbm_s</td>
<td>63.81</td>
</tr>
<tr>
<td>644.nab_s</td>
<td>61.36</td>
</tr>
<tr>
<td>657.xz_s</td>
<td>103.09</td>
</tr>
<tr>
<td><strong>Geometric mean</strong></td>
<td><strong>82.87 %</strong></td>
</tr>
</tbody>
</table>
code blocks and vmfunc instructions to cache lines increases the code size of protected programs as EC-CFI performs this alignment with nop instructions.

### 6.6.2 Runtime Overhead

To measure the performance impact of switching the extended page table pointers, and therefore the view on memory with the extended page tables, with vmfunc, we use the instrumented and uninstrumented binaries generated for the code size evaluation in Section 6.6.1. Here, we executed both versions of the binaries on an Intel CPU supporting the VT building-block of EC-CFI without TME-MK. Note that these instrumented binaries are fully instrumented, i.e., they could be used on a system with support for TME-MK.

Figure 6.7 illustrates the percentual runtime overhead of the instrumented binaries relative to the uninstrumented baseline. Here, we measured a performance impact between x1.18 and x27.05, and a geometric mean of x6.63 for SPEC CPU2017.

Furthermore, we compiled the Embench-IoT [Pat+] benchmark with our custom toolchain and measured the number of cycles with rdtsc [Pao10]. Figure 6.8 highlights the runtime overheads for the Embench-IoT benchmarks. When averaging the cycle count over 10,000 runs and comparing it to the baseline without instrumentation, we measured a geometric mean for the runtime overhead of...
6.6.3 TLB Misses

The EPT aliasing approach requires us to frequently switch the view on memory by switching the EPTP using the `vmfunc` instruction. This switching negatively affects the hit rate of the translation lookaside buffers (TLBs) as the address translation information stored inside these buffers is tagged with the EPTP [22]. Moreover, according to the Intel manual [22], an EPT violation also invalidates the TLB entries associated with the current EPTP. Hence, EPT aliasing increases the pressure on the TLBs. Figure 6.9 depicts the TLB misses for the Embench-IoT benchmarks. Here, we measured with the `perf` tool a geometric mean of x36.46 for the data TLB load misses, x7.08 for the data TLB store misses, and x24.02 for the instruction TLB load misses.

6.7 TME-MK Hardware Change

A minimal-invasive hardware change altering the TME-MK mode of operation can minimize the performance impact of our encryption-based control-flow integrity scheme. Currently, as described in Section 6.1, the TME-MK engine leverages the upper bits of the physical address as the key identifier bits. Hence, to encrypt data with different keys, the identifier needs to be set in the page table or extended page table entries, which requires techniques such as EPT aliasing used in this chapter.

In our proposed hardware change, the TME-MK engine retrieves the key identifier from a user-accessible key identifier register instead from the upper bits of a physical address. The key associated with the key identifier can be rapidly switched by writing to that register. Hence, EC-CFI could be implemented without the EPT aliasing approach, significantly reducing the runtime overhead. With this proposed hardware change, EC-CFI does not induce any additional TLB pressure. Moreover, the key identifier space is no longer limited by the available bits in the physical address, i.e., up to 15 bit, and therefore could be increased to 32 bit.
To measure the runtime overhead of EC-CFI with this hardware modification, we emulated the key switch routine by replacing all `vmfunc` instructions in a protected program with a write to a register. As shown in Figure 6.10, the runtime overhead of EC-CFI for the SPEC CPU2017 benchmark is significantly lower than with the EPT aliasing approach. More specifically, we measured a runtime overhead between x1.02 and x1.51 and a geometric mean of x1.15. Similarly, as shown in Figure 6.11, the proposed hardware change minimizes the runtime overhead of the Embench-IoT benchmarks protected with EC-CFI to a geometric mean of x1.21. We measured a code size overhead between 63.99% and 151.73% and a geometric mean of 86.45% for SPEC CPU2017 and a geometric mean of 24.27% for Embench-IoT.

### 6.8 Related Work

Control-flow integrity [Aba+09] is a generic countermeasure that also can be used to protect programs against software attacks. Here, these schemes assume that the adversary manipulates the control-flow by overwriting control-flow related addresses, such as function pointers or returns, by exploiting a memory safety vulnerability [Sze+13]. To mitigate this threat, CFI schemes [Mas+15; Aba+09; Lil+19; Kuz+14] aim to maintain the integrity of these addresses. CCFI [Mas+15]
protects function pointers and return addresses by calculating a cryptographic MAC of these addresses. By checking the MAC when loading the address, manipulated addresses can be detected. Similar to this approach, PARTS [Lil+19] uses a platform-specific feature, i.e., ARM pointer authentication [Qua], to cryptographically verify the integrity of code and data pointers. However, as the underlying threat model of these countermeasures is weaker (cf. Section 6.2), they can only provide limited protection against fault-induced control-flow hijacks. More specifically, contrary to a software adversary, a fault attacker can also manipulate direct calls and flip bits in the instruction pointer. In addition, as these CFI schemes do not protect the integrity of the instruction pointer, the attacker also can flip bits in rip. Hence, even in the presence of a CFI scheme mitigating software attacks, fault attackers can still manipulate the control-flow.

Therefore, dedicated CFI schemes aiming to protect against faults are commonly used. These schemes [NSM21; OSM02; SNM22a; Rei+05; WS90] derive a signature and, in contrast to EC-CFI, explicitly compare this signature to the signature defined at compile time. Hence, depending on the location of these checks, an adversary capable of redirecting the control-flow still can execute some instructions before the control-flow manipulation is detected. Although within a protection domain, i.e., intra-function, EC-CFI provides similar protection, the protection across function boundaries is stronger. More specifically, when the attacker redirects the control-flow to a function encrypted with a different key, the execution immediately fails. In other schemes, such as FIPAC [SNM22a], the attacker still can execute instructions until the signature is checked, e.g., at the end of a function. Moreover, EC-CFI, with the minimal hardware change, performs similar in terms of runtime overhead than FIPAC, i.e., 15% for EC-CFI and 22% for FIPAC (function end checking policy) for the SPEC CPU2017 geometric mean.

Similar to EC-CFI, SCFP [Wer+18] also implicitly conducts the signature checks by using code encryption. However, SCFP adds a dedicated pipeline stage between the instruction fetch and decode stage for the protection. Moreover, dedicated instructions are added to the instruction set to interact with this pipeline stage. We argue that integrating such intrusive hardware changes into the pipeline of a complex Intel CPU are not feasible as such changes negatively affect area and power consumption as well as they add complexity to the overall functionality of the processor. In comparison, EC-CFI requires no or only minimal-intrusive hardware changes, i.e., changing the behavior or TME-MK, not affecting the general structure of the CPU pipeline.

6.9 Future Work and Limitations

Future Work. The hardware change described in Section 6.7 could be implemented in a system emulator. However, as neither QEMU [Bel05] nor gem5 [Low+20] currently support TME-MK, this hardware extension first needs to be integrated. Moreover, as described in Section 6.4.2, we currently trigger an EPT violation when switching between kernel and user mode to save and
restore the active EPT. As an EPT violation causes a costly \textit{vmexit}, future work could investigate how to avoid these exits. One possibility would be to extend the hypervisor and the kernel. The hypervisor could store the current active EPTP into a kernel-accessible memory region. When entering the kernel from a protected program, the extended kernel then switches to the default kernel EPTP. When leaving the syscall, the kernel could fetch the last active EPTP from the memory region and restore the EPTP. Another option would be to map the kernel address space for each EPTP. Then, when switching from kernel to the user and back, a EPTP switch would not be necessary.

\textbf{Limitations.} In our current prototype implementation, we do not perform a points-to analysis to identify all potential call targets of indirect calls. Instead, we use a default signature for these calls. Although this is a security limitation of the current prototype, this simplification accurately models the runtime and code size overhead. Our current implementation does not support the protection of multiple programs executed on a CPU. To overcome this limitation, the hypervisor needs to be extended to manage the key identifiers and EPTs for each process.

\section{Conclusion}

In this chapter, we have presented EC-CFI, a cryptographically enforced control-flow integrity scheme utilizing recent hardware features of Intel platforms and effective against a fault adversary. EC-CFI prevents that an adversary escapes the call graph of a program by encrypting each function with a different encryption key before executing the application. Only when the execution history is identical to the statically determined control-flow and the call target is within the bounds of the call graph, the decryption key for the called function is correctly derived. On control-flow manipulations outside of the call graph, code is decrypted with the wrong key, which can be detected with no or minimal detection latency. To achieve function-granular encryption on Intel commodity platforms, we have introduced a novel combination of TME-MK and the virtualization technology. In our chapter, we have shown how to utilize our approach based on EPT aliasing to implement EC-CFI and open-source our custom toolchain. Moreover, we have analyzed the EPT switching mechanism using the Embench-IoT and SPEC CPU2017 benchmarks. Finally, we have described and evaluated a TME-MK hardware modification that could significantly reduce the performance impact of EC-CFI.
Protecting Indirect Branches against Faults

Control-Flow Integrity (CFI), as we have shown in Chapter 5 and Chapter 6, is a well-established technique to thwart fault-based control-flow hijacks. Here, CFI schemes restrict the control-flow of the program during the execution to a narrow subset of execution paths. Typically, the set of valid execution paths through the program is statically determined at compile time using the Control-Flow Graph (CFG) extracted from the code. Any control-flow violation, i.e., control-flow deviations from this predefined path, are detected by CFI and hinder the attacker from exploiting the injected fault.

However, state-of-the-art CFI schemes aiming to protect the program execution against fault attacks [VHM03; NM23; Nas+23a; OSM02; LHB14; HLB19; Abe+16; SNM22a; Rei+05] do not explicitly protect code-pointers. A fault into such an address can influence the execution of indirect branches allowing an adversary to bypass CFI and threaten the security of the protected program. Although CFI limits the set of reachable control-flow targets, the statically extracted CFG is only an over-approximation of the actual executed control-flow. Hence, an attacker, especially for indirect branches, could redirect the intended control-flow within the bounds of the approximated control-flow graph.

Hardening a CFI scheme against attacks on unprotected addresses requires adding redundancy to these addresses using data encoding schemes. Such schemes transform addresses to a different representation, allowing the system to detect a certain number of bit-flips induced by faults. While arithmetic codes, such as ANB-codes [For90; Sch+10] or residue codes [Mas64; MG72] allow simple arithmetic operations on the encoded data, software-based schemes induce large runtime overheads. To reduce these large performance overheads, related work [Sch+18a;
MM11] suggests intrusive hardware changes directly in the CPU. However, as custom hardware changes are unrealistic for commercial off-the-shelf systems, a broad range of devices remain unprotected, requiring efficient, software-based countermeasures.

Contribution

In this chapter, we are addressing the issue of insufficient address protection in CFI schemes aiming to thwart control-flow hijacks using fault attacks on commodity devices from ARM. To efficiently protect addresses from targeted faults, which allow attackers to redirect the control-flow, we introduce a software-based hardware-assisted address redundancy scheme capable of detecting such faults. More concretely, we utilize the pointer authentication extension of recent ARM architectures to cryptographically sign code-pointers used by indirect branches. We employ this feature to encode addresses allowing us to detect bit flips injected by a fault rather than protecting against software attacks. To protect addresses at program execution but also when stored in the binary, we encode code-pointers at compile time and verify the integrity at runtime. We showcase how the verification of these addresses using dedicated indirect branch instructions detects bit flips injected by a fault.

Our analysis of state-based CFI schemes further reveals that indirect control-flow transfers are insufficiently protected, allowing an attacker to hijack the control-flow. We propose [NSM21] an enhanced state update mechanism creating a link for such control-flow transfers to mitigate this attack vector. To automatically protect indirect branches from targeted fault attacks, we integrate the address encoding scheme and the hardened state update function into the LLVM-based toolchain of FIPAC [SNM22a], a previously introduced CFI scheme for ARM devices. To evaluate the performance overhead, we compiled a subset of the SPEC2017 benchmark suite with our custom toolchain. Our performance measurement for protecting indirect branches against fault attacks shows a negligible overhead of less than 2.34% on average for protecting indirect branches against fault attacks.

In summary, our contributions are as follows:

- We utilize the ARM pointer authentication feature to protect all addresses used by indirect branches from fault attacks.

- We propose a new state update function for signature-based CFI schemes protecting the link between indirect control-flow transfers.

- We integrate our control-flow protection mechanisms into the LLVM-based toolchain of FIPAC.

- We verify the functional correctness and the performance overhead of our scheme using the SPEC2017 benchmark and discuss security guarantees.
7.1 Background

This section highlights data redundancy schemes protecting against faults and introduces the ARM pointer authentication feature, which can be used to enforce control-flow integrity.

7.1.1 Data Redundancy

To counteract faults on data or addresses, temporal or spatial redundancy is required. While temporal redundancy processes or stores data multiple times [Mor+14; BCR16], spatial redundancy adds additional bits to the data itself, i.e., error detection codes [Ham50]. Data encoding transforms the data to a different representation, such that bit flips up to a certain number of faults are detectable. Examples are binary linear codes [Ham50], ANB-codes [For90; Sch+10], or residue codes [Mas64; MG72], which support the protection during storage but also allow to perform arithmetics on the encoded data. However, software implementations of those encoding schemes are expensive and yield large overheads. For example, ANB-codes have runtime overhead factors between 3 and 140. Similar to that, performing multiple module operations required for residue codes is also costly and thus not suitable for software implementations. To compensate for this performance penalty, dedicated hardware support is required [Sch+18a; MM11]. While these approaches reduce the performance penalty, they require intrusive changes to the processor architecture. Thus, they are not realistic for commodity devices, especially for existing ARM devices.

7.1.2 ARM Pointer Authentication

ARM Pointer Authentication (PA) [Qua] was introduced with the ARMv8.3 architecture and is used to sign and verify the integrity of code- and data-pointers. ARM extended the instruction set with the dedicated sign instructions pac* to sign code- and data-pointers with key A or B, which can be set in privileged mode. Internally, these instructions use the tweakable block cipher QARMA [Ava17] to calculate the MAC of the address, a modifier used as tweak, and the key. This MAC is then truncated to $PAC_{size}$ bits and used as Pointer Authentication Code (PAC). ARM stores this PAC in the upper, unused bits of a pointer to avoid any additional storage overhead. Although this method reduces the virtual address space, systems, such as Linux, anyway limit the address space size to 32- or 48-bit [Mar]. To verify the integrity of a signed pointer, the aut* instructions recompute the MAC of the address and compare it to the PAC stored in the upper bits. If the pointer integrity verification succeeds, the PAC is stripped from the pointer and can be used. In case of an integrity failure, an exception is triggered in ARMv8.6 [Limb]. The ARMv8.3 extension further includes instructions that verify the signed pointer before usage, e.g., the braa recomputes and compares the PAC and then branches to the destination.

The ARM PA feature is already used for pointer integrity in commercial products, beginning with the Apple iPhone XS [Aza]. Additionally, academic
projects demonstrated that the PA extension can also be used to enforce software CFI [Lil+19; Lil+21] and even fault CFI [SNM22a].

7.2 Problem Definition

In this section, we analyze state-of-the-art CFI schemes that protect from fault attackers and discuss exploitable design weaknesses.

7.2.1 A Motivating Example

To protect a program, such as the one in Listing 7.1, from control-flow hijacking attacks triggered by faults, CFI schemes aim to detect control-flow violations.

Most of the CFI schemes [VHM03; NM23; Nas+23a; Aba+09; OSM02; SNM22a; WWM15; Wer+18] aiming to mitigate fault-induced control-flow attacks statically extract the CFG of the program at compile time. When considering a fault CFI scheme enforcing control-flow integrity at basic block level, the extracted CFG is similar to the one illustrated in Figure 7.1.

To protect the program from fault-based control-flow hijacking attacks, these schemes restrict the control-flow to only valid edges of the CFG. In Figure 7.1, basic block $B$ can only be reached from basic block $A_1$ and basic block $C$ and $D$ from basic block $A_5$. As basic block $E$ is never executed in the program, a control-flow redirection to this basic block violates the CFG.

Listing 7.1: Code snippet vulnerable to fault-based control-flow hijacking attacks.

```c
void B(), C(), D(), E();

void A(string password)
{
    void (*fun_ptr)(void) = NULL;
    ... A_1
    B();
    ... A_2
    if (password == "secret")
        fun_ptr = &C;
    else
        fun_ptr = &D
    ... A_3
    (*fun_ptr)();
    ... A_4
    ... A_5
}
```

To enforce the CFG at runtime, signature-based CFI schemes internally maintain a global state $S$ to accumulate the execution history of the program. This state is typically a counter [OSM02] or a cryptographic chain [SNM22a]. Figure 7.1 depicts the basic principle of updating $\mathcal{U}$, patching $\mathcal{P}$, and checking $\mathcal{C}$ this global CFI state $S$, which are explained as follows.

Updating: On each control-flow transfer, the current state $S$ is updated when
entering the next basic block.

\[ S_N = f_U(S_C, ID_{BB}) = S_C \oplus ID_{BB} \]  

Equation (7.1) shows the state update mechanism consisting of the current state \( S_C \), the next state \( S_N \), a unique basic block identifier \( ID_N \), and the accumulating state update function \( f_U() \). In the example in Figure 7.1, on the control-flow transfer from \( A_1 \) to \( B \), the state is updated from \( S = S_{A_1} \) to \( S = S_B = S_{A_1} \oplus ID_B \).

**Patching:** In most programs, the execution diverges to multiple execution flows and then merges again later. As this would create different states for each individual execution flow, signature-based CFI schemes require to patch the state to a common state on control-flow merges.

\[ S_N = f_P(S_C, Patch_{BB}) = S_C \oplus Patch_{BB} \]  

In Equation (7.2), this patching functionality is depicted. Similar to the state update function, the current state \( S_C \) is updated with the patch value \( Patch_{BB} \). This patching mechanism assures that in basic block \( A_5 \) the state \( S = S_{A_5} \) is generated by both branches. Here, the control-flow path through \( A_3 \) generates the state \( S = S_{A_5} = S_{A_3} \oplus ID_{A_5} \) and the path through \( A_4 \) the same state \( S = S_{A_5} = S_{A_4} \oplus Patch_{A_4} \oplus ID_{A_5} \).

**Checking:** To detect control-flow violations, regular checks of the global state \( S \) are required. These checks compare the actual state \( S_C \) with the precomputed state \( S_{expected_{BB}} \) and trigger an exception on a mismatch.

### 7.2.2 Security Analysis

The security of signature-based CFI schemes is based on the enforcement policy of the scheme, the precision of the extracted CFG, and the capability of the signature to detect state mismatches reliably. The enforcement policy is responsible for restricting the control-flow to the nodes of the CFG and is determined by the
number and placement of the state updates and checks. While a larger number of updates and checks clearly increases the enforcement precision of the scheme, it also increases the runtime overhead of the scheme. Hence, software-based CFI schemes \[VHM03; OSM02; SNM22a\] typically update and check the state at basic block granularity to detect inter basic block control-flow hijacks. To also prevent intra basic block control-flow hijacks, e.g., instruction skipping, hardware-assisted schemes \[Wer+18; Cle+17\] even update the state at instruction granularity.

The precision of the CFG is the fundamental cornerstone for the security of CFI schemes. However, this CFG typically is statically extracted at compile time and only offers a limited accuracy. As determining a precise set of valid targets for indirect branches is hard, the CFG used for CFI enforcement is possibly over-approximated and contains multiple target edges \[CV17\].

### Indirect Branches

This over-approximation mainly affects indirect branches, where, in comparison to direct branches, the target address is not known at compile time. The statically extracted CFG is only an approximation of the actually executed control-flow and includes multiple targets for indirect branches. The indirect call in Line 14 in Listing 7.1 highlights the problematic of multiple, valid targets. Although CFI limits valid transitions from $A_5$ to $C$ or $D$, an attacker still could redirect the control-flow within the set of valid targets using two possible fault targets:

**IB1** Faulting Addresses: A targeted fault on the address used in an indirect branch allows the adversary to hijack the actual control-flow and redirect it to another target. The target address in `fun_ptr` could be tampered to point to $D$ instead of $C$. As both basic blocks are in the list of valid targets, a genuine state is generated in both branches and the state verification mechanism cannot detect the control-flow hijack.

**IB2** Faulting the Branch: Even if the address is not modified by a fault, a targeted fault directly injected during the execution of the branch could redirect the control-flow. One example of such an attack would be a fault into the program counter update performed when executing the branch instruction. Here, the target address in `fun_ptr` used by the indirect branch points to $C$ but a fault into this branch redirects the control-flow to $D$. Again, as both targets are possible paths through the CFG, the control-flow redirection remains undetected by the CFI scheme.

### 7.2.3 Threat Model

Similar to threat models of related CFI schemes \[VHM03; OSM02; LHB14; HLB19; Abe+16; SNM22a\] protecting the control-flow of the program against fault attacks, we are considering an attacker having physical access to the system. This attacker is capable of inserting a fault using, e.g., a clock or a voltage glitch, and aims to hijack the control-flow of the program to redirect it to other sensitive code parts. We assume that the system already features a state-based
CFI scheme, such as, for example, FIPAC [SNM22a], thwarting fault attacks on the control-flow. In addition to this threat model, we are considering an attacker aiming to bypass the protection of indirect branches using the fault targets IB1 or IB2. These targets can be attacked by inducing a fault during program runtime or directly into the code stored in the instruction memory. Here, we consider faults flipping bits in addresses used by indirect branches stored in registers, in the immediate field of instructions, or directly in the code segment.

## 7.3 Design

To address the threat model in Section 7.2.3 and counteract the identified weaknesses IB1 and IB2, we show in this section how to thwart fault attacks on indirect branches on recent ARM architectures supporting PA.

### Address Protection

Using a targeted fault IB1 on an address used in an indirect branch allows an adversary to hijack the control-flow of the program. To counteract such attacks, protecting and verifying the integrity of these addresses throughout the program execution is required. However, as indicated in Section 7.1.1, software-based redundancy schemes induce large performance overheads [Sch+10] and hardware-assisted schemes minimizing these performance overheads require intrusive hardware changes [Sch+18a], making it difficult to deploy these schemes on off-the-shelf devices.

To efficiently and securely protect addresses used for indirect branches without hardware changes, we utilize the pointer authentication feature of recent commodity devices from ARM. This feature introduces, as described in Section 7.1.2, dedicated instructions capable of cryptographically signing and verifying pointers. Figure 7.2 depicts a 64-bit pointer signed with the \texttt{pac*} instruction. This instruction uses an optional modifier and a preconfigured key $K$ to calculate the MAC of the 64-bit address. In the latest ARMv8.6-A architecture, the result is then XORed with the original address and stored in the upper, unused bits of the pointer. To verify the integrity of the signed pointer before usage, ARM provides dedicated \texttt{aut*} instructions and combined instructions, such as \texttt{blr*} or \texttt{br*}.

To protect all addresses used by indirect branches, we utilize ARM’s pointer authentication to sign and verify these addresses. More concretely, we exploit the PA feature to efficiently store address redundancy information, \textit{i.e.}, the pointer authentication code, next to the actual address inside the pointer using hardware support. At compile time, we replace the unprotected addresses with their signed equivalent, \textit{i.e.}, the address with the corresponding PAC. Furthermore, we replace all indirect branch instructions, \textit{i.e., branch register} or \textit{branch and link register}, with their PA equivalent, which automatically verify the PAC before usage. Although signing code-pointers at runtime with the \texttt{pac*} instructions would be possible, an attacker still could induce a bit flip into the unprotected
address before the PA instruction protects the address, or directly into the code segment of the program.

The approach of replacing addresses with their PA protected version and verifying them before usage yields, compared to software-based address protection schemes, several advantages. By embedding the redundancy information directly into the corresponding pointer, the design avoids the usage of additional registers and additional register pressure. Furthermore, as ARM’s PA utilizes features of the underlying architecture, the generation and verification of the address redundancy information can be realized using a single instruction, avoiding large performance penalties. Most important, compared to related address protection schemes [Sch+18a], our design does not require custom hardware changes and can be deployed on off-the-shelf hardware from ARM.

**Linking the Branch**

Independently whether the address is protected or not, inducing a bit flip when a code-pointer is used by the indirect branch instruction allows the adversary to hijack the control-flow of the program. Even though ARM PA provides dedicated branch instructions, e.g., `blr*` or `br*`, these instructions first verify the integrity of the address by recomputing and comparing the PAC, removing the PAC from the pointer, and then use this unprotected address for the jump. Hence, a targeted bit flip still enables the attacker to redirect the control-flow to another valid edge in the control-flow graph, which cannot be detected by CFI.

To mitigate this attack vector IB2, we propose to enhance state-update functions of existing state-based CFI schemes. By merging the target address into the global CFI state at the caller side and by removing this address at the callee, we are creating a link between the indirect control-flow transfer. More concretely, we are inducing the target address into the state using a XOR, and we remove this added address by XORing the current address at the callee into the state.

**7.4 Implementation**

This section first introduces FIPAC, the state-based CFI scheme we enhance with our indirect branch protection mechanisms. Then, we elaborate on how we integrate our pointer authentication-based address protection scheme and the hardened state update function linking indirect control-flow transfers into
7.4. Implementation

FIPAC.

7.4.1 FIPAC

FIPAC [SNM22a] is a CFI scheme protecting the control-flow against a software and fault attacker. Similar to other software-based CFI schemes, FIPAC enforces control-flow integrity at basic block granularity using a statically extracted CFG. This CFI scheme exploits hardware features of the ARMv8.6-A architecture to efficiently derive a cryptographic state at each basic block entry. In FIPAC, the state update function uses the pointer authentication (c.f. Section 7.1.2) instructions to create a MAC chain with the execution history.

\[
S = \text{pacia}(S_P, PC, K_A) = MAC_{K_A}(S_P, PC) \oplus S_P
\]  (7.3)

Equation (7.3) shows the update function, which cryptographically links the previous basic block with the current basic block, using the \text{pacia} instruction. The \text{pacia} instruction generates a MAC using the previous state \(S_P\), the current program counter, and a key \(K_A\) and XORs the result to the previous state. This global CFI state \(S\) is then stored into a register exclusively reserved for FIPAC. To verify the integrity of the executed basic blocks, FIPAC uses the \text{autiza} instruction as checking mechanism. On an integrity verification failure, \text{i.e.}, a control-flow redirection outside the CFG at basic block granularity, this instruction triggers an exception. FIPAC automatically protects programs by providing a custom LLVM-based toolchain.

7.4.2 Address Protection

Faulting a code-pointer in the program allows the adversary to redirect the control-flow for indirect branches within the bounds of the CFG. To thwart this attack scenario \text{IB1}, we extend FIPAC to provide protection for all addresses used by indirect branches by exploiting hardware features of the underlying ARM architecture. More concretely, we replace these addresses at compile time with their encoded version embedding the PAC in the upper bits. In addition, we substitute all branch instructions with their \text{PA} equivalent.
Listing 7.2: Replacement of unprotected addresses and branch instructions with their PA protected version.

```assembly
1 <main>:
2 ...
3 ; adr x8, #function //defPAC
4    mov x8,#function\textsubscript{PAC}[15:0]
5    movk x8,#function\textsubscript{PAC}[31:16], lsl #16
6    movk x8,#function\textsubscript{PAC}[47:32], lsl #32
7    movk x8,#function\textsubscript{PAC}[63:48], lsl #48
8 ...
9 ; br x8 //usePACBranch
10  braaz x8
```

To realize the address protection scheme, we introduce a custom ModulePass in the LLVM [LA04] middle-end of the FIPAC toolchain performing a data-flow analysis. We scan each function for indirect calls and track the corresponding virtual register in the Intermediate Representation (IR) containing the address information. By exploiting the def-use property of the Static Single Assignment (SSA) form of LLVM, i.e., each used virtual register is defined at exactly one position, we find the initial store instruction copying the address into a virtual register. If this address is a global value of the type function, we use the LLVM metadata functionality to tag the store with \texttt{defPAC} and the indirect branch instruction with \texttt{usePACBranch}. Inside the IRTranslator and the AArch64InstructionSelector pass in the LLVM backend, we utilize this metadata to replace the indirect branch instructions marked with \texttt{usePACBranch} with their PA equivalent, i.e., \texttt{braaz} or \texttt{blraaz}. These instructions, which use a zero modifier, verify the integrity of the address by recomputing and comparing the PAC stored in the pointer before invoking the address. If the verification fails, an error is triggered. Furthermore, we replace all \texttt{adr} instructions marked with the \texttt{defPAC} tag with four consecutive \texttt{mov} instructions to load a 64-bit immediate value into a register. While the first \texttt{mov} instruction in Line 4 in Listing 7.2 clears the register and stores the first 16-bits of the address into the register, the other 16-bit values of the address are shifted using three \texttt{movk} instructions. As the address information is only available in the linker, we add a custom relocation target to the address stored in the register. This relocation target then is resolved in the linker, where we replace the actual address with the protected version, i.e., we compute the PAC of the address and store it in the upper bits of the pointer.
Listing 7.3: Passing protected addresses to functions.

```<main>:
...  
; adr x8, #function // defPAC
mov x8,#function_PAC[15:0]
movk x8,#function_PAC[31:16], lsl #16
movk x8,#function_PAC[47:32], lsl #32
movk x8,#function_PAC[63:48], lsl #48
str x8, [sp, #16]
...
ldr x0, [sp, #16] ; //usePAC
autiza x0
b #function
<function>:
paciza x0
str x0, [sp, #16] ; // defPAC
...
ldr x8, [sp, #16]
; br x8 ; // usePACBranch
braaz x8
```

Furthermore, we utilize the SSA of the IR in our custom ModulePass to find instructions loading a global address of the function type into a virtual register, which is not used by an indirect branch instruction in the current function. In the backend, we identify these instructions tagged with usePAC and insert an autiza instruction afterwards, as highlighted in Line 11 in Listing 7.3. This instruction uses a zero modifier to verify the PAC and stores the unprotected address back to the target register. As performing a data-flow analysis over function boundaries is challenging, e.g., functions in external libraries or other C files, which cannot be accessed by the ModulePass, we use this mechanism to pass unprotected addresses as function arguments.

Inside a function, we scan for indirect branch instructions and use the SSA to check, if the target address is passed as a function argument. In this case, we tag the branch instruction with usePACBranch to translate it later to a protected branch instruction and tag the store instruction with defPAC. In the backend, we extend all store instructions tagged with defPAC with a paciza, as shown in Line 14 in Listing 7.3. The paciza instructions calculates the PAC of the address using key $K_A$ and the zero modifier and stores this PAC into the upper bits of the pointer.

### 7.4.3 Linking the Branch

To protect indirect branches from threat IB2, we induce the target address of the indirect branch into the global CFI state at the caller and remove this address at the callee, which allows us to detect control-flow hijacking attempts.
Listing 7.4: Target address insertion into the global CFI state S.

```assembly
<main>:
...
state_update(S)
eor  S, S, x8
br   x8
<function>:
adrx27, #function
eor S, S, x27
state_patch(S)
```

Listing 7.4 highlights the basic principle of this protection mechanism. After the actual state update `state_update(S)` of the CFI scheme, we merge the target address of the indirect branch, which is stored in a register, into the state S using an XOR. At the entry of the called function, we first remove the induced address by XORing the state with the address of the current function. Eventually, the CFI scheme patches the state using `state_patch(S)`.

To integrate this mechanism into the LLVM toolchain of FIPAC, we created a custom `MachineFunctionPass` in the LLVM backend scanning for indirect branch instructions. Between the already existing CFI state update function and the indirect call instruction, as depicted in Line 4 in Listing 7.4, we insert a bitwise exclusive OR (`eor`) instruction inducing the target address stored in register `x8` into the global state register. To remove this address from the state, our LLVM pass extends the function header of the callee with two instructions, as illustrated in Line 7 and 8 in Listing 7.4. We utilize the `adr` instruction, which allows to form a PC-relative address using a (negative) offset, to determine the starting address of the current function. Then, we again use an `eor` to add this address to the current state `S`. As this function could be invoked from different callers and state-based CFI schemes require to have a single unique CFI state at a certain position, removing the address induced into the state at the callee is necessary.

### 7.4.4 Combination

When combining both approaches, the caller side XORs the protected address containing the PAC in the upper bits of the pointer to the global CFI state S. At the callee, the unprotected address of the invoked function is determined using the `adr` instruction. To compute the encoded pointer, we insert an additional `paciza` between the `adr` instruction and the XOR correction the state. On a valid indirect control-flow transfer, this instruction computes the same PAC as on the caller side.
7.4.5 Key Management

The precomputation of the encoded addresses at compile time requires the toolchain to have access to the key for the PAC generation. As the system at runtime needs to have the same key for verifying the encoded addresses using the `autiza`, `blraaz`, and `braaz` instructions, we use a pre-shared key $K_A$. This PA key is configured by a custom Linux kernel module running in the kernel mode. We discuss security implications and alternative key sharing approaches in Section 7.5.4.

7.4.6 Compatibility with other CFI schemes

The protection of addresses used by indirect branches can be integrated into other CFI schemes related to FIPAC, if they are deployed on hardware supporting ARM PA. State-based CFI schemes, e.g., CFCSS [OSM02], ACFC [VHM03], or SWIFT [Rei+05], update and verify a global state on each basic block. As our approach thwarting $IB_1$ encodes addresses used by indirect branches and does not influence this state generation and verification mechanism of the underlying CFI scheme, our scheme is fully compatible with these schemes. To protect from attack vector $IB_2$, the state update mechanism of these CFI schemes could be extended to induce the target address into the state at the caller side and remove this address at the callee. By using an XOR, this approach works for schemes using a counter [HLB19] or signatures [WWM15].

7.5 Evaluation

In this section, we first evaluate the performance impact and the code size overhead of hardening indirect branches against fault attacks. Then, we demonstrate the functional correctness of our proposed changes on an emulator supporting the required pointer authentication instructions. Finally, we analyze security guarantees of our enhanced CFI scheme thwarting fault attacks on indirect branches.

7.5.1 Performance Evaluation

To evaluate the performance overhead introduced by the additional protection of indirect branches, we compile a set of benchmarks and execute them on an ARM development board. However, currently, there is no open development board available supporting the ARM pointer authentication instructions introduced in ARMv8.3-A. Although Apple offers this feature in their mobile processors [Inc], iOS restricts the usage of PA by custom software [Aza]. While pointer authentication currently is not broadly available, with the announcement of ARM supporting PA in the upcoming ARMv9-A architecture [Har21], we expect more devices featuring this extension in the near future. Due to the lack of openly available hardware, we emulate the PA instructions on the Raspberry Pi 4 Model B [Fou] consisting of a 64-bit ARMv8-A SoC. To emulate these instructions,
we reuse the cycle accurate emulation approach introduced by PARTS [Lil+19] and used by related work [SNM22a; Lil+21]. This PA-analogue consists of four consecutive XORs simulating the cycles needed for a PA instruction and one memory access.

To quantify the performance overhead introduced by the protection of indirect branches, we execute the SPECspeed2017 [Corb] benchmark suite on the Raspberry Pi. More concretely, we compile all C-based benchmarks without OpenMP support of the SPECspeed2017 Integer test suite with our customized toolchain using the cycle accurate emulation approach for different protection configurations. In Figure 7.3, we depict the performance overhead induced by the protection of indirect branches on top of the baseline, i.e., the SPEC2017 benchmark compiled with the FIPAC toolchain. On average, encoding addresses and verifying them at each indirect branch using the dedicated b1raaz and braaz instructions yields a performance overhead of 1.50%. The protection of the link between indirect control-flow transfers induces a runtime overhead of 0.83% on average. For the combination of both protection mechanism, we measured an average performance overhead of 2.34%.

7.5.2 Code Size Overhead Evaluation

For the code size overhead, we measured an average overhead of 0.16% for IB1, 1.30% for IB2, and a combined overhead of 1.92% for the SPEC2017 benchmark. As the FIPAC toolchain automatically creates two function entry points, one for indirect and one for direct calls, we automatically extend the indirect function header entry with the adr and eor instruction. Hence, the code size overhead is larger for protecting the link between indirect control-flow transfers than for the address encoding using the PA feature.

7.5.3 Functional Evaluation

To verify the functional correctness of our proposed CFI enhancements for FIPAC, we executed the protected SPEC2017 benchmark on a recent Linux kernel on the
QEMU [Bel05] emulator. However, as FIPAC requires the pointer authentication features of the ARMv8.6-A architecture and the latest QEMU 6.0 version only offers PA provided in ARMv8.3-A, we enhanced QEMU to support the EnhancedPAC2 and FPAC features [SNM22a]. The successful execution of SPEC2017 compiled with our extended LLVM compiler hardening indirect branches validates the functional correctness of our proposed CFI policy refinement.

7.5.4 Security Evaluation

As highlighted in Section 7.2, a fault attacker can hijack the control-flow of a program even if a CFI scheme is in place. By inducing a targeted fault into an indirect branch, an attacker can redirect the control-flow within the bounds of the CFG. In this section, we analyze how the protection of indirect branches improves security guarantees for attacker models IB1 and IB2.

Address Protection

In a CFI scheme without indirect branch protection, a single fault into an address used by an indirect branch allows the adversary to hijack the control-flow. When protecting these addresses using the ARM pointer authentication feature, an attacker now needs to induce two faults, one in the address and one in the pointer authentication code. As this PAC, in comparison to other data redundancy schemes, is calculated using a keyed MAC, an attacker, without having access to this key, cannot predict a valid PAC for a target address. Hence, even when having the capability of inducing two precise faults, the likelihood of generating a valid address and PAC pair with a fault is low.

For IB1, a bit flip in a protected address is detected by the next verification instruction, e.g., blraaz. These instructions recompute the PAC using the given address and compare it with the PAC stored in the upper bits of the pointer. If either the address or the PAC is erroneous, the comparison fails and, in the ARMv8.3-A architecture, an error bit is set in the pointer. When using this corrupted pointer, e.g., in blraaz, an error is triggered. As a single error bit can easily be flipped by a fault, we recommend using the ARMv8.6-A architecture, where an authentication error automatically triggers an exception directly in the authentication mechanism.

Linking the Branch

To address attacker model IB2, we merge the target address of the indirect branch at the caller side into the global CFI state and remove this address at the callee. Now, to bypass this protection mechanism, an adversary needs to induce an additional, precise fault on the XOR, which removes the address from the global state. Combining this concept with the address protection, i.e., XORing the PA protected address into the state and remove this address by recomputing the PAC using the paciza instruction, further improves security guarantees.
Key Management

Protecting addresses by replacing them at compile time with their protected PA equivalent requires the toolchain to have access to the used key as the key needs to be identical for the verification at runtime. However, in our threat model considering a physical attacker, we argue that a static key still provides strong protection against faults. An attacker without knowing the key needs to induce a targeted fault into the address and the PAC with the goal of crafting a valid PAC. This requires to flip up to $PAC_{size}$ bits in the PAC. When extending our threat model with an adversary being capable of extracting the PA keys from the highest privilege level, this attacker could precompute a valid PAC. However, the adversary still needs to have the capabilities of inducing a precise fault into the address and also into the PAC. Additionally, the underlying CFI scheme already restricts the control-flow to only valid edges of the CFG. To support dynamic keys, a kernel module configuring ephemeral keys for the PA feature and recomputing and replacing all PACs in the program using binary rewriting could be integrated into the OS [Sch+22].

Function Arguments

In our prototype, we do not conduct an exhaustive data-flow analysis over function boundaries. Currently, addresses passed as function arguments in a register or on the stack are, for a short moment, unprotected, allowing an adversary to induce a bit flip. However, in a future prototype, this attack vector can be avoided by using statically linked libraries and performing a data-flow analysis on the compiled binary and replace unprotected addresses using binary rewriting. Note that this approach also reduces the performance impact of mitigating attacker model IB1, as no additional autiza and paciza instructions are required.

Conditional Branches

Thwarting faults on conditional branches requires the protection of the operands, the comparison, and the branch itself [SWM18]. While the PA feature could be used to protect operands, performing arithmetic operations or comparisons directly on the protected values are not possible. In addition, storing the PAC in the upper bits of the value reduces the data size. Hence, CFI schemes, independently if they are extended with our indirect branch protection, need to be complimented with addition countermeasures addressing attacks on conditional branches.

7.6 Related Work

Code-pointer integrity schemes [Mas+15; Kuz+14] store and validate metadata to protect code-pointers from memory vulnerabilities. However, a fault still can tamper the metadata or the unprotected pointers in registers, allowing to redirect the control-flow. PARTS [Lil+19] utilizes the PA feature to sign and verify all
code- and data-pointers in the program. Although this approach is similar to our work, PARTS only considers a software adversary in their threat model. Hence, PARTS only signs these pointers at runtime, allowing a fault attacker to either induce a fault directly in the address before the pointer is signed or directly into the code segment. Additionally, PARTS is vulnerable to attack vector \textbf{IB2}, as after the verification of the PAC, the unprotected address is used for the jump. In comparison to PARTS, our approach protects addresses used by indirect branches through the whole execution life cycle and the binary in the instruction memory by replacing unprotected addresses with their protected version during compile time. By merging the target address of indirect branches into the global CFI state and removing this address at the callee, we further prevent an attacker from exploiting \textbf{IB2}.

7.7 Conclusion

In this chapter, we have utilized pointer authentication of recent ARM architectures to embed a MAC into addresses to efficiently protect indirect branches. By replacing unprotected addresses and all indirect branch instructions at compile time with their protected equivalent, we thwart fault attacks on these addresses stored in registers, in the immediate field of instructions, or directly in the binary. Additionally, we have enhanced the state update mechanism of signature-based CFI schemes to protect the link between indirect control-flow transfers. To demonstrate how these defense mechanisms improve the protection of state-of-the-art CFI schemes, we have integrated our address encoding and linking strategy into FIPAC. The integration into the LLVM-based toolchain of FIPAC allows the automatic protection without user interaction. Our evaluation with the SPEC2017 benchmark has shown a runtime overhead of less than 2.34% on average.
Control-Flow Integrity for Finite-State Machines

In the course of this thesis, we have demonstrated that Control-Flow Integrity (CFI) is a powerful mitigation technique to protect software against fault-induced control-flow attacks. However, also general hardware primitives, such as Finite-State Machines (FSMs), can be targeted by fault attacks. These controllers are lucrative fault targets, as these fundamental hardware blocks are responsible of controlling systems and their datapaths. By hijacking the execution flow of the FSM using faults, an adversary can manipulate the FSM to enter states which cannot be reached from the current state. Hence, due to the severity of these attacks, security-sensitive state machines need dedicated protection against faults.

A common fault defense strategy is to encode the FSM states such that they are separated with a certain Hamming Distance (HD) [AHS09; CFT21; CTF21]. However, this can only mitigate attackers aiming to induce faults into the state registers. Other defense strategies [DSS05] introduce monitors which check whether the conducted state transition is in the list of valid state transitions. Leveugle et al. [LS90] dynamically verifies that the state transitions stay within the intended execution flow, which is determined during synthesis using the Control-Flow Graph (CFG) of the FSM. There, on each state transition, a signature is derived, and a monitor checks whether the signature matches the predetermined signature of the CFG. However, faults induced either into the next-state logic or into the FSM’s control signals still enable adversaries to redirect the control-flow within the bounds of the CFG. Moreover, the fault detection latency of monitor-based schemes is high and the error coverage is often insufficient [RLS95].
Redundantly instantiating the next-state logic and comparing the resulting states typically requires manual effort by the RTL designer. Moreover, this approach requires an additional redundant next-state logic for each additional fault protection layer. Hence, the area overhead of redundancy-based protection mechanisms scales poorly, especially when considering multi-fault attacks, e.g., quadruple laser fault injection [ALP].

**Contribution**

In this chapter, we introduce SCFI [Nas+23b], a scalable mitigation approach probabilistically protecting the control-flow of finite-state machines against multi-fault attacks. SCFI ensures that any control-flow deviation from the intended control-flow is detected with a high probability by substituting the unprotected next-state logic of the controller with a fault-hardened next-state logic. Internally, this hardened logic absorbs the control signals and the execution history and only generates a valid next state when these inputs are not tampered by faults. When either the control signals, the current state (i.e., the execution history), or the next-state logic itself is targeted by faults, the logic ensures that these faults corrupt the next state output to a degree which can be detected. To ensure this behavior, SCFI uses a lightweight diffusion layer, which is based on a Maximum Distance Separable (MDS) matrix multiplication. We integrate SCFI into the Yosys synthesis suite to automatically protect arbitrary FSMs against fault attacks without any user interaction and open-source\(^1\) the modified toolchain.

In order to evaluate the area and timing overhead, we synthesized several FSMs used in the industry-driven OpenTitan project with our modified synthesis suite. Our comparison with a redundancy-based protection approach of the FSM’s next-state logic shows that SCFI scales better in terms of area-time product for different fault protection levels than classical redundancy-based protection approaches. Finally, we utilize a pre-silicon fault analysis tool to formally verify the fault resiliency of the hardened FSMs.

**Outline**

This chapter is organized as follows. Section 8.1 gives an introduction to fault attacks and finite-state machines. In Section 8.2, we discuss the assumed threat model, provide an attacker description, and formulate the goal of protecting the FSM’s next-state function. Section 8.3 gives an overview of the SCFI design and Section 8.4 reveals implementation details of SCFI. In Section 8.5.3, we discuss security properties of SCFI and in Section 8.5 we analyze FSMs protected with SCFI. Finally, Section 8.7 concludes this chapter.

\(^1\)https://extgit.iaik.tugraz.at/sesys/scfi
8.1 Background

This section provides fundamental background on finite-state machines required for the subsequent chapters.

8.1.1 Finite-State Machines

FSMs are sequential circuits responsible for controlling systems and their datapaths. Internally, an FSM maintains a finite set of states, and a state-transition into the next state that is controlled by the input signals, i.e., the control signals and the current state. The outputs of a Mealy-type FSM are defined by the current state and the input signals, and the outputs of a Moore-type FSM only depend on the current state.

![Figure 8.1: General structure of a finite-state machine.](image)

As depicted in Figure 8.1, an FSM is described using the 5-tuple \( \{ S, X, Y, \phi, \lambda \} \). The \(|S|\) states of an FSM are represented as a \( s \)-bit symbol \( S \), where the size \( s \) needs to be at least \( s = \lceil \log_2(|S|) \rceil \) bits to comprise the entire state space. Furthermore, the FSM consists of \( n \)-bit control signals \( X \) and \( m \)-bit output signals \( Y \). The FSM uses the next-state function \( S_N = \phi(X, S_C) \) to derive the next state \( S_N \) from the current state \( S_C \) and the control signals \( X \). For a Mealy machine, the output \( Y \) depends on the current state \( S_C \) and the input signals \( X \) and is described using the output function \( Y = \lambda(X, S_C) \).

The execution-flow of an FSM can be described using a directed graph, as shown in Figure 8.2. This graph, which is also called a CFG, comprises all valid transitions \( t \in CFG \) the FSM can perform. A valid transition is defined by the valid \( \{S_C, X\} \) pairs and the next-state function \( \phi \).
8.2 Threat Model

We consider a powerful adversary capable of injecting $N - 1$ faults in different clock cycles and at different locations into the device under attack. These faults can be induced independently of the used fault methodology, i.e., we consider local and remote injecting techniques. Similar to related work, we model the impact of a fault as a transient, i.e., a bit-flip, or a permanent, i.e., a stuck-at, effect. The spatial dimension of the induced fault comprises wires as well as combinational and sequential elements of the logic.

8.2.1 Attacker Description

Within this threat model, an attacker aims to hijack the execution-flow of a security-sensitive state machine in the circuit. Based on the general description of a state machine provided in Section 8.1.1, the adversary can achieve this goal by inducing faults into the next-state logic. A fault into the next-state logic allows an adversary to hijack the execution flow of the FSM and to indirectly change the output signals. This fault target can be modeled using the modified next-state logic $S_{N} = \phi(S_C, X, F_N)$, where $F_N$ describes one or multiple faults. Based on this formula, an adversary can induce faults into different fault targets (FT):

- **FT1** State Registers: A fault into the state registers allows the adversary to arbitrarily redirect the control-flow of the FSM inside $t \in CFG$ or outside $t \not\in CFG$ the control-flow graph. For the CFG in Figure 8.2, the adversary could flip bits in the state registers to directly jump from $S_0$ to $S_3$.

- **FT2** Control Signals: By inducing bit-flips into the control signals, the adversary can manipulate the control-flow of the FSM within the borders of the CFG. For example, a fault into the control signal $x_0$ or into the comparison logic can hijack the execution $S_0 \rightarrow S_1$ to $S_0 \rightarrow S_2$ in Figure 8.2.
8.3. Design

FT3 Next-State Logic: When directly targeting the logic of the next-state function, the adversary can arbitrarily redirect the control-flow of the FSM within or outside the CFG.

8.2.2 Goal - Fault Secure FSM

In order to comprehensively protect the control-flow of finite-state machines against fault attacks, dedicated fault countermeasures must consider all fault targets $FT1$, $FT2$, and $FT3$. The goal is that a fault-protected controller $FSM_F$ influenced by faults detects any control-flow deviations from the control-flow of an identical copy $FSM_{\bar{F}}$ which is not affected by faults, i.e., $\phi_F(S, X, F_N) \neq \phi_{\bar{F}}(S, X, 0)$.

8.3 Design

To comprehensively protect finite-state machines against control-flow hijacks, with SCFI, we maintain the integrity of the control-flow by introducing a fault-hardened next-state logic $\phi_{FH}$. This hardened next-state logic prevents that a fault into $FT1$, $FT2$, or $FT3$ enables the adversary to redirect the control-flow inside or outside the boundaries of the CFG. This function $\phi_{FH}$ is internally constructed using a Multi-Input Signature Register (MISR) and it links the entire execution history in a compressed format to detect control-flow deviations. To enter the next valid state, the execution history as well as the control signals need to be genuine. As shown in Figure 8.3, $\phi_{FH}$ maps a valid tuple $\{X, S_C\}$, which includes the execution history in $S_C$, into a valid next state $S_N$. When an adversary induces faults into $FT1...FT3$, i.e., either into the tuple $\{X, S_C\}$ or into the $\phi_{FH}$ logic, $\phi_{FH}$ forces the FSM into a non-escapable terminal error state. Figure 8.4 depicts the transformation of an unprotected next-state logic of an example FSM into a protected version. The unprotected FSM is susceptible to faults, as a single fault into the state registers, the comparison logic, or the control signals can change the execution-flow of the FSM. SCFI closes these attack vectors by deriving the next state using $\phi_{FH}$. If the current state, the control signals, or the next-state logic is tampered with a fault, $\phi_{FH}$ produces an invalid state and enters the non-escapable default error state. To achieve this protection degree, the next-state function and its inputs and outputs need to fulfill requirements $R1$ to $R3$:

**R1** Encoded Control Signals: All control signals $X$ are encoded to $X_e$. The encoding needs to guarantee that the attacker needs at least $N$ bit-flips to manipulate a valid control-signal codeword to another valid codeword.

**R2** Encoded States: All states $S$ are encoded to $s_e$-bit states $S_e$. Similar to the control signals, the encoding needs to guarantee a minimum Hamming Distance between valid states of $N$.

**R3** Hardened Next-State Function: The hardened next-state function $\phi_{FH}$ generates an encoded next state $S_{Ne}$ fulfilling $R2$ for each encoded control
Figure 8.3: Mapping of valid and invalid input tuples to a valid or invalid next state.

**unique case (SC)**

\[ S_0: \begin{align*}
&\text{SN} = S_0; \\
&\text{if (x0)} \\
&\quad \text{SN} = S_1; \\
&\text{else if (x1)} \\
&\quad \text{SN} = S_2; \\
&\text{end}
\end{align*} \]

\[ S_1: \begin{align*}
&\text{SN} = S_1; \\
&\text{if (x2)} \\
&\quad \text{SN} = S_3; \\
&\text{end}
\end{align*} \]

**ERROR**

\[ \text{SN} = \text{ERROR}; \]

**default**

\[ \text{fsm_alert} = \text{err_signal}; \\
\text{SN} = \text{ERROR}; \]

\[ \text{end} \]

**unique case (SC)**

\[ S_0: \begin{align*}
&\text{SN} = \phi_{FH}(SC,X); \\
&\text{end}
\end{align*} \]

\[ S_1: \begin{align*}
&\text{SN} = \phi_{FH}(SC,X); \\
&\text{end}
\end{align*} \]

**ERROR**

\[ \text{SN} = \text{ERROR}; \]

**default**

\[ \text{fsm_alert} = \text{err_signal}; \\
\text{SN} = \text{ERROR}; \]

\[ \text{end} \]

Figure 8.4: Unprotected and protected next-state logic of an example FSM.

The hardened next-state function needs to ensure that up to \( N - 1 \) bit-flips into its circuit or into the input space affect the output in such a way, that the faults can be detected, i.e., an invalid state \( S_{N_e} \) is generated (R3.2).

Due to requirement R3, the state derived in different paths merging at some point also produces different encoded states. For example, the path \( S_1 \rightarrow S_3 \) in Figure 8.2 derives a different state than the path \( S_2 \rightarrow S_3 \). As maintaining different state symbols for a single state is costly, we add an additional requirement:

**R4** Collision Capability: The hardened next-state function needs to produce the same encoded next state for different paths using a modifier, i.e., \( \phi_{FH}(SC_{1e},X_{1e},Mod_1) = \phi_{FH}(SC_{2e},X_{2e},Mod_2) \) for \( SC_{1e} \neq SC_{2e} \) and \( X_{1e} \neq X_{2e} \). The modifiers \( Mod_1 \) and \( Mod_2 \) are used to produce a state collision.

In the following section, we discuss one possible selection for the hardened next-state function.
8.3. Design

8.3.1 Selection of the Hardened Next-State Function

In SCFI, the hardened next-state function $\phi_{FH}$ is based on a lightweight diffusion function used in cryptographic primitives. This function $\phi_{FH}$, as shown in Figure 8.5, maps the input space consisting of the encoded control signals $X_e$ (R1), the encoded current state $S_{Ce}$ (R2), and the modifier $Mod$ (R4) to a next encoded state $S_{Ne}$ (R3.1). The properties of the underlying diffusion function imply that any fault at the input space or within the logic maximally affect the output, thereby substantially decreasing the probability of a successful fault attack and probabilistically fulfilling (R3.2). Overall, SCFI’s hardened next-state function consists of three layers:

Mix Layer In this layer, the input triple is split into $k$ $l$-bit vectors $L$. For this, the encoded current state, the encoded control signals, and the modifier are split into $k$ shares and each share is placed into the vectors, as shown in Figure 8.5.

Diffusion Layer Then, in the diffusion layer, the vectors $L$ are absorbed by $k$ diffusion functions. These functions conduct a linear transformation $D(L) = M \cdot L$ which is a matrix multiplication of vector $L$ with matrix $M$ in a specific field. This transformation, depending on the choice of matrix $M$, yields a strong diffusion. Ideal choices of this matrix are called MDS matrices, maximizing the diffusion property. Hence, by choosing such an ideal MDS matrix, a fault at the input or within the function maximally propagates to the output, destroying the next state with a high probability (R3.2).
Unmix Layer The output of the diffusion layer is stored into $k \cdot l$-bit vectors. The concatenation of the first $s_e/k = s_k$-bits of each output vector results in the encoded next state $S_{Ne}$ (R3.1). As the size $k \cdot l$ of the output space is larger than the size $s_e$ of the encoded state, $k \cdot l - s_e$ bits are free, providing the collision property (R4). Additionally, SCFI uses, depending on the required fault security, the $e$ topmost free bits of each output vector as error detection bits $E$. Here, by choosing a corresponding modifier $Mod$, $\phi_{FH}$ sets these bits to a predefined value, i.e., 1. In the error logic, the logical AND of $S_{Ne}$ and $E$ infects the next state when a fault-induced error happens.

8.4 Implementation

We open-source a modified version of the Yosys [Wolb] open synthesis suite capable of automatically protecting arbitrary FSMs with SCFI. The protection can be enabled globally or selectively for the unprotected FSMs in the design flow with a certain fault protection level $N$. Our implementation adds a new Yosys pass to the suite operating in between of other optimization passes before the design is mapped to the logic gate level. Note that the RTL designer only needs to manually encode the control signals with a Hamming Distance of $N$-bits in the modules driving these signals.

8.4.1 Next-State Logic

First, our custom FSM protection pass identifies the unprotected FSM by utilizing the existing Yosys FSM passes. Then, the FSM’s state variables are re-encoded so that the Hamming Distance between these variables is $N$. Afterwards, our pass extracts the CFG of the FSM and stores the current state, the next state, and the control signals for each control-flow edge. With this information, the modifier $Mod$ for state transition is determined, satisfying the equation $MDS(S_{Ce}, X_e, Mod) = S_{Ne}$.

For the MDS diffusion function, we use a lightweight construction with a minimal gate count proposed by Duval et al. [DL18]. As shown in Figure 8.6, this function splits the 32-bit input space into 4 8-bit chunks, performs the matrix multiplication, and returns 4 8-bit vectors which form the 32-bit output. In SCFI, we selected the $M^{5, 3}_{4, 6}$ [DL18] MDS matrix operating in the field $F_2[\alpha]$ with $\alpha = X^8 + X^2 + 1$. This particular matrix has a low XOR count with a slightly larger logical depth compared to other matrices in the $4 \times 4$ category. We note that the choice of MDS matrix can be changed according to design requirements, i.e., area or timing constraints.

Having the modifiers, our pass describes the logic of the next-state function in the internal Yosys Register-Transfer Level Intermediate Language (RTLIL). As depicted in Figure 8.7, first 1, the active control signal $X_{active}$ is determined by performing a pattern match of the control signal and the current state $S_{Ce}$. Then, using this signal and $S_{Ce}$, the modifier for this input is selected 2. In the mix layer 3, the wires of the triple $\{S_{Ce}, X_{active}, Mod_{active}\}$ are distributed
8.5. Evaluation

To evaluate the effectiveness of SCFI in terms of area, timing, and security when protecting security-sensitive FSMs of an industry-driven project, we integrate our custom Yosys pass into the design flow of the OpenTitan [Joh+18] secure element. This chip, which is entirely open-source, acts as a secure root-of-trust and provides a key storage and cryptographic accelerators.

Figure 8.6: Internal structure of the MDS matrix multiplication [DL18]. All elements operate on 1-bytes each.

Figure 8.7: The next-state logic hardening pass.

to the $k$ 32-bit input MDS diffusion functions. These lightweight diffusion functions consist of only XOR gates. In the unmix layer, the next state $S_{Ne}$ is concatenated and the error bits $E$ are selected. By connecting $S_{Ne}$ and $E$ using AND gates, a fault infectively destroys the next state.

After the execution of the FSM protection pass, the hardened next-state function described in RTLIL is mapped to logic gates by the subsequent Yosys passes.

8.5 Evaluation
8.5.1 Area Overhead

In order to evaluate the area overhead introduced by SCFI, we analyzed unprotected (i), manually protected (ii), and automatically protected (iii) FSMs. As the reference (i) for our evaluation, we selected several FSMs of OpenTitan and synthesized the entire corresponding module with Yosys using the open-source Nangate45 standard cell library. For the manually protected (ii) FSMs, we encoded the control signals with a Hamming Distance of \( N \)-bits and instantiated the next-state logic of the FSM \( N \) times. To detect control-flow hijacks triggered by faults, we designed a small error logic monitoring the state registers of the redundant FSMs and raising an error signal when one or more state values mismatch. Finally, we automatically protected (iii) the reference (i) FSMs by calling the SCFI Yosys pass in the design flow. Similar to the manually protected (ii) FSMs, we encoded the control signals with a HD of \( N \)-bits and configured SCFI that at least \( N \) faults are required to hijack the FSM. Table 8.1 illustrates the area overheads for the three configurations for different FSMs and different protection levels \( N \) ranging from 2 to 4. For the manual redundancy approach, the geometric mean of the area overhead is 17.5% for \( N = 2 \), 42.9% for \( N = 3 \), and 67.6% for \( N = 4 \). In comparison, the geometric mean area overhead for the FSMs protected with SCFI is 9.6% for \( N = 2 \), 21.8% for \( N = 3 \), and 27.1% for \( N = 4 \). Note that for smaller input spaces \( \{ S_C, X, Mod \} \) the area overhead for SCFI could be higher than for a redundancy approach (cf. otbn_controller in Table 8.1) as SCFI needs to instantiate a MDS matrix with a 32-bit input.

8.5.2 Timing Overhead

SCFI affects the timing of the next-state logic by introducing the fault-hardened next-state function \( \phi_{FH} \). However, the timing overhead is minimal, as the logical depth of \( \phi_{FH} \) comprises four XOR layers for the MDS multiplication and an AND layer for the error masking. We successfully synthesized all modules in all configurations depicted in Table 8.1 for OpenTitan’s target frequency of 125 MHz.

<table>
<thead>
<tr>
<th>Protection Level</th>
<th>Unprotected Area [GE]</th>
<th>Redundancy Area [%]</th>
<th>SCFI Area [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>adc_ctrl_fsm</td>
<td>1019</td>
<td>38</td>
<td>76</td>
</tr>
<tr>
<td>aes_control</td>
<td>632</td>
<td>13</td>
<td>44</td>
</tr>
<tr>
<td>i2c_fsm</td>
<td>2729</td>
<td>38</td>
<td>70</td>
</tr>
<tr>
<td>ibex_controller</td>
<td>537</td>
<td>29</td>
<td>75</td>
</tr>
<tr>
<td>ibex_lsu</td>
<td>933</td>
<td>10</td>
<td>21</td>
</tr>
<tr>
<td>otbn_controller</td>
<td>2857</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>pwrmgr_fsm</td>
<td>301</td>
<td>89</td>
<td>184</td>
</tr>
<tr>
<td>Geometric Mean</td>
<td>17.5</td>
<td>42.9</td>
<td>67.6</td>
</tr>
</tbody>
</table>

Table 8.1: Area overhead for protecting different FSMs using redundancy or SCFI.
### 8.5. Evaluation

![Figure 8.8: Area-time product for the adc_ctrl_fsm module in different configurations.](image)

with Yosys and the open-source standard cell library.

Figure 8.8 illustrates the area-time (AT) product for the unmodified, the redundancy-protected, and the SCFI-hardened adc_ctrl_fsm module. In this plot, we increased the clock period from 3200 ps to 6000 ps and measured the area in kGE of the design synthesized by Cadence Genus and a proprietary cell library. For this experiment, we switched from Yosys to the Cadence synthesis suite as Yosys and the internally utilized yosys-abc tool only provides basic area and time optimization functionality. As shown in Figure 8.8, Cadence was able to meet the timing for a maximum frequency of 312 MHz for the base design, 308 MHz for the design using redundancy, and 294 MHz for SCFI. However, this slightly decreased frequency is typically not problematic, as the critical path of a design is usually not in an FSM. Moreover, as depicted, SCFI achieves a better AT product for protecting the next-state logic of the FSM in the adc_ctrl fsm module than the redundancy approach.

### 8.5.3 Security Evaluation

By encoding the control signals and the state variable, an adversary cannot hijack the state machine by inducing faults into fault targets FT1 and FT2. As the input pattern matching logic \( \phi_{FH} \) operates on these encoded signals, the attacker needs to induce \( N \) faults into this block to manipulate the active, encoded control signal. While a fault into the modifier selection block \( \phi_{sel} \), which consists of multiplexers, could select a different modifier, the attacker cannot exploit this injected fault. More specifically, a fault would yield a combination of control signal, state, and modifier which creates a non-valid next state. Internally, the mix layer \( \phi_{mix} \) consists of a rewiring of the encoded control signals, the state, as well as the modifier. Hence, this layer can resist up to \( N - 1 \) faults. The idea of the diffusion layer \( \phi_{diff} \) is that a small change at the input causes a significant change at the output, i.e., the avalanche effect. To achieve this property, SCFI internally uses MDS matrix multiplication yielding optimal diffusion guarantees. These MDS matrices propagate a bit-flip in a single input byte to all four output bytes, i.e., they have a branch number of 5. Hence, one or multiple bit-flips into the input triple \( \{S_{Ce}, X_{active Mod_{active}}\} \) propagate through this function affecting
multiple output bits. By effecting the next state $S_{Ce}$ or the error bits $E$, an invalid state is generated in the unmix layer and the FSM enters the default error state. Precisely, there are only $|S_{Ne}| + |E|$ valid output states; an attacker who induces $N$ faults on the next-state function inputs, \{$X, S_{C}$\}, would have a success probability of $P = \frac{|S_{Ne}| + |E|}{2^{|2S_{Ne} + |E|}}$. However, considering $|S_{Ce}| + |E| << k \cdot 2^{32-(|S_{Ne} + |E|)}$, the success probability is very small. For attacks within the next-state function, the MDS property of the diffusion layer ensures that the success probability still remains quite low, albeit it is higher than the previous case. As shown in Figure 8.6 depicting the construction of the MDS matrix, faults in the first three XOR layers propagate to at least two output bytes. Although a fault at the last layer only affects one output byte, all valid output states $S_N$ are still encoded with a Hamming Distance of $N$, requiring that the adversary needs to induce $N$ bit-flips.

### 8.5.4 Formal Security Analysis

We formally analyzed the resilience of the diffusion layer consisting of the MDS matrix multiplication by utilizing SYNFI (cf. Chapter 9), a recently introduced pre-silicon fault analysis tool operating at the netlist. For the analysis, we synthesized an FSM with 14 state transitions and configured SCFI with a protection level of 2 bits (HD). We used SYNFI to analyze whether it is possible to hijack one of the state transitions and enter another next state using faults. In total, we injected 7644 single bit-flips exhaustively into all available gates in the MDS matrix multiplication and 32 (0.42%) of these faults enable an adversary to hijack the execution-flow of the FSM.

Note that analyzing the resilience of FSMs against faults is also necessary when using other protection approaches. For example, when redundantly instantiating the next-state logic to mitigate faults, a synthesis tool aiming to meet timing and area constraints could weaken the security when optimizing the design.

### 8.6 Limitation & Future Work

A potential future work could extend SCFI to adapt the MDS matrix size to the size of the \{$S_c, X, Mod$\} input triple to further improve the area-time product. In addition, the formal analysis could be integrated into the Yosys pass to increase security guarantees of SCFI. Finally, a future work could investigate how SCFI could be extended to also provide protection for the output logic.

A limitation of the current prototype implementation is that the selector signals of the MUXes used in the input pattern matching logic are 1-bit signals. This would allow an adversary to redirect the control-flow within the bounds of the CFG. To mitigate this attack vector, an updated version of the SCFI Yosys pass could introduce encoded selector signals.
8.7 Conclusion

In this chapter, we have presented SCFI, a methodology capable of protecting the control-flow of finite-state machines against fault attacks. SCFI substitutes the next-state logic of FSMs with a fault-hardened function only deriving a next valid state in a fault-free scenario. We have integrated SCFI into the Yosys synthesis suite and open-sourced our modified toolchain. Our evaluation has shown that the area overhead for an FSMs protected with SCFI is lower than for traditional protection approaches.
Pre-Silicon Fault Countermeasure Analysis

Security-critical devices, such as secure elements, feature dedicated hardware-based countermeasures to protect against fault attacks. Hence, as the resistance of the circuit against faults relies on these countermeasures, it must be assured that they provide the expected security guarantees.

To ensure the correctness of the countermeasures, hardware engineers responsible for designing secure chips must analyze the circuit when influenced by faults in the design phase. This pre-silicon evaluation needs to comprise two central analysis points: First, can induced faults influence the input-output relation of a security-critical circuit and can the countermeasures detect them? Here, the hardware designer wants to reveal whether a fault affects the circuit and to verify that the countermeasure achieves the promised security level, i.e., can handle up to a certain number of simultaneously induced faults specified in the threat model. Second, can the embedded countermeasures hinder an adversary from entering a specific, security-critical circuit state using faults? An example of such a state is the debug mode of a secure element allowing the adversary to escalate privileges.

Testing the resilience of the circuit and its countermeasures against faults needs to be conducted in the last stage of the front-end design, i.e., at synthesized gate-level netlist. This approach ensures that (i) defective countermeasures are detected as early as possible avoiding long design turnaround times. Additionally, at this (ii) level of abstraction, the design uses the standard cell library provided by the manufacturer and, therefore, is already close to the final circuit sent to the fab for the tape-out. Furthermore, performing the security assessment at the netlist ensures (iii) that flaws introduced by the tooling can be detected. Here, especially
the logic synthesis design flow step mapping the Register-Transfer Level (RTL) model to the synthesized gate-level netlist could negatively affect countermeasures using redundancy to detect or mitigate faults. Here, the synthesis optimization passes aiming to meet design constraints, e.g., the area consumption, could be responsible of reducing security guarantees.

One approach of analyzing the resilience of the circuit against faults at the netlist level is to manually induce faults and to analyze their effect in the simulation phase. However, as the names of the wires and cells in the netlist are renamed or mangled by synthesis tools, manually inducing faults in the testbench is an error-prone task. Additionally, the analysis process is very time-consuming since the simulation needs to be restarted for each induced fault. Hence, this process is often at risk of being foregone in the verification phase of the design due to development schedule pressure.

In order to verify the functionality of fault countermeasures embedded into the chip, a framework capable of automatically performing a pre-silicon analysis based on the synthesized gate-level netlist is needed. It is crucial for such a tool to be capable of handling industry-grade netlists using proprietary standard-cell libraries without imposing any restrictions on the netlist and the design. Otherwise, such restrictions would render the tool practically irrelevant, especially for commercial projects that rely on established hardware design flows with a multi-stakeholder design team.

Recently published tools [Bur+17; Arr+20; Ric+21; BN08; SKK13] cannot be used to analyze unmodified netlists of industry-driven projects, as these frameworks impose invasive requirements to the design. Tools, such as FIVER [Ric+21], limit (a) the supported gates in the netlist to a small set, preventing the usage of complex, proprietary standard cell libraries. Furthermore, most of these frameworks [Ric+21] require (b) that the given netlist does not include any cycles, i.e., the hardware designer needs to manually unroll the design before the evaluation. Additionally, related work often demands (c) that the netlist is fully flattened, i.e., does not include any submodules or hierarchy, does not support (d) all language features, or is not (e) open-source. Finally, as most fault injection frameworks [Bur+17; Arr+20; BN08] exclusively focus on analyzing cryptographic primitives (f), it remains unclear whether these frameworks also can be used to assess the security of more generic hardware designs consisting of a diverse set of hardware IP components, especially in respect with the two analysis points described above.

**Contribution**

In this chapter, we present SYNFI [Nas+22], a versatile framework capable of performing a pre-silicon fault analysis of synthesized gate-level netlists. SYNFI allows a hardware designer to automatically analyze the resilience of a circuit and its countermeasures against fault attacks with minimal setup overhead. More specifically, SYNFI enables hardware designers and security engineers to study the impact of faults on the circuit, to analyze the functionality of tailored fault countermeasures, and to investigate which cells are the most critical attack
targets and need special protection. This information can be used to find logical flaws in the design as well as defects introduced by the hardware design flow tools before the tape-out of the chip.

The SYNFI framework is capable of performing the pre-silicon fault analysis on unmodified netlists generated with proprietary or open design flows and standard cell libraries of designs using common hardware design patterns. For the fault experiment, the security engineer needs to provide information about the circuit to analyze and the fault model. SYNFI supports fault models comprising single and multiple faults injected into various locations in the circuit and different fault effects, i.e., transient or stuck-at effects. With this configuration, SYNFI automatically extracts the circuit to analyze from the netlist and injects faults according to the fault model. In the analysis phase, SYNFI reveals whether a fault affects the input-output relation of the circuit, shows whether the embedded countermeasures can detect faults up to a certain number, and verifies whether a fault could enable an adversary to enter a security-critical state.

To emphasize the importance of conducting a pre-silicon fault analysis before an upcoming tape-out, we utilize SYNFI to analyze components of the OpenTitan secure element. In particular, we focus on analyzing the fault-resiliency of the most security-critical components, such as the AES primitive, the life cycle controller, the lockstep mode of the processor, and several other fault hardened IP. For our assessment, we study the impact of single and multiple faults induced into different parts of the modules for various fault effects. We conduct our analysis on the unmodified netlist generated with the internal, proprietary hardware design flow of OpenTitan including a commercial standard cell library as well as on the netlist synthesized with open-source tools. We utilize SYNFI to (i) reveal the impact of faults to unprotected circuits, to (ii) verify that the redundancy-based countermeasures are not removed by the synthesis tool, and (iii) to verify whether certain security-critical states cannot be entered using faults without triggering the countermeasures. Our in-depth analysis of the tested modules revealed that the AES module is highly susceptible to fault attacks. More concretely, our evaluation disclosed that already a single fault into the AES round counter, the handshake signals, or certain finite-state machines allow an adversary to break the security of the module. To mitigate the encountered security violations, we developed several fault hardening mechanism and integrated them into the OpenTitan project. We ensured the correctness of these countermeasures by reassessing the hardened module using SYNFI. For fault-hardened modules, such as the life cycle controller, we were able to formally verify the expected fault-resiliency.

In summary, our contributions are:

- We present and implement SYNFI, an open-source1 framework capable of performing a pre-silicon fault analysis at the gate-level. SYNFI allows security engineers to automatically (i) reveal whether a fault affects the input-output relation of a circuit and its countermeasures and (ii) assess if an adversary can enter a particular circuit state without triggering the

1https://github.com/lowRISC/synfi
countermeasures. In contrast to related work, the SYNFI framework is able to process unmodified netlists of hardware designs making use of a variety of design patterns and synthesized with commercial and open-source synthesis tools.

- We identified several fault attack vectors for the unprotected AES module used in the OpenTitan secure element allowing an adversary to threaten the security of the encryption primitive. To prevent the exploitation of these flaws in the final taped-out chip, we implemented, reassessed, and contributed several fault hardening techniques to the upstream project.

- We verified with SYNFI that a selection of the most security-critical Open-Titan IP blocks hardened against faults provides the expected security. In particular, we verified, among other modules, that an adversary cannot hijack the life cycle controller to enter the RMA debug state from the production state and that a fault into the program counter of the processor is detected by the lockstep mode of the CPU.

Outline

In Section 9.1, we highlight the design and the implementation of the SYNFI framework. We demonstrate the capabilities of our framework in Section 9.2 by analyzing security-critical fault countermeasures of OpenTitan. In Section 9.3, we highlight related work and compare these frameworks to SYNFI. Finally, Section 9.4 discusses current limitations and possible future work and Section 9.5 concludes this chapter.

9.1 Design and Implementation

This section describes the fundamental concepts of the SYNFI framework along with the design rationale. We first give a high-level overview of the framework and then provide an in-depth description of all the design stages of SYNFI.

9.1.1 Overview

To analyze the effects of one or multiple faults to the input-output relation of a circuit and its fault countermeasures, the gate-level netlist, the used standard cell library, as well as a fault specification need to be provided to the SYNFI framework.

Netlist & Cell Library: The first input of the SYNFI framework is the unmodified netlist of the module to analyze and the standard cell library that the design is mapped against. As shown in the block diagram in Figure 9.1, the synthesis design flow step, which is not part of the SYNFI framework, is responsible for transforming the RTL design into the netlist using the standard cell library.
9.1. Design and Implementation

Figure 9.1: Block diagram of the SYNFI framework.

Fault Specification: The second input of the framework is the fault specification file responsible for describing the fault experiment the designer wants to perform.

Listing 9.1: Fault specification.

```
"Fault Specification": {
  "Target Circuit": {
    "inputs": ["in_port1": "2'b00", "in_port2": "2'b01"],
    "outputs": ["out_port1": "2'b11", "in_port2": "2'b10"]
  },
  "Fault Model": {
    "Simultaneous Faults": 2,
    "Fault Locations": ["gate1", "gate2", "..."],
    "Fault Mappings": ["NAND2": ["AND2", "OR2"], "XNOR2": ["XOR2"]]
  }
}
```

As shown in Listing 9.1, the fault specification file is split into (i) the description of the subcircuit the designer wants to evaluate and (ii) the fault model describing the faults injected into this subcircuit. The subcircuit to analyze (i) is defined by the user by providing input and output nodes, e.g., input or output ports, cells, or submodules of the design. Furthermore, the user needs to assign inputs and expected output values for the provided nodes. The fault model (ii) describes all faults which are induced into the subcircuit. Here, the fault model consists of the (a) number of faults injected into the circuit, the (b) location, and the (c) effects of the faults. With the number of faults (a), the user can specify how many faults are simultaneously injected into the subcircuit. In SYNFI, a fault is injected into (b) a certain location, i.e., a gate. Here, the user can either provide a list of gates which are attacked or select an exhaustive approach where SYNFI automatically injects faults into all gates. The last parameter is the (c) fault effect. Similar to [RSG21], we model the effect of a fault induced into a certain gate by replacing the boolean function of the gate type according to a mapping. For example, the mapping NAND2=AND2 replaces a gate of type NAND2
during the attack phase with an AND2. Here, by inverting the boolean function, SYNFI is capable of modeling a transient fault effect. To model a stuck-at 0 or 1 fault, the boolean function of the corresponding gate can be set to a 0 or 1 in the fault mapping. By providing multiple entries in the fault mapping, e.g., NAND2=[AND2, 0], SYNFI can be used to analyze the circuit when influenced by transient or stuck-at faults. Summarized, this mapping enables SYNFI to model transient, stuck-at, or more advanced fault effects. For each subcircuit the user wants to analyze, a new fault specification file needs to be provided and SYNFI needs to be started again.

SYNFI: With the unmodified netlist, the standard cell library, and the fault specification, the tool starts the two-phase transformation and analysis process depicted in Figure 9.1. In Phase 0, the framework transforms the netlist into a directed multigraph and converts the cell library into a format the subsequent steps of the SYNFI framework support. In Phase 1, the subcircuit to analyze, i.e., the target graph, is extracted from the circuit according to the fault specification file. Afterwards, for each fault location and fault mapping combination, a separate process is started. In these processes, two copies of the target graph are created, i.e., the faulty and non-faulty target graph. SYNFI induces faults according to the fault mapping (number of simultaneous faults, location, and mapping) into the faulty target graph by replacing the boolean functions of the target gates according to the mapping. By combining the faulty and non-faulty target and adding an input and output layer responsible for analyzing the effects of the induced faults, the differential graph is created. This differential graph is used by SYNFI to evaluate if a fault is effective, i.e., the fault manipulates the outputs of the faulty target graph and is not detected by the countermeasures. Although the detection of faults by the countermeasures is implementation specific, these countermeasures typically raise an error signal, which SYNFI uses to evaluate whether the fault was detected or not. Finally, the differential graph is converted to a boolean formula and a SAT solver utilizes this mathematical model representing the circuit to reason about the effectiveness of the induced faults. In the end, the framework provides a detailed report summarizing the outcome of the fault analysis.

9.1.2 Phase 0 - Cell Library & Netlist Converter

The first step the framework conducts is the transformation of the (i) standard cell library and the (ii) gate-level netlist. The goal of this transformation step is to support arbitrary netlists generated with different hardware design flows and standard cell libraries.

First (i), SYNFI converts the provided standard cell library from the liberty format to a Python library.
9.1. Design and Implementation

Listing 9.2: Cell library entry for an AOI21_X2 cell.

```
"Cell Library": {
    "AOI21_X2": {
        "input_pins": ["A1", "B1", "B2"],
        "output_pins": "ZN",
        "boolean_function": "ZN = !(A1 & (B1 | B2))"
    }
}
```

For this conversion, SYNFI opens the provided standard cell library and extracts the name, the boolean function, and the input and output pins of the cells, as shown in Listing 9.2. SYNFI supports all cells with a boolean function, including compound gates, such as AOI cells. Cells that are used due to their electrical rather than for their logical behavior, e.g., filler cells, are not handled by SYNFI as they are not used in the gate-level netlist.

Afterwards (ii), SYNFI transforms the unmodified netlist into a directed multigraph using a Python library [HSS08].

Listing 9.3: Graph representation of the netlist.

```
"Nodes": {
    "U1": { "type": "NAND2" },
    "U2": { "type": "AOI21" }
},
"Edges": {
    "1": {
        "out": { "node": "U1", "port": "ZN" },
        "in": { "node": "U2", "port": "A1" }
    }
}
```

In this graph, nodes represent ports, cells, and submodules and each of these nodes consists of a name and a type. The type, e.g., a NAND2 gate or a port, defines the behavior of the node and the corresponding boolean function is provided by the cell library. Similar to gates and ports, submodules are also represented as nodes and the corresponding boolean function needs to be provided by the user. These nodes are connected using edges, which store information about the input and output port. Listing 9.3 shows an example graph where the output port ZN of the gate U1 is connected with the input port A1 of the gate U2.

9.1.3 Phase 1 - Target Graph Extraction

The target graph extraction step consists of the (i) extraction and (ii) preprocessing phase.
Extraction

The goal of the target graph extraction (i) is to simplify the subsequent analysis phase by extracting the subcircuit the user wants to analyze with SYNFI from the overall circuit. The definition of the target graph is provided in the fault specification file. Here, the user needs to define input and output nodes and the corresponding input and expected output values in the fault specification file. These nodes can be any cell, port, or submodule in the circuit.

With this information, the SYNFI framework starts the automatic target graph extraction process. Here, the tool finds all paths, consisting of combinational and sequential logic, between the defined inputs and outputs. Due to this extraction step, some nodes, e.g., gates, are missing one or multiple inputs as the corresponding connecting gates are not part of the extracted circuit. For all of these missing inputs, SYNFI introduces auxiliary input nodes.

Preprocessing

The goal of the preprocessing phase (ii) is to remove any time-dependencies in the extracted target graph. This is necessary as the graph is converted into a time-independent mathematical model, i.e., a boolean equation, in the last step described in Section 9.1.6. In SYNFI, we automatically break time-dependencies by (a) replacing registers used in pipeline stages with pass-through elements and (b) by removing loops and replacing registers in iterative designs and state machines. These pass-through elements are time-independent, i.e., do not have a clock port, and map the input to the (negated) output. SYNFI automatically distinguishes between the two register types by checking whether the register is the start and end of a cycle, i.e., a register used in an iterative design. This preprocessing phase enables SYNFI to handle circuits that were not manually unrolled by the hardware designer. However, when aiming to analyze multiple loop iterations, e.g., multiple rounds in an iterative AES implementation, SYNFI needs to evaluate each round individually.

Preprocessing and Extraction Example

We illustrate the extraction (i) and preprocessing (ii) phase in the example circuit in Figure 9.2 and Figure 9.3. Figure 9.2 depicts a circuit consisting of three input ports ($In_1...In_3$), one output port ($Out_1$), two registers ($U_3,U_6$), and a set of combinational gates. For the target graph extraction step ➊, the user needs to provide input and output nodes and a corresponding circuit state, i.e., values for these nodes. In this example, we set the register $U_3 = 1$ (blue) as the input and the register $U_3 = 2$ and $Out_3 = 2$ (green) as the output in the fault specification file.

As the circuit contains a register used in a pipeline stage ($U_6$) and a register used in a sequential loop ($U_3$), SYNFI removes these time-dependencies in the graph. The registers are, as shown in Figure 9.3, replaced ➋ with pass-through elements and the loop between $U_3$ and $U_1$ is removed ➌. Then, SYNFI finds all paths between the input node ($U_3$) and the output nodes ($U_3,Out_1$), i.e., all
nodes except $U_2$, $In_1$, $In_2$, and $In_3$. To avoid that certain gates have unconnected inputs, i.e., $U_1$, the frameworks adds auxiliary input nodes $\oplus$ and connects them with the corresponding nodes. Finally, the framework adds input and output nodes $\oplus$ for the user-defined input and output values.

### 9.1.4 Phase 1 - Fault Injection

After transforming the netlist into a graph and extracting the target graph, the injection phase starts. For each fault combination, i.e., number of simultaneous faults, fault locations, and fault mappings, defined in the fault model, SYNFI starts a new process. Inside these processes, SYNFI creates two copies of the extracted target graph - the faulty and non-faulty target graph. While the non-faulty target graph is used as a reference circuit in the subsequent steps, SYNFI induces faults into the faulty target graph. Here, the framework replaces the boolean function at a fault location according to the fault mapping.
To limit the configuration effort, SYNFI already provides a default fault mapping for a large set of gates. Furthermore, as the fault location is an optional parameter allowing the security engineer to attack specific gates, SYNFI supports an exhaustive injection approach automatically targeting all available gates in the target graph. Hence, at a minimum, the user only needs to specify the number of simultaneous faults injected into the gate-level netlist in the fault model.

9.1.5 Phase 1 - Differential Graph Creation

For each fault combination process, SYNFI creates a differential graph consisting of a faulty and non-faulty target graph. These differential graphs are responsible for evaluating the impact of faults on the circuit. As depicted in Figure 9.4, the differential graph consists of the faulty and non-faulty target graph. To this differential graph, we add an input layer and an output layer. In the input layer, we assign the input nodes added in the extraction phase the values provided by the user in the fault specification. As the user does not need to provide all possible input values of the analyzed circuit, SYNFI automatically connects the non-defined inputs of the faulty target graph with the non-defined inputs of the non-faulty target graph. The SAT solver, which is used in the evaluation step and described in Section 9.1.6, then automatically assigns values to these non-defined inputs. The output layer is used to analyze the effect of a fault. This layer consists of a logic comparing the output values produced by the faulty and non-faulty target graph with the output values provided by the user in the fault specification. Depending on the attack objectives and the implemented countermeasures, SYNFI allows the hardware designer to define two different types of effective faults, which are defined by the impact of the fault to the output. The detection of these two different effective fault types is implemented in the output layer.

Unspecific Fault Effects

This type of effective fault enables SYNFI to generically reveal whether a fault influences the input-output relation of a circuit. For circuits without counter-
9.1. Design and Implementation

measures, SYNFI defines a fault to be effective, if this fault manipulates one or multiple output bits of the analyzed subcircuit, i.e., the non-faulty and faulty target graph produce different output values.

\[ \text{Output Logic} = (O_{NF} = O_E) \land (O_F \neq O_E) \]  \hspace{1cm} (9.1)

Equation (9.1) depicts the logic in the output layer used to detect this type of fault effect. The first part of the equation ensures that the output \( O_{NF} \) of the non-faulty (\( NF \)) graph produces the expected output value \( O_E \) defined in the fault specification. More specifically, this part of the equation ensures that the SAT solver only assigns values to the non-defined inputs (cf. Section 9.1.5) of the differential graph which generate the expected output circuit state \( O_E \). The second part of the equation is responsible for ensuring that the output \( O_F \) of the faulty (\( F \)) circuit does not match the expected output value. If the output logic produces a logical 1, an effective fault is found.

\[ \text{Output Logic} = ((O_{NF} = O_E) \land (O_{NFA} = 0)) \land ((O_F = O_E) \land (O_{FA} = 0)) \]  \hspace{1cm} (9.2)

For circuits with dedicated fault countermeasures, SYNFI considers a fault to be effective, if this fault manipulates one or multiple output bits of the analyzed subcircuit and the alert signal of the countermeasure was not triggered. If the alert signal was triggered, the countermeasure works as intended and the fault is considered to be ineffective. The output logic in Equation (9.2) models this behavior by ensuring that the alert signal \( O_{FA} \) was not triggered in both parts of the formula.

Specific Fault Effects

This type of fault effect allows SYNFI to check whether a fault enables the adversary to enter a specific circuit state. Here, SYNFI considers a fault to be effective, if the output of the faulty target graph matches the expected output value defined in the fault specification.

\[ \text{Output Logic} = (O_{NF} = O_E) \land (O_{FA} = O_{EF}) \]  \hspace{1cm} (9.3)

Equation (9.3) shows the logic in the output layer capable of detecting this fault effect type. Here, the first part of the equation ensures that the outputs \( O_{NF} \) produced by the non-faulty target graph match the expected output values \( O_E \) provided in the fault specification. The second part of the equation ensures that the outputs of the faulty graph match the expected fault output value \( O_{EF} \) specified in the fault specification.

\[ \text{Output Logic} = ((O_{NF} = O_E) \land (O_{NFA} = O_{EA})) \land ((O_F = O_{EF}) \land (O_{FA} = O_{EA})) \]  \hspace{1cm} (9.4)

For circuits consisting of a fault countermeasure designed to detect a fault, the alert signal \( O_{EA} \) is also incorporated in the output logic, as shown in
Equation (9.4). Here, the output layer ensures that the non-faulty circuit produces the expected output values \( O_{NF} = O_E \) and that the alert was not triggered in the reference circuit, i.e., \( O_{NFA} = O_{EA} \). For the faulty target graph, the equation ensures that the output value matches the expected fault output value \( O_F = O_{EF} \), and the alert was not triggered.

9.1.6 Phase 1 - Transformation & Evaluation

After creating the differential graph, the SYNFI framework converts this graph into a mathematical model. As each node is assigned a boolean function, the tool uses the Tseitin transformation [Tse83] to automatically transform the differential graph into a boolean formula in Conjunctive Normal Form (CNF). The extracted boolean formula then is handed over to a SAT solver for the evaluation. As shown in the differential graph in Figure 9.4, the inputs of the boolean formula are either set in the input layer to values provided by the user in the fault specification or are left unconnected. For these unconnected input values, which are shared by the faulty and non-fault target graph, the SAT solver can set these values freely as long as the reference circuit produces the expected output values. This is ensured by the output layer of the differential graph (cf. Section 9.1.5). If the logic in the output layer produces a logical 1, an effective fault is found. For the report, the framework collects the number of effective faults, their location, and fault mapping.

Selection of the SAT Solver

For our Python-based tool, we use the PySAT [IMM18] framework as an interface to the SAT solver. To determine the fastest solver for our purpose, we executed several fault injection verification experiments with the provided solvers [Bie+20; ES03; Bie17; AS18; LM; Lia+18] as a custom benchmark and decided to use MiniSAT22 [ES03] in the end.

9.1.7 SYNFI Guarantees

SYNFI provides hard security guarantees for a specific fault experiment conducted on the analyzed circuit. This fault experiment is defined by the security engineer analyzing the circuit in the fault specification and is in line with the threat model of the design. The fault specification consists of the (i) definition of the fault model and the (ii) description of the target circuit.

In the fault model (i), the SYNFI user defines the fault capabilities of the attacker, which are specified in the threat model of the analyzed circuit. This definition comprises the number of faults the attacker can simultaneously inject into the circuit, the effects, and the locations of the faults. For the fault locations, the security engineer can either target specific gates or instrument SYNFI to exhaustively inject faults into all gates of the circuit. SYNFI injects a fault into these targeted gates by replacing the boolean function of the gate according to
the fault mapping specified in the fault model. Here, SYNFI supports transient or permanent fault effects.

The target circuit (ii) is the subpart of the overall circuit containing the security-critical logic and the corresponding fault countermeasure the security engineer aims to analyze with SYNFI. This circuit is defined in the fault specification by providing the names of input and output ports of a module or certain gates. SYNFI then automatically extracts the target circuit between these inputs and outputs. In addition to the names of these ports or gates, the SYNFI user needs to specify a specific circuit state, i.e., values for the inputs and outputs.

Depending on the configuration, SYNFI can reveal whether a fault has (a) an unspecific (cf. Section 9.1.5) or (b) a specific (cf. Section 9.1.5) effect. More concretely, SYNFI can formally verify (a) whether or not any fault specified in the fault model can change the input-output relation of the target circuit without triggering the fault countermeasures. Additionally, SYNFI can formally show (b) whether or not it is possible to enter a specific circuit state from a given circuit state without triggering the countermeasures using a fault.

Note that SYNFI is designed to detect faults manipulating the input-output relation of the analyzed circuit. Hence, classes of fault attacks not impacting this relation, e.g., safe error [YJ00] or ineffective attacks [Dob+18b], are not in the scope of SYNFI.

When the input circuit state space is small, e.g., a counter logic, multiple fault experiments for each possible circuit state can be conducted. Then, SYNFI provides comprehensive security guarantees for the analyzed circuit. For larger circuit state spaces, the security engineer needs to focus on verifying specific states which are particularly security-sensitive or are a representative of the possible states.

False-positive Results. SYNFI can produce false-positive results when the target circuit is too loosely specified. As described in Section 9.1.5, the SYNFI user does not need to provide the entire input circuit state in the fault specification. The non-defined inputs provide more freedom to the SAT solver and the solver can freely set these inputs as long as the non-faulty target graph produces the specified output circuit state.

However, in some circuits, the SAT solver could find a circuit state which cannot occur during normal operation. Then, a false-positive result is returned, requiring a manual inspection of unexpected effective faults.

Note that the approach of SYNFI is to consider states that occur during normal operation and analyze how a fault changes the behavior. Using a faulty starting state means analyzing how a fault can change a faulty starting state. This is a fault that is beyond the defined fault model and actually corresponds to a stronger fault model. When a false-positive like this occurs, the security engineer can simply manually exclude it or constrain SYNFI more tightly to avoid the need for manual inspection. In some cases, the false positive may also provide a hint to the security engineer about faults that can occur with stronger fault models and this can be an input for an extended analysis with a stronger
fault model.

Overall, in our analysis, fault positives have not turned out to be a severe limitation as effective faults have occurred rarely in fault-hardened circuits and it was possible to handle them by constraining SYNFI more tightly to specific circuit states.

False-negative Results. SYNFI cannot produce false-negative results within the bounds of the fault specification. The security engineer only needs to ensure that the fault specification matches the threat model of the analyzed circuit. For example, when the threat model considers an attacker capable of injecting faults with permanent or transient effects, the fault model also needs to model these faults in the fault mappings.

9.2 Analysis of OpenTitan

The OpenTitan chip will be deployed in hostile environments allowing an adversary to gain physical access and attempt to inject faults into the device to break its security. Therefore, in this chapter, we utilize SYNFI to actively contribute to the security of the OpenTitan chip before the tape-out by performing a pre-silicon fault analysis. As analyzing the entire chip consisting of a wide variety of IP blocks is far beyond the scope of this chapter and not all modules actually need to provide fault-resiliency, we selected, together with the OpenTitan project team, the most security-critical modules for our analysis. In particular, we focused on analyzing (i) the unprotected AES module and (ii) the protected life cycle controller, the lockstep mode of the CPU, and generic, fault-hardened building blocks. We utilized SYNFI to (FE) reveal the faults’ effect to an unprotected module, to (FD) check whether faults can be detected by the countermeasures, and to (FS) verify that faults cannot enable an adversary to enter a specific state without triggering the countermeasures. For all experiments, we injected up to a certain number of simultaneous faults specified in the threat model of each module into the circuit. Our analysis is conducted on the unmodified netlist synthesized with the internal OpenTitan hardware design flow consisting of the Synopsys DC synthesis tool and a proprietary standard cell library.

Results. With SYNFI, we revealed that the (i) AES module is susceptible to single faults enabling an adversary to perform attacks on a round-reduced AES, extract temporary encryption results over the software interface, or hijack the execution-flow of the AES Finite-State Machine (FSM). For the other analyzed modules (ii), our analysis showed that they provide adequate protection, i.e., all modules can withstand or detect at least single fault attacks.

9.2.1 AES

The AES module of OpenTitan is a hardware accelerator providing a secure encryption and decryption mechanism for protocols used by the chip. As this IP block is one of the most crucial elements of the RoT element, we analyze in detail
Table 9.1: Verification results for the AES round counter performed on a 16-core machine.

<table>
<thead>
<tr>
<th>Target Setting</th>
<th>Effective [%]</th>
<th>Total [#]</th>
<th>Execution [s]</th>
<th>Circuit [GE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>① Unprotected Round Counter FE 1 55.56 18 4.4 20.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>② Unprotected Round Counter FS 2 2.65 302 4.47 20.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>③ Protected Round Counter FD 2 0.13 34,652 366.81 156</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the behavior of the most security-critical parts of the module when influenced by faults. In comparison to related work (cf. Section 9.3), we focus on assessing generic hardware primitives, such as state machines and counters, instead of performing specific cryptographic data-flow attacks, such as SIFA [Dob+18b] or DFA [PQ03].

**Results.** Our analysis revealed several fault attack vectors for the unprotected AES module. In particular, SYNFI showed that single faults into the AES round counter, handshake signals, and certain FSMs could enable an adversary to break the security of the module. Based on these verification results, we developed several fault hardening techniques, reassessed their security, and contributed them to the OpenTitan project.

**AES Round Counter**

The AES [DR99] block cipher performs, depending on the mode of operation, a certain number of encryption rounds. This round counter, which is generated in an FSM, is security-critical, as a fault hijacking the counter value could weaken the cryptographic strength of the AES [Bir04].

**Unprotected round counter.** To analyze the resilience of the round counter against faults, we first ① utilize SYNFI to reveal if the round counter circuit is generally susceptible to faults, i.e., it is possible to arbitrarily manipulate the counter value. Then ②, we determine how many simultaneous faults are required to manipulate the counter to a specific value.

**Listing 9.4:** Fault specification for the round counter.

```plaintext
1 "Fault Specification":
2 "Target Circuit":
3   "inputs": ["rnd_ctr_q": "4'b0001"],
4   "outputs": ["rnd_ctr_d": "4'b0010"]
5 "Fault Model":
6   "Sim. Faults": 1 or 2,
7   "Fault Locations": ["*"]
```

To conduct this analysis, we describe the circuit of interest and the fault model in the fault specification file as shown in Listing 9.4. We configure SYNFI
to analyze the logic in between the `rnd_ctr` register responsible for incrementing the value and set the input value of the counter circuit to 1 and the expected output value to 2. For the fault model, we instrument SYNFI to exhaustively induce one or two simultaneous faults into all available gates of the circuit.

Table 9.1 shows the evaluation report generated by SYNFI. The setting column in the table specifies how SYNFI considers a fault to be effective. In the (FE) mode, any fault having an arbitrary effect to the input-output relation of the circuit is considered to be an effective fault. In (FD), an effective fault is a fault manipulating the circuit’s output and the fault countermeasures did not detect, i.e., did not trigger the alert signal, this fault. Finally, (FS) refers to a fault changing the output of the circuit to a specific state without triggering the countermeasures. Moreover, the table shows the total number of injected faults and the percentage of the effective faults. The effective fault percentage number indicates how many of the total number of injected faults SYNFI considers to be effective. Finally, the table highlights the execution time of SYNFI and the circuit size in Gate Equivalent (GE). Note that the circuit size refers to the fault affected target circuit extracted by the SYNFI framework, which is a subcircuit of the whole circuit.

As shown in the first row $\textcircled{1}$, a single fault into the circuit enables a fault attacker to manipulate the round counter value. To manipulate the round counter to a specific value, SYNFI reveals in the second row $\textcircled{2}$ that an adversary needs to induce at least two simultaneous faults.

**Hardened round counter.** To enhance the resilience of the counter against faults, we extend the FSM to generate an up counting (the round counter) and a redundant down counting counter value. We redundantly instantiate this FSM, combine the generated counters, and add an error logic capable of detecting an ongoing fault attack.
9.2. Analysis of OpenTitan

Listing 9.5: Round counter protection in the aes_cipher_control module.

```verilog
// Instantiate redundant FSMs.
for (genvar i = 0; i < 3; i++) begin : gen_fsm
    aes_cipher_control_fsm u_aes_cipher_control_fsm_i (  
        .rnd_ctr_q_i ( rnd_ctr_q ),  
        .rnd_ctr_d_o ( mr_rnd_ctr_d[i] ),  
        .rnd_ctr_rem_q_i ( rnd_ctr_rem_q ),  
        .rnd_ctr_rem_d_o ( mr_rnd_ctr_rem_d[i] ),  
        ...);
end
// Combine counter signals.
always_comb begin : combine_counter_signals
    for (int i = 0; i < 3; i++) begin
        rnd_ctr_d |= mr_rnd_ctr_d[i];  
        rnd_ctr_rem_d |= mr_rnd_ctr_rem_d[i];
    end
end  
// Generate sum.
assign rnd_ctr_sum = rnd_ctr_q + rnd_ctr_rem_q;  
assign rnd_ctr_err = (rnd_ctr_sum != num_rounds_q) ? 1'b1 : 1'b0;
```

To ensure that the synthesis tool does not weaken the redundancy-based protection mechanism shown in Listing 9.5, we reassess its security using SYNFI. In particular, we utilize the framework to evaluate whether the circuit is capable of detecting a single fault arbitrarily manipulating the counter value (FD).
Listing 9.6: Fault specification file for the aes_cipher_control_fsm round counter experiment.

```json
{
    "fimodels": {
        "aes_cipher_control_fsm_rnd_cntr": {
            "simultaneous_faults": 1,
            "stages": {
                "stage_cntr": {
                    "inputs": [ "rnd_ctr_q_i" ],
                    "outputs": [ "rnd_ctr_d_o" ],
                    "type": "inout"
                }
            },
            "input_values": {
                "rnd_ctr_q_i": {
                    "i": {
                        "0": 1, "1": 0, "2": 0, "3": 0
                    }
                }
            },
            "output_values": {
                "rnd_ctr_d_o": {
                    "o": {
                        "0": 0, "1": 1, "2": 0, "3": 0
                    }
                }
            },
            "output_fault_values": {
                "rnd_ctr_d_o": {
                    "o": {
                        "0": 0, "1": 0, "2": 1, "3": 0
                    }
                }
            },
            "alert_values": { },
            "node_fault_mapping": {
                "NAND": [ "AND" ]
            },
            "fault_locations": {
                "Gate_189": [ "stage_cntr" ]
            }
        }
    }
}
```
Listing 9.6 shows the fault specification used for the verification of the hardened cipher control round counter FSM. The `simultaneous_faults` parameter, which also can be overwritten by a command line argument, defines the number of simultaneous faults injected into the extracted circuit. To specify the extracted target circuit, the input and output elements need to be defined using the `stages` parameter. These elements can be any gate, register, or input and output port of the circuit. For the example in Listing 9.6, the circuit between the `rnd_ctr` register with the input \( Q \) and the output port \( D \) is defined as the circuit of interest. The SYNFI tool then uses this information to automatically extract the target circuit by finding all paths between the defined input and output element. This circuit can consist of combinational and sequential logic. If multiple stages are provided, SYNFI automatically connects them. To constrain the SAT solver, the user needs to provide input values with the `input_values` parameter. In order to allow the output layer to evaluate the effect of a fault, the fault model also provides information about the expected, expected fault output value, and the alert value. The `node_fault_mapping` parameter defines the mapping function of a target gate. During the fault injection process, the boolean function of the target gate is replaced according to this mapping. The target gate can be defined using the `fault_locations` entry. If the fault evaluator does not have an intuition about the critical gates which need to be analyzed, the SYNFI tool is also capable of exhaustively targeting all gates in the extracted circuit.

SYNFI could formally verify that, in a specific circuit state, a single fault cannot manipulate the counter value without triggering the alert signal. This specific circuit state comprises a fixed counter input value of 1 and a counter output value of 2, all other non-defined inputs of the circuit are automatically set by the SAT solver SYNFI internally uses. We argue that testing a single circuit state, i.e., an input-output pair, is sufficient to verify that the tooling of the design flow does not remove the redundancy-based countermeasures. For two simultaneous faults, SYNFI reveals in Row ③ in Table 9.1, that at least one fault into the error logic and one fault into the input or output shared round counter register are required to tamper the counter value without raising the alert.

**AES Handshake Signals**

Internally, the AES IP consists of a variety of handshake signals responsible for influencing the data- and control-flow of the encryption. As manipulating the `out_valid_o` signal would allow an adversary to leak temporary encryption data to the software interface of the AES, we exemplarily focus on analyzing this signal. More specifically, we instrument SYNFI to show whether it is possible to manipulate this signal to a specific value \( \text{FS} \), i.e., from a logical 0 to a logical 1. Here, we configure SYNFI to inject a single fault into the FSM circuit responsible for driving this signal. The verification result in Row ⑨ in Table 9.2 shows that already a single fault induced into the circuit enables an adversary to tamper the handshake signal.
Table 9.2: Verification results for the AES handshake signal on a 16-core machine.

<table>
<thead>
<tr>
<th>Target Setting</th>
<th>Simult. Faults</th>
<th>Effective [%]</th>
<th>Total [#]</th>
<th>Execution [s]</th>
<th>Circuit [GE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>① Unprotected Handshake Signal</td>
<td>FS</td>
<td>1</td>
<td>83.83</td>
<td>31</td>
<td>4.49</td>
</tr>
<tr>
<td>② Protected Handshake Signal</td>
<td>FS</td>
<td>3</td>
<td>15.32</td>
<td>8436</td>
<td>270.68</td>
</tr>
</tbody>
</table>

Listing 9.7: `out_valid_o` signal generation in the `aes_cipher_control_fsm` module.

```vhdl
module aes_cipher_control_fsm (
    output logic out_valid_o,
    input logic [3:0] rnd_ctr_q_i,
    ...
);

assign num_rounds_regular = num_rounds_q_i - 4'd1;
unique case (aes_cipher_ctrl_cs)
    ROUND: begin
        advance = (dec_key_gen_q_i | sub_bytes_out_req_i) &
            key_expand_out_req_i;
        if (advance) begin
            if (advance) begin
                out_valid_o = 1'b1;
                ...
            if (rnd_ctr_q_i == num_rounds_regular) begin
                if (dec_key_gen_q_i) begin
                    out_valid_o = 1'b1;
                    ...
        ...
```

The detailed verification summary reporting the fault-affected cells shows that the adversary can induce faults either (i) directly into the `out_valid_o` signal (Line 14 in Listing 9.7), the (ii) comparisons in the output logic (Line 12 in Listing 9.7), or the (iii) control signals (Line 13 in Listing 9.7) of the FSM. Hence, to comprehensively protect the handshake signal, we must consider all three attack vectors.

**Multi-bit encoding.** To protect critical handshake signals (i), we extend the AES IP to adopt the multi-bit encoding the OpenTitan project uses in other hardware modules.

Listing 9.8: Encoded multi-bit signals.

```vhdl
typedef enum logic [2:0] { SP2V_HIGH = 3'b011,
    SP2V_LOW = 3'b100 } sp2v_e;
```

In the multi-bit encoding approach shown in Listing 9.8, a 1-bit signal is encoded into a 3-bit signal resulting in a Hamming distance of three. Here, the
first two bits represent the logical value and the Most Significant Bit (MSB) is the inverse of the value to encode.

To verify that the synthesis step does not weaken the security guarantees of multi-bit signals by simplifying the encoding in the optimization phase, we use the SYNFI framework to inject faults into the encoded out_valid_o signal. In particular, we use SYNFI to reveal if it is possible to manipulate the encoded signal to a specific value (FS), i.e., from SP2V_LOW to SP2V_HIGH. The verification result in Row ② in Table 9.2 confirms the expected security bound of three, i.e., SYNFI could not find an effective fault when inducing one or two simultaneous faults.

**Multi-rail FSM.** As a single fault into the output logic (ii) of the FSM, e.g., the comparison in Line 12 in Listing 9.7, is enough to tamper the out valid signal, we design and deploy a redundant multi-rail FSM scheme.

![Multi-rail FSM approach](image)

**Figure 9.5:** Multi-rail FSM approach.

The multi-rail scheme, as shown in Figure 9.5, instantiates the unmodified FSM in a triple modular redundancy mode. Unencoded m-bit signals are independently processed by the three state machines and an output logic is responsible for combining and checking the resulting signals. On a comparison mismatch, an alert signal is triggered. For multi-bit encoded signals (cf. Listing 9.8), the first two bits are processed by the positive FSM\(_P0\) and FSM\(_P1\) rail. The inverted third bit is processed by the negative FSM\(_N0\) rail. Combining these signals at the output again produces an encoded multi-bit signal. If a fault in one or two FSMs modifies the signal, an invalid code word is produced, which is detectable by a checker unit.

```vhdl
assign sp_dec_key_gen_q = {dec_key_gen_q}
// For every bit in the Sp2V signals, one separate rail is instantiated.
for (genvar i = 0; i < 3; i++) begin : gen_fsm
  if (SP2V_LOGIC_HIGH[i] == 1'b1) begin : gen_fsm_p
    aes_cipher_control_fsm_p u_aes_cipher_control_fsm_i (
      .rnd_ctr_q_i ( rnd_ctr_q ),
      .num_rounds_q_i ( num_rounds_q ),
      .dec_key_gen_q_i ( sp_dec_key_gen_q[i] ),
      .out_valid_o ( sp_out_valid[i] )
    );
  end else begin: gen_fsm_n
    aes_cipher_control_fsm_n u_aes_cipher_control_fsm_i ( 
      .rnd_ctr_q_i ( rnd_ctr_q ),
      .num_rounds_q_i ( num_rounds_q ),
      .dec_key_gen_q_i ( sp_dec_key_gen_q[i] ),
      .out_valid_o ( sp_out_valid[i] )
    );
  end
// Convert sparsified outputs to sp2v_e type.
assign out_valid_o = sp2v_e'(sp_out_valid);
```

Listing 9.9 shows the multi-rail FSM approach integrated into the AES module. In this approach, three redundant FSMs are instantiated where the two `aes_cipher_control_fsm_p` FSMs produce a positive output and the `aes_cipher_control_fsm_n` FSM a negated output. Combined, they form a multi-bit signal with a Hamming distance of three. As the multi-rail approach requires to instantiate FSMs redundantly, the area increases from 211.15GE for the `aes_cipher_control_fsm_p` FSM to 908.81GE for the whole `aes_cipher_control` module including the redundant FSMs, the combination logic, as well as other countermeasures, such as the counter error logic introduced in Section 9.2.1.

To verify that the synthesis tool does not remove the redundant FSMs, we utilize SYNFI to analyze the resilience of the multi-rail approach against faults. In particular, we instrument the framework to reveal whether there exists a fault enabling an adversary to manipulate the `out_valid_o` signal to a specific value (FS), i.e., from `SP2V_LOW` to `SP2V_HIGH`, without triggering the fault countermeasure.
9.2. Analysis of OpenTitan

Table 9.3: Verification results for the AES multi-rail FSM on a 16- or 72-core* machine.

<table>
<thead>
<tr>
<th>Target Setting Simult.</th>
<th>Effective % Total #</th>
<th>Execution Circuit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-Rail loose config</td>
<td>FS 1 2.46 122</td>
<td>8.2 s 96.5</td>
</tr>
<tr>
<td>Multi-Rail tight config</td>
<td>FS 1 0 573</td>
<td>266.87 s 355.75</td>
</tr>
<tr>
<td>Multi-Rail tight config</td>
<td>FS 2 0 170,982</td>
<td>1.01 h 355.75</td>
</tr>
<tr>
<td>Multi-Rail tight config</td>
<td>FS 3 0.02 35,222,293</td>
<td>38.27 h 355.75</td>
</tr>
</tbody>
</table>

Listing 9.10: Fault specification for the multi-rail FSM.

```
"Fault Specification":
  "Target Circuit":
    "inputs": ["rnd_ctr": "2"],
    "outputs": ["out_valid_o": "SP2V_LOW"],
    "expected fault outputs":
      ["out_valid_o": "SP2V_HIGH"],
    "alerts": ["rnd_ctr_err": "0"]
```

Using the fault specification file shown in Listing 9.10, SYNFI automatically extracts the circuit between the round counter register and the out valid signal, i.e., the multi-rail FSM including the input, output, and error logic. By setting the counter value to 2, we force the circuit in a state where the out valid signal is set to a logical 0 in the fault-free setting. As shown in Row ① in Table 9.3, in this SYNFI configuration, already a single fault can be sufficient to produce an effective fault, i.e., the out valid signal is set to a logical 1 and the error was not triggered. The framework reports that all of these four effective faults occur when faulting the num_rounds register value. Since the value of this register is used for the comparison in Line 12 in Listing 9.7 in all redundant FSMs, the out valid signal is set to a logical 1. Nevertheless, this verification result can only provide a limited statement about the security of the multi-rail approach as the SYNFI framework was minimally constrained. By only defining the input value of the round counter register, the SAT solver is loosely constrained (cf. Section 9.1.7) and automatically sets the dec_key_gen_q_i and advance signals in the boolean formula of the differential graph to a logical 1. Setting these variables is possible, as in the non-faulty reference circuit the out_valid_o always stays at SP2V_LOW when the round counter value is 2. Hence, as all redundant FSMs set the out_valid_o to the same value, the error signal is not set.

To avoid these false-positive results, we more tightly configure SYNFI by further defining the inputs dec_key_gen_q_i = 0 and keyexpand_out_req_i =
150 Chapter 9. Pre-Silicon Fault Countermeasure Analysis

In the fault specification file. In this configuration, the extracted circuit SYNFI analyzes increases from 96.5 GE to 355.75 GE, as the tool finds more paths from the defined inputs to the outputs. Now, as expected and depicted in Row ② and ③ in Table 9.3, one or two simultaneous faults cannot manipulate the output valid signal. Starting with three simultaneous faults into the circuit, we observe effective faults (cf. Row ④ in Table 9.3). These effective faults manipulating the output valid signal are caused by inducing bit-flips into variables used by the redundant FSMs. To demonstrate the possibility of scaling SYNFI to the cloud and as the number of possible fault combinations for three simultaneous faults, i.e., fault location and fault mapping, explodes, we conducted this experiment on a 72-core server. We measured a total run time of 38.27 h and a maximum memory consumption of less than 8 GB for injecting 35,222,292 faults into the circuit of a size of 355.75 GE.

Shadow registers. As discussed in the initial experiment in Section 9.2.1, handshake signals also can be tampered by faulting (iii) control signals used by the FSM. To protect security-critical control signals, which are provided by the software over a register interface, the AES modules stores them in dedicated shadow registers. These registers constantly compare the two values and on a comparison mismatch caused by, for example a fault, an alert is raised forcing the AES module in a terminal state.

Sparsely Encoded State Machines

Finite-state machines are, as seen in Section 9.2.1, security-critical hardware elements as they are responsible for setting control signals used by the data path.

```plaintext
Listing 9.11: Finite-state machine with a state encoding vulnerable to faults.
1 typedef enum logic [3:0] {
2   IDLE, INIT, ROUND, ..., ERROR
3 } aes_cipher_ctrl_e;
4
5 aes_cipher_ctrl_e aes_cipher_ctrl_ns, aes_cipher_ctrl_cs;
6
7 always_comb begin : aes_cipher_ctrl_fsm
8   unique case (aes_cipher_ctrl_cs)
9     IDLE: begin
10     if <condition>:
11       aes_cipher_ctrl_ns = INIT;
12     end
13   end
14   INIT: begin
15   ...
```

In a state machine, the next-state logic derives the next state from the current state and a set of inputs. As seen in Listing 9.11, the state variable stored in the state register is typically represented as a simple enum. However, as the minimum Hamming distance between two states is 1, a single fault into the state
9.2. Analysis of OpenTitan

Table 9.4: Verification results for the AES FSM state encoding on a 16-core machine.

<table>
<thead>
<tr>
<th>Target Setting</th>
<th>Simult. Faults</th>
<th>Effective [%]</th>
<th>Total [#]</th>
<th>Execution [s]</th>
<th>Circuit [GE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoded FSM states</td>
<td>FS</td>
<td>3</td>
<td>15</td>
<td>29</td>
<td>16.09</td>
</tr>
</tbody>
</table>

 registers would allow an adversary to hijack the control-flow of the FSM, \( i.e., \), skip or enter a normally non-reachable state.

In order to mitigate this threat, we deploy the sparse FSM state encoding technique used by different OpenTitan modules into the AES.

Listing 9.12: Sparsely encoded FSM state.

```c
typedef enum logic [5:0] {
    IDLE = 6'b001001,
    INIT = 6'b100011,
    ROUND = 6'b111101,
    FINISH = 6'b010000,
    PRNG_RESEED = 6'b100100,
    CLEAR_S = 6'b111010,
    CLEAR_KD = 6'b001110,
    ERROR = 6'b010111
} aes_cipher_ctrl_e;
```

The encoding, which is shown in Listing 9.12, assures a minimum Hamming distance between the states of 3, increasing the resistance against faults. Additionally, we introduce a default error state, which is entered when the state value does not match the `aes_cipher_ctrl_e` enum. Now, if a fault flips bits in the state variable, with a high probability, the terminal error state is entered.

To verify that an aggressive synthesis setting does not reduce the security by altering the state encoding, we utilize SYNFI to analyze the `aes_cipher_control_fsm` FSM. In particular, we determine, how many faults are required to hijack the control-flow of the FSM by skipping a certain state and directly enter a normally not reachable state, \( i.e., \), (FS). For this experiment, we instrument SYNFI to analyze the next-state logic and to inject faults directly into the state registers. SYNFI shows that one or two simultaneous bit-flips into the state registers triggers the alert signal, \( i.e., \), the FSM enters the error state. When inducing three simultaneous faults, as shown in Table 9.4, the attacker is able to redirect the control-flow of the FSM.

**FSM optimizations.** Several synthesis tools also apply optimization passes to state machines. Yosys, for example, removes unused control signals, merges states, and recodes the FSM state variables stored in the state registers [Wola]. To analyze the impact of these optimization on the security of the sparsely encoded states, we synthesize the `aes_cipher_control_fsm` module with Yosys using an aggressive optimization strategy. Similar to the previous experiment, we configure SYNFI to skip a state and directly enter a typically non-reachable state. Our result shows that, in comparison to the verification in Table 9.4, now 2 instead of 3 simultaneous faults are already sufficient to skip the FSM state,
### Table 9.5: Verification results for entering the RMA state on a 16-core machine.

<table>
<thead>
<tr>
<th>Target Setting</th>
<th>Simult. Faults</th>
<th>Effective [%]</th>
<th>Total [#]</th>
<th>Execution [s]</th>
<th>Circuit [GE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token comparison in TokenHashSt Token FS 1 0 313 32.56 215</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Token comparison in TokenCheck0St FS 1 0 349 35.87 248.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Token comparison in TokenCheck1St FS 1 0 310 28.26 204.75</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Skip token check states FS 7 100 7 17.44 214.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[1\text{e.},\] the FSM optimization weakens the encoding. To prevent these optimizations, Yosys can be parameterized with the `nofsm` flag. In summary, this experiment shows that synthesis optimizations configured by different stakeholders, e.g., trying to minimize the area of the design, could have fatal security implications.

#### 9.2.2 Life Cycle Controller

OpenTitan can be transferred into different operational states depending on where the device is deployed, \(i.e.,\) at the customer or the manufacturer. The state switching mechanism is implemented in hardware in the life cycle controller module. As certain states, e.g., the Return Material Authorization (RMA) state, enable security-sensitive features, such as access to the debug port, the life cycle controller is hardened against fault attacks.

**Results.** SYNFI verified that the analyzed fault-hardened primitives of the life cycle controller provide adequate protection. Specifically, we could confirm that the countermeasures prevent the exploitation of single or double faults for all critical attack vectors.

### Entering the RMA State

The core mechanism of the life cycle controller IP is an FSM responsible for determining the life cycle state of OpenTitan. To hinder an adversary from entering the security-sensitive RMA state, which is used to debug the RoT chip when returned to the manufacturer, this state transition is only permitted when possessing a 128-bit unlock token. Internally, the state machine checks the validity of the token in three different FSM states. This redundancy-based mechanism and the state encoding technique guaranteeing a minimum Hamming distance of 7 between the state symbols form the fault protection strategy of the controller.

Based on this description of the fault-hardening mechanisms, we identified two major attack vectors for a fault attacker: \(i\) glitch the token comparisons three times or \(ii\) hijack the execution of the FSM by glitching the state symbols.

**Glitching the comparisons.** Glitching the three token checks requires a strong
9.2. Analysis of OpenTitan

adversary capable of injecting three faults in three clock cycles. We utilize SYNFI to test whether these three comparisons are susceptible to a single fault each.

For this verification, we configure SYNFI to analyze if it is possible to induce a fault into the next-state logic of the FSM changing the next valid state (FS). In the fault specification file, we instrument the tool to exhaustively induce a single fault into all gates of the next-state logic for each of the three fault experiments. Row ①-③ in Table 9.5 shows that SYNFI could not find a single fault allowing the attacker to enter the next state (from TokenHashSt to TokenCheck0St, ...) without possessing the required token.

Skip the token check states. In order to verify that the synthesis step does not weaken the 16-bit FSM state encoding, we utilize SYNFI to check whether it is possible to induce faults into the state register allowing the attacker to directly enter the RMA state (FS). Here, we configure SYNFI to analyze the FSM and to inject 1 to 7 simultaneous faults into the state register. As shown in Row ④ in Table 9.5, at least 7 simultaneous faults are required to enter the target state. This matches the security expectation, i.e., a Hamming distance of 7 for the state symbol encoding.

Flash Erase Mechanism

Before entering RMA, the life cycle controller erases the flash to hinder an adversary from accessing previously created data. Although entering RMA only is possible when knowing a secret, device dependent token, this hardware-backed flash erasing mechanism is meant to be a second line of defense. Internally, the flash erasing command is directly triggered in the FSM of the life cycle controller. To ensure that the flash was erased before entering RMA, the acknowledgment sent by the flash controller is checked three times in the FSM. If one of the acknowledgements was not received, e.g., due to a fault, the FSM remains in the current state.

Glitching the Encoded Flash Handshake Signal. An attacker with access to a valid RMA token aiming to bypass the flash erasing mechanisms needs to suppress the flash erasing command as well as the acknowledgement signal or the corresponding check three times. However, as the initiate and acknowledgement signal is encoded with a Hamming distance of 4, the adversary theoretically needs to flip 4-bits four times. To confirm this behavior, i.e., the synthesis tool did not tamper the encoding, we exemplary analyze the resilience of the flash erase initiate signal. In particular, we instrument SYNFI to reveal whether it is possible to induce faults manipulating this signal to a specific value (FS), i.e., from an encoded On = 4'b1001 to an encoded Off = 4'b0110. This analysis showed that already two simultaneous faults injected into the encoded signal allow an adversary to hijack the initiate signal. Since bit 0 and 3 as well as 1 and 2 in the encoding are always the same, the synthesis tool decided to merge these signals and only instantiate two instead of four registers driving the bits of the signal. Hence, the security of the encoding is reduced to a Hamming distance of 2.

In order to prevent that the four registers instantiated in the HDL code
Table 9.6: Verification results for glitching the locking mechanism performed on a 16- or 72-core* machine.

<table>
<thead>
<tr>
<th>Target Setting</th>
<th>Simult. Faults</th>
<th>Effective [%]</th>
<th>Total [#]</th>
<th>Execution Circuit [GE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prevent counter incr.</td>
<td>FS</td>
<td>1</td>
<td>17.81</td>
<td>853</td>
</tr>
<tr>
<td>Reset counter value*</td>
<td>FS</td>
<td>3</td>
<td>0</td>
<td>1,000,000</td>
</tr>
<tr>
<td>Skip CntProgSt</td>
<td>FS</td>
<td>7</td>
<td>100</td>
<td>7</td>
</tr>
</tbody>
</table>

are merged in the synthesized netlist, we augmented the design flow with the `set_dont_touch` parameter. Now, as shown in Row ④ in Table 9.5, the encoding works as expected and an attacker needs at least four simultaneous faults to tamper the encoded signal.

Locking Mechanism

OpenTitan limits the number of state transitions and transition attempts to 24. Once this number is reached, the life cycle controller rejects further attempts, effectively locking the device into its current state. In the life cycle hardware IP block, the counter increment is conducted in the `lc_ctrl_state_transition` module and the counter value is programmed into the One Time Programmable (OTP) memory of OpenTitan in the `lc_ctrl_fsm` FSM. Hence, an adversary aiming to increase the number of state transitions attempts either needs to fault the counter increment (i) or the programming (ii) of the value into the OTP.

**Skip the Counter Increment.** Inside the `lc_ctrl_state_transition` module, an FSM is responsible for updating the counter value.

```
Listing 9.13: Life cycle controller counter increment FSM.

1 module lc_ctrl_state_transition ( lc_cnt_e _lc_cnt_i, lc_cnt_e next_lc_cnt_o, ... );
2 unique case ( _lc_cnt_i )
3     LcCnt0: next_lc_cnt_o = LcCnt1;
4     LcCnt1: next_lc_cnt_o = LcCnt2;
5     ... 
6     LcCnt23: next_lc_cnt_o = LcCnt24;
7     endcase
```

As illustrated in Listing 9.13, this FSM uses a strong state encoding technique to mitigate fault attacks. Each state of type `lc_cnt_e` consists of 384-bit with a Hamming distance of 269 between `LcCnt0` and `LcCnt24` and a Hamming of 3 between `LcCnt23` and `LcCnt24`. To verify that the synthesis flow does not weaken the encoding, we utilize SYNFI to verify that a fault cannot manipulate the `next_lc_cnt_o` variable to a specific value (`FS`). More concretely, we aim to
bypass the counter increment from $LcCnt23$ to $LcCnt24$. In this scenario, the attacker aims to avoid that the counter increments to the final $LcCnt24$ value and locks the life cycle controller.

As shown in Row ① in Table 9.6, already a single fault allows the adversary to avoid that the counter is incremented. Supported by the generated analysis results, we were able to track back the single point of failure responsible for enabling an attacker flipping the three bits, i.e., the Hamming distance between $LcCnt23$ and $LcCnt24$, with a single fault. Since the three gates driving the three targeted bits of the $next_{lc\_cnt\_o}$ output port are driven by a single gate, attacking this gate or drivers of this gate allow an adversary to manipulate the output counter value. However, as the counter increment is only prevented once, an attacker only could initiate one additional state transition, making this attack impractical in reality.

Since resetting the counter value to $LcCnt0$ enables the attacker to initiate more additional state transitions, the encoding is also stronger, i.e., a Hamming distance of 269 between $LcCnt23$ and $LcCnt0$. To ensure that this strong encoding between these two counter values is correct after the synthesis, we test with SYNFI whether it is possible to switch to the specific $LcCnt0$ value from $LcCnt23$ with faults (FS). Similar to the previous experiment, we configure the framework to exhaustively inject 1, 2, and 3 simultaneous faults into the circuit responsible for incrementing the counter value. Since the possible fault combinations explode, we limited the number of injected faults to 1M and performed the experiment on a 72-core server in the cloud. Within this fault threat model, SYNFI could not find a single, effective fault, as shown in Row ② in Table 9.6.

**Prevent the Programming of the Counter.** The programming of the counter value into OTP is initiated in the $CntProgSt$ state in the life cycle controller FSM. We utilize SYNFI to validate that an adversary cannot bypass this state and directly switch to the next state (FS). As shown in Row ③ in Table 9.6, at least 7 faults, i.e., the Hamming distance the encoding guarantees, are required to skip the state.

**Life Cycle Escalation Signal**

When the life cycle controller detects an ongoing attack, the 4-bit encoded $lc\_escalate\_en$ signal is triggered. This signal is consumed by other hardware modules, e.g., the AES IP, and transfers them into a non-escapable error state. To validate that the optimization passes in the synthesis does not weaken the encoding of the signal, we inject faults into the escalation signal driven in the $lc\_ctrl\_signal\_decode$ module. In the fault model used by SYNFI, we set the input value to $lc\_escalate\_en = On = 4'b1001$, the expected output value to $lc\_escalate\_en\_o = On = 4'b1001$, and the fault output to $lc\_escalate\_en\_o = Off = 4'b0110$ (FS). Without constraining the synthesis flow with the `set_dont_touch` parameter (cf. Section 9.2.2), the security of the encoding is reduced to a Hamming distance of 2, as Synopsys removes the redundant flip-flops. When applying this constraint to the $lc\_escalate\_en$ flip-flop, at least four simultaneous faults are required to suppress the escalation
### Table 9.7: Verification results for the Ibex processor on a 16-core machine.

<table>
<thead>
<tr>
<th>Target</th>
<th>Setting</th>
<th>Simult. Faults</th>
<th>Effective [%]</th>
<th>Total Execution [s]</th>
<th>Circuit [GE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>① Glitch the PC</td>
<td>FE</td>
<td>1</td>
<td>78.1</td>
<td>557</td>
<td>185.07</td>
</tr>
<tr>
<td>② Glitch the PC</td>
<td>FS</td>
<td>2</td>
<td>0.02</td>
<td>309,500</td>
<td>0.72h</td>
</tr>
<tr>
<td>③ Lockstep mode</td>
<td>FS</td>
<td>1</td>
<td>6.31</td>
<td>111</td>
<td>10.22</td>
</tr>
</tbody>
</table>

Privilege Escalation in the PROD State

When OpenTitan is shipped to the customer, the device is put into the PROD state. In this state, certain features are activated, such as the CPU, and security-critical features, such as debug functionalities, are disabled. Instead of directly hijacking the life cycle state of OpenTitan, an adversary also could aim to switch on such features in the PROD state. All features are activated in the `lc_ctrl_signal_decode` module by setting the corresponding signal from `Off` to `On`. This signal then is transmitted to the corresponding hardware module responsible for activating or deactivating the feature. Similar to the escalation signal described in Section 9.2.2, the OpenTitan project uses a 4-bit encoding technique with a Hamming distance of 4 between `Off` and `On`. For the fault verification of the encoded signal, we configure the input to `lc_hw_debug_en = Off = 4'b0110`, the expected output value to `lc_hw_debug_en_o = Off = 4'b0110`, and the fault output to `lc_hw_debug_en_o = On = 4'b1001` (FS). Similar to the previous experiments, the `set_dont_touch` constraint needs to be applied to the registers responsible for driving the `lc_hw_debug_en` signal to maintain a Hamming distance of 4. Then, at least four simultaneous faults are required to enable the debug mode. While a transient fault only can active the debug functionality for a brief moment, a permanent stuck-at fault could allow an adversary to enable this feature permanently.

#### 9.2.3 Ibex

The RISC-V Ibex processor is the core element of the OpenTitan chip. In this section, we utilize SYNFI to analyze the behavior of the CPU when injecting faults. To demonstrate the ability of SYNFI to handle different netlists, we, contrary to the previous verification setups, analyze the netlist synthesized with the open-source Yosys synthesis tool and the open Nangate45 cell library.

**Results.** Our analysis showed that the error logic of the Ibex lockstep mode is capable of detecting a fault into the program counter.

**Glitching the Program Counter**

A fault into the Program Counter (PC) allows an attacker to arbitrarily redirect the control-flow of the program executed on the processor [TSW16]. We utilize SYNFI to ③ analyze whether the instruction fetch stage of the Ibex is generally
9.2. Analysis of OpenTitan

Table 9.8: Verification results for the `prim_double_lfsr` and `prim_count` modules.

<table>
<thead>
<tr>
<th>Target</th>
<th>Setting</th>
<th>Simult. Faults</th>
<th>Effective [%]</th>
<th>Total Execution [s]</th>
<th>Circuit [GE]</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>prim_count</code></td>
<td>FD</td>
<td>2</td>
<td>10.82</td>
<td>1552</td>
<td>37.46</td>
</tr>
<tr>
<td><code>prim_double_lfsr</code></td>
<td>FD</td>
<td>2</td>
<td>0.05</td>
<td>7827</td>
<td>74.44</td>
</tr>
<tr>
<td><code>prim_double_lfsr</code></td>
<td>FD</td>
<td>3</td>
<td>0.09</td>
<td>340,692</td>
<td>2114.69</td>
</tr>
</tbody>
</table>

susceptible to a fault arbitrarily changing the PC (FE). Row ① in Table 9.7 shows that a single fault already is sufficient to manipulate the PC and to redirect the control-flow. Although targeting a NOP slide, i.e., a consecutive sequence of NOP instructions, does not require an adversary to accurately manipulate the PC, randomly glitching the PC makes it hard for the attacker to exploit the induced fault. Therefore, we analyze ② if it is possible to change the program counter to a specific PC, i.e., from the boot address to 32’h40400, using a fault (FS). The analysis of SYNFI in Row ② in Table 9.7 shows that, with two simultaneously induced faults, glitching the PC to a specific value is hard. More specifically, for this fault specification, SYNFI shows that only 62 (0.02 %) out of 309,500 injected faults manipulate the program counter to the specified value.

Lockstep Mode

To protect the execution of software on Ibex from faults, OpenTitan instantiates the CPU redundantly in a dual-core lockstep mode. In this approach, the input used for the main core is delayed, provided to the redundant core, and the delayed output is compared to the output of the main core. On a mismatch, a hardware monitor raises an alert. Similar to the verification setup in Section 9.2.3, we consider an adversary aiming to redirect the control-flow by glitching the program counter. For the verification, we assume that the attacker already managed to flip a bit in the instruction address generated by the main core but not in the shadow core. As the error detection logic should raise an error due to the mismatch, a fault attacker needs to additionally inject a fault into this error detection circuit. SYNFI reveals that (i) the error detection logic actually raises an error, i.e., the synthesis tool did not remove the redundant logic, and that (ii) one fault could enable an attacker to suppress the error flag (FS), as shown in Row ③ in Table 9.7.

9.2.4 Generic Primitives

The OpenTitan project provides a set of generic hardware building blocks which are reused in several hardware modules. In this section, we analyze the fault protected generic primitives, i.e., the counter and the LFSR, using SYNFI.

Results. Our analysis with SYNFI confirms that the inspected primitives provide the expected security, i.e., a single fault into the protected counter or the LFSR triggers the alert signal of the countermeasures.
Counter

The prim_count module provides a fault protected generic counter primitive which is used by different modules. This module offers a flexible parameterization interface allowing the hardware designer to define the mode, the start value, and the bit width of the counter. In order to mitigate faults manipulating the counter value, the prim_count module implements the double count or cross count protection mode. While in the double count mode two redundant counters are compared, in the cross count mode the values of the up counting counter and the down counting counter are added and compared to a constant. On a comparison mismatch, a fault is detected and an error is triggered.

To ensure that the synthesis does not remove the redundant counter, we use the SYNFI framework to test the resilience of the module against faults. In particular, we check whether the countermeasure can detect a fault arbitrarily changing the output of the counter value (FD). For this, we configure SYNFI to inject faults into the counter logic, the counter registers, and the comparison logic. With this description, SYNFI automatically extracts the target circuit (29.75 GE) from the overall counter circuit (32.75 GE).

When injecting one fault into the netlist, SYNFI finds two effective faults manipulating the output counter value without triggering the error logic. These effective faults occur when faulting the counter increment signal or the counter clear signal, which are used by both counter instances. Since this behavior is documented in the description of the module, we further analyze the effect of two faults into the counter. In this setting, our tool shows in Row 1 in Table 9.8, that 10.82% of all injected faults are effective, i.e., manipulate the output counter value to an arbitrary value but do not trigger the error signal.

Double LFSR

As the Linear-Feedback Shift Registers (LFSRs) are used in OpenTitan as the primary source of randomness, they require a strong protection against fault attacks. The prim_double_lfsr module, which is used by several hardware IP blocks in the project, instantiates an LFSR twice and triggers an exception if the comparison of the two generated values mismatches.

In order to verify that a potentially aggressive synthesis setup does not remove the redundancy used as a fault protection, we use SYNFI to induce faults into the netlist and observe the behavior of the circuit. Here, we are interested if the error detection logic is capable of detecting a fault arbitrarily manipulating the output of the LFSR (FD). SYNFI confirms that a fault into the circuit manipulating the LFSR value is detected by the error logic. When inducing two simultaneous faults into the netlist, SYNFI finds, as shown in Row 2 in Table 9.8, 4 effective faults either suppressing the error signal and changing the output LFSR value or manipulating the LFSR value in both LFSR modules. Increasing the number of simultaneous faults to three increases the number of faults injected into the circuit of a size of 116.75 GE to 340,692, which takes 35 min on a 16-core setup. Based on these results, forging the LFSR output to an attacker controllable value
with three or less simultaneous faults is hard to achieve.

9.3 Related Work

Fault injection verification frameworks can be categorized into simulation- or verification-based approaches operating either on the RTL model or on the gate-level netlist. As indicated in the introduction of this chapter, frameworks [Gei20; Jen+94] working on the HDL description of a module only can provide security assumptions for this level of abstraction. In particular, the transformation of the RTL model into the gate-level netlist, i.e., the synthesis, can be responsible for inducing flaws into redundancy-based fault countermeasures by applying optimization passes. To also detect flaws potentially introduced in this design phase, various frameworks conduct their fault experiments at the netlist level [Bur+17; Arr+20; Ric+21; BN08; SKK13]. The disadvantage of simulation-based frameworks [Arr+20; BN08; SKK13] is that they require an input stimuli covering all inputs of the circuit. Verification-based frameworks, such as SYNFI, FIVER [Ric+21], and AutoFault [Bur+17], can achieve higher fault coverage as a SAT solver is responsible for probing all the undefined inputs. Similar to SYNFI, AutoFault and FIVER transform the gate-level netlist into a different representation and extract the equation of the circuit. FIVER first transforms the circuit into a DAG and then converts this graph into a binary decision diagram to perform the symbolic fault injection. As fault attacks originally focused on breaking cryptographic primitives, most fault injection frameworks [Bur+17; Arr+20; BN08], including FIVER, concentrate on analyzing such schemes. However, when using these frameworks to analyze more generic circuits, such as a silicon design of a root-of-trust element including a broad range of fault countermeasures, there are some limitations to overcome. First, some tools only provide support for a subset of VHDL descriptions [Bur+17] and others limit the number of supported gates [Ric+21] to a small set. Especially for industry-grade designs using long-established digital design flows, this constraint is severe as it is unlikely to adapt these hardware design flows. SYNFI overcomes these limitations by automatically processing and including arbitrary standard cell libraries into the framework and by translating the Verilog netlist into a unified model, i.e., a directed multigraph. This approach allows the framework to also support submodules when the boolean formula is provided. Second, FIVER requires that the given netlist does not include any cycles, i.e., the hardware designer needs to manually unroll the design before the evaluation. As described in Section 9.1.3, SYNFI is able to also handle such designs and automatically unfolds cycles found in the graph. To overcome computational limitations for larger circuits, the architecture of SYNFI makes heavy usage of multiprocessing, allowing the distribution of large workloads into the cloud. Finally, and in contrast to other frameworks [Bur+17; Gei20; Jen+94], we release an open-source version of SYNFI to encourage the verification of other security-critical designs.

Similar to related work [Ric+21], SYNFI can also be used to analyze the resilience of cryptographic primitives against fault attacks. For example, when
analyzing a round of the LED block cipher [Guo+11] protected by a detection-based countermeasure, the SYNFI user needs to provide a plaintext-ciphertext pair in the fault configuration file. Then, depending on the configuration, SYNFI can detect (i) whether it is possible to induce a fault with any effect on the ciphertext without triggering the countermeasure or (ii) whether it is possible to flip certain bits in the ciphertext without triggering the countermeasure.

9.4 Limitations and Future Work

This section summarizes current limitations of SYNFI and highlights potential future work.

**Fault Specification.** In the current prototype implementation of SYNFI, the user needs to manually specify input and expected output values in the fault model configuration. A possible future work could be to automate this process by parsing these values from traces generated by the simulation tools. This parser fetches the values of the circuit of interest for a specific amount of clock cycles and automatically writes these values into a separate fault model for each clock cycle. As SYNFI is already capable of successively analyzing multiple fault models, no additional changes in the existing framework would be required.

To assist the security engineer to specify the target circuit in the fault specification file, the SYNFI repository\footnote{https://github.com/lowRISC/synfi} contains an experimental feature automatically creating this file. When using this feature, the SYNFI user directly can specify the input and output nodes and their values of the target circuit in the HDL code using code annotation. The experimental tool then extracts this annotated information from the netlist and automatically describes the target circuit in the fault specification file, i.e., the tool defines the inputs and outputs and the state of the circuit.

**Preprocessing.** SYNFI automatically extracts a time-independent mathematical model of the circuit to analyze in the preprocessing phase. This time-independent model is created by replacing registers used in pipeline stages with pass-through elements and by removing cycles introduced by sequential logic. Hence, SYNFI is capable of analyzing the effect of a fault in multiple clock cycles. In addition, by automatically processing register stages and sequential loops, the framework can handle designs that the designer did not manually unroll. However, when aiming to analyze multiple loop iterations, e.g., multiple rounds in an iterative AES implementation, SYNFI needs to be configured for each round individually. Nonetheless, as SYNFI allows using multiple fault configurations in a single fault model specification file executed in one verification run, this is only a minor limitation. Nevertheless, a possible extension of the framework could automatically unroll the circuit instead of removing the loop.
9.5 Conclusion

Fault Effects & Layout. SYNFI and related frameworks [Bur+17; Arr+20; BN08] model a fault at the logical and not at the electrical level. Consequently, these frameworks cannot analyze transient faults occurring within a clock cycle and they also cannot consider the propagation delay between gates. Additionally, these tools, including SYNFI, operate on the gate-level netlist after the synthesis step and not on the layout after place and route. As some backend tools provide the possibility to simplify and optimize the netlist before the actual place and route step, SYNFI needs to be reapplied to this netlist to confirm the evaluation results. A future work could extend SYNFI to operate on the layout to also take the position of the gates into account for the analysis.

Performance. One of the main performance impact factors is the extraction and preprocessing phase (cf. Section 9.1.3). In this phase, SYNFI extracts (i) the target graph by finding all paths from the input and output nodes specified in the fault specification and handles (ii) registers used in iterative designs by finding cycles including a register. These operations on the graph could be improved by switching to a faster Python graph library or by porting SYNFI to C or C++.

Another performance limitation is the number of fault combinations, i.e., fault locations and fault effects. For an exhaustive fault analysis over all gates and multiple simultaneous injected faults, the number of fault combinations explodes. As SYNFI, for each fault combination, needs to create the differential graph, convert this graph into a boolean formula, and uses a SAT solver to evaluate the effectiveness of the faults, the number of fault combinations primarily affects the runtime. To improve this evaluation performance, the optimizations proposed by FIVER [Ric+21] could be integrated. Here, FIVER uses a fault propagation path analysis and a clustering technique to minimize the computational overhead.

9.5 Conclusion

In this chapter, we have presented SYNFI, a pre-silicon framework capable of inducing faults and analyzing their effects on the gate-level netlist. The framework enables hardware designers and security engineers to analyze the resilience of designs against fault attacks. As SYNFI conducts the security assessment directly on the unmodified netlist, the framework assures that (i) the same netlist is used for the evaluation as for the next steps in the digital hardware design flow with the final tape-out step and that (ii) potential security weaknesses still can be fixed before the chip gets manufactured. For the evaluation, SYNFI extracts the circuit of interest and injects fault into this circuit according to the fault model. To evaluate the effect of induced faults, the framework constructs a differential graph, transforms this graph into a mathematical model, and uses a SAT solver to study the behavior of the circuit when affected by faults. SYNFI is capable of (i) revealing whether faults affect the input-output relation of a circuit and its countermeasures and (ii) showing whether it is possible to enter a security-critical state using a fault without triggering the countermeasures. We have utilized the framework to assess the security of several hardware modules of OpenTitan, a
secure RoT chip. Our evaluation results presented in Section 9.2 showed that the unprotected AES module is highly susceptible to single faults, our proposed fault-hardening techniques increased the security, and that the other protected hardware blocks provide a strong resilience against fault attacks.
Computing systems, ranging from embedded devices to high-performance cloud services, are frequently targeted by memory safety vulnerabilities as well as physical fault attacks. As the consequences of these attacks are devastating, strong mitigation techniques are needed. In this thesis, we made significant contributions to protecting systems against these threats by designing novel countermeasures utilizing cryptography as a main building block.

More specifically, we have shown that we can realize memory safety and control-flow integrity using memory encryption. With CrypTag, we have demonstrated that cryptographic memory safety provides strong security guarantees with a reasonable performance overhead, outperforming existing schemes. Moreover, we have discussed that memory encryption is also suitable for defending systems against physical fault attacks. Although there exist cryptographically enforced control-flow integrity schemes, they require intrusive hardware changes making it difficult to deploy them at a larger scale. Within the context of this thesis, we have shown that we can achieve cryptographic control-flow integrity with minimal hardware modifications on OpenTitan or no hardware changes on Intel CPUs. The introduced control-flow integrity schemes prevent an adversary from redirecting the control-flow outside the call graph with reasonable performance overheads. As memory encryption is currently gaining enormous popularity on commodity systems, this thesis provided significant new contributions allowing systems to address different threats. Based on our work on control-flow integrity, we have revealed weaknesses of well-established CFI schemes and have used cryptography to mitigate the identified vulnerabilities. Furthermore, we also have shown that concepts used in encryption primitives can be leveraged to harden the control-flow of finite-state machines against faults with a lower area overhead than existing redundancy-based techniques. Finally, we have advanced the state-
of-the-art of fault countermeasure verification by introducing SYNFI. In contrast to existing fault frameworks, SYNFI is capable of proving the effectiveness of fault countermeasures directly at industry-grade gate-level netlists before the tape-out. In the following, we detail the contribution points of this thesis in more detail:

In the first part of this thesis, we have investigated the protection of systems against memory safety vulnerabilities. To that end, we have demonstrated that an already installed memory encryption engine, in addition to providing protection against physical attacks, can also be utilized to mitigate these vulnerabilities. We have developed a hardware/software co-design consisting of a modified RISC-V core and a custom LLVM compiler to analyze security guarantees and the performance overhead.

In the second part of this thesis, we have focused on researching novel countermeasures protecting systems against fault attacks. First, we have shown in Chapter 4 that fault attacks enable adversaries to redirect the control-flow, even in the absence of software bugs. Our demonstrated attack targeted an automotive platform running AUTOSAR, a popular real-time operating system broadly deployed in vehicles. By injecting messages into the automotive bus infrastructure and inducing faults into the communication stack, we have achieved full malicious code execution on the target device.

Based on this threat model, in Chapter 5, we have analyzed the security guarantees offered by hardware- and software-based fault countermeasures installed in the OpenTitan secure element. To increase the resilience of this chip against faults, we have introduced SCRAMBLE-CFI, an encryption-based control-flow integrity scheme. By utilizing the existing memory encryption unit of OpenTitan, we showcase that we have achieved strong security guarantees with only requiring minimal hardware changes.

Recent attacks showed that fault attacks also can be performed remotely in software, broadening the attack surface of this threat vector. Hence, in Chapter 6, we have researched a cryptographically enforced control-flow integrity scheme for recent Intel processors requiring no hardware changes. In this work, we have introduced a novel concept allowing us to utilize Intel’s memory encryption engine beyond its intended usage. Using this approach, we have provided fine-grained memory encryption on commodity hardware that can be utilized for control-flow integrity.

In Chapter 7, based on our work on control-flow integrity, we have identified weaknesses enabling adversaries to still redirect the control-flow of protected software. To that end, we have introduced a custom compiler that automatically protects code pointers used by indirect branches from fault attacks. By utilizing a hardware extension of recent ARM systems, i.e., the pointer authentication feature, we have demonstrated that our solution efficiently and securely protects software against the identified threat.

Chapter 8 has shown that control-flow integrity is a versatile technique to also protect hardware blocks against fault-induced control-flow attacks. By adapting control-flow integrity for hardware and leveraging building blocks
used in cryptographic primitives, we have shown that we can probabilistically protect finite-state machines against faults. To enable hardware designers to automatically protect arbitrary finite-state machines, we have integrated our concept into an open-source synthesis suite. Our evaluation has revealed that our protection approach improves the area-time product in comparison to related countermeasures.

Chapter 9 has highlighted the importance of conducting a pre-silicon analysis to ensure that fault countermeasures provide the expected security guarantees. In this context, we have introduced a fault injection framework capable of helping hardware designers to verify the correctness of countermeasures. To demonstrate the capabilities of our tool, we have utilized our open-source framework to analyze security-critical parts of the OpenTitan secure element. Based on our analysis results, we have hardened the design and contributed the changes to the open-source repository.

10.1 Outlook

Although, in this thesis, we have made progress in enhancing the resilience of computing systems against software vulnerabilities and fault attacks using cryptography, there are still open questions to address and new research directions available. In the following paragraphs, we summarize possible future work.

**Memory Encryption.** In this thesis, we have demonstrated that memory encryption is a versatile technique to mitigate memory safety vulnerabilities and fault attacks. Future work could investigate whether this primitive also can be leveraged for other use cases, such as in-process isolation and sandboxing. To fuel research in this direction, new open-source memory encryption frameworks compatible with larger processors are required. Although MEMSEC (cf. Section 2.5.2) can already be utilized in this context, the large overheads induced by memory encryption prevent widespread usage. Hence, the integration of lightweight and low-latency encryption primitives into memory encryption frameworks could be a future research direction.

**Protecting Indirect Branches on other Architectures.** Currently, the control-flow integrity schemes introduced in Chapter 5 and Chapter 6 do not explicitly protect code pointers from fault attacks. As we have demonstrated in Chapter 7, this attack vector enables an adversary to redirect the control-flow within the bounds of the call graph. To mitigate this attack, future work could investigate how to mitigate this threat on RISC-V and x86 without the pointer authentication feature.

**Control-Flow Integrity Verification.** In Chapter 5 and Chapter 6, we have provided an exhaustive security discussion of our cryptographically enforced control-flow integrity schemes. However, to determine hard security guarantees,
analyzing these schemes using a fault injection framework would be needed. Although we made first steps in this direction by introducing the SYNFI tool (cf. Chapter 9), new frameworks allowing to also analyze more complex processors, such as Intel CPUs, are required. To handle the complexity of larger designs, future frameworks could investigate new analysis approaches combining verification and simulation.
List of Contributions

Author’s Publications

The following publications are scientific contributions of the author that were discussed in this thesis.


Further Publications

The following list of publications comprises collaborations and contributions of the author, which are not part of this thesis.


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
<th>Conference/Journal</th>
<th>Pages</th>
<th>DOI</th>
</tr>
</thead>
</table>


[Ope] OpenTitan. CHIP_EARLGREY_ASIC Synthesis Results. URL: https://reports.opentitan.org/hw/top_earlgrey/syn/2022.07.02_00.42.20/results.html (visited on 03/22/2023).


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. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.