KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Technology

Improving Software-Implemented Hardware Fault Tolerance for Embedded Systems

Ing. Brent De Blaere

Supervisors:
Prof. dr. ing. J. Boydens
Dr. ing. J. Vankeirsbilck

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

November 2024

Improving Software-Implemented Hardware Fault Tolerance for Embedded Systems

Ing. Brent DE BLAERE

Examination committee:
Prof. dr. ir. F. Desplentere, chair
(KU Leuven, dept. of Materials Engineering)
Prof. dr. ing. J. Boydens, supervisor
(KU Leuven, dept. of Computer Science)
Dr. ing. J. Vankeirsbilck, supervisor
(KU Leuven, dept. of Computer Science)
Prof. dr. T. Holvoet
(KU Leuven, dept. of Computer Science)
Prof. dr. ir. P. Leroux
(KU Leuven, dept. of Electrical Engineering)
Ir. B. Vanderbeke
(Imec)
Dr. L. Daniel
(KU Leuven, dept. of Computer Science)
Prof. dr. M. Violante
(Politecnico di Torino university, Italy,
dept. of Control and Computer Engineering)

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

November 2024

© 2024 KU Leuven – Faculty of Engineering Technology
Uitgegeven in eigen beheer, Ing. Brent De Blaere, Spoorwegstraat 12, B-8200 Brugge (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Use of Generative AI
I did not use generative AI assistance tools during the research/writing process of my thesis, except for mere language assistance. The text, code, and images in this thesis are my own (unless otherwise specified). Generative AI has only been used in accordance with the KU Leuven guidelines and appropriate references have been added. I have reviewed and edited the content as needed and I take full responsibility for the content of the thesis.

Preface

Before you lies my PhD manuscript entitled “Improving Software-Implemented Hardware Fault Tolerance for Embedded Systems”. The completion of this dissertation marks the culmination of years of dedicated study, experimentation, and reflection. The journey of completing this PhD has been one of the most enriching experiences of my life. It has been an immensely rewarding process, full of both technical challenges and opportunities for personal growth. Working in the field of fault tolerance has been a fascinating and rewarding experience. I realize that, even after four years, there is still so much to discover and learn in this field.

My path has been shaped and supported by the remarkable people at the KU Leuven university in Bruges, where I have had the privilege of working alongside some of the most dedicated researchers, educators, and supporting staff. These people make KU Leuven more than just an academic institution, but a vibrant and inspiring community where excellence in research and education is a focal point. Being surrounded by such exceptional people – each of whom is driven by a commitment to knowledge and innovation – has been an experience I will carry with me long after I leave these walls. I am ever so thankful to have been part of this community.

I am deeply grateful to my supervisors, Prof. dr. ing. Jeroen Boydens and Dr. ing. Jens Vankeirsbilck, not only for providing me with the opportinity to pursue a PhD degree, but also for their guidance, support, and encouragement throughout the whole process. Jeroen, I am forever indebted for the mentorship you have provided me with. You have been a source of inspiration and encouragement. Thank you for the trust you have placed in me, and for the freedom you have given me to explore and experiment. Jens, you are a true expert in the field of software-implemented fault tolerance. Without your insignts and expertise, my work would not have been possible. Thank you for guiding me, for challenging me, and for always being there when I needed help.

Next to my supervisors, I would also like to thank Prof. dr. Tom Holvoet, Prof. dr. ir. Paul Leroux, and Ir. Bart Vanderbeke to have been part of my supervisory committee. Your feedback and insights have been invaluable to me. Thank you also to Prof. dr. Massimo Violante, Dr. Lesly-Ann Daniel, and Prof. dr. ir. Frederik Desplentere for being part of my examination committee and for providing me with valuable feedback on my manuscript.

Also, a special thanks to my colleagues at KU Leuven in Bruges, who have been a source of inspiration, support, and friendship. Thank you to the members of the DistriNet research group and to the members of the M-Group for the many interesting discussions and collaborations. The many enjoyable moments we have shared have made my time at KU Leuven all the more memorable. I am sure that the friendships I have made here will last a lifetime.

Finally, I wish to express my deepest gratitude to my family, friends, and my dearest Delphine, for being my steadfast support system throughout this journey. It is from you that I learned perseverance – a quality without which I would never have completed this trajectory. Thank you for your encouragement, understanding, and motivation during the most demanding phases of my research. Your unwavering belief in me, your encouragement, and your understanding provided the strength and motivation I needed to pursue this path to its conclusion.

This work is the product not only of my own efforts, but of the collective guidance of my mentors and collegues, who have supported me throughout this journey. Additionally, it stands on the foundation of countless researchers and thinkers whose contributions have shaped this field. In the words of Sir Isaac Newton: “If I have seen further, it is by standing on the shoulders of giants”.

Brent De Blaere
Bruges, November 2024

Abstract

In modern programmable systems, factors like higher component density, lower voltage levels, and an increased number of transistors have amplified the susceptibility of embedded systems to transient bitflip faults. These faults can occur due to external disturbances such as radiation or electromagnetic interference. This electromagnetic interference can be both conducted or radiated. These soft errors often pass by unnoticed, but their consequences can be severe since they lead to unpredictable system behavior. Specifically, two types of errors emerge, namely corruption of data (Data Flow Errors, DFEs) and unintended jumps in the program flow (Control Flow Errors, CFEs). Given the adoption of such systems in safety-critical and mission-critical applications, ensuring their reliability becomes paramount.

To mitigate the impact of bitflips, researchers have explored Software-Implemented Hardware Fault Tolerance (SIHFT) techniques. Unlike traditional hardware-based approaches, SIHFT aims to detect errors caused by transient faults directly within the software. By introducing redundant instructions to verify the program state, SIHFT techniques can identify corrupted data and control flow anomalies at runtime. However, much of the existing research focuses on simplified scenarios. These techniques struggle to scale up for larger, more realistic applications, making their adoption challenging or even impossible.

We investigated the limitations of state-of-the-art SIHFT techniques when applied to large case studies. We identified that many of the major techniques could not be applied to real-world applications due to their reliance on a large number of dedicated CPU registers. More specifically, instruction duplication techniques — one of the most prominent error detection techniques in the state-of-the-art — need these registers to store redundant values. Unfortunately, this can cause a shortage of available registers, leading to a compilation failure for real-world applications.

To solve this issue, we propose a set of SIHFT techniques that can reliably detect both DFEs and CFEs while merely utilizing three dedicated CPU registers. Opposed to the classical instruction duplication approach, our techniques rely on re-execution and checkpointing to detect bitflips. While the state-of-the-art DFE detection techniques fail to compile for numerous of our case studies, the techniques developed in this thesis can be implemented for all applications, regardless of their size or complexity. The proposed techniques were rigorously evaluated through extensive fault injection campaigns across various case studies, showcasing their effectiveness. Comparing our techniques to existing state-of-the-art methods, we found that they achieve a comparable error detection ratio for both DFEs and CFEs. The novel techniques effectively cover errors that cause unpredictable system behavior while disregarding faults with no impact on system functionality. However, the evaluation also shows that the re-execution-based approach can lead to considerable overhead in execution time and program size for certain applications.

To address the high overhead, firstly, the well-known strategy of selective implementation was used. In this approach, only a selection of the CPU registers are protected as opposed to all of them. Secondly, the possibilities of SIHFT extensions for the rising open-source RISC-V instruction set architecture were explored. Due to the extendable nature of RISC-V, new possibilities for developing optimized, reliable, and secure systems arise. As a step toward adopting software-implemented tolerance techniques within the RISC-V ecosystem, this thesis proposes a specialized RISC-V extension that supports one of our re-execution-based techniques. While SIHFT techniques can be implemented on any processor architecture, the RISC-V extension offers a more efficient implementation, promising significantly reduced overhead without impacting error detection rates.

In addition to the novel SIHFT techniques, we provide tool support to facilitate the adoption of these techniques in the form of a publicly available GCC plugin under a GPL license model. This plugin allows developers to easily apply the proposed SIHFT techniques, as well as numerous state-of-the-art techniques to their applications. The plugin supports multiple target architectures and is designed to be easily extendable to provide support for other target architectures and SIHFT techniques.

By providing new SIHFT techniques that can be applied to real-world applications and providing the tool support to implement these techniques, this thesis aims to make SIHFT techniques more accessible to the broader software engineering community.

Beknopte samenvatting

Door factoren als een hogere componentendichtheid, lagere spanningsniveaus en een groter aantal transistoren zijn moderne programmeerbare systemen gevoeliger geworden voor bitflip-fouten. Deze fouten kunnen optreden als gevolg van externe storingen zoals straling of elektromagnetische interferentie. De gevolgen van deze zogeheten *soft errors* kunnen ernstig zijn, omdat ze leiden tot onvoorspelbaar systeemgedrag. Concreet komen er twee soorten fouten voor, namelijk corruptie van gegevens (Eng: Data Flow Errors, DFE's) en onbedoelde sprongen in de programmastroom (Eng: Control Flow Errors, CFE's). Aangezien dergelijke systemen veel voorkomen in veiligheids- en missiekritieke toepassingen, is het van het allergrootste belang dat hun betrouwbaarheid wordt gegarandeerd.

Het onderzoek naar zogeheten *Software-Implemented Hardware Fault Tolerance* (SIHFT) technieken heeft als doel om de impact van deze bitflips te beperken. In tegenstelling tot traditionele benaderingen — die focussen op het introduceren van nieuwe hardware — streeft SIHFT ernaar fouten veroorzaakt door soft errors direct in de software te detecteren. Door redundantie te introduceren en extra controlevariabelen te gebruiken, kunnen SIHFT-technieken corrupte gegevens identificeren en afwijkingen in de uitvoervolgorde van het programma verifiëren. Veel van het bestaande onderzoek richt zich echter op vereenvoudigde scenario's die niet schalen naar meer realistische toepassingen. Daardoor is de adoptie van state-of-the-art SIFHT technieken een uitdaging of zelfs onmogelijk.

Wij onderzochten de beperkingen van de modernste SIHFT-technieken bij toepassing op grotere, meer realistische casestudies. We stelden vast dat veel state-of-the-art technieken niet kunnen toegepast worden op deze toepassingen, omdat de SIHFT-technieken een groot aantal CPU-registers vereisen. Meer specifiek vereisen instructieduplicatietechnieken — een van de meest prominente errordetectietechnieken in de bestaande literatuur — een groot aantal registers om de geduppliceerde toestand op te slaan. Helaas kan dit een gebrek aan beschikbare registers veroorzaken, wat leidt tot een compilatiefout.

Om dit probleem op te lossen, introduceren wij een set van SIHFT-technieken die zowel DFE's als CFE's betrouwbaar kunnen detecteren en daarvoor slechts drie CPU-registers gebruiken. In tegenstelling tot de klassieke instructieduplicatiemethode, baseren onze technieken zich op heruitvoering en checkpointing om bitflips te detecteren. Waar de state-of-the-art DFE-detectietechnieken niet kunnen compileren voor veel van onze casestudies, kunnen de technieken die in dit proefschrift zijn ontwikkeld, geïmplementeerd worden voor alle toepassingen, ongeacht hun grootte of complexiteit. De nieuwe technieken werden rigoureus geëvalueerd door middel van uitgebreide foutinjectiecampagnes voor verschillende casestudies. Door onze technieken te vergelijken met bestaande state-of-the-art-methoden, tonen we aan dat ze een vergelijkbare foutdetectieratio bereiken voor zowel DFE's als CFE's. De evaluatie toont echter ook aan dat de op heruitvoering gebaseerde benadering kan leiden tot aanzienlijke overhead in uitvoeringstijd en programmagrootte voor bepaalde toepassingen.

Om de hoge overhead aan te pakken, introduceren we eerst een selectieve implementatie van de techniek. Dit betekent in deze context dat slechts een subset van de CPU-registers beschermd wordt. Anderzijds werden de mogelijkheden van SIHFT-extensies voor de opkomende open-source RISC-V-instructiesetarchitectuur onderzocht. Door de uitbreidbare aard van RISC-V ontstaan er nieuwe mogelijkheden voor het ontwikkelen van geoptimaliseerde, betrouwbare en veilige systemen. Dit proefschrift stelt een gespecialiseerde RISC-V-extensie die een van onze nieuwe technieken ondersteunt voor. Hoewel SIHFT-technieken op elke processorarchitectuur kunnen worden geïmplementeerd, biedt deze RISC-V-extensie een efficiëntere implementatie, die aanzienlijk lagere overhead belooft zonder impact te hebben op de foutdetectieratio.

Naast de nieuwe SIHFT-technieken, introduceren we ook verschillende tools om de adoptie van deze technieken te vergemakkelijken. Dit komt onder andere in de vorm van een openbaar beschikbare GCC-plug-in onder een GPL-licentiemodel. Deze plug-in stelt ontwikkelaars in staat om talloze SIHFT-technieken eenvoudig toe te passen op hun applicaties. De plug-in ondersteunt meerdere architecturen en is ontworpen om eenvoudig uitbreidbaar te zijn om ondersteuning te bieden voor nog niet geïmplementeerde architecturen en SIHFT-technieken.

Door de introductie van de nieuwe SIHFT-technieken die kunnen worden toegepast op grote en complexe toepassingen en door de toolondersteuning om deze technieken te implementeren aan te bieden, wilt dit proefschrift SIHFT-technieken toegankelijker maken voor de bredere softwareontwikkelingsgemeenschap.

List of Abbreviations

ABI	Application Binary Interface
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BC	Bit Count
BS	Bubble Sort
CFCSS	Control Flow Checking by Software Signatures
CFE	Control Flow Error
CFED	Control Flow Error Detection
CFG	Control Flow Graph
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSO	Code Size Overhead
CU	Cubic solver
DETECTOR	Soft Error Detection through Low-level Re-execution
DFE	Data Flow Error
DFED	Data Flow Error Detection
DIJ	Dijkstra's algorithm
Distr	Distribution station
DMR	Dual Modular Redundancy
DUT	Device Under Test
ECC	Error-Correcting Code
EMI	Electromagnetic Interference
ETO	Execution Time Overhead

FDSC	Full Duplication and Selective Comparison
FFT	Fast Fourier Transform
FPGA	Field-Programmable Gate Array
GAS	GNU Assembler
GCC	GNU Compiler Collection
GNU	GNU's Not Unix (recursive acronym)
GPL	GNU General Public Licence
HIL	Hardware-In-the-Loop
HWD	Hardware Detected
I/O	Input/Output
IP	Intellectual Property
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
MM	Matrix Multiplication
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
NEF	No Effect
OVP	Open Virtual Platforms
P-DETECTOR	Soft Error Detection through Low-level Re-execution using Parity checking
PCB	Printed Circuit Board
QS	Quicksort
RACFED	Random Additive Control Flow Error Detection
RISC	Reduced Instruction Set Computer
RO	Research Objective
RTL	Register Transfer Level
S-DETECTOR	Selective Soft Error Detection through Low-level Re-execution
SaaS	Software as a Service
SDC	Silent Data Corruption
SDK	Software Development Kit

SEE	Single Event Effect
SEL	Single Event Latchup
SEU	Single Event Upset
SIHFT	Software-Implemented Hardware Fault Tolerance
SoC	System on a Chip
Sort	Sorting station
SOTA	State-Of-The-Art
SWD	Software Detected
SWIFI	Software-Implemented Fault Injection
SWIFT	Software Implemented Fault Tolerance
Test	Testing station
TMR	Triple Modular Redundancy
UML	Unified Modeling Language
XOR	Exclusive OR

Contents

Abstract iii

Beknopte samenvatting v

List of Abbreviations ix

Contents xi

List of Figures xv

List of Tables xxi

1 Introduction 1
- 1.1 Single Event Upsets 1
- 1.2 Soft Error Mitigation Techniques 6
- 1.3 Research Objectives 8
- 1.4 Contributions 9
- 1.5 Structure of the thesis 10

2 State-Of-The-Art 11
- 2.1 Control Flow Error Detection 11
  - 2.1.1 Principles of Control Flow Error Detection 11
  - 2.1.2 State-Of-The-Art Control Flow Error Detection Techniques 14
- 2.2 Data Flow Error Detection 16
  - 2.2.1 Principles of Data Flow Error Detection 16
  - 2.2.2 State-Of-The-Art Data Flow Error Detection Techniques 18
- 2.3 Hybrid Error Detection 20
- 2.4 Limitations of the State-Of-The-Art 21
- 2.5 Conclusion 22

3 Experimental Setup 25

3.1 Case Studies 25
  3.1.1 Data-Driven Case Studies 25
  3.1.2 I/O-Driven Case Studies 27
3.2 Experiment Methods 28
  3.2.1 Fault Injection Result Classification 29
  3.2.2 Simulation-Based Fault Injection Framework 30
  3.2.3 Hardware-In-The-Loop Based Fault Injection Framework 33
3.3 Overhead Analysis 35
3.4 Conclusion 35

4 Soft Error Detection Through Low-level Re-execution 37
4.1 Core Concepts 37
  4.1.1 Vulnerable Sections 38
  4.1.2 Checkpoints and Re-execution points 39
4.2 Implementation 42
4.3 Evaluation 46
  4.3.1 Register Availability 46
  4.3.2 Fault Injection Results for the Data-Processing Case Studies 47
  4.3.3 Fault Injection Results for the I/O-Driven Case Studies 49
  4.3.4 Overhead Evaluation 52
4.4 Conclusion 54

5 Optimizing the DETECTOR Technique 57
5.1 Selective Implementation 57
  5.1.1 Core Concepts 58
  5.1.2 Implementation 58
  5.1.3 Impact Analysis 60
5.2 Parity Checking 61
  5.2.1 Core Concepts 62
  5.2.2 Implementation 63
  5.2.3 Impact Analysis 65
5.3 Evaluation 67
  5.3.1 Dynamic Register Analysis 67
  5.3.2 Fault Injection Results for the Data-Processing Case Studies 68
  5.3.3 Fault Injection Results for the I/O-Driven Case Studies 69
  5.3.4 Overhead Evaluation 71
5.4 Conclusion 72

6 The RISC-V architecture 73
6.1 What is RISC-V? 73
6.2 Extensions 75
6.3 Registers 76
6.4 Instruction Formats 77

6.5 From ISA Definition to Hardware Implementation 79
6.6 Fault-Tolerant RISC-V Cores 80

7 Using RISC-V Extensions to Support SIHFT Techniques 81
7.1 Hardware Support for SIHFT Techniques 81
7.2 An Extension to Support P-DETECTOR 82
7.2.1 Special Purpose Registers 83
7.2.2 Duplicated Register Bank 84
7.2.3 Checkpoint Immediate Instruction 85
7.2.4 Parity Instruction 86
7.2.5 Re-execute Immediate Instruction 87
7.2.6 Parity Zero And Link Instruction 89
7.2.7 Example Usage 92
7.3 Impact Analysis 94
7.4 Changes to the GNU Assembler 95
7.5 Validating the Risc-V Extension Through Simulation 96
7.6 Evaluation 97
7.6.1 Fault Injection Results 97
7.6.2 Overhead Evaluation 98
7.7 Challenges for Micro-Architectural Verification 100
7.8 Conclusion 101

8 A GCC Plugin to Automatically Implement SIHFT Techniques 103
8.1 Introduction to the Compiler Plugin 104
8.2 Usage 105
8.3 GCC Intermediate Representations 107
8.4 Structure of the GCC Plugin 107
8.4.1 Support Classes 108
8.4.2 Entry Point 109
8.4.3 Techniques 110
8.4.4 Targets 111
8.5 Implementing the (P-)DETECTOR Technique 114
8.5.1 Locating the Vulnerable Sections 114
8.5.2 Finding the Checkpoints 116
8.5.3 Inserting the Instructions 117
8.6 Conclusion 118

9 Valorization 119
9.1 The GCC SIHFT Plugin 119
9.2 Fault Injection Frameworks 121
9.3 RISC-V Extension 123

10 Directions for Future Work and General Conclusions 125

10.1 Future Work 125
10.1.1 Borrowing Ideas from Signature Monitoring 125
10.1.2 From Error Detection to Error Correction 126
10.1.3 Synthesizing the RV32IXpdetector Core 127
10.1.4 Further Research on Hardware-Supported SIHFT Techniques 128
10.2 Concluding Remarks 128

A Overview of All Fault Injection Results 133
A.1 Fault Injection Campaigns on the Data Processing Case Studies on ARMv7-M 134
A.2 Fault Injection Campaigns on the I/O-Driven Case Studies on ARMv7-M 141
A.3 Fault Injection Campaigns on RISC-V 145

B Calculating the Added Instruction Cycles per Vulnerable Section 149
B.1 DETECTOR 150
B.2 S-DETECTOR 152
B.3 P-DETECTOR 154
B.4 P-DETECTOR Using the Xpdetector RISC-V Extension . . . 156

C Result of the Dynamic Register Analysis 157

D Assembly Reference 159
D.1 ARM Assembly 159
D.2 RISC-V Assembly 162

Bibliography 165

Publications 177

Curriculum Vitae 179

List of Figures

1.1 Moore's Law: the number of transistors on a microchip doubles approximately every two years. Data source: [2] 2
1.2 A radiation event (fast-charged particle) can create an ionized track in the substrate or silicon it passes through, generating electron-hole pairs. These charges can produce a parasitic transient current that disrupt the device. This effect can result in a transient pulse within logic circuits, creating a bitflip in memory components. 3
1.3 Simplified overview of the typical microcontroller components: (1) the memory and peripherals, (2) the buses, and (3) the CPU core. 5

2.1 The flow of control in a program can be represented in a control flow graph: (a) shows the bit-count program of the MiBench benchmark suite [29] in ARMv7-M assembly language and (b) shows the control flow graph of this program. 12
2.2 The three types of control flow errors depicted on a CFG, namely (a) inter-block, (b) intra-block, and (c) out-of-CFG control flow errors. 12
2.3 The principle of signature monitoring, here illustrated on a control flow graph, is a common technique to detect inter-block CFEs. An illegal jump from BB1 to BB2 is detected by the signature assertion at the end of BB2. 13
2.4 The concept of coarse-grained instruction duplication and comparison is a common technique to detect DFEs. Each instruction in the program is duplicated and executed on separate shadow registers. At the end of basic block 2, the registers are compared to their shadow registers to verify their correctness. 17

2.5 Trying to implement a SOTA DFED technique on larger, more industrially applicable case studies can lead to register allocation errors during the compilation process. 22

3.1 The miniature factory used for the I/O-driven case studies. The factory consists of three stations: the distribution station, the testing station, and the sorting station. Each station is controlled by an NXP LPC1768 microcontroller containing an ARMv7-M Cortex-M3 processor. 28
3.2 The simulated SWIFI framework uses the Imperas Instruction Set Simulator to simulate a target processor, the model of which is provided by Open Virtual Platforms (OVP). The fault injection framework injects faults into the register file of the simulated target using the Imperas API. 30
3.3 The flow and different error handlers of a data-processing case study used for the simulation-based fault injection framework. By monitoring the various error handlers, the framework can classify the outcome of each fault injection. 32
3.4 The hardware-in-the-loop-based fault injector injects faults into the Device Under Test (DUT) via the on-chip debugger. The environment of the DUT is simulated using the HIL simulator. The HIL simulator also validates the output of the DUT and sends the report to the fault injector. 34

4.1 UML activity diagram representing the flow of a program protected with DETECTOR. The protected section — the code between the checkpoint and a re-execution point — is re-executed. Next, the result of executions are compared to verify an error-free execution, after which the vulnerable section is executed. . . . 40
4.2 The control flow graph of the bubble sort algorithm in ARMv7-M assembly. The vulnerable sections are colored gray. 42
4.3 The DETECTOR technique implemented on the bubble sort algorithm in ARMv7-M assembly. 43
4.4 The `compare` subroutine used by the DETECTOR technique. . 45
4.5 Boxplots of the CFE single-bit fault injection results performed on the data-processing case studies protected with CFCSS, RACFED, and DETECTOR on the ARMv7-M ISA, showing the error detection ratio and SDC ratio. 48
4.6 Boxplots of the DFE single-bit fault injection results performed on the data-processing case studies protected with SWIFT, FDSC, and DETECTOR on the ARMv7-M ISA, showing the error detection ratio and SDC ratio. 48

4.7 The results of the single-bit CFE fault injection campaigns on the I/O-driven case studies protected with CFCSS, RACFED, and DETECTOR on the ARMv7-M ISA. 50
4.8 A CFE that jumps to a checkpoint can often remain uncaught by DETECTOR. 51
4.9 The results of the single-bit DFE fault injection campaigns on the I/O-driven case studies protected with DETECTOR on the ARMv7-M ISA. 51
4.10 Boxplots showing the Code Size Overhead (CSO) and Execution Time Overhead (ETO) of SWIFT, FDSC, CFCSS, RACFED, and DETECTOR. The CSO is evaluated over all case studies on the Cortex-M3 processor. The ETO is only evaluated for the data-processing case studies since the ETO for the I/O-driven case studies is negligible. 52

5.1 The re-execution flow of a program protected with DETECTOR, showing the locations of the push and pop operations to the shadow stacks (S_1 and S_2) and the `compare` subroutine. The push and pop operations onto and from S_2 (highlighted with a thick border) can be optimized by S-DETECTOR. 59
5.2 Comparison of the `compare` subroutine of DETECTOR and S-DETECTOR using the ARMv7-M assembly language. 60
5.3 The P-DETECTOR technique implemented on the bubble sort algorithm. The instructions that differ from the DETECTOR technique are indicated in bold. 64
5.4 The `calculateParity` subroutine of the P-DETECTOR technique in ARMv7-M assembly language. 65
5.5 In contrast to DETECTOR, control flow errors that jump to a checkpoint can often be caught by P-DETECTOR. 66
5.6 Boxplots of the CFE single-bit fault injection results performed on the data-processing case studies protected with DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA, showing the error detection ratio and SDC ratio. 68
5.7 Boxplots of the DFE single-bit fault injection results performed on the data-processing case studies protected with DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA, showing the error detection ratio and SDC ratio. 69
5.8 The results of the single-bit CFE fault injection campaigns on the I/O-driven case studies protected with DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA. . 70
5.9 The results of the single-bit CFE fault injection campaigns on the I/O-driven case studies protected with DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA. . 70

5.10 Boxplots showing the Code Size Overhead (CSO) and Execution Time Overhead (ETO) of DETECTOR, S-DETECTOR, and P-DETECTOR on the Cortex-M3 processor. 71

6.1 The RISC-V logo 73
6.2 The six instruction formats of the RISC-V instruction set architecture. 78

7.1 The Xpdetector extension introduces the `sr` register, `pr` register and `xs1`-`xs31` register bank to RISC-V. 84
7.3 The operation of the Checkpoint Immediate (`CPI`) instruction. 85
7.2 The encoding of the Checkpoint Immediate (`CPI`) instruction, using the I-type instruction format. 85
7.4 The encoding of the Parity (`PAR`) instruction, using the I-type instruction format. 86
7.5 The operation of the Parity (`PAR`) instruction. 87
7.7 The operation of the Re-Execute Immediate (`REI`) instruction. 88
7.6 The encoding of the Re-Execute Immediate (`REI`) instruction, using the new U0-type instruction format. 88
7.8 The encoding of the Parity Zero And Link (`PZAL`) instruction, using the J-type instruction format. 89
7.9 The operation of the Parity Zero And Link (`PZAL`) instruction. 90
7.10 The control flow graph of the bubble sort algorithm in RV32I assembly. The vulnerable sections are colored gray. 92
7.11 The P-DETECTOR technique implemented on the bubble sort algorithm in RV32IXpdetector assembly. 93
7.12 The P-DETECTOR technique implemented on the bubble sort algorithm in RV32I assembly. 94
7.13 The components of a custom processor model using the Imperas SDK. 96
7.14 The behavior of the intercept library in the Imperas ISS. . . . 97
7.15 Boxplots of the CFE single-bit fault injection results performed on the BC, BS, CRC, MM, and QS case studies protected with P-DETECTOR on the RV32I and RV32IXpdetector ISAs, showing the error detection ratio and SDC ratio. 98
7.16 Boxplots of the DFE single-bit fault injection results performed on the BC, BS, CRC, MM, and QS case studies protected with P-DETECTOR on the RV32I and RV32IXpdetector ISAs, showing the error detection ratio and SDC ratio. 98
7.17 The Code Size Overhead (CSO) and estimated Execution Time Overhead (ETO) of P-DETECTOR on the RV32I and RV32IXpdetector ISAs for each evaluated case study. 99

8.1 Manually implementing a SIHFT technique in assembly language is a time-consuming and error-prone process. 104
8.2 The different intermediate languages of the GCC compiler toolchain and the entry point of the GCC plugin. 106
8.3 The UML component diagram of the SIHFT GCC plugin. . . . 108
8.4 UML class diagram of the *Support Classes* component. 109
8.5 UML class diagram of the *Entry Point* component. 109
8.6 UML class diagram of the *Techniques* component. 110
8.7 UML class diagram of the *Targets* component. 112
8.8 Finding the source vulnerable blocks using a control flow graph. 116
8.9 Depending on which instructions are supported by the ISA and how the CFG of the program is structured, the instructions to implement the DETECTOR technique can be inserted in two different ways. 117

10.1 Borrowing ideas from signature monitoring techniques can improve the (P-)DETECTOR technique. 126

A.1 The results of the single-bit CFE fault injection campaigns on the data-processing case studies protected with CFCSS, RACFED, DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA. 136
A.2 The results of the single-bit DFE fault injection campaigns on the data-processing case studies protected with SWIFT, FDSC, DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA. 138
A.3 The results of the single-bit CFE fault injection campaigns on the I/O-driven case studies protected with CFCSS, SWIFT, DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA. 142
A.4 The results of the single-bit DFE fault injection campaigns on the I/O-driven case studies protected with DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA. . . 142
A.5 The results of the single-bit CFE fault injection campaigns on the BC, BS, CRC, MM, and QS case studies protected with P-DETECTOR on the RV32I and RV32IXpdetector ISAs. . . . 146
A.6 The results of the single-bit DFE fault injection campaigns on the BC, BS, CRC, MM, and QS case studies protected with P-DETECTOR on the RV32I and RV32IXpdetector ISAs. . . . 146

List of Tables

1.1 Failure reports by terrestrial neutrons in various industries. This shows that single event effects can cause a wide range of failures in various industries. Adapted from [8, Table 3.3] 4

2.1 A summary of the CFED techniques discussed in this section. The table shows which types of CFEs each technique can detect. 15
2.2 A summary of the DFED techniques discussed in this section. The table summarizes the properties of each DFED technique. 19

4.1 Overview of which case studies do (✓) and do not (✗) compile when each tested SIHFT technique is applied. 47
4.2 The average SDC ratio of the CFEs injected in the I/O-driven case studies, classified based on the source (from) and destination (to) location of the injected fault. 50

6.1 The base integer instruction sets of the RISC-V standard. . . . 74
6.2 Standard unprivileged extensions of RISC-V. 75
6.3 Unprivileged registers of the RISC-V base integer ISAs. 77

7.1 The opcodes used by the Xpdetector extension. 83
7.2 Using the `cpi` mnemonic in assembly to create a checkpoint. . . 86
7.3 Using the `par` mnemonic in assembly to calculate the parity of the register. 87
7.4 Using the `rei` mnemonic in assembly to re-execute parts of the program. 89
7.5 Using the `pzal` mnemonic in assembly to verify that the parity register is zero and jump to an error handler if it is not. 91

A.1 Details of the fault injection campaigns on the data processing case studies on the ARMv7-M ISA. 134

A.2 The average results of the single-bit CFE fault injection campaigns on the data-processing case studies protected with CFCSS, RACFED, DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA. 137

A.3 The average results of the single-bit DFE fault injection campaigns on the data-processing case studies protected with SWIFT, FDSC, DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA. 139

A.4 Details of the fault injection campaigns on the I/O-driven case studies on the ARMv7-M ISA. 141

A.5 The average results of the single-bit CFE fault injection campaigns on the I/O-driven case studies protected with CFCSS, RACFED, DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA. 143

A.6 The average results of the single-bit DFE fault injection campaigns on the I/O-driven case studies protected with DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA. 143

A.7 Details of the fault injection campaigns on the BC, BS, CRC, MM, and QS case studies on the RV32I and RV32IXpdetector ISAs. 145

A.8 The average results of the single-bit CFE fault injection campaigns on the BC, BS, CRC, MM, and QS case studies protected with P-DETECTOR on the RV32I and RV32IXpdetector ISAs. 147

A.9 The average results of the single-bit DFE fault injection campaigns on the BC, BS, CRC, MM, and QS case studies protected with P-DETECTOR on the RV32I and RV32IXpdetector ISAs. 147

B.1 The added instruction cycles for each added instruction cycle per vulnerable section for DETECTOR using the ARMv7-M Cortex-M3 processor. 150

B.2 The added instruction cycles for the `compare` subroutine of DETECTOR using the ARMv7-M Cortex-M3 processor. 151

B.3 The added instruction cycles for each added instruction cycle per vulnerable section for S-DETECTOR using the ARMv7-M Cortex-M3 processor. 152

B.4 The minimum and maximum added instruction cycles per vulnerable section added by S-DETECTOR on a Cortex-M3 processor ($N = 12$). 153

B.5 The added instruction cycles for each added instruction cycle per vulnerable section for P-DETECTOR using the ARMv7-M Cortex-M3 processor. 154

B.6 The added instruction cycles for the `calculateParity` subroutine of P-DETECTOR using the ARMv7-M Cortex-M3 processor. . 154
B.7 The added instruction cycles for each added instruction cycle per vulnerable section for P-DETECTOR using the Xpdetector RISC-V extension. 156

C.1 Results of the dynamic register analysis. The table shows how often each register is used for memory access operations in each case study performed on the Cortex-M3 processor. 158

Chapter 1

Introduction

1.1 Single Event Upsets

Today's modern society relies heavily on the use of embedded systems, both in industrial application fields and in home appliances. Microcontrollers, and microprocessors, are now able to perform complex tasks at lightning speeds with prolonged battery lives, driving the deployment of principles like the Internet of Things and Industry 4.0.

An embedded system comprises hardware and software components that interact with the physical environment via sensors and actuators to perform a specific task. The software component can include two subcomponents: the application and an optional (real-time) operating system. The necessity of an operating system depends on the type of applications the embedded system needs to run. Embedded systems without an operating system are known as bare-metal embedded systems, where the application interacts directly with the hardware. Designers can choose to work with a bare-metal system to eliminate the overhead of an operating system, thereby requiring a more simple and cheaper hardware.

Embedded systems have become the core of contemporary technology, with advancements in microelectronics technology enabling the integration of more functionality in smaller packages. Over the past five decades, the semiconductor industry has consistently adhered to Moore's Law (Figure 1.1), continually shrinking transistor sizes from approximately 10 micrometers to as small as 5 nanometers [1]. Knowing that a single silicon atom is about 0.2 nanometers in diameter puts the scale in perspective. Meanwhile, the power consumption of these devices has decreased, while their performance has increased. This trend

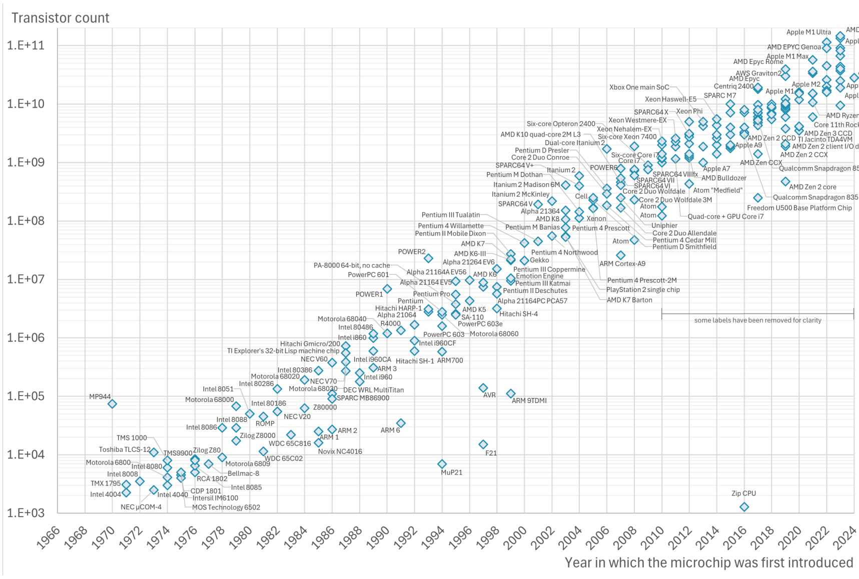


Figure 1.1: Moore's Law: the number of transistors on a microchip doubles approximately every two years. Data source: [2]

has enabled the development of more powerful and energy-efficient devices, which have become ubiquitous in our daily lives.

Since embedded systems are commonly used in safety-critical and mission-critical systems, their reliability is of utmost importance. However, the decreasing transistor sizes and operating voltages have increased the electronics' susceptibility to radiation and other environmental factors [3, 4]. Baumann et al. show how radiation from alpha particles, cosmic rays, muons, and neutron-induced ^{10}B fission can impact electronic components and corrupt the transistor state, causing a Single Event Upset (SEU) [3, 5, 6]. Such a radiation event is depicted in Figure 1.2. Radiation can cause localized ionization events, either directly or indirectly, which can disrupt internal data states.

An SEU is a type of Single Event Effect (SEE) usually appearing as transient pulses in logic or support circuitry, or as bitflips in memory cells or registers. In this thesis, the latter is the primary focus. SEEs are disturbances in the normal operation of a device caused by a single energetic particle. Some types of SEEs are hard errors (i.e. errors causing permanent damage to the device), like Single Event Latchups (SELs) and burnouts of power MOSFETs. However, SEUs are soft errors, meaning they are transient and thus do not cause permanent damage to the device. Simply cycling the power of the device or rewriting

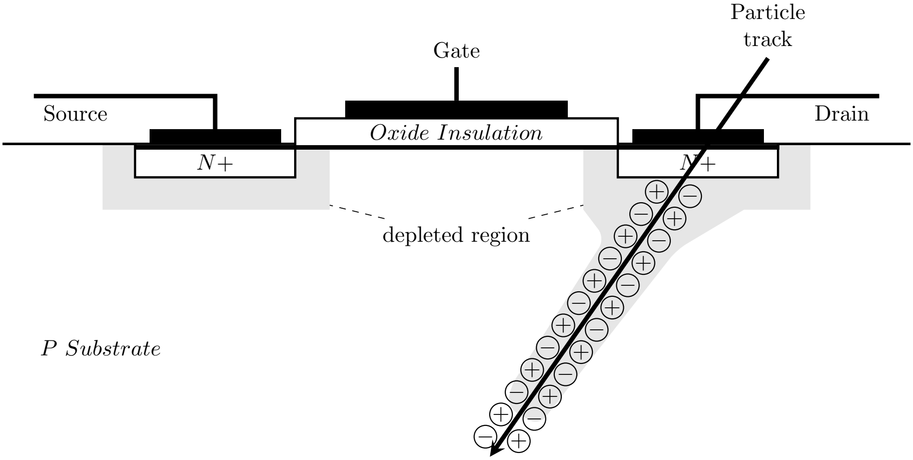


Figure 1.2: A radiation event (fast-charged particle) can create an ionized track in the substrate or silicon it passes through, generating electron-hole pairs. These charges can produce a parasitic transient current that disrupt the device. This effect can result in a transient pulse within logic circuits, creating a bitflip in memory components.

the bit of the affected memory cell or register will remove all traces of the original fault. This, however, does not mean that SEUs cannot cause significant damage or disruption to a system. The effect of a single event upset can cause wrong and unpredictable behavior in the system, which can lead to catastrophic consequences and physical harm in safety-critical systems [7].

In 2019, E. H. Ibe et al. documented the spreading concerns in various fields of industry related to SEEs [8]. Their results are summarized in Table 1.1. The table shows that SEEs can cause a wide range of failures in various industries, from the avionics industry to the automotive industry.

While the effects of single event upsets are well-documented, tracing a failure back to a soft error is challenging due to its transient nature. A system might behave incorrectly due to an SEU, but the affected memory cell might be overwritten well before the system is analyzed. Moreover, if the system is turned off and the faulty register is not written to permanent memory, the cause of the fault is lost. This makes diagnosis even more challenging. Therefore, it is only in rare cases – like the malfunctioning of the ADIRU (Air Data Inertial Reference Unit) of Qantas Flight 72 in 2008 causing two uncommanded rapid nose-down pitch events [9] – that the cause of the failure is traced back to SEUs. This, however, does not mean that the risks of radiation-induced

Table 1.1: Failure reports by terrestrial neutrons in various industries. This shows that single event effects can cause a wide range of failures in various industries. Adapted from [8, Table 3.3]

Field	Application	Failure symptom
Avionics	Fly by wire	Reboot
Railway	GTO[1]/IGBT[2]	Out-of-service
Network	Server Router Power supply	Data corruption/Reboot Reboot/Address change Out-of-service
Supercomputer		Unrecognizable wrong calculation
Automobile	Brake by wire Power steering Engine control CAN[3]/LIN[4] IGBT[2] Pedestrian detection	Non-stop/Sudden stop Stuck/Unexpected rotation Sudden acceleration/No operation Communication error Out-of-service No pedestrian detection

[1] Gate Turn-Off thyristor [2] Insulated Gate Bipolar Transistor

[3] Controller Area Network [4] Local Interconnected Network

errors go unnoticed by industry experts. For example, the 2022 edition of the IEEE International Roadmap for Devices and Systems warns of issues related to muon-induced soft errors in the long-term (2029-2037) in its *More Moore* report [10].

Bitflips in registers or memory components can also result from Electromagnetic Interference (EMI), which can introduce charges on PCB traces and transistors, changing their state [11, 12, 13]. This can both be conduced and radiated EMI. Research shows that the sensitivity of a transistor to EMI and particles increase when the transistor is subjected to higher temperatures [14].

Additionally, although this thesis focuses on unintentional upsets caused by environmental factors, attackers can intentionally introduce bitflips to extract critical data or bypass security protocols. Van Woudenberg et al. showed how voltage and clock perturbation and optical fault injection methods can be used to exploit modern microcontrollers and to bypass, among others, security condition checks such as PIN correctness [15].

Many components of a microcontroller can be affected by these faults. Figure 1.3 represents a simplified overview of the typical components of a microcontroller.

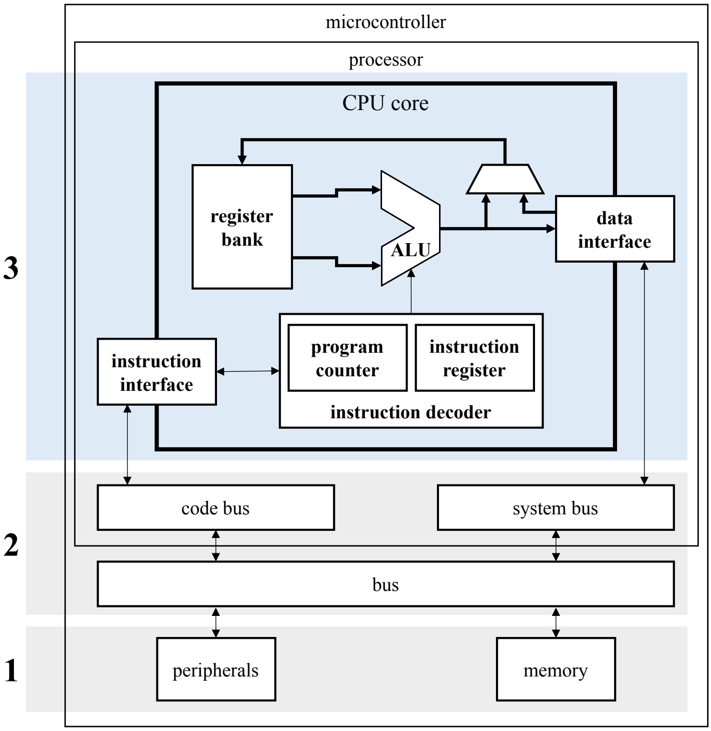


Figure 1.3: Simplified overview of the typical microcontroller components: (1) the memory and peripherals, (2) the buses, and (3) the CPU core.

Three sets of components vulnerable to bitflips can be identified.

The first set of components is the memory and peripherals. Transient faults in these components can corrupt data, instructions or interrupt signals. These types of faults have been known to occur for decades and can effectively be protected against by implementing Error-Correcting Code (ECC) in the memory and peripherals. Such codes add redundancy to the stored data, enabling error detection and error correction.

The second set of components is the many buses used to transmit the data between the different hardware components. Data residing on the bus can be corrupted by external disturbances. The data transmitted on the bus can again be protected by applying ECC such as hamming codes [16].

The components within the CPU core make up the third group. Bitflips in the register bank, arithmetic logic unit, and data interface will directly result in corrupted data in the register bank. These types of errors are called Data Flow Errors (DFEs). Bitflips in the program counter will cause an illegal jump in the program, resulting in a corrupted control flow. These errors are therefore called Control Flow Errors (CFEs). Bitflips in the instruction interface or instruction register can result in both types of corruption, depending on the instruction being executed and how that instruction is affected. This thesis focuses on disturbances in the components of the CPU core.

1.2 Soft Error Mitigation Techniques

To harden microprocessors against soft errors, countermeasures can be implemented across multiple levels of design abstraction, including the transistor level, circuit level, and program level [17].

Transistor-level solutions primarily depend on process technology to shield against radiation events (like alpha and neutron strikes) and to minimize the amount of charge that accumulates in a transistor node during such events [17]. This can, for example, be accomplished by shielding the electronics with a layer of depleted boron and mounting them on insulating substrates, like done for the 2020 Perseverance Mars rover [18]. These solutions are very costly and lag several generations behind Commercial Off-The-Shelf (COTS) processors [19]. Therefore, they are only used in applications where both the level of radiation and the cost of failure are very high (e.g. in space applications).

The circuit-level solutions largely focus on the use of redundant components to enable detection and recovery from soft errors. Another solution on this level is simply increasing the supply voltage or capacitance of the circuit, thereby increasing the energy required to flip a bit [17]. While effective, these hardware-based countermeasures also represent a high cost to implement as they need extra components, power and space. They also lack flexibility since they cannot be changed once the system has been deployed.

This thesis focuses on program-level mitigation techniques that make use of software resilience techniques to detect corrupted data or control flow. These techniques, often called Software-Implemented Hardware Fault Tolerance (SIHFT) techniques, are typically implemented solely on the software level by inserting checks throughout the program to verify an error-free execution. There are a number of reasons why SIHFT techniques could be considered over hardware-based solutions.

Firstly, many mission- and safety-critical systems have incorporated commercial off-the-shelf components to reduce costs and development time [20]. However, these components are not specifically designed, manufactured, and validated for the deployment in such critical applications. In these cases, SIHFT techniques can be used to mitigate the risks of transient errors. Since implementing hardening techniques at this level does not require costly hardware changes, they can be implemented on existing COTS systems.

Secondly, existing systems that are already deployed can be hardened by modifying the software through a firmware update. Although for safety- and mission-critical systems, this involves an extensive validation process, it is still significantly more cost-effective compared to recalling a product to replace its hardware components. The latter is often also infeasible, e.g. for space applications where, once launched, the system cannot be physically accessed anymore.

Thirdly, software-implemented techniques allow for a more flexible approach to the level of protection. The level of protection can be changed by modifying the software or can even be switched on or off at various stages in the program. For example, a mission-critical part of the program can be enhanced through a full protection scheme, while less critical parts can be left unprotected to save resources or speed up execution.

Applications that could benefit from SIHFT techniques include, but are not limited to:

- Automotive systems, where a soft error could cause a sudden acceleration or deceleration of the vehicle;
- Industrial control systems in high radiation environments such as nuclear power plants, where a soft error could cause a machine to operate incorrectly, potentially causing physical harm to operators;
- Medical devices in radiated environments such as MRI scanners, where a bitflip fault could cause the device to operate incorrectly, potentially causing physical harm to patients and healthcare workers;
- Space applications, where a SEU could comprimize the operation of a satellite or spacecraft, potentially causing the loss of the mission.

SIHFT techniques can be implemented in multiple ways. Many implement redundancy by executing the same program on multiple threads or processes and comparing the results of those threads [21, 22, 23]. Such redundancy techniques require either an operating system or a superscalar processor. However, market studies show that approximately 26% of the companies in the embedded systems market solely use bare-metal devices [24]. For these systems, single-threaded instruction-level redundancy techniques are used. They operate by adding instructions to the program at compile time to verify the correctness of the program execution. This can, for example, be done by implementing signature-checking mechanisms or duplicating variables and comparing their results.

When an error is detected by a SIHFT technique, an error handler is called. This error handler should be defined by the application programmer and should take appropriate action to put the system in a safe state or, if possible recover from the error. The implementation of this error handler is application dependent and out-of-scope for this thesis.

Although there are instruction-level redundancy techniques that are applied at high-level code, such as C or C++ [25, 26, 27], the majority of these techniques are implemented in low-level code, such as assembly language to ensure that the redundancy measures added by the SIHFT technique are not removed by compiler optimizations. This work focuses on these low-level instruction-level redundancy techniques.

1.3 Research Objectives

Over the past decades, a wide range of SIHFT techniques have been developed, each with its focus, advantages, and disadvantages. However, the research on these techniques typically focuses on small toy-like problems and simplified examples. Novel techniques are rarely tested on larger and more realistic problems, making their effective application in real-world scenarios uncertain. Therefore, the first objective of this thesis is to evaluate the applicability of state-of-the-art SIHFT techniques on larger, more industrially applicable case studies. This will provide a better understanding of the effectiveness of these techniques in real-world scenarios and will help to identify the strengths and weaknesses of different types of SIHFT techniques.

This research will reveal the limitations of the current SIHFT techniques and will provide insights into the development of new techniques. The second objective of this thesis is to develop new SIHFT techniques that address these limitations. These new techniques will be evaluated on the same case studies as the state-of-the-art techniques to compare their effectiveness. The aim is to create a set of techniques that provide full protection against both control flow errors and data flow errors. Since implementing these techniques in low-level code is complex and error-prone, the novel techniques will be implemented in a GCC compiler plugin [28].

Another way to improve the state-of-the-art of the SIHFT techniques is to create hardware support for the SIHFT techniques, thereby combining the flexibility of software-based techniques with the speed of hardware-based techniques. For this, the extendable RISC-V Instruction Set Architecture (ISA), which has recently disrupted the microprocessor market, can be explored. The extendable nature of this ISA opens the door to a range of specialized processors, tailored to specific applications and needs. This work will show that this can be useful for protecting against SEUs, as the ISA can be extended with instructions that are specifically designed to support SIHFT techniques. Therefore, the third objective of this thesis is to investigate how this can be achieved by creating a custom RISC-V ISA extension that supports one of the developed SIHFT techniques. This extension will be implemented on the Imperas Instruction Set Simulator (ISS), a widely used RISC-V simulator, to evaluate the effectiveness of SIHFT techniques supported by the RISC-V extension against the traditional SIHFT techniques.

To summarize, the main Research Objectives (ROs) of this thesis are as follows:

RO1. Evaluate the applicability of state-of-the-art SIHFT techniques on more industrially applicable case studies;

RO2. Develop new SIHFT techniques that address the limitations discovered in RO1;

RO3. Investigate how a custom RISC-V extension can be created to support SIHFT techniques.

1.4 Contributions

The main contributions of this thesis are as follows:

- The effectiveness of state-of-the-art SIHFT techniques on larger, more realistic case studies were evaluated, providing insights into the limitations of these techniques. This research shows that many of the State-Of-The-Art (SOTA) techniques – specifically Data Flow Error Detection (DFED) and hybrid techniques – cannot be applied to many case studies since they require a large amount of CPU registers dedicated to the technique.
- Three SIHFT techniques that address the limitations of the SOTA techniques were developed. These novel techniques were evaluated on the same case studies as the SOTA techniques. Whereas most common DFED and hybrid techniques focus on some form of code duplication, the first novel technique is a re-execution-based technique called DETECTOR.

Next, this technique was improved upon by introducing parity checking, resulting in the P-DETECTOR technique. Finally, versions of these techniques that can be selectively implemented were created. Various experiments indicate that the novel techniques provide full protection against both CFEs and DFEs, showing similar error detection capabilities as the SOTA techniques, while only requiring three CPU registers.

- The novel SIHFT techniques were implemented in the GCC compiler plugin framework of the M-Group research team, providing a more user-friendly way to implement these techniques in low-level code. Using this framework, the techniques can be easily applied to any program without the need for manual code changes. This can aid in both further research endeavors and the implementation of these techniques in industrial applications.
- The possibility of extending the RISC-V ISA to support SIHFT techniques was investigated. This work shows how the extendable nature of RISC-V allows for the creation of custom functionality to support SIHFT techniques by creating a custom RISC-V ISA extension that supports the P-DETECTOR technique. This extension was implemented as a custom CPU model on the Imperas RISC-V ISS. Its effectiveness was evaluated by comparing the implementation of the P-DETECTOR technique supported by the RISC-V extension to the normal P-DETECTOR implementation.

1.5 Structure of the thesis

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of the state-of-the-art in the field of SIHFT techniques and shows how these techniques have their limitations. Chapter 3 describes the tools and methodologies used for all experiments throughout this thesis. Next, Chapter 4 presents the novel SIHFT technique, called DETECTOR, and evaluates its effectiveness on a set of case studies. This is followed by Chapter 5, which describes two optimizations to DETECTOR. Chapter 6 introduces the RISC-V instruction set architecture and how its modular design is designed to support various standard and nonstandard extensions. This is the basis for Chapter 7, where a novel RISC-V extension to support the P-DETECTOR technique is described. Next, Chapter 8 describes how the implementation of SIHFT techniques can be automated using a GCC plugin, after which Chapter 9 details how this GCC extension and the other research results can be valorized. Finally, Chapter 10 concludes the thesis and provides directions for future research.

Chapter 2

State-Of-The-Art

As mentioned in Chapter 1, this thesis focuses on software-implemented hardware fault tolerance techniques, which comprise data flow error detection techniques, control flow error detection techniques, or a combination of both. This chapter provides an overview of the state-of-the-art in control flow, data flow, and hybrid error detection techniques in Sections 2.1 to 2.3. The chapter concludes with Section 2.4, which dives into the limitations with the SOTA that this thesis aims to address.

2.1 Control Flow Error Detection

2.1.1 Principles of Control Flow Error Detection

A control flow error is defined as the corruption of the execution order of instructions. More concretely, a CFE is a violation of the Control Flow Graph (CFG) of a program. A CFG is a directed graph that represents the flow of control in a program. It consists of basic blocks, which are sequences of instructions that are always executed in the same order and edges that represent the flow of control between basic blocks. By its nature, each basic block has one entry point and one exit point. Multiple edges can originate from one exit point and multiple edges can terminate at one entry point.

To illustrate this, Figure 2.1a shows a program written in ARMv7-M assembly language. Its corresponding CFG is shown in Figure 2.1b. Since instruction 0xf04 is a conditional branch instruction (`cbz`, *compare and branch on zero*),

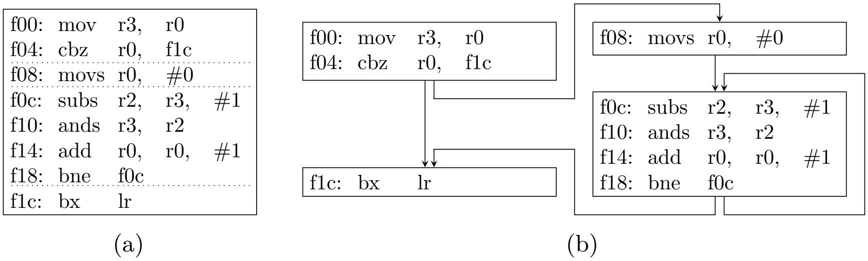


Figure 2.1: The flow of control in a program can be represented in a control flow graph: (a) shows the bit-count program of the MiBench benchmark suite [29] in ARMv7-M assembly language and (b) shows the control flow graph of this program.

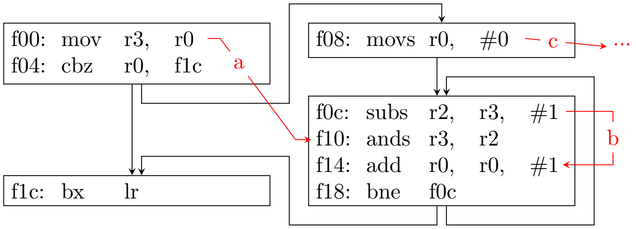


Figure 2.2: The three types of control flow errors depicted on a CFG, namely (a) inter-block, (b) intra-block, and (c) out-of-CFG control flow errors.

the program can either jump to instruction 0xf1c or continue to instruction 0xf08. Hence, this is the end of the first basic block. Two edges start from this exit point: one to instruction 0xf1c and one to instruction 0xf08. The same goes for instruction 0xf18 (`bne`, *branch if not equal*), which can loop back to instruction 0xf0c or continue to instruction 0xf1c. This creates an exit point at instruction 0xf18 and an entry point at instructions 0xf0c. Instruction 0xf1c is a return instruction (`bx`, *branch and exchange*), identifying the end of the subroutine and the CFG.

During a fault-free execution, the program executes each instruction of each basic block sequentially and follows the edges to jump between basic blocks. Thus, a CFE is an error during which the program (a) jumps between two basic blocks without following the correct edges, (b) jumps forwards or backward within the same basic block, or (c) jumps outside the control flow graph. The only exceptions to this rule are return (`bx`) instructions and function calls (`bl`, *branch with link* instructions) which, by jumping to different subroutines, can move between different control flow graphs. These so-called inter-block,

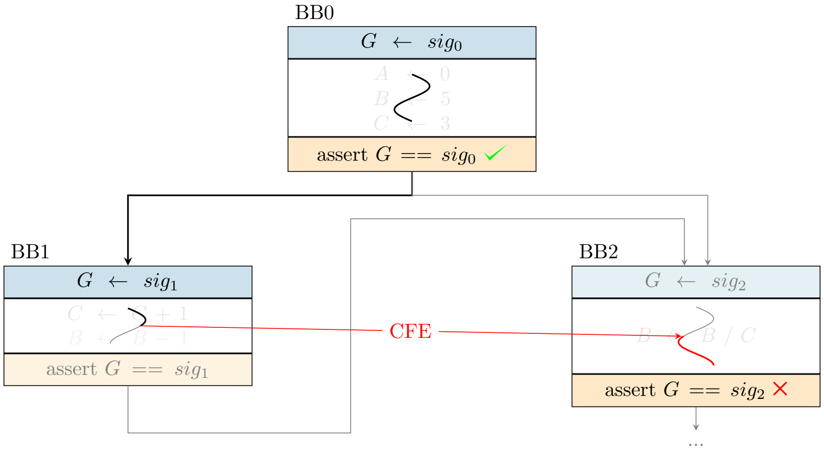


Figure 2.3: The principle of signature monitoring, here illustrated on a control flow graph, is a common technique to detect inter-block CFEs. An illegal jump from BB1 to BB2 is detected by the signature assertion at the end of BB2.

intra-block, and out-of-CFG control flow errors are shown in Figure 2.2.

Specialized Control Flow Error Detection (CFED) techniques exist for both inter-block and intra-block CFEs. Out-of-CFG CFEs cannot be targeted by specific techniques, as they are, by definition, not part of the CFG. They can, however, be detected by both inter-block and intra-block CFED techniques, depending on their nature. If the out-of-CFG CFE ends in another subroutine or program, the error can be detected by that program's inter- or intra-block CFED. If the error ends up in unused memory, the error could be detected by the system's memory protection systems. Alternatively, the unused memory can be filled with a jump instruction to a predefined error handler to safely handle the error. For this reason, out-of-CFG CFEs are not discussed in this thesis.

Figure 2.3 shows the concept of *signature monitoring*, a common technique to detect inter-block CFEs. Signature monitoring techniques use a signature to verify the correctness of the control flow. Each basic block i is assigned a signature sig_i. At the start of each basic block, this signature is assigned to a signature variable G. At specific points in the basic blocks, signature assertions are inserted. These assertions verify that the signature variable G matches the signature sig_i of the current basic block. If this assertion fails, a CFE is detected and an error handler is triggered. In the example in Figure 2.3, the signature assertion is performed at the end of each basic block.

Figure 2.3 also shows how this approach can detect a CFE. The program execution starts at basic block 0 and moves to basic block 1. During the execution of basic block 1, the program makes an erroneous jump to basic block 2. When the program reaches the end of basic block 2, the signature variable G is checked against the expected signature of basic block 2 (sig_2). Since G still holds the signature of basic block 1 (sig_1) this assertion fails. This triggers an error handler, which can log the error, reset the system, or take any other action, depending on the requirements of the system.

Apart from signature monitoring, another common technique to detect CFEs is *instruction monitoring*. While signature monitoring techniques typically only keep track of the basic block's signature, instruction monitoring techniques monitor each executed instruction to detect intra-flow CFEs.

2.1.2 State-Of-The-Art Control Flow Error Detection Techniques

Over the past decades, several CFED techniques have been proposed, each with unique characteristics and detection capabilities. What follows is a non-exhaustive list of the most prominent CFED techniques. A summary of the CFED techniques discussed in this section is shown in Table 2.1.

The **Enhanced Control flow Checking using Assertions (ECCA)** inter-block CFE detection technique was proposed by Alkhalifa et al. in 1999 [30]. It uses three compile-time variables for each basic block, identifying the basic block and the two successor basic blocks, as well as a run-time signature and helper variable.

Control Flow Checking by Software Signatures (CFCSS) was proposed by Oh et al. in 2002 [31]. The technique uses a compile-time signature and a *valid predecessor* value calculated at compile-time by the XOR operation of the current signature and the signature of the first predecessor basic block. While CFCSS can detect illegal jumps between (inter) basic blocks, it cannot detect wrong branch decisions.

Software Implemented Error Detection SIED was proposed by Nicolescu et al. [32] in 2004 and uses a unique identifier for each basic block, a list of signatures for all successor basic blocks and a value indicating the number of instructions that should be executed in the basic block. The technique uses a signature monitoring variable updated at the start and end of each basic block, as well as an instruction monitoring variable that counts down after each executed instruction to verify that the correct number of instructions has been executed. This combination of instruction monitoring and signature monitoring

Table 2.1: A summary of the CFED techniques discussed in this section. The table shows which types of CFEs each technique can detect.

Technique	Inter-block CFEs	Intra-block CFEs	Wrong branch decisions
ECCA	✓		
CFCSS	✓	✓	
SIED	✓	✓	✓
YACCA	✓		
RSCRC	✓	✓	
SEDSR	✓		
SCFC	✓		✓
RASM	✓		✓
RACFED	✓	✓	✓

enables SIED to detect both inter- and intra-block CFEs. Additionally, the update of the signature monitoring variable can be done conditionally, enabling the technique to also detect wrong branch decisions.

Yet Another Control flow Checking using Assertions (YACCA) is, like the name suggests, another CFE detection technique. This technique, proposed in 2004 by Goloubeva et al. [33], uses three to four compile-time variables for each basic block to verify the signature both at the beginning and at the end of each basic block. While YACCA can detect a limited number of CFEs that do not cross the boundaries of basic blocks, it is still considered an inter-block CFE detection technique.

The **Relationship Signatures for Control Flow Checking (RSCFC)** technique is designed to detect both inter-block and intra-block CFEs. It was proposed by Li et al. in 2007 [34] and uses three compile-time variables for each basic block: a signature, a CFG locator and a cumulative signature. The signature is generated by making a mask of the valid successor basic blocks. The CFG locator indicates the position of the basic block in the CFG, using the same type of mask. The cumulative signature is a bit sequence indicating the number of instructions to be executed for each basic block. By using bitwise operations on these three values, two run-time variables can be used to detect inter-block and intra-block CFEs. The technique can, however, not detect wrong-branch decisions.

Software Error Detection using Software Redundancy SEDSR, proposed by Asghari et al. [35] in 2012, uses a similar approach to RSCFC, using a compile-time signature calculated by creating a bit sequence that shows the valid successor basic blocks of the current basic block. However, SEDSR only focuses on inter-block CFEs, using a single run-time variable to verify that the current basic block is a valid successor of the previous basic block. Just like RSCFC, SEDSR does not detect wrong branch decisions.

SEDSR was later optimized by Asghari et al. [36] in 2014 to create **Software-based Control Flow Checking SCFC**. It uses the same compile-time signature as SEDSR but uses two run-time variables to verify the signature. This enables SCFC to also detect wrong branch decisions, enabling it to detect more CFEs. This technique was later again improved upon in [37] by analyzing the critical data path and by using timers to detect additional CFEs.

Random Additive Signature Monitoring RASM, proposed by Vankeirsbilck et al. in 2017 [38], uses two random compile-time values per basic block: a signature and a subtraction value. A single run-time variable is then used to verify the inter-block control flow. At the end of each basic block, the run-time variable is incremented so that its new value equals the sum of both compile-time variables of the successor basic block. This operation can be executed conditionally to enable RASM to also detect wrong branch decisions. At the start of each basic block, the variable is decremented by the subtraction value of the current basic block, ensuring that the run-time variable equals the signature value for error-free runs.

The final CFED technique that will be discussed is **Random Additive Control Flow Error Detection (RACFED)**, also proposed by Vankeirsbilck et al. in 2018 [39]. The technique is an extension to the RASM technique to detect inter-block and intra-block CFEs. It accomplishes this by using the same compile-time and run-time variables as RASM but updating the run-time variable at the end of each instruction instead of at the end of each basic block. This way, the technique remains elegantly simple, while detecting the full range of inter-block and intra-block CFEs.

2.2 Data Flow Error Detection

2.2.1 Principles of Data Flow Error Detection

A data flow error is defined by the corruption of input, intermediate, or output data. When approaching this from the level of the CPU, a DFE is an error that, either directly or indirectly, causes a wrong value in a register value of the CPU. For example, a soft error occurring in the register bank will directly cause one of the registers to hold an incorrect value, which may lead to incorrect results in the program. A soft error in the Arithmetic Logic Unit (ALU) of the processor,

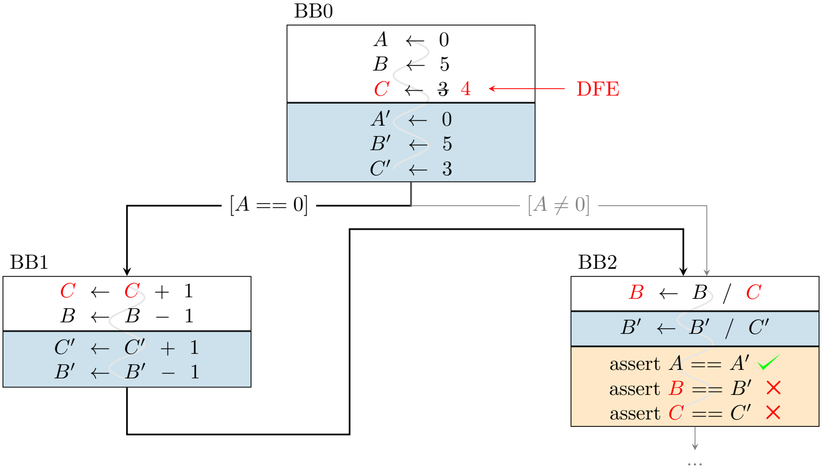


Figure 2.4: The concept of coarse-grained instruction duplication and comparison is a common technique to detect DFEs. Each instruction in the program is duplicated and executed on separate shadow registers. At the end of basic block 2, the registers are compared to their shadow registers to verify their correctness.

on the other hand, may cause the result of an operation to be incorrect, which in turn also corrupts the output register of the ALU.

Thus, since a DFE affects the register values, DFED techniques aim to verify that the register values are correct. This is typically done by using instruction duplication and comparison techniques.

The concept of instruction duplication and comparison is shown in Figure 2.4. The program is depicted in a CFG. In each basic block, all instructions are duplicated and placed at the end of the basic block. This is called a coarse-grained instruction duplication, as opposed to fine-grained instruction duplication, where each instruction is duplicated immediately after the original instruction. In the duplicated instructions, the original registers A, B, and C are replaced by shadow registers A', B' and C'. At the end of basic block 2, the original register values are compared to the shadow register values. A mismatch indicates a DFE and triggers an error handler.

The figure also shows how a DFE corrupting C can propagate throughout the program. The erroneous value of C causes a wrong calculation of register B in basic block 2. However, the shadow registers B' and C' remain unaffected. Therefore, when the assertions at the end of basic block 2 are executed, the

DFE is detected.

2.2.2 State-Of-The-Art Data Flow Error Detection Techniques

DFED techniques can differ significantly in their implementation, from the location where instructions are duplicated to the location or method used to compare the original and shadow registers. The following is a non-exhaustive list of the most prominent DFED techniques. A summary of the DFED techniques discussed in this section is shown in Table 2.2.

The **Error Detection by Duplicated Instructions (EDDI)** technique was proposed by Oh et al. in 2002 [40]. The technique uses fine-grained duplication and inserts assertions before branch and store instructions. This is done to ensure that decisions based on register values are correct and that final values, i.e. values that are written to memory, are correct. By using two distinct memory locations for all original and duplicated memory access instructions, almost every data manipulation or transfer instruction is duplicated. However, this also means that the memory overhead of the protected program significantly increases.

Oh et al. also proposed **Error Detection by Diverse Data and Duplicated Instructions (ED^{4}I)** in 2002 [41], which uses the same concepts, but puts the assertions after each duplicated instruction. This means that a DFE is detected as soon as possible. The technique does, however, introduce a lot of assertions, increasing the code size of the target program.

Similarly, **Instruction Level Duplication and Comparison (ILDC)**, proposed by Thati et al. in 2018 [42], duplicates all instructions except branch instructions. ILDC inserts assertions before store, branch, load, and move instructions to maximize the error detection coverage.

In 2012, **Critical Block Duplication (CBD)** introduced (coarse-grained) selective duplication, where instead of duplicating instructions in all basic blocks, only instructions in basic blocks deemed most *critical* are duplicated. This is done to reduce the overhead of the DFED technique. The definition of what makes instructions critical is an optimization problem and varies between different techniques. With CBD, Abdi et al. define critical basic blocks as those with the maximum amount of fan-out edges [43]. They reason that the results of these blocks propagate to many parts of the program, affecting a large part of it.

Arasteh et al. proposed an alternative to find the most critical blocks in 2015,

Table 2.2: A summary of the DFED techniques discussed in this section. The table summarizes the properties of each DFED technique.

Technique	Granularity	Full/Selective	Assertion placement
EDDI	Fine	Full	Before branch and store instructions
ED^{4}I	Fine	Full	After each duplicated instruction
ILDC	Fine	Full	Before store, branch, load, and move instructions
CBD	Coarse	Selective – protect basic blocks with the maximum amount of fan-out edges	At the end of the duplicated basic block
GA	Fine	Selective – protect the most vulnerable sequence of basic blocks determined through an iterative analysis	After each duplicated instruction
FDSC	Fine	Full duplication, Selective assertion – Only place assertions in basic blocks with more than one predecessor	After each duplicated instruction (if inside a vulnerable block)

which they called a **Genetic Algorithm (GA)** [44]. GA tries to precisely identify the smallest subset of basic blocks that are the most vulnerable and error-derating. The algorithm identifies an executable subset of the basic blocks that have a higher impact on the program results. Through various iterations, a fitness function evaluates how vulnerable each sequence of basic blocks is.

Full Duplication and Selective Comparison (FDSC), proposed by Thati et al. [45] in 2018, merges the advantages of full duplication and selective duplication techniques. Using FDSC, the entire codebase of the original program is duplicated in a fine-grained manner, but the assertions are only inserted in a few selected critical basic blocks. In this case, critical basic blocks are those with two or more incoming edges since these are more likely to be included in many execution paths throughout the target algorithm.

Finally, while most techniques – including the techniques that will be discussed in this thesis – focus on error detection and leave the error handling to the application developers, some techniques provide triple redundancy to introduce error recovery. In 2006, Chang et al. propose a set of techniques: **Software Implemented Fault Tolerance with Recovery (SWIFT-R)**, **Triple Redundancy Using Multiplication Protection (TRUMP)** and **Merely Asserting Statistically Known facts (MASK)** [46]. These techniques make three copies of the original program instructions and use majority voting mechanisms to automatically recover from a DFE.

2.3 Hybrid Error Detection

To fully protect systems against all types of soft errors, techniques need to protect against both CFEs and DFEs. Therefore, hybrid techniques are needed. In the state-of-the-art, this is accomplished by combining CFED and DFED techniques from Sections 2.1 and 2.2. Sometimes, this combination can be optimized to somewhat limit the overhead or to increase the error detection coverage

The following is a non-exhaustive list of the most prominent hybrid techniques.

Detecting Errors using a Software Approach (SA) was proposed by Nicolescu et al. [27] in 2003. This technique combines coarse-grained instruction duplication on all basic blocks for DFED with a basic signature checking for CFED.

The **Software Implemented Fault Tolerance (SWIFT)** technique was proposed by Reis et al. in 2005 [47]. This technique combines EDDI with CFCSS and makes some optimizations to both techniques. However, in this thesis, its optimized DFED mechanism is considered as a standalone DFED technique, as it is still considered as the baseline for a wide range of SOTA techniques. While EDDI uses two distinct memory locations for all original and duplicated store instructions and duplicates all memory access instructions, the authors of SWIFT choose to not duplicate store instructions at all. Reis et al. state that this does not reduce the fault detection coverage, but reduces the memory overhead of the protected program significantly.

In their discussion of RSCFC (2007), discussed in Section 2.1, Li et al. also show that the RSCFC can be combined with a DFED technique to create a hybrid technique called **Relationship Signatures for Control Flow Checking with Data Validation (RSCFCDV)** [34]. Although the authors do not go in much detail, they describe a coarse-grained instruction duplication technique similar to SA. Similarly, the authors of SEDSR (2012) show that their CFED mechanism can be combined with CBD to introduce selective DFED [35].

Finally, **near Zero silent Data Corruption (nZDC)** optimizes features of SWIFT to create a hybrid technique [48]. The technique, proposed by Didehban et al., uses a full fine-grained instruction duplication and comparison mechanism to detect DFEs, trying to optimize the error detection ratio while sacrificing overhead.

To compare the novel techniques that will be discussed in this thesis, separate techniques are used to evaluate the CFED and DFED mechanisms. This is done to ensure that the CFE and DFE detection mechanisms of the novel techniques are as effective as possible, as hybrid techniques are often less effective in detecting both types of errors compared to standalone techniques.

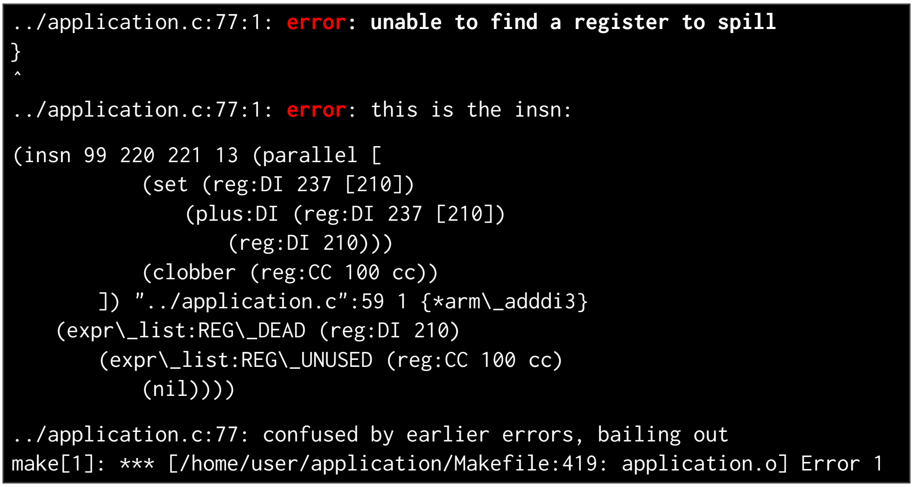
For the control flow error detection techniques, the CFCSS and RACFED techniques are used for the evaluation. CFCSS is one of the most prominent CFED techniques and is used as a baseline for many other CFED techniques. RACFED is a more recent technique and is considered to be one of the most effective inter- and intra-block CFED techniques.

For the data flow error detection techniques, SWIFT and FDSC is used. Like CFCSS, SWIFT is very often used as a baseline for DFED techniques. FDSC is used because it uses a selective comparison mechanism, which is comparable to the selective assertions that will be used in the techniques discussed in this thesis.

2.4 Limitations of the State-Of-The-Art

While the SOTA SIHFT techniques are effective in detecting soft errors, DFEDs techniques heavily limit the number of general-purpose registers that are available for the compiler to use. Since these techniques require a shadow register for each register used in the program, only half of the available general-purpose registers can be used in the original program. Since hybrid techniques combine DFED and CFED techniques, this limitation also applies to them. To account for this, the protected program should be compiled with a reduced set of registers.

Small programs, mostly used in research experiments, compile fine with a reduced set of general-purpose registers. In these cases, the compiler is able to cope with the reduced register set by spilling registers to memory where needed. However, during our research, we experienced that this does not hold for more elaborate programs. Since at least half of the available registers have to be reserved for error detection techniques, the register allocation process of the compiler can fail. When it does fail, the compiler explicitly mentions that this



```
../application.c:77:1: error: unable to find a register to spill
}
^
../application.c:77:1: error: this is the insn:

(insn 99 220 221 13 (parallel [
         (set (reg:DI 237 [210])
            (plus:DI (reg:DI 237 [210])
               (reg:DI 210)))
         (clobber (reg:CC 100 cc))
      ]) ”../application.c”:59 1 {*arm\_adddi3}
   (expr\_list:REG\_DEAD (reg:DI 210)
      (expr\_list:REG\_UNUSED (reg:CC 100 cc)
         (nil))))

../application.c:77: confused by earlier errors, bailing out
make[1]: *** [/home/user/application/Makefile:419: application.o] Error 1
```

Figure 2.5: Trying to implement a SOTA DFED technique on larger, more industrially applicable case studies can lead to register allocation errors during the compilation process.

register spilling process has failed, as shown in Figure 2.5 by the error “unable to find a register to spill”.

The DFED techniques discussed in Sections 2.2 and 2.3 were applied on eleven case studies for the Cortex-M3 processor (these case studies will be discussed in more detail in Chapter 3). This ARMv7-M processor contains 13 general-purpose registers, excluding the stack pointer and link register. Five of the case studies compiled fine. However, the other five case studies, including the I/O-driven case studies, failed to compile. This major oversight in the current state-of-the-art is the main motivation for the development of the techniques discussed in this thesis.

2.5 Conclusion

In this chapter, the state-of-the-art techniques for software-implemented hardware fault tolerance were explored, focussing on CFE, DFE, and hybrid detection techniques. The methodologies used by these techniques were discussed, showing how these can detect soft errors occurring in the program at runtime.

Several CFED techniques that monitor the flow of control in a program to detect any deviations caused by faults were examined. These techniques vary in their approaches, from using compile-time variables and runtime signatures to instruction monitoring, each aiming to ensure that the program execution follows the intended control flow graph.

DFED techniques, which focus on the integrity of data as it flows through the program, were also discussed. These techniques all base themselves on the duplication and comparison concept, differing in the granularity of the duplication, the scope of their protection, and the location of the inserted assertions.

Recognizing the need to protect systems comprehensively against both CFEs and DFEs, hybrid techniques have been developed. These techniques combine elements of the CFED and DFED techniques to offer robust protection.

Despite the advancements, the current state-of-the-art techniques have notable limitations. A significant issue is their reliance on many *shadow* registers, i.e. general purpose registers that are reserved to facilitate redundancy and can therefore not be used in the original program anymore. This issue, which is particularly prevalent in DFED and hybrid error detection techniques, can lead to register allocation errors during the compilation process. This limitation is a critical barrier to the practical implementation of these techniques in real-world applications. Therefore, this thesis will focus on developing techniques that do not suffer from this register availability problem.

Chapter 3

Experimental Setup

The experimental setup described in his chapter has been used and described in the following publications: [99, 100, 101, 102]. Since publication of these papers, the setup has been fine-tuned. Hence, this chapter reflects the latest version of the experimental setup.

This chapter describes the experimental setup used to evaluate the techniques presented in this thesis. First, the case studies that will be used throughout this thesis are presented. Next, methods used to evaluate the effectiveness of the techniques are described. This entails the classification of the fault injection results and the two fault injection frameworks used to conduct these experiments. Finally, the method used to evaluate the overhead of the techniques in both execution time and program size is discussed.

3.1 Case Studies

To evaluate the techniques presented in this thesis, several case studies were used. These case studies can be divided into two categories: data-processing case studies and I/O-driven case studies.

3.1.1 Data-Driven Case Studies

Data-processing case studies are programs that calculate a result based on input data. Given a specific input, the output is deterministic and invariant.

Eight data-processing case studies were used to evaluate the techniques presented in this thesis. For each data-processing case study, five datasets were used to guarantee that all control flow paths were covered.

- **Bit Count (BC):** This algorithm counts the hamming weight, i.e., the number of bits set to '1', in a given input word. This functionality is, amongst others, used to calculate a parity bit in communication domains and to calculate keys in cryptographic applications [49].
- **Bubble Sort (BS):** This algorithm sorts a list of numbers in ascending order using the bubble sort algorithm. Sorting algorithms like bubble sort are used in a wide range of applications, e.g., to assign priorities or to enable faster analysis of data [50].
- **Cyclic Redundancy Check (CRC):** This algorithm, mainly used to add error detection information to data transmissions, calculates the CRC of a given input array. This algorithm is mainly used to add error detection information in data transmissions [51].
- **Cubic solver (CU):** This algorithm solves an algebraic cubic equation $ax^3 + bx^2 + cx + d = 0$. Cubic solvers are typically used in various scientific applications, e.g. to find the acidity of a buffer solution [52] or to model the pressure of a gas [53].
- **Dijkstra's algorithm (DIJ):** Dijkstra's algorithm is used to find the shortest path between two nodes in a graph. It is therefore useful for many routing applications, e.g. in computer networks [54].
- **Fast Fourier Transform (FFT):** This algorithm computes the discrete Fourier transform of a given input array. This is typically used in digital signal processing applications, e.g. in digital recording, sampling, additive synthesis and pitch correction software [55].
- **Matrix Multiplication (MM):** This algorithm multiplies two $n \times n$ matrices. It is widely used in areas such as network theory, solving linear systems of equations, transforming coordinate systems and population modeling [56].
- **Quicksort (QS):** This algorithm sorts a list of numbers in ascending order using the quicksort algorithm. Just like BS, it is used in a wide range of applications. However, quicksort is generally faster than bubble sort, operating in $O(n \log n)$ time on average, compared to bubble sort's $O(n^2)$ time complexity.

These case studies were previously used by members of the KU Leuven M-Group research group to conduct a comprehensive evaluation of SOTA techniques [57, 58], as well as to evaluate several novel techniques [38, 39, 42, 45]. The BC, CRC, CU, DIJ, and FFT case studies are selected from the MiBench version 1.0 benchmark suite [29]. BS, MM, and QS use a custom implementation. They were chosen because the algorithms are used in a wide range of applications within the embedded systems domain, from digital signal processing to cryptographic applications. These case studies have varying basic block and edge distributions, which enables us to evaluate the techniques for different types of control flow graphs [59]. Furthermore, they have varying complexity in terms of number and type of instructions. This makes them suitable for evaluating the techniques on a wide range of programs.

The data-processing case studies are executed on a simulated ARMv7-M Cortex-M3 processor using the Imperas Instruction Set Simulator (ISS) [60]. This speeds up the evaluation process by allowing us to run the case studies at host speeds, rather than the slower speeds of the target hardware.

3.1.2 I/O-Driven Case Studies

Apart from the eight data-processing case studies, three I/O-driven case studies were used. While the data-processing case studies calculate a result based on input data, the I/O-driven case studies interact with the environment by reading input data from sensors and driving actuators to complete their task. These case studies comprise three stations of the Festo-Didactic MPS series [61], which combined, form the miniature factory shown in Figure 3.1.

- **Distribution station (Distr):** This case study uses a swiveling arm with a vacuum gripper to take workpieces from a magazine and distribute them to a downstream station.
- **Testing station (Test):** This case study tests each workpiece for faults, discards the faulty workpieces and delivers the good workpieces to a downstream station.
- **Sorting station (Sort):** This case study sorts the workpieces based on their color (black, metallic, and red) using two color sensors.

Each case study consists of a set of sensors and actuators controlled by an NXP LPC1768 microcontroller containing an ARMv7-M Cortex-M3 processor.

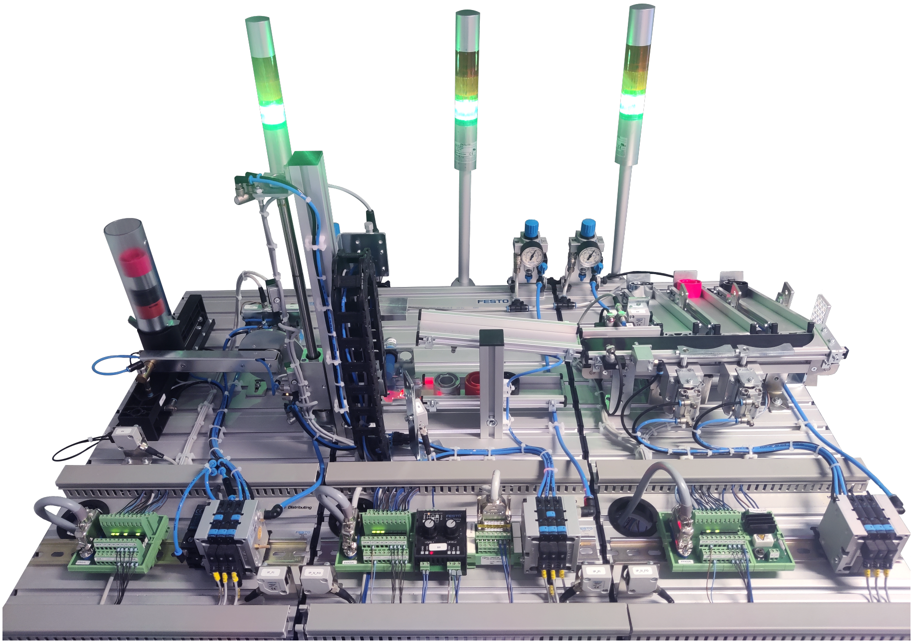


Figure 3.1: The miniature factory used for the I/O-driven case studies. The factory consists of three stations: the distribution station, the testing station, and the sorting station. Each station is controlled by an NXP LPC1768 microcontroller containing an ARMv7-M Cortex-M3 processor.

3.2 Experiment Methods

To evaluate the effectiveness of the techniques presented in this thesis, their ability to detect soft errors must be evaluated. While some researchers have conducted radiation experiments to test the robustness of their systems [62, 63], the use of fault injection experiments has become the standard in evaluating the effectiveness of SIHFT techniques. This technique enables researchers to evaluate the effectiveness of their techniques in a controlled environment while being able to deterministically evaluate each weak spot of the system.

In computer science, fault injection is a testing technique used to observe how computing systems behave when stressed in unusual ways. Specifically, when evaluating SIHFT techniques, soft errors are intentionally introduced into a system at runtime to evaluate the behavior of the system in these unexpected conditions. This is done for the original (*unprotected*) case studies as a reference and for the case studies with the SIHFT techniques implemented (*protected*).

To protect the case studies with a SIHFT technique, the SIHFT GCC plugin is used to automatically insert the required error detection mechanisms into the code. This plugin is explored in more detail in Chapter 8.

3.2.1 Fault Injection Result Classification

In order to accurately present the effectiveness of the evaluated techniques, the results of the fault injection experiments are classified into four categories:

- **Software Detected (SWD):** The faults that were detected by the implemented SIHFT countermeasure (**higher is better**).
- **Hardware Detected (HWD):** The faults that were detected by the several internal fault handlers of the processor, designed to detect faults such as memory access violations.
- **Silent Data Corruption (SDC):** The faults that were not detected (neither by the SIHFT technique nor by the internal fault handlers) and induced a corrupted result. A good protection against soft errors minimizes the amounts of faults in this category (**lower is better**).
- **No Effect (NEF):** The faults that were not detected by any measure, but also did not affect the outcome of the program.

This enables us to not only consider the error detection ratio of the SIHFT techniques, but also the number of remaining corruption events. This is important because the error detection ratio is measured relative to the fault-space of each program, but the fault-space of the unprotected programs is not the same as that of the protected programs [64]. Moreover, safety-critical systems aim to reduce the risk of failure, which is in this context the risk of a corruption event. Therefore, the absence of corruption events, rather than the error detection ratio, is the most important aspect for safety-critical and mission-critical applications.

To conduct these fault injection experiments, two in-house Software-Implemented Fault Injection (SWIFI) frameworks were used: a simulation-based fault injection framework and a Hardware-In-the-Loop (HIL) based fault injection framework.

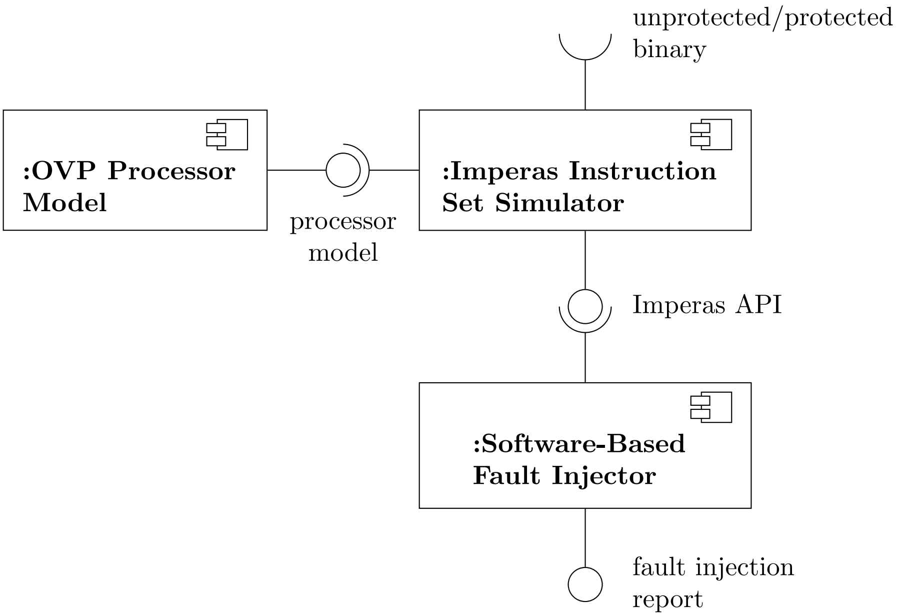

unprotected/protected binary
:OVP Processor Model
processor model
:Imperas Instruction Set Simulator
Imperas API
:Software-Based Fault Injector
fault injection report

Figure 3.2: The simulated SWIFI framework uses the Imperas Instruction Set Simulator to simulate a target processor, the model of which is provided by Open Virtual Platforms (OVP). The fault injection framework injects faults into the register file of the simulated target using the Imperas API.

3.2.2 Simulation-Based Fault Injection Framework

The simulation-based fault injection framework is a fault injection framework designed to inject bitflip faults into the register file of a simulated target running on the Imperas Instruction Set Simulator. It is used for the data-processing case studies.

The framework, originally developed by Vankeirsbilck et al. [65, 66] is shown in Figure 3.2. It uses the C++ API of Imperas to interface with the simulated target, the model of which is provided by Open Virtual Platforms (OVP) [67]. Depending on if the evaluated technique is a CFED, DFED, or hybrid technique, CFEs, DFEs, or both are injected into the target.

Injecting Control Flow Errors

The control flow error injection process used in the simulation-based fault injection framework was originally proposed by Vankeirsbilck as the *extended CFE injection process* [65].

The process starts by evaluating the control flow graph of the target program based on its disassembled binary. Using this CFG, the process evaluates all possible program counter corruptions for each instruction in the program using a single (or multiple, if so desired) bitflip fault. This results in four possible illegal jumps:

- Inter-block CFEs
- Intra-block CFEs
- Out-of-CFG CFEs
- Misaligned addresses

Since misaligned addresses will always result in a processor error, they can be excluded from the fault injection process. Additionally, as mentioned in Chapter 2, out-of-CFG CFEs can also be discarded. This creates a hash map of all possible inter-block and intra-block CFEs for each instruction in the program.

After this analysis step, the framework executes the program x steps, with x starting at 0. The program counter at that location is then read to obtain the address of the current instruction. Using this address as the key, an inter- or intra-block CFE is selected from the hash map generated in the analysis step.

Next, the program counter is rewritten to the selected CFE address and the program is resumed. When the program finishes (or after a timeout is exceeded), the outcome of the corrupted execution is evaluated and the result is saved.

Finally, the fault injection process resets the target and follows the same procedure, gradually going through all possible inter- or inter-block CFEs for the selected instruction. When all possible CFEs for the current step have been injected, x is incremented and the same steps are repeated for the next step of the program. This is repeated until the end of the program is reached.

By going through the full program and injecting all possible n-bit CFEs for each program instruction, the framework ensures complete coverage of all effective CFEs in the program.

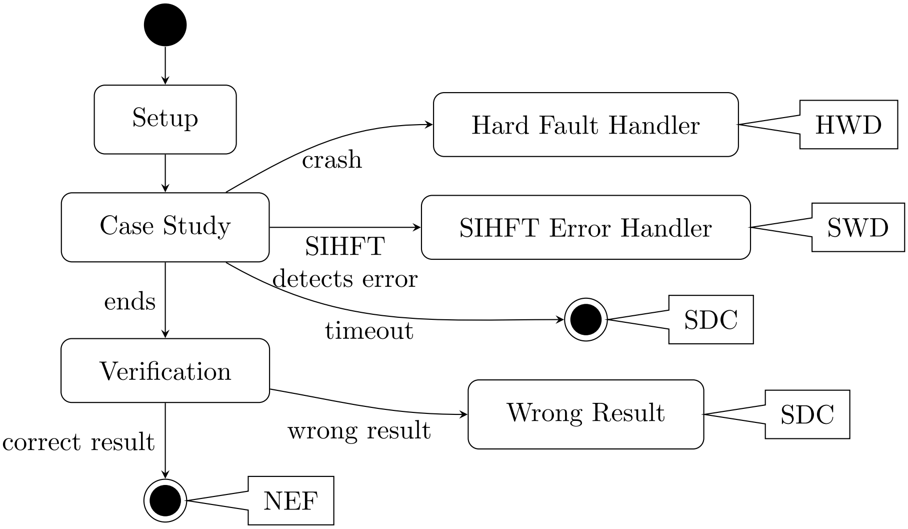


Figure 3.3: The flow and different error handlers of a data-processing case study used for the simulation-based fault injection framework. By monitoring the various error handlers, the framework can classify the outcome of each fault injection.

Injecting Data Flow Errors

The data flow error injection process used in the simulated SWIFI framework works similarly to the CFE injection process. First, during the analysis step, the framework determines which registers are used by each instruction in the program. This is again placed in a hash map, with the instruction address as the key and the registers used by the instruction as the value. For each step (x) in the program, the framework injects all possible n-bit DFEs for each register used by the instruction at that step.

Classifying the Fault Injection Results

To determine the outcome of each fault injection, the data-processing case studies include a verification step at the end of their program. This is shown in Figure 3.3. This verification function is used to verify the correctness of the calculated result.

If the result is incorrect, the function will branch to the *wrong result* handler, the location of which is known by the fault injection framework. The framework will detect this and classify the fault as a silent data corruption. If the result is correct, the program will simply exit, which will also be detected by the framework, classifying the fault as a *no effect* fault. The framework will also classify the fault as a silent data corruption if the program executes longer than a predetermined timeout.

Finally, the locations of the hard fault handler and SIHFT error handler are also known by the framework, enabling it to classify the fault as a *hardware detected* or *software detected* fault if the program ends up in one of these error handlers.

3.2.3 Hardware-In-The-Loop Based Fault Injection Framework

While the simulation-based fault injection framework uses a simulated target, the hardware-in-the-loop-based fault injection framework uses the on-chip debugger of a physical microcontroller to inject bitflips into the target.

This framework is used for the I/O-driven case studies. These case studies interact with the environment using sensors and actuators. While it would theoretically be possible to use the system in a real environment – i.e., the Festo-Didactic MPS series stations – during the fault injection experiments, this would be impractical.

Firstly, the injected soft errors cause the system to behave in unexpected ways, which could lead to damage to the hardware components. Secondly, an injected fault could leave the system in an unusual state, requiring manual intervention to reset the system. For example, a workpiece could be left on a conveyor belt, requiring it to be manually removed before the next fault injection can be performed. Thirdly, the actuators of the factory are slow, taking seconds to move from one position to another. This would make the fault injection process very slow.

Therefore, the environment of the I/O-driven case studies is simulated using a HIL simulator. This is shown in Figure 3.4. The HIL simulator receives the actuator signals of the Device Under Test (DUT) and dynamically simulates the sensors based on these actuator signals. This creates a closed-loop system where the HIL simulator imitates the real-world environment of the DUT.

For each injected fault, the HIL simulator runs through several scenarios to ensure that all possible control flow paths are covered. For example, in the testing station, both a good and a faulty workpiece will be simulated to ensure that the DUT is evaluated for both scenarios.

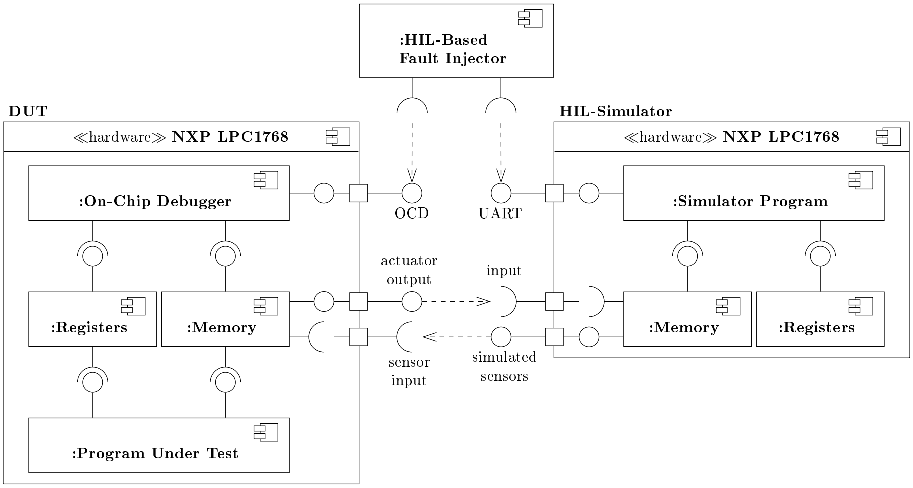


Figure 3.4: The hardware-in-the-loop-based fault injector injects faults into the Device Under Test (DUT) via the on-chip debugger. The environment of the DUT is simulated using the HIL simulator. The HIL simulator also validates the output of the DUT and sends the report to the fault injector.

The HIL simulator operates in *virtual time*, making the execution of the case studies magnitudes faster than operating in real time. However, the fault injection process is still slower than the simulation-based fault injection framework. This is due to the slower speeds of the physical targets and the different scenarios being simulated for each fault injection. Therefore, this framework uses a random fault injection process rather than the exhaustive fault injection process used in the simulation-based fault injection framework. While this means that not all possible faults are injected, it does show a general image of the effectiveness of the tested SIHFT technique.

The random fault injection process halts the program at a random location throughout the program under test. Next, it injects a fault in the program counter or in one of the registers used by that instruction, depending on the type of fault being injected.

Apart from simulating the environment of the DUT, the HIL simulator also validates the output of the DUT. Since the HIL simulator has an accurate simulation model of the environment, it can determine if the operation of the DUT is within the expected bounds. If it is not, the HIL simulator will send a report to the fault injector, which will classify the fault as an SDC. This will also happen if the HIL simulator does not receive a response from the DUT within a predetermined timeout, which indicates that the DUT is in a stalled state.

The fault injector detects errors caught by the internal fault handlers (HWD) and SIHFT technique (SDC) by setting breakpoints in their subroutines. If the program does not halt in one of these breakpoints and the HIL simulator does not report an error, the fault is classified as an NEF fault.

3.3 Overhead Analysis

Apart from evaluating the effectiveness of the SIHFT techniques, the overhead they introduce must also be measured. This overhead comes in two forms: Code Size Overhead (CSO) and Execution Time Overhead (ETO).

The code size overhead is determined by the number of instructions in the program before ($I_{\text{unprotected}}$) and after ($I_{\text{protected}}$) the implementation of the SIHFT technique, which can be obtained by analyzing the disassembled binary.

$$CSO = \frac{I_{\text{protected}}}{I_{\text{unprotected}}} \tag{3.1}$$

Similarly, the execution time overhead is determined by the time it takes for the program to execute before ($t_{\text{unprotected}}$) and after ($t_{\text{protected}}$) the implementation of the SIHFT technique.

$$ETO = \frac{t_{\text{protected}}}{t_{\text{unprotected}}} \tag{3.2}$$

Since the Imperas ISS is not a cycle-accurate simulator, physical hardware is used to evaluate the ETO of the SIHFT techniques for both the data-processing and I/O-driven case studies. This is done by utilizing the onboard hardware timer of the NXP LPC1768 microcontroller to measure the time it takes for the data-processing case studies to calculate the result or the time it takes to process one workpiece in the I/O-driven case studies.

3.4 Conclusion

In this chapter, the selected case studies, fault injection setup, and method for evaluating the overhead of the SIHFT techniques were presented. Eight data-processing case studies and three I/O-driven case studies were selected to evaluate the effectiveness of the techniques in both a data-processing and I/O-driven environment.

To evaluate the effectiveness of the SIHFT techniques, inter-block CFE, intra-block CFE, and DFE fault injections can be performed on these case studies using one of two fault injection frameworks. The simulation-based fault injection framework uses the Imperas ISS to inject faults into the data-processing case studies, while the HIL-based fault injection framework uses the on-chip debugger of a physical target and a hardware-in-the-loop simulator to inject faults into the I/O-driven case studies.

The evaluation of the SIHFT techniques takes both the results of the fault injection and the overhead analysis into account. More specifically, the number of corrupted results (SDC) should be minimized for a protected case study, which normally correlates to a higher error detection ratio. Meanwhile, the overhead of both the code size and execution time should be kept as low as possible.

Chapter 4

Soft Error Detection Through Low-level Re-execution

The content of this chapter has been published in [100]. Since publication of this paper, the results have been updated to reflect the latest version of the technique. Hence, this chapter reflects the latest version of the technique.

In this chapter, the re-execution-based DETECTOR technique is presented. First, the core concepts of DETECTOR are presented in Section 4.1. Next, Section 4.2 explains how these concepts can be used to implement the DETECTOR technique. The chapter concludes with Section 4.3, in which DETECTOR is evaluated using the fault injection experiments and overhead measurements as discussed in Chapter 3.

4.1 Core Concepts

As mentioned in Section 2.4, SOTA DFE and hybrid detection techniques suffer from limited implementation possibilities due to the need for many shadow registers. To overcome this limitation, DETECTOR was developed. DETECTOR stands for *Soft Error Detection Through Low-level Re-execution*. Instead of adding redundancy by using instruction duplication, DETECTOR adds redundancy through re-execution. This eliminates the need for a shadow register for each register in the processor.

The general idea of DETECTOR is to back up the program state and execute the program until it reaches a *vulnerable section*. Once such a vulnerable section is reached, DETECTOR temporarily stores the currently calculated values in memory, reloads the original values, and re-executes the instructions leading up to the vulnerable section. After the re-execution, the values of both cycles are compared to determine if a soft error occurred. If not, the program continues executing until a new vulnerable section is reached or until the complete program has been executed.

4.1.1 Vulnerable Sections

In a compiled program, registers are used to hold temporary values for computing. Once a final value is calculated, these register values are written to memory. This means that a corrupted register value does not immediately result in corrupted program results. Only when a corrupted value is written to memory or to an I/O device, can it manifest in a corrupted program output. Therefore, DETECTOR defines *vulnerable instructions*.

A Vulnerable Instruction is an instruction whose execution may cause the consolidation of the error in a silent data corruption. One or more consecutive vulnerable instructions make one *vulnerable section*. To avoid a corrupted program output, the register values must be correct when a vulnerable section is executed.

For the purposes of this work, three types of instructions are considered vulnerable instructions: memory write instructions, subroutine calls, and return instructions. Memory write instructions are considered vulnerable since writing to memory can result in a corrupted output as these values could be the result of a computation. Since the Cortex-M3 processor used for the experiments contains memory-mapped I/O, I/O operations are also covered by this definition. Subroutine calles are considered vulnerable instructions since the called subroutines need correct argument values to execute correctly. Return instructions are considered vulnerable instructions, as the register values should be correct when the function returns.

Finally, if a vulnerable instruction is executed conditionally, the compare instruction on which that instruction relies is also included in the vulnerable section. This is because this compare instruction sets the necessary flags for the conditional execution of the vulnerable instruction. This detail will be expanded upon in Section 4.2.

The remainder of the original program is called the *protected sections*. These are the parts of the program that will be re-executed before entering the vulnerable

Algorithm 4.1 The DETECTOR operations performed during a checkpoint.

```
$G \leftarrow sig_i$
$S_1$.pushAll $(registers)$
$checkpointLabel_i$ :
```

sections. Thus, any program consists of an alternation between protected and vulnerable sections.

4.1.2 Checkpoints and Re-execution points

At the end of a protected code section, DETECTOR inserts a *re-execution point*. At the start of each protected code section, it inserts a *checkpoint*. The mechanisms of DETECTOR enable a program to be re-executed between checkpoints and re-execution points.

The re-execution flow of DETECTOR is shown in Figure 4.1. When the program reaches a re-execution point, the current state of the register values is backed up. Next, the register values from the last reached checkpoint are restored and the program re-executes starting from that checkpoint. When the re-execution point is reached again, the register values of both executions are compared. If no corruption has occurred, the register values of both executions should be identical.

Operations of the Checkpoints

A checkpoint comprises three operations, shown in Algorithm 4.1. Each checkpoint is assigned a unique random nonzero compile-time signature sig_i. A value at a hamming distance greater than one from zero should be used to avoid that `G` becomes zero after a single bitflip. First, this signature is stored in a run-time variable G variable. Next, the register values are pushed on a shadow stack in memory. The variable S_1 represents the stack pointer to this shadow stack. Finally, a checkpoint label ($checkpointLabel_i$) is added at the end of the checkpoint. This checkpoint represents the start of the protected section, which is entered after performing the operations of the checkpoint.

Operations of the Re-execution Points

The operations performed during the re-execution point are shown in Algorithm 4.2. Since the checkpoint sets the signature variable G to a nonzero

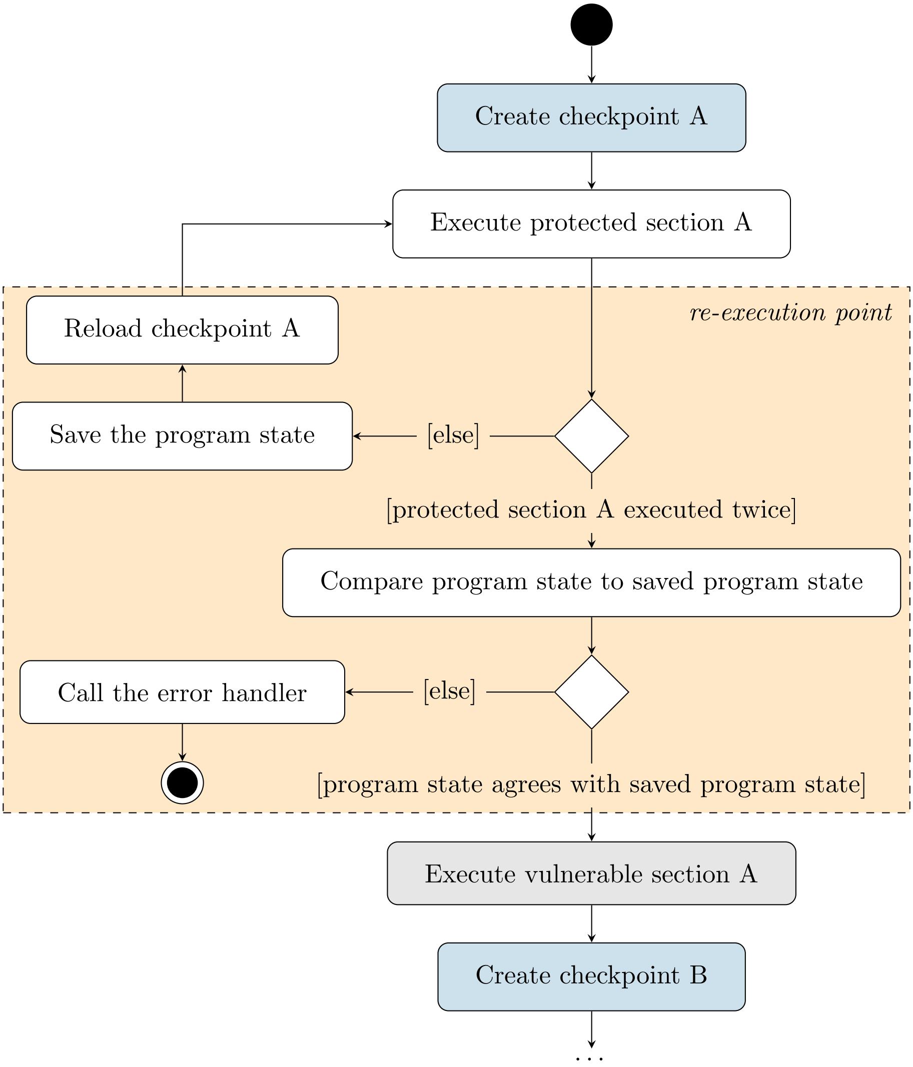

Create checkpoint A
Execute protected section A
re-execution point
Reload checkpoint A
Save the program state
[else]
[protected section A executed twice]
Compare program state to saved program state
Call the error handler
[else]
[program state agrees with saved program state]
Execute vulnerable section A
Create checkpoint B
...

Figure 4.1: UML activity diagram representing the flow of a program protected with DETECTOR. The protected section — the code between the checkpoint and a re-execution point — is re-executed. Next, the result of executions are compared to verify an error-free execution, after which the vulnerable section is executed.

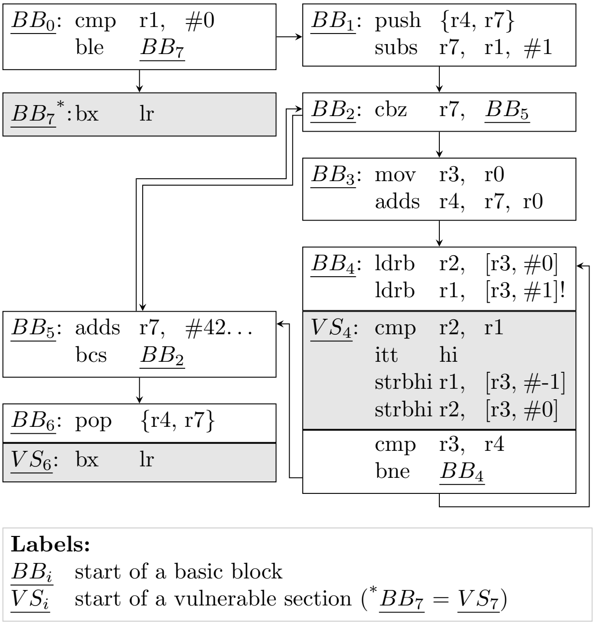
Algorithm 4.2 The DETECTOR operations performed during a re-execution point.

```
 1: if $G \neq 0$ then
 2:     $S_2$.pushAll($registers$)
 3:     $registers \leftarrow S_1$.popAll()
 4:     for all $sig_i \in sources$ do
 5:         if $G = sig_i$ then
 6:             $G \leftarrow 0$
 7:             goto $checkpointLabel_i$
 8: else
 9:     for all $register \in registers$ do
10:         $temp \leftarrow S_2$.pop()
11:         if $temp \neq register$ then
12:             errorHandler()
```

value, operations 1 through 7 are performed. First, all registers are pushed on a second shadow stack using a second shadow stack pointer S_2. Next, the original register values are restored by popping them from the first shadow stack. Then, based on the value of the signature register G (line 5), the program will jump back to the correct checkpoint label $checkpointLabel_i$ in line 7. Before this jump, the signature register G is cleared in line 6 to mark the start of the second execution. From there, the protected section is re-executed. Since the same register values are used, a fault-free run should follow the same path as the first execution.

When the program reaches the re-execution point again, the signature value is zero and thus, Algorithm 4.2 executes the comparison phase in lines 8 through 12. All registers are individually popped from the second shadow stack and compared to their current register value. A mismatch causes the error handler to be called. Only if all the values match can the program continue into the vulnerable section.

Note that DETECTOR does not re-execute the vulnerable section itself and thus will not detect transient faults happening within the vulnerable section. It safeguards the correct execution of the protected section before the vulnerable section.



```
BB0:  cmp    r1, #0
      ble    BB7
BB1:  push   {r4, r7}
      subs   r7, r1, #1
BB7*: bx     lr
BB2:  cbz    r7, BB5
BB3:  mov    r3, r0
      adds   r4, r7, r0
BB4:  ldrb   r2, [r3, #0]
      ldrb   r1, [r3, #1]!
VS4:  cmp    r2, r1
      itt    hi
      strbhi r1, [r3, #-1]
      strbhi r2, [r3, #0]
      cmp    r3, r4
      bne    BB4
BB5:  adds   r7, #42...
      bcs    BB2
BB6:  pop    {r4, r7}
VS6:  bx     lr
```

Labels:

$\underline{BB_i}$ start of a basic block

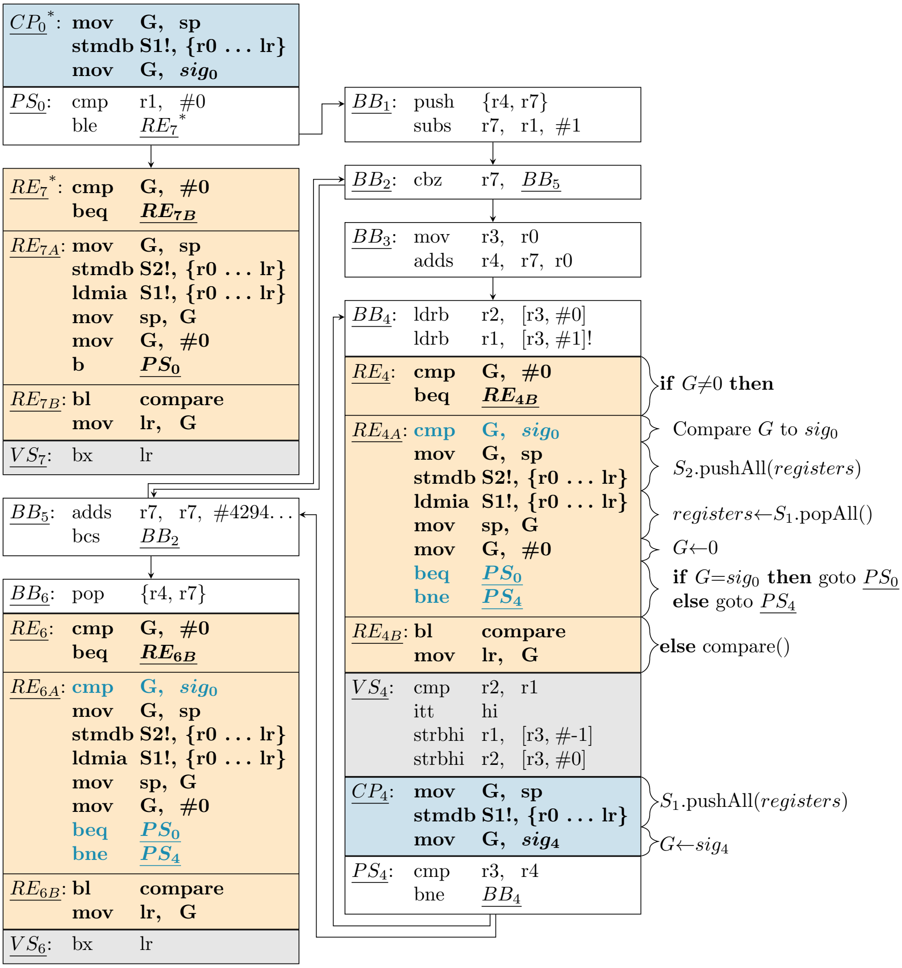
$\underline{VS_i}$ start of a vulnerable section ($^*\underline{BB_7} = \underline{VS_7}$)

Figure 4.2: The control flow graph of the bubble sort algorithm in ARMv7-M assembly. The vulnerable sections are colored gray.

4.2 Implementation

To explain the low-level implementation of DETECTOR, an example of the bubble sort algorithm will be used. Figure 4.2 shows the CFG of the algorithm in the ARMv7-M assembly language. The vulnerable sections are colored gray. In this example, there are only conditional store instructions (`strbhi`) and return instructions (`bx`). Notice the vulnerable section in basic block 4 ($\underline{BB_4}$). Here, the compare (`cmp r2, r1`) and if-then (`itt hi`) instructions are also part of the vulnerable section, as they are part of the conditional execution of the store instruction. If not included in the vulnerable sections, the condition flags would be disturbed by the added instructions in the re-execution points.

The implementation of DETECTOR on the bubble sort algorithm is shown in Figure 4.3. A first dedicated register `S1` is used to point to the first shadow stack. At the beginning of the program, the first checkpoint is inserted. All the values of the general purpose registers, the stack pointer, and the link register are pushed to the first shadow stack with the instruction `stmdb`. However, the ARM assembler does not allow the use of the stack pointer in the register list.



```
CP0*:  mov    G,  sp
       stmdb  S1!, {r0 ... lr}
       mov    G,  sig0
PS0:   cmp    r1,  #0
       ble    RE7*
RE7*:  cmp    G,  #0
       beq    RE7B
RE7A:  mov    G,  sp
       stmdb  S2!, {r0 ... lr}
       ldmia  S1!, {r0 ... lr}
       mov    sp, G
       mov    G,  #0
       b      PS0
RE7B:  bl     compare
       mov    lr, G
VS7:   bx     lr

BB1:   push   {r4, r7}
       subs   r7,  r1,  #1
BB2:   cbz    r7,  BB5
BB3:   mov    r3,  r0
       adds   r4,  r7,  r0
BB4:   ldrb   r2,  [r3, #0]
       ldrb   r1,  [r3, #1]!
RE4:   cmp    G,  #0
       beq    RE4B
RE4A:  cmp    G,  sig0
       mov    G,  sp
       stmdb  S2!, {r0 ... lr}
       ldmia  S1!, {r0 ... lr}
       mov    sp, G
       mov    G,  #0
       beq    PS0
       bne    PS4
RE4B:  bl     compare
       mov    lr, G
VS4:   cmp    r2,  r1
       itt    hi
       strbhi r1,  [r3, #-1]
       strbhi r2,  [r3, #0]
CP4:   mov    G,  sp
       stmdb  S1!, {r0 ... lr}
       mov    G,  sig4
PS4:   cmp    r3,  r4
       bne    BB4

BB5:   adds   r7,  r7,  #4294...
       bcs    BB2
BB6:   pop    {r4, r7}
RE6:   cmp    G,  #0
       beq    RE6B
RE6A:  cmp    G,  sig0
       mov    G,  sp
       stmdb  S2!, {r0 ... lr}
       ldmia  S1!, {r0 ... lr}
       mov    sp, G
       mov    G,  #0
       beq    PS0
       bne    PS4
RE6B:  bl     compare
       mov    lr, G
VS6:   bx     lr
```

if $G \neq 0$ **then**
Compare G to sig_0
S_2.pushAll($registers$)
$registers \leftarrow S_1$.popAll()
$G \leftarrow 0$
if $G = sig_0$ **then** goto PS_0
else goto PS_4
else compare()
S_1.pushAll($registers$)
$G \leftarrow sig_4$

Labels:

BB_i	start of a basic block ($^*CP_0 = BB_0$, $RE_7 = BB_7$)
VS_i	start of a vulnerable section
PS_i	start of a protected section
CP_i	start of a checkpoint
RE_i	start of a re-execution point
RE_{iA}	part of the re-execution point executed after the first execution of PS_i
RE_{iB}	part of the re-execution point executed after the second execution of PS_i

Figure 4.3: The DETECTOR technique implemented on the bubble sort algorithm in ARMv7-M assembly.

Hence, a different register is used to temporarily hold the stack pointer, which can be pushed to the stack pointer instead. For the purposes of this thesis, this register will be called `G`. This can be seen in Figure 4.3 at label $\underline{BB_0}$.

```
mov G, sp
stmdb S1!, {r0, r1, r2, r3 ... r12, G, lr}
```

The opposite can later be done to pop the registers and restore the stack pointer using the `ldmia` instruction (e.g. at label $\underline{RE_{7A}}$).

```
ldmia S1!, {r0, r1, r2, r3 ... r12, G, lr}
mov sp, G
```

Section 4.1.2 described how a signature variable is used to determine whether the program is in its first or second execution stage. Register `G` can again be used to fulfill this purpose. For the first execution, `G` is initialized with a uniqe nonzero value sig_0, as illustrated at the end of the checkpoint in $\underline{BB_0}$. After executing the code until the vulnerable section, the value of `G` is checked, as shown in label $\underline{BB_7}$. Since the register will have a nonzero value sig_0 after the first execution, the conditional branch `beq` $\underline{RE_{7B}}$ will not be taken. The program will simly continue to $\underline{RE_{7A}}$.

At label $\underline{RE_{7A}}$, the register values are pushed to the second shadow stack and the original values are popped from the first shadow stack. Register `S2` is used to hold a pointer to the second shadow stack.

```
mov G, sp
stmdb S2!, {r0, r1, r2, r3 ... r12, G, lr}
ldmia S1!, {r0, r1, r2, r3 ... r12, G, lr}
mov sp, G
```

The first execution concludes by setting the signature register `G` to zero, which marks the start of the second execution, before jumping back to the start of the protected section.

Since processes often take different paths through their CFG, it is necessary to keep track of which checkpoint to use when starting a re-execution. This is done using the unique signature value of each checkpoint. By checking the value of `G`, the technique knows to which instruction it should jump to re-execute a part of the code. This is the function of the additional instructions in the re-execution points at labels $\underline{RE_{4A}}$ and $\underline{RE_{6A}}$.

After the re-execution of the protected section, `G` is again compared to zero at label $\underline{BB_7}$. This time, `G` is indeed zero and thus, the conditional branch

```
compare:
CMPr0:   ldr   G,   [S2],#4   @ r0
         cmp   r0,  G
         bne   ERR
..................................................
                     ...
..................................................
CMPr12:  ldr   G,   [S2],#4   @ r12
         cmp   r12, G
         bne   ERR
..................................................
CMPsp:   ldr   G,   [S2],#4   @ G (=sp)
         cmp   sp,  G
         bne   ERR
..................................................
RET:     ldr   G,   [S2],#4   @ lr
         bx    lr
--------------------------------------------------
ERR:     bl    errorHandler
```

Figure 4.4: The `compare` subroutine used by the DETECTOR technique.

to $\underline{RE_{7B}}$ taken. This means that the *compare* function is called (`bl`). The implementation of *compare* is shown in Figure 4.4. In this function, the values of the first execution are popped one by one from the second shadow stack and temporarily stored in `G`, after which they are compared with their respective registers. If a mismatch occurs, an error is detected and the program branches to the error handler at label $\underline{ERR}$.

When the *compare* function is called with the instruction `bl compare`, the link register `lr` will be loaded with the address of the instruction after the function call (instruction `mov lr, G` in Figure 4.3 in this case). By doing this, the program can jump back to the address stored in the link register when the function returns (`bx lr` in Figure 4.4). However, this implies that the link register value of the original program has been lost. To recover this register, the value of `lr` is popped to register `G` at the end of the *compare* function (label $\underline{RET}$ in Figure 4.4). `G` can then be used to restore the original value of `lr` at the end of the re-execution point. This, however, does mean that DETECTOR cannot detect corruption of the link register since its value is never verified.

```
ldr G, [S2], #4 @ lr
bx lr
```

```
mov G, lr
```

Finally, the vulnerable section of the original code can be executed. If the vulnerable section is an exit section (i.e. it contains a return statement `bx lr` like in $\underline{VS_6}$ and $\underline{VS_7}$), the function ends, and no further steps have to be taken. If it does not, a new checkpoint is inserted like before: push the current registers to the first shadow stack and mark the new checkpoint with a nonzero compile time signature value sig_4 in `G` (label $\underline{CP_4}$).

While in this explanation, the registers `G`, `S1`, and `S2` are used to refer to the registers that DETECTOR uses for its operation, in reality, these registers are to be selected from the available general-purpose registers of the CPU. As with all SIHFT techniques, these registers should be reserved for the error detection technique and should not be used for other purposes in the protected program areas.

4.3 Evaluation

In order to situate the error detection capabilities of DETECTOR within the current SOTA, its performance is compared to both CFED and DFED techniques separately. For CFE detection, DETECTOR is compared against CFCSS [31] and RACFED [39]. For DFE detection, DETECTOR is compared against the DFED of SWIFT [47] and FDSC [45].

4.3.1 Register Availability

As mentioned in Chapter 2 the state-of-the-art techniques for DFE detection require many registers to be reserved for the shadow registers, sometimes causing the compilation to fail due to a lack of available registers. This happened with six of the eleven case studies used in this evaluation, as shown in Table 4.1. Therefore, the results of the SWIFT and FDSC techniques are not available for the CU, DIJ, and FFT data-processing case studies and for the Distr, Test, and Sort I/O-driven case studies.

Since DETECTOR only uses three registers, it does not suffer from this limitation. This enables DETECTOR to be used in all case studies, even though it is capable of detecting both CFEs and DFEs.

Table 4.1: Overview of which case studies do (✓) and do not (✗) compile when each tested SIHFT technique is applied.

Case study	Techniques				
	CFCSS	RACFED	SWIFT	FDSC	DETECTOR
Data-processing case studies					
BC	✓	✓	✓	✓	✓
BS	✓	✓	✓	✓	✓
CRC	✓	✓	✓	✓	✓
CU	✓	✓	✗	✗	✓
DIJ	✓	✓	✗	✗	✓
FFT	✓	✓	✗	✗	✓
MM	✓	✓	✓	✓	✓
QS	✓	✓	✓	✓	✓
I/O-driven case studies					
Distr	✓	✓	✗	✗	✓
Test	✓	✓	✗	✗	✓
Sort	✓	✓	✗	✗	✓

4.3.2 Fault Injection Results for the Data-Processing Case Studies

Single-bit fault injection campaigns were performed on the data-processing case studies discussed in Chapter 3 to evaluate the effectiveness of the DETECTOR technique. In the following sections, a summary of the results will be discussed to show an overview of the performance of the DETECTOR technique. An overview of the details of all fault injection campaigns can be found in Appendix A.

The results of the CFE fault injection campaign are shown in Figure 4.5. DETECTOR detects 59.3% to 94.8% of the injected CFEs, with an average of 69.6%. However, as mentioned in Chapter 3, the reduction in the number of silent data corruptions should also be taken into account. The data shows that the CFEs resulting in an SDC decrease from averagely 54.6% for unprotected case studies to 5.7% for case studies protected with DETECTOR.

Looking at the data, it is clear that DETECTOR outperforms CFCSS on all levels. CFCSS has an average error detection ratio of 46.2% and causes 12.0% of CFEs to classify as an SDC. Comparing the results to RACFED, DETECTOR averagely shows a lower error detection ratio, with RACFED averaging 79.7% detected errors. RACFED outperforms DETECTOR on seven of the eight case

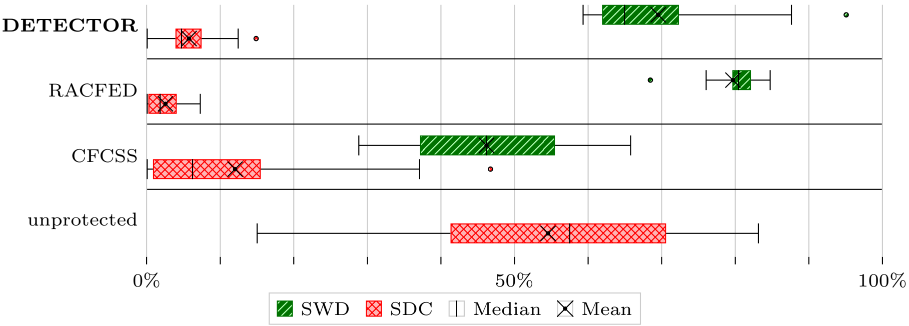


Figure 4.5: Boxplots of the CFE single-bit fault injection results performed on the data-processing case studies protected with CFCSS, RACFED, and DETECTOR on the ARMv7-M ISA, showing the error detection ratio and SDC ratio.

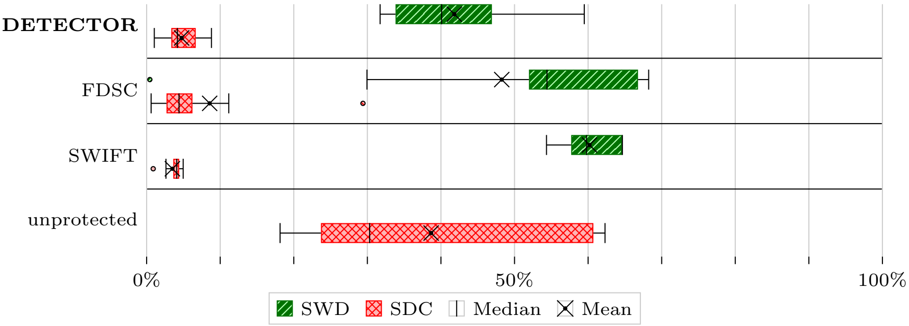


Figure 4.6: Boxplots of the DFE single-bit fault injection results performed on the data-processing case studies protected with SWIFT, FDSC, and DETECTOR on the ARMv7-M ISA, showing the error detection ratio and SDC ratio.

studies. Similarly, RACFED also shows a lower SDC percentage in six of the eight case studies, with DETECTOR only showing better results for the Bubble Sort (BS) and Cyclic Redundancy Check (CRC) case studies.

When analyzing the DFE fault injection results in Figure 4.6, a significantly lower detection ratio can be seen: 37.7% to 59.5% with an average of 41.8%. However, the SDC reduction sheds a different light, with an SDC reduction from averagely 38.7% on unprotected code to 4.7% on code protected with DETECTOR. This indicates that DETECTOR prioritizes errors that cause a corrupted output, while errors that have no effect are masked more often.

The choice to only verify the register values before the vulnerable section, as opposed to before every branch and store instruction, appears to be the cause of this behavior.

To compare DETECTOR to SWIFT and FDSC, only the BC, BS, CRC, MM, and QS case studies should be considered since these are the only case studies where all three techniques can be applied. SWIFT outperforms DETECTOR when it comes to error detection ratio, with an average of 60.2% detected errors, compared to 47.1% for DETECTOR. Only for the CRC case study does DETECTOR outperform SWIFT when it comes to the error detection ratio. The SDC reduction is on average also better for SWIFT. The SDC ratio of 49.0% is reduced to 3.4% when using SWIFT, compared to 4.7 when using DETECTOR. Here, DETECTOR scores better for two of the five case studies. A similar story can be told when comparing DETECTOR to FDSC, having an error detection ratio of 48.3%. However, FDSC's average SDC ratio is only 8.5% on average due to its poor performance on the BC case study.

4.3.3 Fault Injection Results for the I/O-Driven Case Studies

Using the HIL-based fault injection method described in Chapter 3, CFE and DFE fault injections were performed on the three I/O-driven case studies.

The results of the CFE fault injection campaigns are shown in Figure 4.7. These show that DETECTOR, while detecting on average 38.5% of the injected CFEs, does not reduce, but rather increase the SDC ratio of two of the three case studies. This result is in stark contrast to the data-processing case studies, where DETECTOR was able to reduce the SDC ratio significantly. Meanwhile, classic CFE detection techniques like CFCSS and RACFED are able to detect a large portion of the injected CFEs, showing that this issue is specific to the approach of DETECTOR.

To gain an insight into this phenomenon, the CFE fault injection results were analyzed. Each CFE can jump from and to:

- checkpoints (added by DETECTOR after each vulnerable section and at the start of the function),
- vulnerable sections,
- re-execution points (added by DETECTOR before each vulnerable section),
- other parts of the code (i.e. instructions not added by DETECTOR and not part of a vulnerable section).

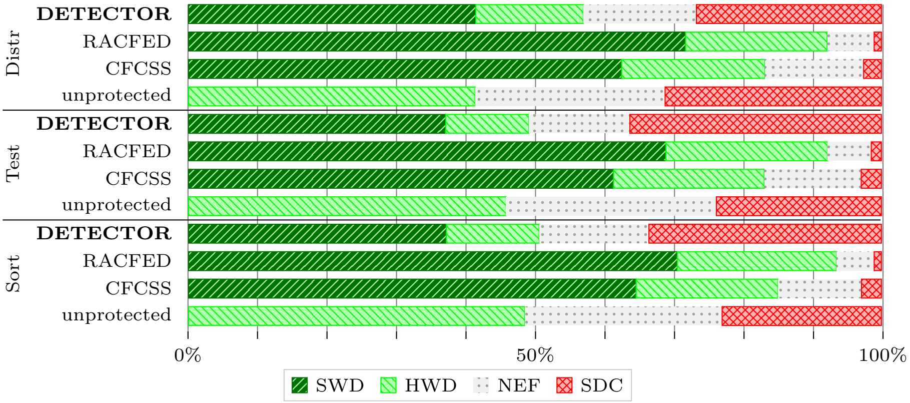


Figure 4.7: The results of the single-bit CFE fault injection campaigns on the I/O-driven case studies protected with CFCSS, RACFED, and DETECTOR on the ARMv7-M ISA.

Table 4.2: The average SDC ratio of the CFEs injected in the I/O-driven case studies, classified based on the source (from) and destination (to) location of the injected fault.

to / from	checkpoint	vulnerable section	re-execution point	other
checkpoint	**54**%	22%	39%	46%
vulnerable section	**66**%	31%	**60**%	**78**%
re-execution point	**54**%	16%	28%	31%
other	49%	20%	18%	12%

Each CFE was classified based on the source and destination location of the injected fault and for each of these classes, the SDC ratio was calculated, resulting in the matrix shown in Table 4.2.

The results show that most SDC results are caused by CFEs that jump from a vulnerable section to a non-related part of the code. Since a vulnerable section is not protected by DETECTOR, this is expected. However, the results also show that DETECTOR is not able to detect many CFEs that jump to a checkpoint. This is explained by Figure 4.8.

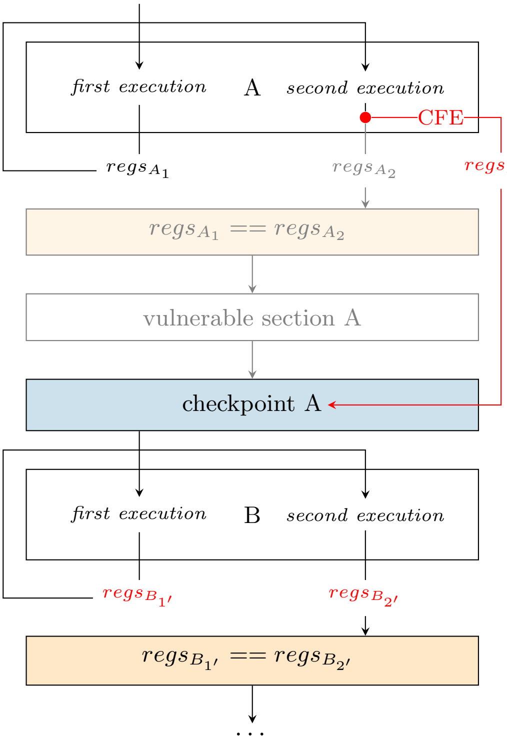


Figure 4.8: A CFE that jumps to a checkpoint can often remain uncaught by DETECTOR.

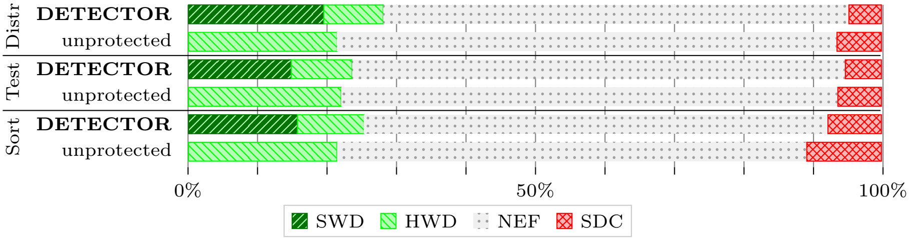


Figure 4.9: The results of the single-bit DFE fault injection campaigns on the I/O-driven case studies protected with DETECTOR on the ARMv7-M ISA.

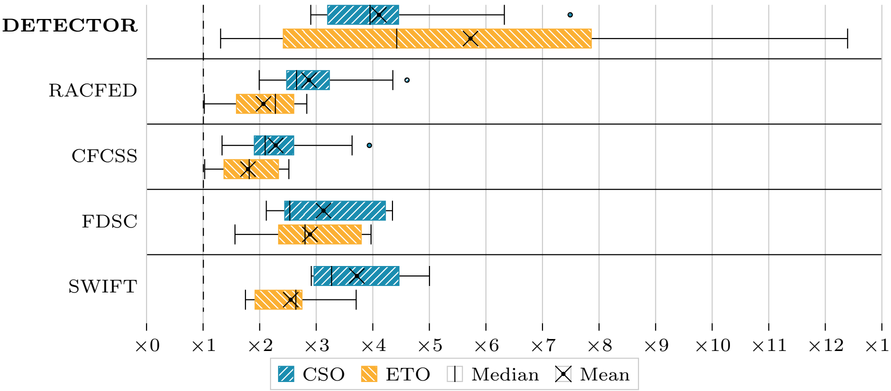


Figure 4.10: Boxplots showing the Code Size Overhead (CSO) and Execution Time Overhead (ETO) of SWIFT, FDSC, CFCSS, RACFED, and DETECTOR. The CSO is evaluated over all case studies on the Cortex-M3 processor. The ETO is only evaluated for the data-processing case studies since the ETO for the I/O-driven case studies is negligible.

When the comparison step of DETECTOR is skipped by an erroneous jump, there is a large chance that the error will remain undetected. In Figure 4.8, an erroneous jump to the checkpoint occurs during the second execution of the A block. The remainder of the program is thus executed with corrupted registers $regs_{B_1'}$. Since the next B block is executed twice with these corrupted registers, the comparison step after these two executions does not catch the error. Due to the high number of vulnerable sections – and with that, the high number of checkpoints – in the I/O-driven case studies, this issue is more prevalent in these case studies than in the data-processing case studies.

The results of the DFE fault injection campaigns are shown in Figure 4.9. Here, a small reduction in the SDC ratio is observed, from an average of 6.9% on unprotected code to 6.0% on code protected with DETECTOR.

4.3.4 Overhead Evaluation

Adding any SIHFT technique increases the code size and execution time. These overhead types were measured for all case studies. The results of the overhead measurements are shown in Figure 4.10.

As is to be expected for a technique that employs re-execution, the execution time overhead of DETECTOR is significant. For most case studies, the overhead of DETECTOR is higher than all SOTA techniques. This can be attributed to several factors. Firstly, with DETECTOR, every instruction of the protected code is re-executed. This already doubles the execution time. Secondly, DETECTOR pushes and pops to and from shadow stacks in memory regularly. This does not come cheap, as this can cause a pipeline stall, where the pipeline in a CPU is temporarily halted or delayed due to a data dependency. This further increases the execution time overhead [68]. Finally, the instructions in each checkpoint and re-execution point also add up, further increasing the ETO.

These issues are even more pronounced when vulnerable sections reside in a hot loop, which is the case for the BS, MM, and QS case studies, resulting in very high ETO numbers for these case studies.

The ETO of the I/O-driven case studies is ×1 (i.e., the execution time is the same as the original program) for all techniques (these ETO values are excluded from the boxplot in Figure 4.10). There is no measurable difference in execution time between the unprotected code and the code protected by any technique. This is because these programs spend a large percentage of their first execution waiting for physical actuators to move and sensors to trigger. The time spent waiting makes the overhead due to added instructions negligible. Furthermore, with DETECTOR, the actuators are already in the right position during the second execution, eliminating the time spent in the busy waiting loop. The code size overhead measurements of DETECTOR show that DETECTOR has a similar CSO to FDSC and SWIFT, albeit slightly higher in most cases.

Interestingly the implementation of DETECTOR on the QS case study shows a CSO of ×2.9, which is lower than SWIFT (×3.3), FDSC (×4.4), and RACFED (×3.1). Only CFCSS has a lower CSO of ×2.7. However, the ETO of DETECTOR for this case study was far out the worst of all techniques, with an ETO of ×6.4, compared to ×3.7, ×3.8, ×2.8, and ×2.8 for SWIFT, FDSC, CFCSS, and RACFED, respectively. This further establishes that the ETO of DETECTOR is highly dependent on the location where its instructions are inserted.

The results of the overhead measurements show that, just like the error detection ratio and SDC ratio, the CSO and the ETO can differ greatly depending on the target application, especially for DETECTOR. All four factors should therefore be tested and taken into consideration when choosing an error detection technique for a specific application.

4.4 Conclusion

In this chapter, the DETECTOR technique was introduced, emphasizing its approach of utilizing low-level re-execution to ensure data integrity. Unlike traditional methods that rely on extensive shadow registers, DETECTOR enhances fault tolerance through strategic re-execution.

The core concepts of DETECTOR were first explained including the identification of vulnerable sections within a program and the mechanisms for checkpoints and re-execution points. These foundational elements enable DETECTOR to periodically validate the integrity of register values, ensuring errors are detected before they can affect the program output.

Section 4.2 demonstrated how DETECTOR was implemented in ARMv7-M using the bubble sort algorithm as a concrete example. The example illustrated the insertion of checkpoints and re-execution points, as well as the processes for comparing execution states to detect discrepancies. It also shows that no matter the size or complexity of the program, DETECTOR only uses three registers for its operation. This is a significant advantage over other techniques DFED and hybrid techniques.

In Section 4.3, DETECTOR's performance was assessed through fault injection experiments and overhead measurements. The results show that DETECTOR reduces the SDCs occurring in a program with similar effectiveness to the state-of-the-art when tested on data-processing case studies. When testing on I/O-driven case studies, DETECTOR showed poor results for CFE detection, as it increased the SDC ratio for two of the three case studies. Analysis showed that this is due to CFEs that jump to a checkpoint, which can remain undetected by DETECTOR.

Section 4.3 also revealed that DETECTOR incurs a significant Execution Time Overhead (ETO), especially in cases involving hot loops. This is a direct consequence of its re-execution strategy, which, while thorough, can substantially increase the number of CPU cycles. The Code Size Overhead (CSO) was found to be comparable to other techniques, though it varied depending on the specific application.

Finally, Section 4.3 again highlighted the register availability advantage of DETECTOR over other techniques, as it requires only three registers for operation. This enables DETECTOR to be applied to all case studies, unlike other DFED techniques that fail to compile on several case studies due to a lack of available registers.

In conclusion, DETECTOR presents a viable solution for soft error detection with the notable advantage that it can be applied to a wide range of applications without the need for extensive register resources. While the technique does incur a significant execution time overhead on applications with vulnerable sections in hot loops, its effectiveness in reducing SDCs in the data-processing case studies shows that the technique is a valuable option for applications where data integrity is crucial.

Chapter 5

Optimizing the DETECTOR Technique

This chapter is based on the work published in [101] and [102]. Since publication of these papers, the results have been updated to reflect the latest version of the techniques. Hence, this chapter reflects the latest version of the techniques.

As mentioned in Chapter 4, DETECTOR uses two shadow stacks to achieve runtime redundancy. However, the continuous memory access instructions create a pipeline stall, which results in a large execution time overhead. This is especially an issue when a hot loop in the program contains vulnerable instructions. This causes a checkpoint to be inserted in the hot loop, thus, introducing many memory access instructions into the program.

To reduce this overhead and further optimize DETECTOR, two optimizations were developed. The first optimization uses the concept of selective implementation in order to reduce overhead. This is discussed in Section 5.1. The second optimization is a variation on DETECTOR that uses a parity-checking mechanism instead of the second shadow stack. This is discussed in Section 5.2.

5.1 Selective Implementation

A common way to reduce overhead in SIHFT techniques is to selectively apply the technique to only the most vulnerable parts of the program [69]. By protecting only a subset of elements, e.g., a few CPU registers or blocks of instructions, it is possible to achieve significant overhead reductions with minimal effects on overall reliability. The parts to protect should be selected based on their vulnerability and contribution to overhead. Based on this concept, a selective implementation of the DETECTOR technique was developed, called S-DETECTOR.

5.1.1 Core Concepts

The selective implementation of DETECTOR only protects a subset of the registers. In Chapter 4, the concept that registers are used to hold temporary values for computing and final values are written to memory was introduced. With this in mind, S-DETECTOR only protects the registers that are often used in memory operations.

Based on a dynamic analysis [70] of the register usage in the program, a map of how often each register is used to write to memory can be created. This analysis steps through the program and counts the number of times each register is used to write to or from memory. For loads from memory, only registers containing the source address are counted. This is because the destination register will be overwritten by the load operation. For stores, both the data and destination register are counted. The registers that interact most often with memory should be prioritized as the most vulnerable registers and should therefore be protected by S-DETECTOR.

Selecting the number of registers to protect is a trade-off between reliability and overhead. The more registers that are protected, the more reliable the technique will be, but the more overhead will be introduced. For the purpose of this thesis, the number of registers to protect are manually selected based on the results of the dynamic analysis.

5.1.2 Implementation

As explained in Chapter 4, DETECTOR uses two shadow stacks. This is shown in Figure 5.1. The first shadow stack, S_1, is used to store the register values at the checkpoints ($S_1.\text{pushAll}(registers)$). These values are restored before starting the second execution cycle ($registers \leftarrow S_1.\text{popAll}()$). These stores and writes cannot be performed selectively, as this would break the re-execution mechanism of DETECTOR.

The second shadow stack, S_2, is used to store the values calculated after the first execution cycle ($S_2.\text{pushAll}(registers)$). They are then compared to the values calculated after the second execution cycle to verify the data integrity

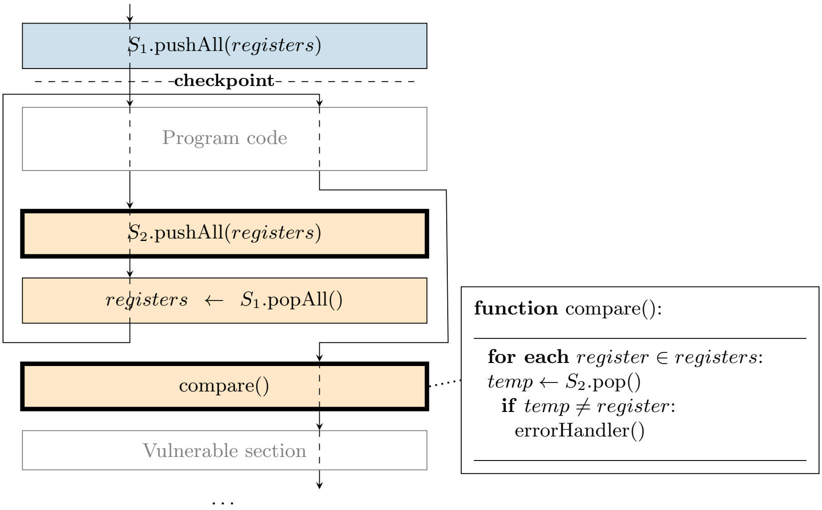

S_1.pushAll($registers$)
checkpoint
Program code
S_2.pushAll($registers$)
$registers \leftarrow S_1$.popAll()
compare()
Vulnerable section
...
function compare():
for each $register \in registers$:
$temp \leftarrow S_2$.pop()
if $temp \neq register$:
errorHandler()

Figure 5.1: The re-execution flow of a program protected with DETECTOR, showing the locations of the push and pop operations to the shadow stacks (S_1 and S_2) and the `compare` subroutine. The push and pop operations onto and from S_2 (highlighted with a thick border) can be optimized by S-DETECTOR.

in the `compare` subroutine. These are the parts that can be optimized by S-DETECTOR.

By only pushing the vulnerable registers to shadow stack S_2 after the first execution cycle, the number of memory write operations is already reduced. More importantly, however, is that this also reduces the number of CPU cycles needed in the comparison stage of DETECTOR.

In Figure 5.2, the `compare` subroutine of the full DETECTOR implementation is compared to that of the selective implementation when only protecting one register (`r0` in this case). Notice that the number of instructions for this comparison has been reduced by a factor of six. Moreover, a large amount of nonconsecutive memory read instructions (`ldr`) have been removed. Eliminating these instructions removes the multiple pipeline stalls that come with memory operations, further decreasing the amount of CPU cycles executed during the `compare` subroutine.

```
compare:
CMPr0:  ldr   G,   [S2], #4   @ r0
        cmp   r0,  G
        bne   ERR
.................................................
                 ...
.................................................
CMPr12: ldr   G,   [S2], #4   @
r12
        cmp   r12, G
        bne   ERR
.................................................
CMPsp:  ldr   G,   [S2], #4   @ sp
        cmp   sp,  G
        bne   ERR
.................................................
RET:    ldr   G,   [S2], #4   @ lr
        bx    lr
-------------------------------------------------
ERR:    bl    errorHandler
```

(a) The `compare` subroutine of DETECTOR.

```
compare:
CMPr0:  ldr   G,   [S2], #4   @ r0
        cmp   r0,  G
        bne   ERR
.................................................
RET:    ldr   G,   [S2], #4   @ lr
        bx    lr
-------------------------------------------------
ERR:    bl    errorHandler
```

(b) The `compare` subroutine of S-DETECTOR when only protecting register `r0`.

Figure 5.2: Comparison of the `compare` subroutine of DETECTOR and S-DETECTOR using the ARMv7-M assembly language.

5.1.3 Impact Analysis

To demonstrate how S-DETECTOR reduces the overhead of DETECTOR, the additional instruction cycles for each vulnerable section will be calculated. This analysis will be conducted using the ARMv7-M architecture.

Instruction Cycles Added by DETECTOR

All pushAll() and popAll() operations of DETECTOR take $N + 1$ instruction cycles, with N being the number of registers in the register list to be pushed [68]. The real culprit, however, is the `compare` subroutine. For each register to be checked, a load instruction, a compare instruction, and a conditional *not equal* branch instruction are added. Since each load instruction takes two instruction cycles, this adds up to a total of $4N$ additional instruction cycles for the `compare` subroutine. After adding up all the remaining instructions that are added for the DETECTOR technique, the total amount of added instruction cycles for each vulnerable section is:

$$I^{+}_{VS_{detector}} = 7N + 4P + C + 14 \tag{5.1}$$
$$\text{where } \{N, P \in \mathbb{N} \mid P \leq 3 \mid C \in \mathbb{N}_0 \mid C \leq 3\}$$

Appendix B.1 provides a more detailed explanation of how this formula is derived. In Equation (5.1), N is the aforementioned number of registers to be pushed or popped, P is the number of cycles required for a pipeline refill, ranging from 1 to 3 [68], and C varies between 0 and 3, depending on the number of possible re-execution paths for a single vulnerable block. In the case of DETECTOR on an ARMv7-M-based processor, $N = 12$, and thus results in Equation (5.2):

$$I^{+}_{VS_{detector}} \in [102, 113]\,|_{N=12} \tag{5.2}$$

Instruction Cycles Added by S-DETECTOR

S-DETECTOR only protects the most vulnerable registers. Therefore, the `compare` subroutine of S-DETECTOR only adds $4M$ instruction cycles, with M being the number of registers to be protected. The same goes for the push to shadow stack $S2$ during the re-execution point, changing from $N + 1$ to $M + 1$. This results in Equation (5.3):

$$I^{+}_{VS_{s-detector}} = 5M + 2N + 4P + C + 14 \tag{5.3}$$
$$\text{where } \{M, N, P \in \mathbb{N} \mid P \leq 3 \mid C \in \mathbb{N}_0 \mid C \leq 3\}$$

Like before, P is the number of cycles required for a pipeline refill, ranging from 1 to 3 [68] and C varies between 0 and 3, depending on the number of possible re-execution paths for a single vulnerable block. In the case of DETECTOR on an ARMv7-M-based processor, $N = 12$. Thus, if only one register is protected $(M = 1)$, the result is Equation (5.4):

$$I^{+}_{VS_{s-detector}} \in [47, 58]\,|_{M=1 \ \mid \ N=12} \tag{5.4}$$

For each additional protected register, the added instruction cycles per vulnerable section increase by five. Apart from the derivation of Equation (5.3), Appendix B.2 provides Table B.4, showing how many instruction cycles are added depending on the number of protected registers.

5.2 Parity Checking

Another approach to optimize the DETECTOR technique is changing the way that the data integrity is verified. Instead of storing all calculated registers to a

Algorithm 5.1 The P-DETECTOR operations performed during a re-execution point.

```
1:  if $G \neq 0$ then
2:      for all $register \in registers$ do
3:          $P \leftarrow P \oplus register$
4:      $registers \leftarrow S$.popAll()
5:      for all $sig_i \in sources$ do
6:          if $G = sig_i$ then
7:              $G \leftarrow 0$
8:              goto $checkpointLabel_i$
9:  else
10:     for all $register \in registers$ do
11:         $P \leftarrow P \oplus register$
12:     if $P \neq 0$ then
13:         errorHandler()
```

second shadow stack and comparing them one by one after the re-execution phase, a parity-checking mechanism can be used. This variation on the DETECTOR technique is called P-DETECTOR.

5.2.1 Core Concepts

The P-DETECTOR technique removes the second shadow stack `S2` and its associated push and pop operations, including the `compare()` subroutine. This is replaced with parity calculations and a parity check. Hence, the re-execution point operations are now modified to Algorithm 5.1. The first (and now only) shadow stack with stack pointer S (previously S_1) is still used, together with the signature variable G.

Just like with DETECTOR, operation 1 through 8 are performed after the first execution because of the nonzero value of G. However, instead of pushing all registers to a shadow stack, an *exclusive or* ($\oplus$) operation $P \leftarrow P \oplus register$ is performed for each register (lines 2 and 3). In this way, the parity of all registers is calculated. Next, just like with DETECTOR, the original register values are restored from the shadow stack, the signature variable is set to zero, and the program jumps to the checkpoint label $checkpointLabel_i$ (lines 4 through 8).

After the second execution, the same parity calculation is performed again (lines 10 and 11). Note that now, however, the parity variable P already holds the calculated parity of the first execution. Since register values should agree after two error-free executions, the $parityValue$ should be zero after the second execution ($A \oplus A = 0$ for any value A). This is verified in lines 12 through 13.

Since the equality of the register state is now verified by comparing the parity, the possibility exists that two unequal register states result in the same parity. This can happen if the same bit is corrupted an even number of times in multiple registers. For example, if register $A = 0b10$ gets corrupted to $A' = 0b11$ and register $B = 0b01$ gets corrupted to $B' = 0b00$, parity of these registers will be equal, like shown in Equation (5.5).

$$\begin{aligned} A \oplus B &= 0b10 \oplus 0b01 = 0b11 \\ A' \oplus B' &= 0b11 \oplus 0b00 = 0b11 \\ \Rightarrow A \oplus B &= A' \oplus B' \end{aligned} \tag{5.5}$$

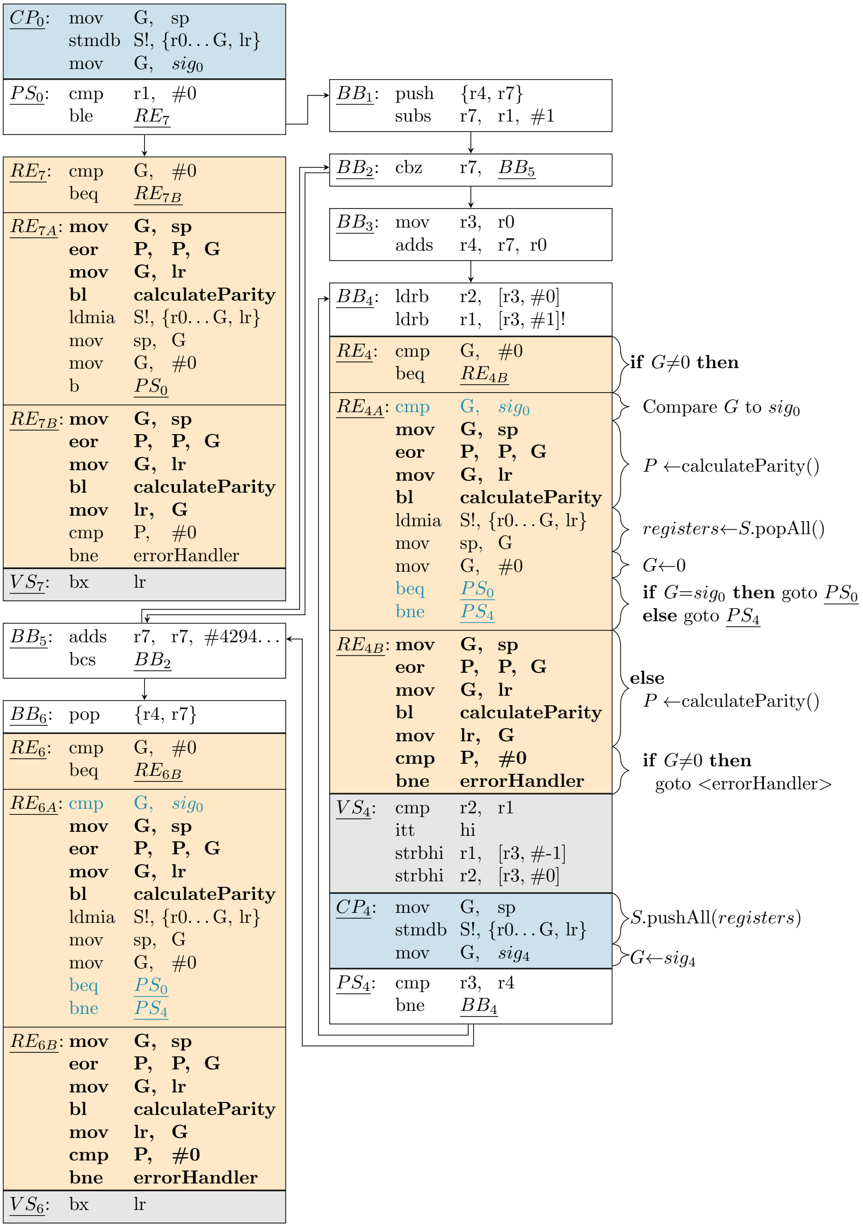
5.2.2 Implementation

Since the P-DETECTOR technique is a variation of the DETECTOR technique, its implementation is largely the same. The same example program as in Chapter 4 will be used to explain the low-level implementation of P-DETECTOR (Figure 4.2). The P-DETECTOR technique implemented on the bubble sort algorithm is shown in Figure 5.3. In this figure, the instructions that differ from the DETECTOR implementation are marked in bold.

The instructions added in the checkpoint are the same as in the DETECTOR implementation. P-DETECTOR still uses one shadow stack to store the register values at the checkpoints and the signature is updated to a unique nonzero signature value. Once again, `G` is used as the signature register. Since there is now only one shadow stack, its pointer is now stored in `S`.

The code added in the re-execution point starts the same: the signature register is checked against zero to determine if the program is in its first or second execution stage (e.g., at label $\underline{BB_7}$). However, instead of pushing the register values to the second shadow stack, the `calculateParity` subroutine is called. This subroutine is shown in Figure 5.4.

An exclusive or (`eor`) operation is performed over each CPU register, the result of which is stored in a parity register (`P` in this instance). However, since the ARM assembler again does not allow the stack pointer to be an operand of the exclusive or operation, the stack pointer is temporarily saved to `G`, which is then used for the exclusive or instead. This is done before the call to the subroutine since the link register is also stored in the same `G` register.



```
CP0:   mov    G,  sp
       stmdb  S!, {r0...G, lr}
       mov    G,  sig0
PS0:   cmp    r1, #0
       ble    RE7
RE7:   cmp    G,  #0
       beq    RE7B
RE7A:  mov    G,  sp
       eor    P,  P,  G
       mov    G,  lr
       bl     calculateParity
       ldmia  S!, {r0...G, lr}
       mov    sp, G
       mov    G,  #0
       b      PS0
RE7B:  mov    G,  sp
       eor    P,  P,  G
       mov    G,  lr
       bl     calculateParity
       mov    lr, G
       cmp    P,  #0
       bne    errorHandler
VS7:   bx     lr
BB5:   adds   r7, r7, #4294...
       bcs    BB2
BB6:   pop    {r4, r7}
RE6:   cmp    G,  #0
       beq    RE6B
RE6A:  cmp    G,  sig0
       mov    G,  sp
       eor    P,  P,  G
       mov    G,  lr
       bl     calculateParity
       ldmia  S!, {r0...G, lr}
       mov    sp, G
       mov    G,  #0
       beq    PS0
       bne    PS4
RE6B:  mov    G,  sp
       eor    P,  P,  G
       mov    G,  lr
       bl     calculateParity
       mov    lr, G
       cmp    P,  #0
       bne    errorHandler
VS6:   bx     lr
```

```
BB1:   push   {r4, r7}
       subs   r7, r1, #1
BB2:   cbz    r7, BB5
BB3:   mov    r3, r0
       adds   r4, r7, r0
BB4:   ldrb   r2, [r3, #0]
       ldrb   r1, [r3, #1]!
RE4:   cmp    G,  #0
       beq    RE4B
RE4A:  cmp    G,  sig0
       mov    G,  sp
       eor    P,  P,  G
       mov    G,  lr
       bl     calculateParity
       ldmia  S!, {r0...G, lr}
       mov    sp, G
       mov    G,  #0
       beq    PS0
       bne    PS4
RE4B:  mov    G,  sp
       eor    P,  P,  G
       mov    G,  lr
       bl     calculateParity
       mov    lr, G
       cmp    P,  #0
       bne    errorHandler
VS4:   cmp    r2, r1
       itt    hi
       strbhi r1, [r3, #-1]
       strbhi r2, [r3, #0]
CP4:   mov    G,  sp
       stmdb  S!, {r0...G, lr}
       mov    G,  sig4
PS4:   cmp    r3, r4
       bne    BB4
```

Figure 5.3: The P-DETECTOR technique implemented on the bubble sort algorithm. The instructions that differ from the DETECTOR technique are indicated in bold.

```
calculateParity:
EOR:    eor  P,  P,  r0
        eor  P,  P,  r1
        eor  P,  P,  r2
        ...
        eor  P,  P,  r12
.......................................
RET:    bx   lr
```

Figure 5.4: The `calculateParity` subroutine of the P-DETECTOR technique in ARMv7-M assembly language.

Next, the register values from before the first execution are popped from the shadow stack and the signature register is updated to zero to mark the start of the second execution, before jumping to the checkpoint.

After the second execution, the program will jump to the second part of the re-execution point (e.g., label $\underline{RE_{7B}}$) to again calculate the parity. As mentioned before, the result of this second parity calculation should be zero for an error-free execution. If this is not the case, the error handler will be called at the end of the re-execution point.

5.2.3 Impact Analysis

Instruction Cycles Added by P-DETECTOR

By replacing the push and pop operations to and from the second shadow stack with a parity calculation subroutine, the added instruction cycles are reduced significantly. When again looking at an ARMv7-M-based processor, the push and pop instructions to the first (now only) shadow stack remain the same, both adding $N + 1$ instruction cycles. The second push to the second shadow stack and the `compare` subroutine are both replaced by a more efficient `calculateParity` subroutine, now only adding N instruction cycles for each call to `calculateParity` (instead of $4N$). Adding all remaining instructions for the P-DETECTOR technique results in Equations (5.6) and (5.7):

$$I^{+}_{VS_{p-detector}} = 4N + 6P + C + 22 \tag{5.6}$$

$$\text{where } \{N, P \in \mathbb{N} \mid P \leq 3 \mid C \in \mathbb{N}_0 \mid C \leq 3\}$$

$$I^{+}_{VS_{p-detector}} \in [76, 91]\,|_{N=12} \tag{5.7}$$

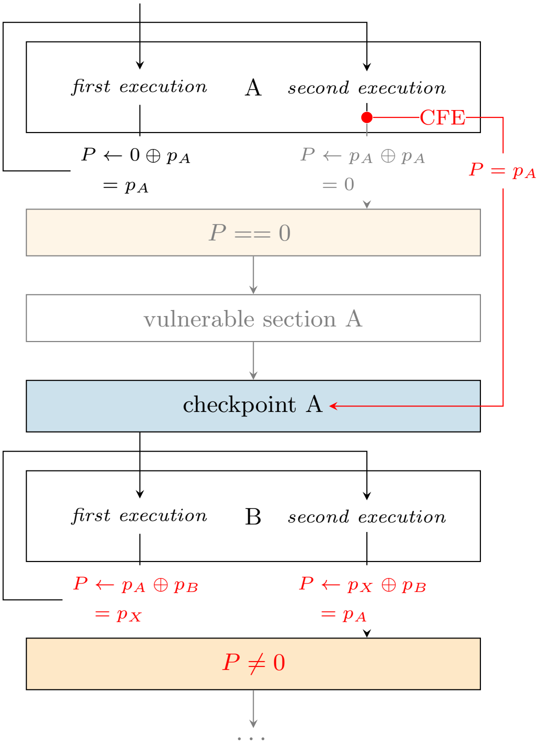


Figure 5.5: In contrast to DETECTOR, control flow errors that jump to a checkpoint can often be caught by P-DETECTOR.

A more detailed explanation of the derivation of Equation (5.6) is attached in Appendix B.3.

Improved CFE Detection

An additional advantage that comes with this approach is an improved CFE detection capability. As established in Chapter 4, when the comparison step of DETECTOR is skipped by an erroneous jump, there is a large chance that the error will remain undetected. This is because DETECTOR is not able to detect a fault if the first and second execution cycle both contain the same corrupted register values.

Figure 5.5 shows how P-DETECTOR behaves in the same scenario. After the first execution of the A block, a parity of p_A is calculated. Since the CFE occurs during the second execution of the A block, the P register still holds this value at the start of the B block. As a result of this, the parity register (P) will hold this nonzero value after the two execution cycles of the B block, resulting in the error handler being triggered in the next step. The persistence of the parity value after the validation step causes a control flow error skipping the verification step to have a high chance of being caught at a later point in the program. This makes P-DETECTOR significantly more effective for these types of CFEs.

Protected Link Register

A final improvement of P-DETECTOR is that it can protect the link register. DETECTOR was unable to do so since the function call to the `compare` subroutine changes the link register to the return address of the subroutine. P-DETECTOR works around this issue by temporarily assigning the link register to the signature register before the function call to `calculateParity`. This feature causes P-DETECTOR to have a slight advantage in data flow error detection.

5.3 Evaluation

Like DETECTOR, S-DETECTOR and P-DETECTOR are evaluated through CFE and DFE fault injection campaigns on the data-processing and I/O-driven case studies described in Chapter 3. In the following sections, the results of S-DETECTOR and P-DETECTOR are compared to the results of DETECTOR. A full overview of the results, including the comparison with CFCSS, RACFED, SWIFT, and FDSC can be found in Appendix A.

5.3.1 Dynamic Register Analysis

Before evaluating S-DETECTOR, the dynamic register analysis should be performed on all case studies to determine which registers to protect. The results of this analysis are shown in Appendix C.

For each case study, one to seven registers are selected for protection, based on how often they interact with memory. The only exception is the bit count program, which does not interact with memory. For this case study, the register (`r0`) was manually selected based on the program's structure.

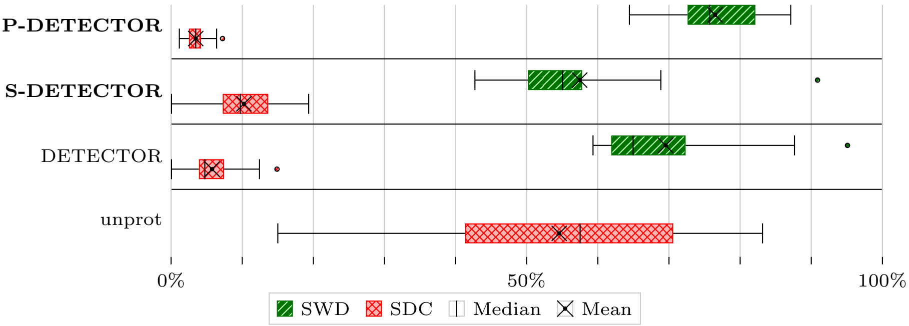


Figure 5.6: Boxplots of the CFE single-bit fault injection results performed on the data-processing case studies protected with DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA, showing the error detection ratio and SDC ratio.

5.3.2 Fault Injection Results for the Data-Processing Case Studies

Before discussing how S-DETECTOR and P-DETECTOR impact the overhead of the DETECTOR technique, the results of the fault injection campaigns will first be discussed. Figure 5.6 shows the results of the CFE fault injection campaign on the data-processing case studies.

As expected, the selective implementation of S-DETECTOR causes the error detection ratio and SDC ratio to worsen compared to DETECTOR. This varies between case studies, depending on how many and which registers are protected and how much the unprotected registers eventually influence the program execution. There will always be a trade-off between the number of registers protected and the error detection capability of the technique.

When looking at P-DETECTOR, the results show an improvement over the original DETECTOR technique, with the error detection ratio increased to 76.5% and the SDC ratio decreased to 3.4%. Detailed analysis confirms that this is indeed due to the advantages of the double parity calculations described in Section 5.2.3.

Figure 5.7 show the results of the DFE fault injection campaigns on the data-processing case studies. The results of the S-DETECTOR technique show a similar image as those from the CFE fault injection results: the error detection ratio and SDC ratio are worse than those of DETECTOR. A notable exception here is the DIJ case study, where even though the error detection ratio has

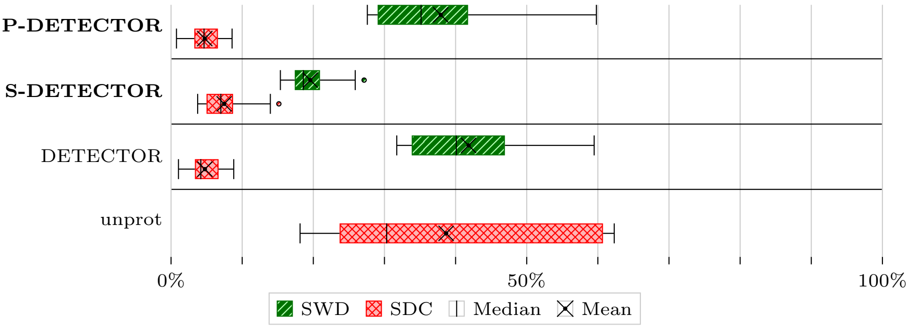


Figure 5.7: Boxplots of the DFE single-bit fault injection results performed on the data-processing case studies protected with DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA, showing the error detection ratio and SDC ratio.

decreased compared to DETECTOR (from 34% to 19%), the SDC ratio has also decreased (from 6.3% to 4.9%). This shows that in certain cases, a selective implementation can be highly effective.

The error detection ratio of the P-DETECTOR technique is slightly lower than that of DETECTOR. This is to be expected since some injected faults can result in an equal parity value, even when the actual register values are not the same. The difference is, however, so small that this is not reflected in the SDC ratio, which is equal to that of DETECTOR. This higher SDC ratio is also the result of P-DETECTOR being able to protect the link register, in contrast to DETECTOR.

5.3.3 Fault Injection Results for the I/O-Driven Case Studies

The results of the CFE fault injection campaign on the I/O-driven case studies are shown in Figure 5.8. As expected S-DETECTOR shows a decrease in error detection ratio and an increase in SDC ratio compared to DETECTOR. Additionally, the same disappointing results from DETECTOR, namely an increased SDC ratio for two of the three I/O-driven case studies, are also present in the S-DETECTOR results. We hypothesized that P-DETECTOR would largely mitigate this issue by catching more erroneous jumps that skip the assertion step of the technique. The results show that this hypothesis is correct, with the error detection ratio increased to an average of 47.4% and, more importantly, the SDC ratio decreased to an average of 11.5%.

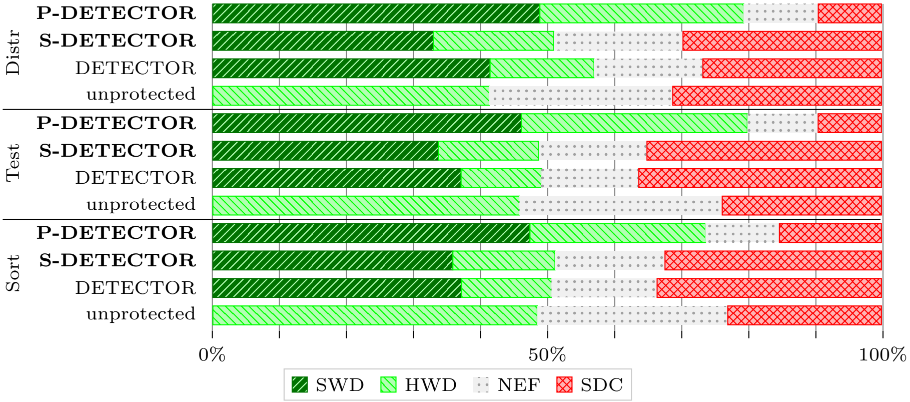


Figure 5.8: The results of the single-bit CFE fault injection campaigns on the I/O-driven case studies protected with DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA.

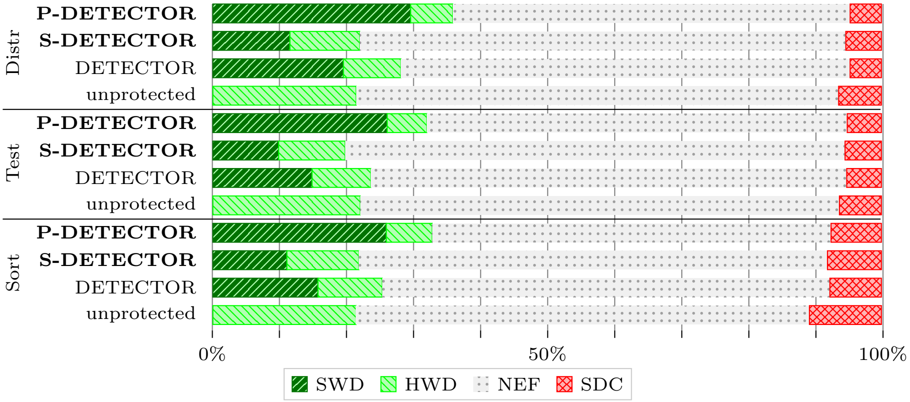


Figure 5.9: The results of the single-bit CFE fault injection campaigns on the I/O-driven case studies protected with DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA.

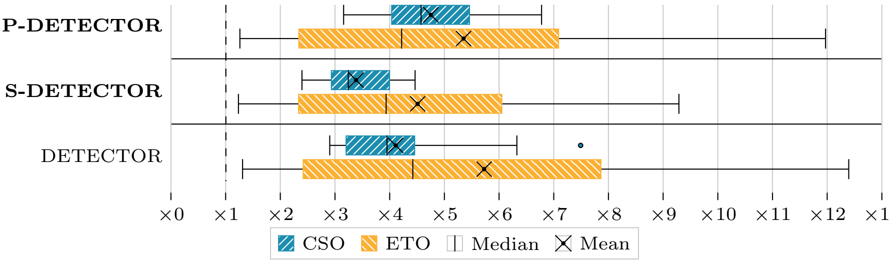


Figure 5.10: Boxplots showing the Code Size Overhead (CSO) and Execution Time Overhead (ETO) of DETECTOR, S-DETECTOR, and P-DETECTOR on the Cortex-M3 processor.

The results of the DFE fault injection campaign on the I/O-driven case studies shown in Figure 5.8 show a similar image as with the data-processing case studies. However, since the effect of DFEs on the I/O-driven case studies is already quite low, the effects of S-DETECTOR and P-DETECTOR are less significant than with the data-processing case studies.

5.3.4 Overhead Evaluation

Figure 5.10 shows the overhead evaluation of S-DETECTOR and P-DETECTOR compared to DETECTOR and other state-of-the-art techniques.

Looking at the results for S-DETECTOR, it is apparent that the selective approach of S-DETECTOR significantly reduces the ETO of the different case studies. However, the overhead is still highly dependent on the case study on which the technique is applied, which can be seen by the high spread in the ETO boxplot, still suffering from high overheads when a vulnerable section is located in a hot loop.

As is to be expected, the effect of the selective approach of S-DETECTOR is less significant when looking at code size overhead. This is because the comparison stage of DETECTOR and P-DETECTOR is implemented as a single subroutine. Reducing the size of this subroutine reduces the dynamic (i.e. executed) instructions significantly, but does not affect the static instructions nearly as much. Only for small case studies, where the comparison stage is large compared to the rest of the program, does the code size overhead of S-DETECTOR show a significant reduction.

Apart from the improvement in CFE detection, the main focus of P-DETECTOR was to reduce the execution time overhead of DETECTOR. Figure 5.10 shows

P-DETECTOR does indeed lower this overhead. However, the overhead is still significantly higher than the SOTA for the case studies that execute a lot of vulnerable sections. In most cases, the CSO of P-DETECTOR is higher than that of DETECTOR. This is because P-DETECTOR adds more instructions per vulnerable section.

5.4 Conclusion

In this chapter, two optimizations of the DETECTOR technique were presented: S-DETECTOR and P-DETECTOR.

S-DETECTOR is a selective implementation of DETECTOR that only protects a subset of the registers. More specifically, only the registers that most often interact with memory are protected. This approach makes a trade-off between the error detection capability of the technique and the overhead it introduces. This is backed up by the fault injection results and overhead analysis presented in this chapter.

P-DETECTOR is an optimization of DETECTOR that replaces the memory operations in the comparison stage with a parity calculation. This not only reduces the execution time overhead but also improves the control flow error detection capabilities.

For the case studies where vulnerable sections are present in a hot loop, the ETO of the DETECTOR techniques is still significantly higher than the SOTA, even when using the optimizations presented in this chapter. Still, just like DETECTOR, S-DETECTOR and P-DETECTOR are able to detect both CFEs and DFEs while only utilizing three registers, making them a valuable alternative when classical SIHFT techniques fail due to a lack of register availability.

Chapter 6

The RISC-V architecture

While the previous chapter optimized DETECTOR by adapting the technique itself, another optimization possibility is to explore hardware support. The possibility to add custom instructions to the RISC-V ISA to support SIHFT techniques will be explored. Before diving into these custom instructions, however, this chapter introduces the RISC-V instruction set architecture and how its modular design enables the development of custom processors for a variety of applications. Section 6.1 first introduces RISC-V ISA and discusses its modular design. Next, Section 6.2 explains how RISC-V utilizes extensions to customize the architecture for various end markets. This is followed by Sections 6.3 and 6.4, which delve deeper into the general-purpose registers and instruction formats used by RISC-V. Next, Section 6.5 describes how the RISC-V instruction set architecture can be implemented in hardware. Finally, Section 6.6 explains how fault tolerance is already present in the state-of-the-art and briefly discusses how this is implemented.

6.1 What is RISC-V?

Figure 6.1: The RISC-V logo

RISC-V (pronounced “risk-five”) is an open-source instruction set architecture used to develop custom processors for a variety of applications, from embedded designs to supercomputers. Originating from the University of California, Berkeley, RISC-V represents the fifth generation of processors based on the Reduced Instruction Set Computer (RISC)

Table 6.1: The base integer instruction sets of the RISC-V standard.

Name	Size	Registers	Version	Status
RV32I	32-bit	32	2.1	Ratified
RV32E	32-bit	16	2.0	Ratified
RV64I	64-bit	32	2.1	Ratified
RV64E	64-bit	16	2.0	Ratified
RV128I	128-bit	32	1.7	Draft

concept. Unlike proprietary architectures, RISC-V is an open-source ISA, enabling developers to adapt and extend without licensing limitations. Its openness and technical strengths have led to its widespread adoption in recent years. The standard is now managed by the nonprofit RISC-V International organization, which reported that more than 13 billion chips containing RISC-V cores had shipped by the end of 2023 [71]. Many implementations of RISC-V are available, both as open-source cores [72, 73] and as commercial IP products [74, 75].

Another key feature of RISC-V is its modular design. The RISC-V standard features a small core set of instructions on which all RISC-V are based. Its optional extensions enable designers to customize the architecture for various end markets. This means that designers can tailor their processors to meet specific application requirements, optimizing power, performance, and area by selecting which modules to use from the standardized extensions. Additionally, RISC-V provides the possibility to create custom nonstandard extensions to the ISA. This flexibility enables designers to create highly customized processors that are optimized for their specific applications, rather than having to use a one-size-fits-all solution.

These factors lead to an increasing interest from industry, with the RISC-V IP expected to grow at a compounded annual growth rate of more than 40% between 2022 and 2030 according to a market report from the SDH Group [76]. This growth is largely from the three main market segments interested in the architecture: IP providers who can offer their own designs, System on a Chip (SoC) teams using commercial IP, and designers building custom RISC-V processor-based SoCs.

Table 6.2: Standard unprivileged extensions of RISC-V.

Subset	Name	Implies
Integer Multiplication and Division	M	Zmmul
Atomics	A	-
Single-Precision Floating-Point	F	Zicsr
Double-Precision Floating-Point	D	F
General (short for IMAFDZicsr_Zifencei)	G	IMAFDZicsrZifencei
Quad-Precision Floating-Point	Q	D
16-bit Compressed Instructions	C	-
Bit Manipulation	B	-
Packed Single Instruction, Multiple Data	P	-
Vector	V	D
Hypervisor	H	-
Integer Multiplication	Zmmul	-
Control and Status Register Access	Zicsr	-
Instruction-Fetch Fence	Zifencei	-
Total Store Ordering	Ztso	-

6.2 Extensions

Although it is common to speak of *the* RISC-V ISA, RISC-V is more accurately described as a family of related ISAs, of which there are currently four ratified (finalized) and one draft base ISAs. The base ISAs, listed in Table 6.1 are characterized by the width of the integer registers (and the corresponding size of the address space) and by the number of general-purpose CPU registers. The two primary base integer variants, RV32I and RV64I, provide 32-bit or 64-bit address spaces respectively with each having 32 general purpose CPU registers [77]. Additionally, their RV32E and RV64E variants are defined to have 16 general purpose CPU registers, which can be useful for smaller microcontrollers.

Each base integer ISA can be enhanced with one or more optional instruction-set extensions, which can be classified as standard, custom, or non-conforming.

Standard extensions are given a name consisting of a single letter or using a single "Z", followed by an alphabetical name. Table 6.2 lists some standard extensions that are currently defined in the RISC-V ISA specification. Any RISC-V instruction-set variant can be described by concatenating the base integer prefix with the names of the included extensions, e.g., "RV32IMAFD". Underscores can be added to separate extensions for clarity.

Some ISA extensions depend on the presence of other extensions, e.g., "D" depends on “F” and “F” depends on “Zicsr”. These dependencies may be omitted from the name as they are implied. For example, “RV32IMAFDQZicsr_Zifencei” can be shortened to “RV32GQ”.

Optionally, extension version numbers can be placed following the extension or base name divided into major and minor version numbers, separated by a “p”. For example, RV32I2p3M2 indicates version 2.3 of the RV32I base ISA and version 2(.0) of the M extension.

Non-standard custom extensions are named using a single “X” followed by an alphabetical name. These extensions are not part of the official RISC-V ISA specification but its guidelines. The RISC-V supports custom extensions by guaranteeing that portions of the encoding space are never used by standard extensions.

The RISC-V ISA supports these nonstandard extensions by defining two opcodes, namely opcodes `0x0b` and `0x2b`, that are reserved for custom extensions and will be avoided by future standard extensions. Additionally, opcodes `0x5b` and `0x7b` are reserved for future use by RV128 but will otherwise be avoided for standard extensions and can therefore also be used for custom instruction-set extensions for the RV32I and RV64I base ISAs.

Nonconforming extensions go beyond the specifications and guidelines of the RISC-V ISA and are not guaranteed to be compatible with current or future RISC-V base ISAs or extensions.

6.3 Registers

Table 6.3 shows the unprivileged register state for the base integer ISAs (RV32I and RV64I) [77]. The 32 `x` registers are each 32 or 64 bits wide, depending on the ISA variant. Register `x0` is hardwired to zero: writing to this register will have no effect, and reading from it will always return zero. When using the RV32E or RV64E variants, only the lower 16 registers (`x0`-`x15`) are available.

RISC-V does not specify a dedicated stack pointer or subroutine return address link register but allows any `x` register to be used for these purposes. However, `x1` is commonly used to hold the return address for a call (with `x5` available as an alternate link register) and `x2` is commonly used as the stack pointer. Additionally, all registers have alternate Application Binary Interface (ABI) names, which are used in the RISC-V assembly language.

Table 6.3: Unprivileged registers of the RISC-V base integer ISAs.

Register	ABI Name	Description
`x0`	`zero`	Hardwired to zero
`x1`	`ra`	Return address
`x2`	`sp`	Stack pointer
`x3`	`gp`	Global pointer
`x4`	`tp`	Thread pointer
`x5`	`t0`	Temporary/alternate link register
`x6-x7`	`t1-t2`	Temporary registers
`x8`	`s0/fp`	Saved register/frame pointer
`x9`	`s1`	Saved register
`x10-x11`	`a0-a1`	Function arguments/return values
`x12-x17`	`a2-a7`	Function arguments
`x18-x27`	`s2-s11`	Saved registers
`x28-x31`	`t3-t6`	Temporary registers

6.4 Instruction Formats

The RV32I ISA defines four base instruction formats: R, I, S, and U [77]. They define how the instruction fields are encoded. These formats are shown in Figure 6.2. The R-type format is used to describe register-register operations, such as the `ADD rd, rd, rs1` and `XOR rd, rs1, rs2` instructions. Similarly, I-type instructions are used for register-immediate operations, such as `ADDI rd, rs, imm` and `XORI rd, rs, imm`. An immediate value is a value that is decoded directly from the machine code from an operand field, rather than being read from a register or loaded from memory. S-type instructions are used for instructions that need two source registers and an immediate value, such as the store instruction `SW rs2, imm(rs1)`, where a source register (`rs2`) is written to the address held by `rs1`, incremented by a signed offset (`imm`). Finally, the U-type instruction format is used to describe purely immediate operations that do not require any source registers, such as the `LUI rd, imm` (load upper immediate) instruction, which loads an immediate value into the upper 20 bits of a register.

Notice that most instruction formats have function fields. These fields can be used in tandem with the opcode to determine the exact operation to be performed. For example, all I-type instructions used by the RV32I base ISA share the same opcode but are differentiated by the `funct3` field. Therefore, these fields are often referred to as minor opcodes.

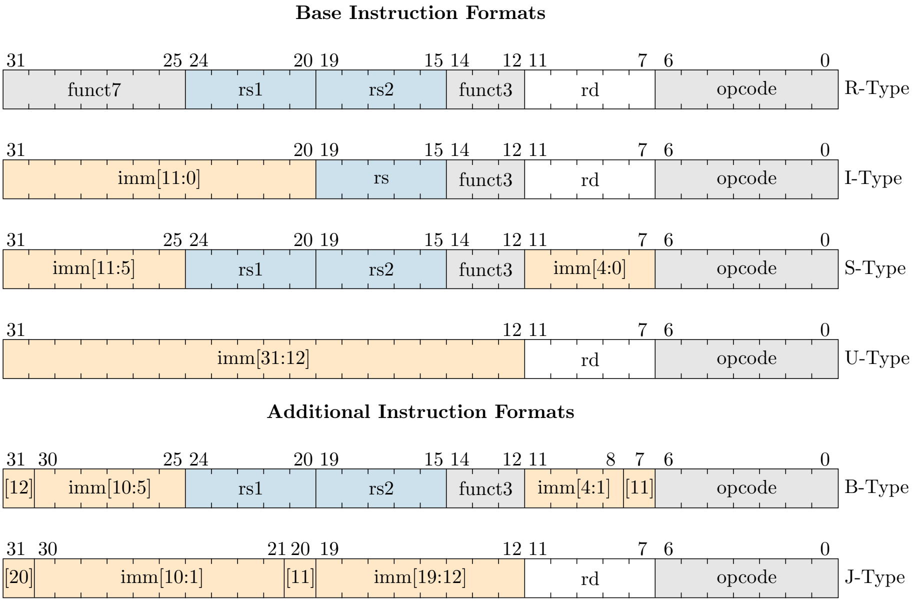

Base Instruction Formats
31 25 24 20 19 15 14 12 11 7 6 0
funct7 rs1 rs2 funct3 rd opcode R-Type
31 20 19 15 14 12 11 7 6 0
imm[11:0] rs funct3 rd opcode I-Type
31 25 24 20 19 15 14 12 11 7 6 0
imm[11:5] rs1 rs2 funct3 imm[4:0] opcode S-Type
31 12 11 7 6 0
imm[31:12] rd opcode U-Type
Additional Instruction Formats
31 30 25 24 20 19 15 14 12 11 8 7 6 0
[12] imm[10:5] rs1 rs2 funct3 imm[4:1] [11] opcode B-Type
31 30 21 20 19 12 11 7 6 0
[20] imm[10:1] [11] imm[19:12] rd opcode J-Type

Figure 6.2: The six instruction formats of the RISC-V instruction set architecture.

Figure 6.2 shows two additional instruction formats, namely the B- and J-type. These solely differ from the S- and U-type formats by the handling of immediates. The B-type format is used to describe conditional branch instructions and the J-type format for unconditional jump instructions. Hence, the 12-bit immediate field of this format encodes branch offsets. In the base RISC-V ISAs, all instructions are aligned to 32-bit boundaries. However, the ‘C’-extension allows for 16-bit compressed instructions, meaning that only a 16-bit alignment is guaranteed. Therefore, the offset of each branch instruction will be a multiple of two bytes. To accommodate this, the immediate field of the B-type format is encoded in multiples of two by keeping the middle bits (`imm[10:1]`) and the sign bit (`imm[12]`) but replacing the lowest most bit of the S format with a high-order bit in the B format (`imm[11]`). A similar strategy is employed for the J-type instruction, shifting the immediate left by 1 bit (`imm[20:1]`). The placement of bits in the U- and J-type immediates maximizes overlap with other formats.

6.5 From ISA Definition to Hardware Implementation

While the RISC-V instruction set architecture specifies the instructions that a processor must support, it does not define how these instructions are implemented. RISC-V implementations refer to the different methods by which the RISC-V ISA is realized in hardware. These implementations vary from *soft-core* designs, which run on programmable logic devices like Field-Programmable Gate Arrays (FPGAs), to *hard-core* designs embedded in custom silicon chips [78]. Each implementation type offers distinct advantages and drawbacks, influenced by considerations like performance, power efficiency, and development cost.

Soft-core RISC-V implementations are the most approachable implementations of RISC-V cores. Since they are designed to run on programmable logic devices like FPGAs their implementation can be easily modified and reconfigured to meet specific design requirements. This flexibility makes them ideal for prototyping and research purposes. It also makes the implementation much more cost-effective than creating a custom silicon chip, especially for small-scale projects or proof-of-concept designs. Additionally, the fact that these designs do not require the lengthy fabrication process associated with custom silicon chips, means that soft-core RISC-V designs have a far faster time-to-market time. However, soft-core implementations are generally slower and less power-efficient than hard-core implementations. One example of such a soft-core RISC-V implementation is the Proteus RISC-V core developed by Bognar et al. [79]. This FPGA implementation in SpinalHDL uses a plugin system to make the processor easily configurable and extensible.

Hard-core RISC-V implementations are designs that are integrated into custom silicon chips, such as Application-Specific Integrated Circuits (ASICs) or SoC devices. These implementations offer higher performance and lower power consumption than soft-core implementations. However, the cost of chip fabrication, testing, and packaging is significant, making this form of implementation far more expensive than soft-lcore designs. The design, production, and testing process for silicon chips is also very time-consuming, resulting in a longer time-to-market time. Therefore, hard-core RISC-V implementations are generally only used for large-scale commercial projects where the performance and power efficiency benefits outweigh the cost and time drawbacks.

6.6 Fault-Tolerant RISC-V Cores

The free and open character of RISC-V lowers the barrier to implementing custom modifications, such as adding fault tolerance features. This research area largely focuses on the use of redundant hardware, namely Dual or Triple Modular Redundancy (DMR/TMR). For example, the Thales Group and Antmicro AB developed a TMR RISC-V demonstrator with three redundant RV32I Rocket cores, voter, fault injector, and system monitor [80].

More extraordinary approaches, e.g., a heterogeneous lockstep system with a RISC-V Rocket and Arm Cortex-A9 core have also been proposed [81]. Other examples include the Klessydra RISC-V core [82], which is based on the PULPino microcontroller system [83]. This core is designed for space applications and explores several fault tolerance features, such as spatial redundancy (including DMR and TMR) and temporal redundancy (e.g. repetition with error detection).

Dörflinger et al. proposed a more generic approach, describing five fault tolerance features and components such as the use of redundant CPUs with N-modular redundancy [84].

It is worth noting that these state-of-the-art solutions emphasize the architectural implementation of existing RISC-V ISAs, rather than introducing ISA extensions aimed at fault tolerance. In contrast, the *Bratter* approach by Park et al. [85] does propose an ISA extension. They discuss a forward control flow integrity extension that uses a dedicated *branch tag register* and two new instructions to provide a signature-checking mechanism. To our knowledge, no such extensions have been proposed in the SOTA for DFED or hybrid techniques.

Chapter 7

Using RISC-V Extensions to Support SIHFT Techniques

The introduction and popularity of the extendable RISC-V ISA opens new opportunities for hardware-supported SIHFT techniques. This chapter will discuss how a custom RISC-V extension can be employed to create processors that provide custom instructions for SIHFT techniques. First, Section 7.1 will explore why hardware support for SIHFT techniques might be beneficial. Next, Section 7.2 will show how a custom RISC-V extension can be created to provide instructions for the P-DETECTOR technique, after which Section 7.3 will show how these new instructions impact the overhead of the P-DETECTOR technique by providing an impact analysis. To test the custom extension, the GNU assembler was extended to support the new instructions, as discussed in Section 7.4, after which the extension was simulated using the Imperas ISS in Section 7.5. This simulation will be used to evaluate the performance of the P-DETECTOR technique using the custom RISC-V processor against a default RISC-V processor in Section 7.6. Finally, Section 7.7 will discuss the challenges of formally verifying a custom RISC-V extension, before concluding this chapter in Section 7.8.

7.1 Hardware Support for SIHFT Techniques

The use of custom instructions to support SIHFT techniques can provide a significant overhead reduction over purely software-based implementations. By providing instructions that are tailor-made for the SIHFT technique, the number of instruction cycles required to perform the technique can be reduced significantly.

Meanwhile, the flexible nature of SIHFT techniques is preserved, enabling the developer to selectively apply the technique to specific parts of the program. For example, safety-critical parts of the application can be protected using a SIHFT technique while less critical parts can be left unprotected to utilize the full performance of the processor or to save energy. This flexibility is especially useful in embedded systems where resources are limited and performance is critical.

7.2 An Extension to Support P-DETECTOR

To show how a custom RISC-V extension could be used to support SIHFT techniques, an extension to support the P-DETECTOR technique was created. This extension, called “Xpdetector” (Xpdetector1p0), is a so-called nonstandard extension but conforms to the RISC-V standard by using the encodings reserved for custom extensions.

Our Xpdetector extension defines:

- Two additional special purpose registers (`sr` and `pr`) to store the signature and parity values;
- A duplicated register bank to store the register values at checkpoints;
- A Checkpoint Immediate (`CPI`) instruction to create the checkpoints;
- A Parity (`PAR`) instruction to calculate the parity of the register values;
- A Re-execute Immediate (`REI`) instruction to re-execute parts of the program;
- A Parity Zero and Link (`PZAL`) instruction to verify that the parities of two execution cycles were equal.

The extension is designed to work on the RV32I, RV32E, RV64I, and RV64E base ISAs. It is implemented utilizing the opcodes `0x0b`, `0x2b`, and `0x5b` reserved for custom nonstandard extensions. In the following sections, these opcodes will be referred to as `OP-CPI/PAR`, `OP-REI`, and `OP-PZAL`, respectively, as shown in Table 7.1.

Note that, just like an ISA specification, the description of this extension does not specify how the instructions shall be implemented in hardware. The

Table 7.1: The opcodes used by the Xpdetector extension.

Name	Opcode
Major opcodes	
`OP-CPI/PAR`	`0x0b`
`OP-REI`	`0x2b`
`OP-PZAL`	`0x5b`
Minor opcodes	
CPI	`0x1`
PAR	`0x0`

implementation of the instructions is left to the hardware designer, as long as the instructions conform to the specification. Schematics of the workings of the instructions in the following sections are merely illustrative and do not represent a real-world implementation.

7.2.1 Special Purpose Registers

The P-DETECTOR technique uses a general-purpose register as a signature register to keep track of which checkpoints should be jumped to when re-executing parts of the program. In Chapter 5, this register was referred to as register G. For this, Xpdetector defines a special 12-bit special purpose signature register (`sr`). The register is only 12 bits wide because the instruction encoding used to set the signature value only allows for a 12-bit immediate value. More on this in Section 7.2.3. The signature register cannot be accessed by the programmer directly but is used by the `CPI` and `REI` instructions to store and retrieve the signature.

Similarly, P-DETECTOR uses another general-purpose register to store and retrieve the parity value. This register was referred to as register P in Chapter 5. For this, Xpdetector defines a special $XLEN$-bit special purpose parity register (`pr`), where $XLEN$ is the width of the registers (32 or 64 bits). Just like `sr`, `pr` cannot be accessed by the programmer directly but is used by the `PAR` and `PZAL` instructions to store and retrieve the parity value.

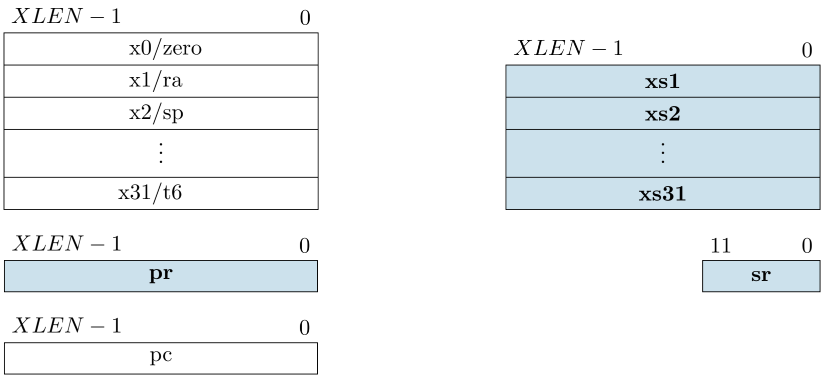


Figure 7.1: The Xpdetector extension introduces the `sr` register, `pr` register and `xs1`-`xs31` register bank to RISC-V.

7.2.2 Duplicated Register Bank

Chapter 5 discussed how the P-DETECTOR technique pushes the register values to a shadow stack in the memory at checkpoints. This enables it to restore the register values when re-executing the program. However, this approach is proven to be slow due to the high memory latency and the large number of registers that sequentially need to be restored.

To speed up this process, the Xpdetector extension defines a duplicated register bank. This 32-bit or 64-bit (depending on if an RV32 or RV64 ISA is used) register bank consists of 31 registers representing a shadow of registers `x1` to `x31`. The `x0` register has no shadow register as it is hardwired to zero. The registers in this shadow register bank will be referred to as registers `xs1` through `xs31`. When using an ISA with a reduced register set, i.e. RV32E or RV64E, the shadow register bank will be reduced to 15 registers (`xs1`-`xs15`) accordingly. Just like the special purpose registers `sr` and `pr`, the shadow registers cannot be accessed by the programmer directly. They are used by the `CPI` and `REI` instructions.

The added `sr` and `pr` registers together with the shadow register bank are shown in Figure 7.1.

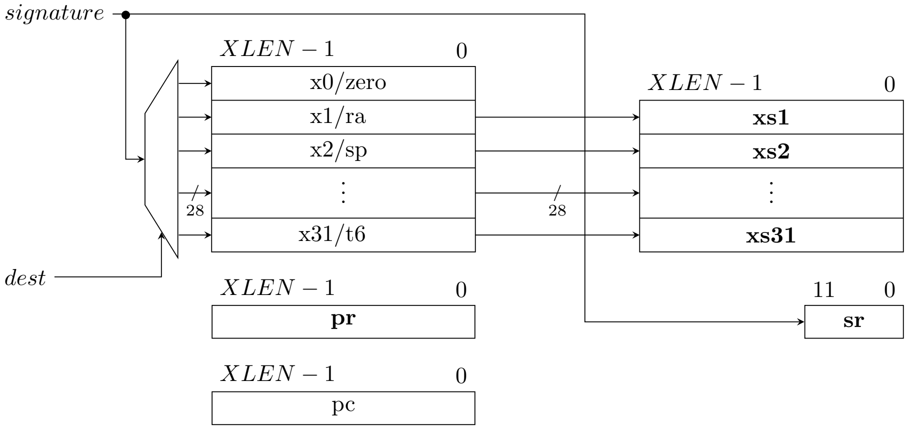


Figure 7.3: The operation of the Checkpoint Immediate (`CPI`) instruction.

7.2.3 Checkpoint Immediate Instruction

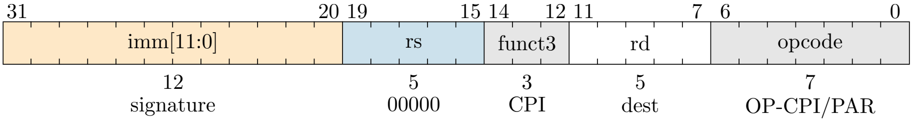


Figure 7.2: The encoding of the Checkpoint Immediate (`CPI`) instruction, using the I-type instruction format.

The Checkpoint Immediate (`CPI`) instruction is used to create a checkpoint in the program. Its encoding is shown in Figure 7.2. The instruction is encoded as an I-type instruction defined by the major opcode `OP-CPI/PAR` and minor opcode `CPI`. It takes a 12-bit immediate value (`signature`) and a 5-bit register specifier (`dest`) as operands.

As illustrated in Figure 7.3, the `CPI` instruction will perform three actions when executed:

- The values of the register bank `x1-x31` will be written to the shadow register bank `xs1-xs31`;
- The immediate value `signature` will be written to the special purpose register `sr`;

Table 7.2: Using the `cpi` mnemonic in assembly to create a checkpoint.

Example	Description
`cpi t1, 0x123`	Create a checkpoint with signature `0x123` and save the signature value to register `t1`.
`cpi 0x123`	Equivalent to `cpi zero, 0x123`. Create a checkpoint with signature `0x123`. Do not save the signature value to a general-purpose register.

- The immediate value `signature` will be written to the register specified by `dest`. If `dest` selects `x0` (`zero`), this action will not be performed.

Table 7.2 shows how the `CPI` instruction can be used in assembly language using the `cpi` mnemonic, followed by the destination register and the signature value. Alternatively, a pseudo-instruction can be used where the destination register is omitted. In this case, register `x0` (`zero`) will be selected, meaning the immediate value will not be saved to a general-purpose register.

7.2.4 Parity Instruction

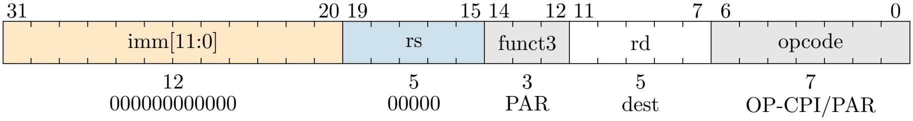


Figure 7.4: The encoding of the Parity (`PAR`) instruction, using the I-type instruction format.

The Parity (`PAR`) instruction is used to calculate the parity of the register values. The instruction is encoded as an I-type instruction, as shown in Figure 7.4. It uses the same opcode as the `CPI` instruction (`OP-CPI/PAR`) but with the minor opcode `PAR`. As operands, the `PAR` instruction only takes a 5-bit register specifier (`dest`).

When executed, the `PAR` instruction will perform two actions, as shown in Figure 7.5:

- An *exclusive or* operation will be performed on the values of the register bank `x1-x31` and the parity register `pr`. The result will be written back to the parity register `pr`;

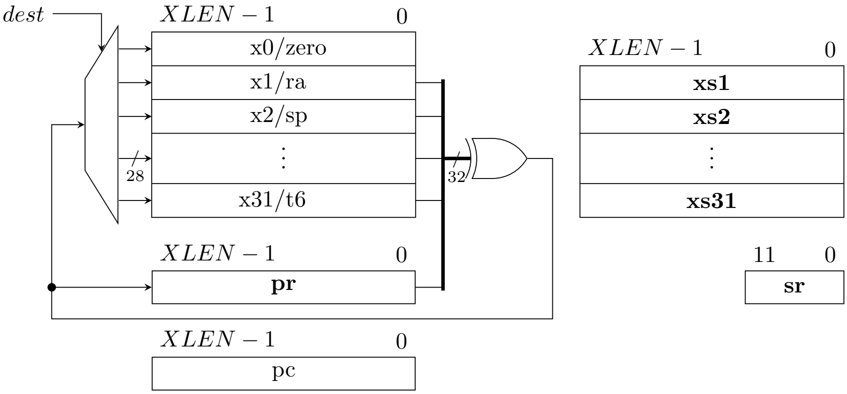


Figure 7.5: The operation of the Parity (`PAR`) instruction.

Table 7.3: Using the `par` mnemonic in assembly to calculate the parity of the register.

Example	Description
`par t1`	Calculate the parity of the register values and save the result to register `t1`.
`par`	Equivalent to `par zero`. Calculate the parity of the register values. Do not save the result to a general-purpose register.

- The result of the above *exclusive or* operation will also be written to the register specified by `dest`. Similarly to the `CPI` instruction, if `dest` selects `x0` (`zero`), this action will not be performed.

To use the `PAR` instruction in assembly, the `par` mnemonic should be used followed by the destination register. This is shown in Table 7.3. Similarly to the `CPI` instruction, a pseudo-instruction can be used to omit the destination register, in which case the result will not be saved to a general-purpose register.

7.2.5 Re-execute Immediate Instruction

The Re-Execute Immediate (`REI`) instruction is used to re-execute parts of the program from a checkpoint. For its encoding, Xpdetector defines a new instruction format. This format, called the *U0-type* instruction format, is based on the U-type format, but omits the `rd` field, creating room for a 25-bit immediate field `imm[31:7]`.

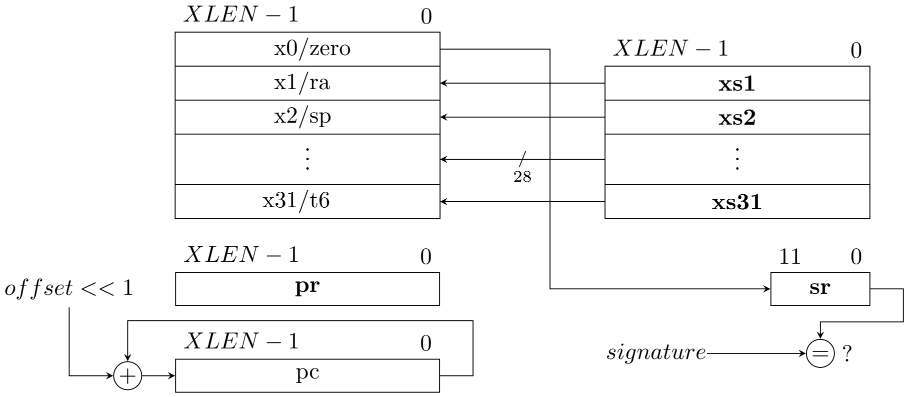


Figure 7.7: The operation of the Re-Execute Immediate (`REI`) instruction.

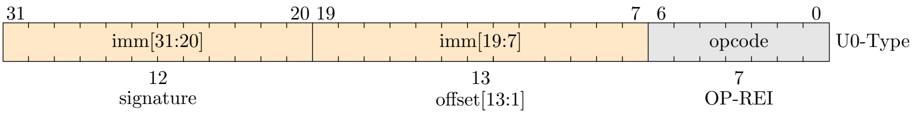


Figure 7.6: The encoding of the Re-Execute Immediate (`REI`) instruction, using the new U0-type instruction format.

Figure 7.6 shows this new *U0-type* instruction format while used to encode the `REI` instruction. The instruction uses the major opcode `OP-REI` and uses the 25-bit immediate field to encode two immediate operands: a 12-bit signature value (`signature`) and a 13-bit signed offset value (`offset`).

The `REI` instruction is a conditional instruction. It will only re-execute the program if the signature value in the special purpose register `sr` matches the signature value in the `signature` immediate field. If the signatures do not match, the instruction will behave as an `NOP` (no operation) instruction. If the condition is met, the instruction will perform three operations, shown in Figure 7.7:

- The values of the shadow register bank `xs1-xs31` will be written back to the register bank `x1-x31`;
- The signature register `sr` will be cleared;
- The program counter will be incremented by the `offset` multiplied by two (i.e. shifted left one bit).

Table 7.4: Using the `rei` mnemonic in assembly to re-execute parts of the program.

Example	Description
`rei 0x123, -0x456`	Conditionally re-execute from the address 0x456 before the program counter
`rei 0x123, myLabel`	Conditionally re-execute from the location indicated by `myLabel`
`rei 0x123, 5b`	Conditionally re-execute from 5 instructions before the program counter

Since the `offset` field is shifted to the left by one bit, it is interpreted as a 14-bit signed offset value. This enables the `REI` instruction to jump to any address within a ±8 KiB range. To put this into perspective, the conditional branch instructions in the RV32I base ISA have an address range of ±4 KiB.

To use the `REI` instruction in assembly, the `rei` mnemonic should be used followed by the signature value and the signed offset value. This is shown in Table 7.4. A pseudo-instruction can be employed where a text label is used in place of the offset value. Alternatively, numeric labels can be used to specify a relative offset. These are suffixed with `f` for a forward reference or `b` for a backward reference.

7.2.6 Parity Zero And Link Instruction

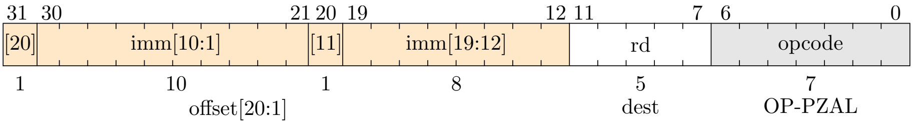


Figure 7.8: The encoding of the Parity Zero And Link (`PZAL`) instruction, using the J-type instruction format.

The Parity Zero And Link (`PZAL`) instruction is used to verify that the parity is equal to zero and jump to an error handler if it is not. Its encoding is shown in Figure 7.8. The instruction is encoded as a J-type instruction using the major opcode `OP-PZAL`. It takes two operands: a 20-bit signed offset value (`offset`) and a 5-bit register specifier (`dest`).

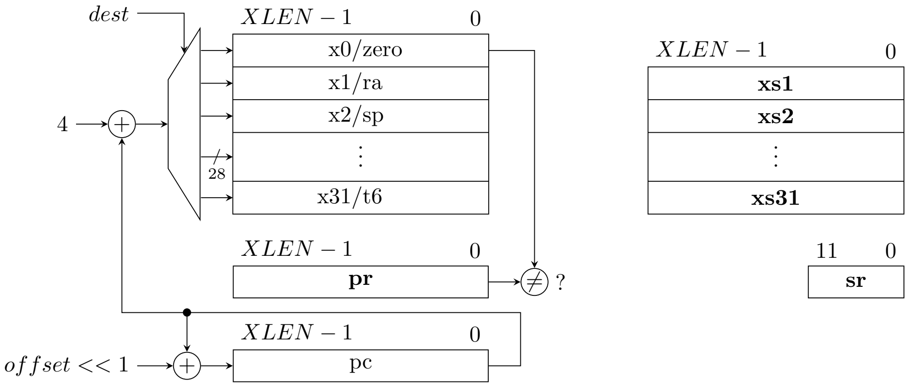


Figure 7.9: The operation of the Parity Zero And Link (`PZAL`) instruction.

The `PZAL` instruction is a conditional instruction that executes if the parity register `pr` is not zero. If the condition is met, the instruction will perform the two operations shown in Figure 7.9.

- The program counter will be incremented by the `offset` multiplied by two (i.e. shifted left one bit);
- The address of the instruction following the jump (`pc`+4) will be written to the register specified by `dest`. If `x0` (`zero`) is specified as the destination register, this action will not be performed.

Since the `offset` field is shifted to the left by one bit, it is interpreted as a 21-bit signed offset value. This enables the `PZAL` instruction to jump to any address within a ±1 MiB range. This is the same range as the unconditional jump (`JAL` instruction in the RV32I base ISA).

An alternative encoding to this instruction could use the I-type instruction format that reuses the `OP-CPI/PAR` major opcode but with a different minor opcode. This would reduce the width of the `offset` field to 12 bits, reducing the range to ±4 KiB, and would reduce the number of major opcodes used by the extension. Moreover, this would enable the Xpdetector extension to be implemented on the RV128I base ISA.

There are two ways to use the `PZAL` instruction in assembly, as shown in Table 7.5. The first way is to use the `pzal` mnemonic followed by the destination register and the signed offset value. Using the instruction in this way makes it act like a function call, saving the return address in the destination register. Similarly to the `REI` instruction, a pseudo-instruction using a text label or a numeric label

Table 7.5: Using the `pzal` mnemonic in assembly to verify that the parity register is zero and jump to an error handler if it is not.

Example	Description
	1. Parity Zero And Link
`pzal t1, -0x456`	Jump to the error handler at the address 0x456 before the program counter if the parities are not equal. Use register `t1` as the link register.
`pzal t1, mylabel`	Jump to the error handler at the location indicated by `mylabel` if the parities are not equal. Use register `t1` as the link register.
`pzal t1, 5b`	Jump to the error handler at 5 instructions before the program counter if the parities are not equal. Use register `t1` as the link register.
`pzal 5b`	Equivalent to `pzal ra, 1b`. Jump to the error handler located 5 instruction before the program counter if the parities are not equal. Use the default link register `ra`.
	2. Parity Zero
`pz -0x456`	Jump to the error handler at the address 0x456 before the program counter if the parities are not equal. Do not save the return address.
`pz mylabel`	Jump to the error handler at the location indicated by `mylabel` if the parities are not equal. Do not save the return address.
`pz 5b`	Jump to the error handler located 5 instructions before the program counter if the parities are not equal. Do not save the return address.

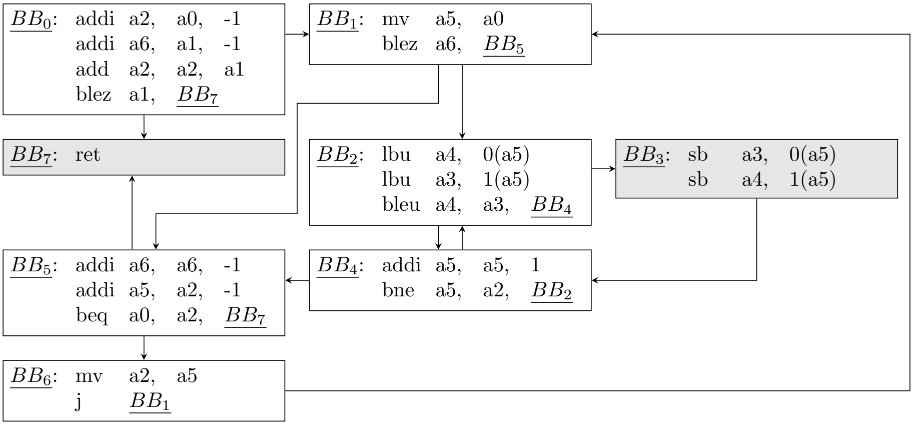

$\underline{BB_0}$: addi a2, a0, -1
addi a6, a1, -1
add a2, a2, a1
blez a1, $\underline{BB_7}$
$\underline{BB_1}$: mv a5, a0
blez a6, $\underline{BB_5}$
$\underline{BB_7}$: ret
$\underline{BB_2}$: lbu a4, 0(a5)
lbu a3, 1(a5)
bleu a4, a3, $\underline{BB_4}$
$\underline{BB_3}$: sb a3, 0(a5)
sb a4, 1(a5)
$\underline{BB_5}$: addi a6, a6, -1
addi a5, a2, -1
beq a0, a2, $\underline{BB_7}$
$\underline{BB_4}$: addi a5, a5, 1
bne a5, a2, $\underline{BB_2}$
$\underline{BB_6}$: mv a2, a5
j $\underline{BB_1}$

Figure 7.10: The control flow graph of the bubble sort algorithm in RV32I assembly. The vulnerable sections are colored gray.

in place of the offset value. When the `pzal` mnemonic is used the destination register can be omitted to use `x1`, the register generally used to store the return address.

The second way is to use the `pz` (*verify*) mnemonic followed by the signed offset value. In this case, the destination register is omitted, but `x0` (`zero`) is used as the destination register, meaning that `pc`+4 will not be saved. Using the instruction in this way makes it act like a conditional branch instruction.

This use of pseudo-instructions is similar to the `JAL` (jump and link) instruction in the RV32I base, which can be used as a function call using the `jal` mnemonic or as a normal jump using the `j` mnemonic [86].

7.2.7 Example Usage

To illustrate how the instructions of the Xpdetector extension can be used to implement the P-DETECTOR SIHFT technique, the same bubble sort program as in Chapters 4 and 5 will be used. However, this time, the program is written in RISC-V assembly language. The control flow graph of the bubble sort program in RV32I assembly language is shown in Figure 7.10. The vulnerable sections are colored gray. Since a different instruction set architecture and accompanying toolchain is now used to compile the program, there are notable differences from the CFG in ARMv7-M assembly from Chapter 4.

Figure 7.11 shows the P-DETECTOR technique implemented on the bubble

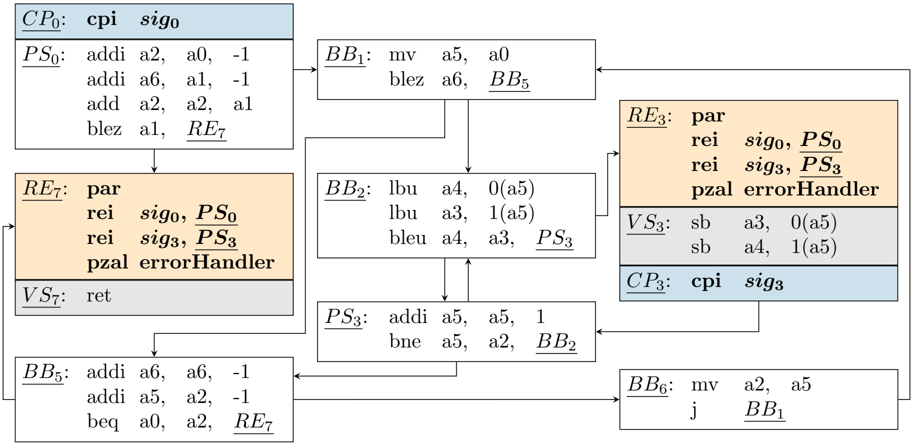


Figure 7.11: The P-DETECTOR technique implemented on the bubble sort algorithm in RV32IXpdetector assembly.

sort program using the Xpdetector extension. The program starts by creating a checkpoint at the beginning of the program using the `CPI` instruction at label $\underline{CP_0}$. The initial values of the registers are saved to the shadow register bank and the signature register is set to the value sig_0.

Next, the program executes until it reaches one of the re-execution points (labels $\underline{RE_3}$ or $\underline{RE_7}$). At these points, the program calculates the parity of the register values using the `PAR` instruction. The result of this operation is stored in the parity register. Next, the first conditional `REI` instruction is executed. The signature register is compared against the signature value sig_0. Since the signatures match, the `REI` instruction will restore the register values from the shadow register bank, clear the signature register, and jump back to the start of the protected section at label $\underline{PS_0}$. This starts the second execution cycle.

In an error-free run, the program ends up back at the same re-execution point (labels $\underline{RE_3}$ or $\underline{RE_7}$). The parity of the register values is once again calculated, resulting in the parity register containing zero if no errors occurred. Since the signature register was cleared at the end of the first execution cycle, the conditional `REI` instructions will act as `NOP` instructions. Finally, the `PZAL` instruction at the end of the re-execution points is executed. This instruction will verify that the parity register is zero, calling an error handler defined by the `errorHandler` text label if it is not. Otherwise, the vulnerable section is entered.

Comparing the implementation with the Xpdetector extension in Figure 7.11 to

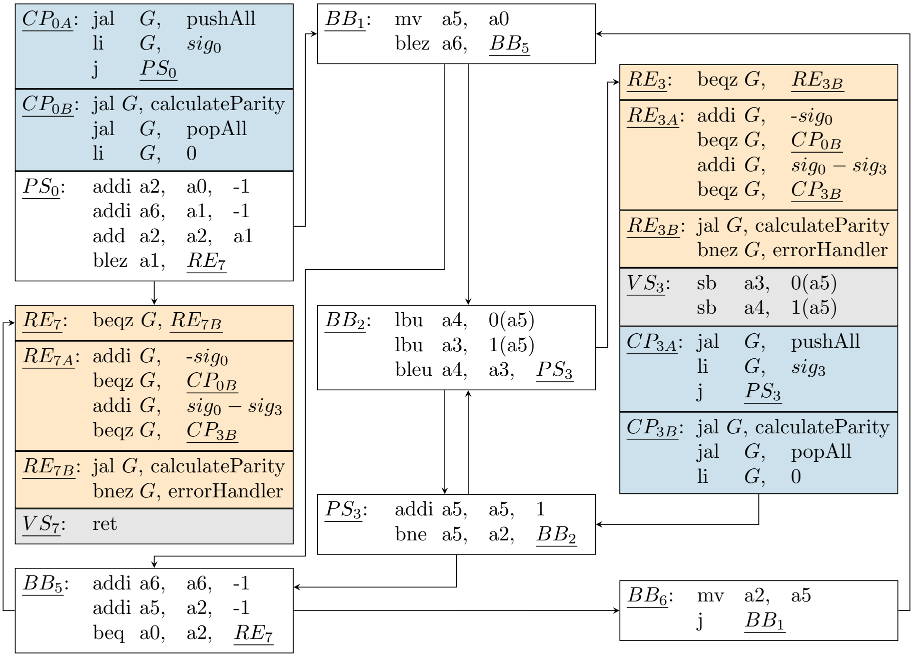


Figure 7.12: The P-DETECTOR technique implemented on the bubble sort algorithm in RV32I assembly.

the implementation on the base RV32I ISA in Figure 7.12 already shows that far fewer instructions are needed when using the extension. However, not shown in Figure 7.12 is the implementation of the `pushAll`, `popAll` and `calculateParity` subroutines that are used by the P-DETECTOR technique. The `pushAll` and `popAll` subroutines add 29 static instructions each, and the `calculateParity` subroutine adds 28 static instructions.

Notice as well that the implementation of P-DETECTOR on Figure 7.12 differs significantly from the implementation shown in Chapter 5. This is because the RISC-V implementation of the DETECTOR family of techniques use an alternative implementation. This will be discussed in more detail in Chapter 8.

7.3 Impact Analysis

Since the Xpdetector extension is described at the ISA level, its hardware implementation is left to the hardware designer. This also means that the number of cycles required to execute each instruction is not defined by this specification.

In the best case, each instruction would be executed in a single clock cycle. However, this might be unfeasible for faster clock cycles due to the complexity of some instructions. Therefore, it is only possible to provide an estimation of the number of instruction cycles added for each executed vulnerable section. For each vulnerable section, the number of added instruction cycles can be calculated as follows:

$$I^{+}_{VS_{Xpdetector}} = 2I_{PAR} + I_{REI} + I_{CPI} + C + 1 \tag{7.1}$$

$$\text{where } \{I_{PAR}, I_{REI}, I_{CPI}, C \in \mathbb{N}\}$$

In Equation (7.1), I_{PAR}, and I_{CPI} are the number of added instruction cycles for the `PAR` and `CPI` instructions, respectively. I_{REI} is the number of added instruction cycles for the `REI` instruction when the condition ($signature = \texttt{sr}$) is met. Like before in Chapter 5, C depends on the number of possible re-execution paths for a single vulnerable block. Appendix B.4 provides a more detailed explanation of how Equation (7.1) is derived.

In the best-case scenario, all added instructions would be executed in a single clock cycle, resulting in $I^{+}_{VS_{Xpdetector}} = 5 + C$. In a more realistic scenario, the `PAR` instruction could, for example, take 4 instruction cycles, the `CPI` instruction 1 cycle, and the `REI` instruction 2 cycles, meaning $I^{+}_{VS_{Xpdetector}} = 12 + C$.

7.4 Changes to the GNU Assembler

The assembler takes the assembly language code and translates it into binary machine code. Therefore, the assembler must be aware of the new instructions defined by the Xpdetector extension before they can be used in assembly code. This can be achieved by extending the assembler to recognize the new mnemonics and encode them into the binary format. To realize this, the GNU Assembler (GAS) of the RISC-V C/C++ cross-compiler has been extended to support the Xpdetector extension.

To add the required instructions to the assembler, two parts of the GNU assembler have been modified. The first part is the opcode library, which defines the encoding of the RISC-V instructions. For the Xpdetector instructions which use the default instruction formats (`CPI`, `PAR`, and `PZAL`), this step is sufficient. However, for the `REI` instruction, the custom U0-type instruction type was used, meaning the assembler's encoding function was also extended.

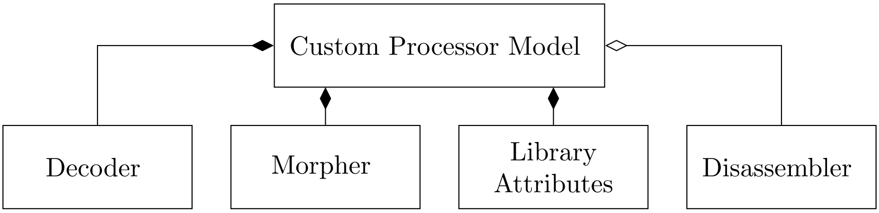


Figure 7.13: The components of a custom processor model using the Imperas SDK.

7.5 Validating the Risc-V Extension Through Simulation

To validate the Xpdetector extension, its behavior was implemented using the Imperas SDK, extending the OVPsim RISC-V model [67]. Using the Imperas SDK, a custom instruction can be implemented using a custom processor model as an intercept library. This entails that the custom processor model is used as an extension of the parent processor model. Figure 7.13 shows the four components defined in the custom processor model:

- The **decoder** is responsible for decoding the binary formatted instruction using a decoding table.
- The **morpher** defines the behavior of the instruction. This function can be written using the OVP VMI Morph Time API [87].
- The **library attributes** encapsulate all required information about the library to interact with the Imperas simulator products.
- The **disassembler** defines the string representation when disassembling the instructions. This component is optional.

This intercept library is compiled as a shared object file that can be added to the Imperas instruction set simulator at runtime. Its behavior is shown in Figure 7.14. For every fetched instruction, the ISS decoder will be called. If the instruction is not recognized, the instruction will be decoded and executed by the parent processor model. Otherwise, the Morpher of the custom model will be called to simulate the instruction.

For the implementation of the Xpdetector extension, OVP's RV32I processor model was used as the parent processor model [88], thus creating an RV32IXpdetector model.

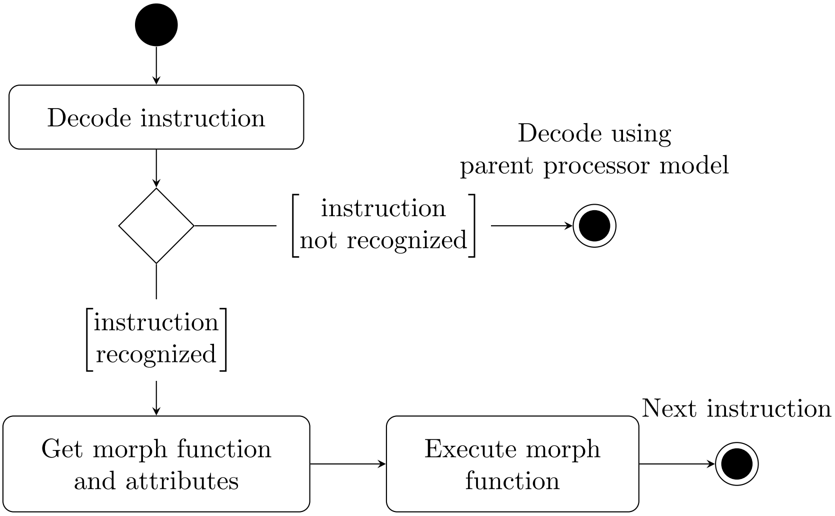

Decode instruction
Decode using parent processor model
[instruction not recognized]
[instruction recognized]
Next instruction
Get morph function and attributes
Execute morph function

Figure 7.14: The behavior of the intercept library in the Imperas ISS.

7.6 Evaluation

To validate the effectiveness of P-DETECTOR on the RV32IXpdetector ISA, five case studies were evaluated using the Imperas ISS: BC, BS, CRC, MM, and QS. Additionally, the P-DETECTOR technique was also implemented on the base RV32I ISA to evaluate the effect of the hardware support provided by the Xpdetector extension.

7.6.1 Fault Injection Results

The results of the single-bit CFE and DFE fault injection campaigns are shown in Figures 7.15 and 7.16. The results indicate that the version using the Xpdetector extension performs slightly worse than the implementation on the base RV32I ISA in both error detection ratio and SDC ratio. This is because in the implementation with the native RV32I ISA, many more instructions are added to the program to implement the P-DETECTOR technique. CFEs or DFEs within these instructions are likely to be detected by the P-DETECTOR mechanism. Moreover, even if they are not detected, they are unlikely to affect the program result and thus do not result in an SDC.

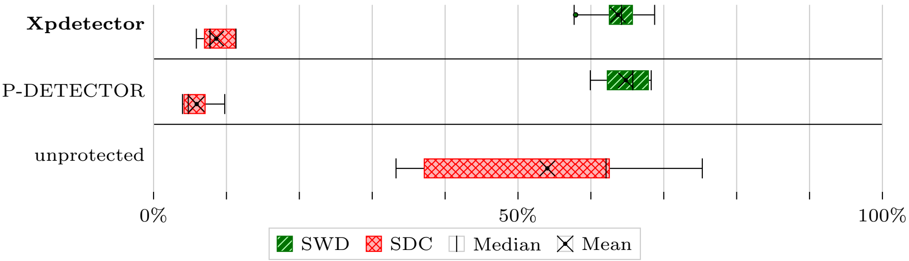


Figure 7.15: Boxplots of the CFE single-bit fault injection results performed on the BC, BS, CRC, MM, and QS case studies protected with P-DETECTOR on the RV32I and RV32IXpdetector ISAs, showing the error detection ratio and SDC ratio.

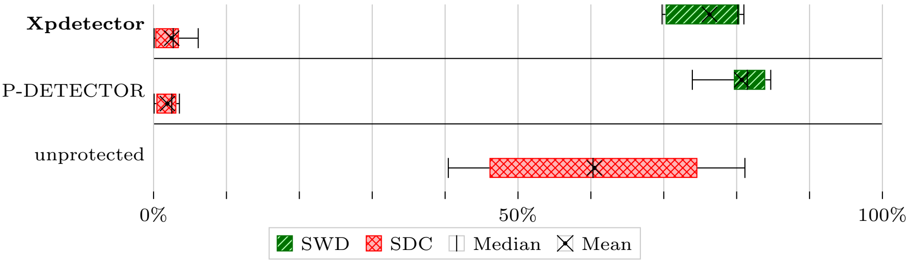


Figure 7.16: Boxplots of the DFE single-bit fault injection results performed on the BC, BS, CRC, MM, and QS case studies protected with P-DETECTOR on the RV32I and RV32IXpdetector ISAs, showing the error detection ratio and SDC ratio.

7.6.2 Overhead Evaluation

As mentioned in Section 7.3, the number of cycles required to execute each instruction is not defined by the specification, nor is it specified in the simulator model. However, the execution time overhead can be estimated by analyzing the execution trace of each case study and assigning an expected number of cycles for each instruction. For this, the following rules have been used:

- Each branch instruction takes two instruction cycles when the branch is taken and one cycle when it is not.
- Each memory access instruction takes three instruction cycles.
- All other instructions take one instruction cycle.

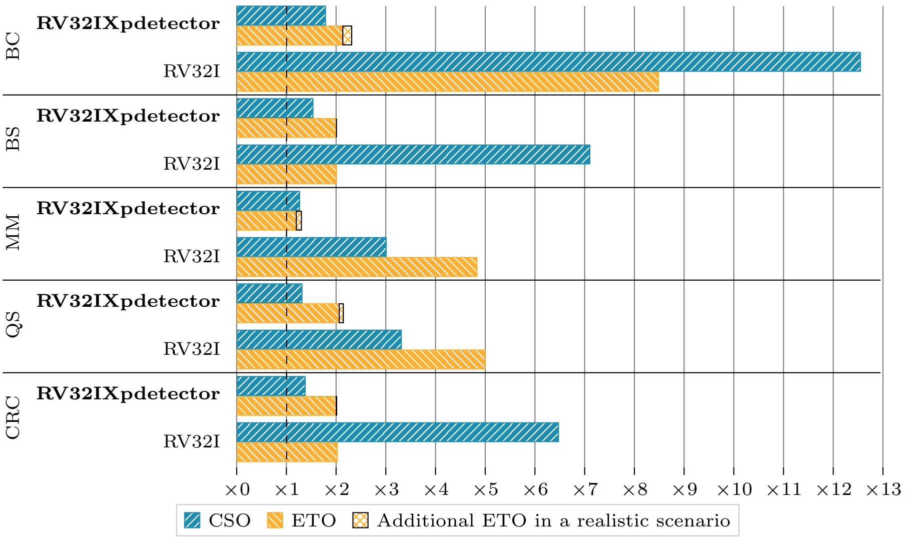


Figure 7.17: The Code Size Overhead (CSO) and estimated Execution Time Overhead (ETO) of P-DETECTOR on the RV32I and RV32IXpdetector ISAs for each evaluated case study.

This is a simplification of the actual execution time, but it provides a rough basis to estimate the ETO. For the custom instructions, both the best case scenario (i.e. all instructions take one cycle) and a more realistic scenario (i.e. `PAR` takes four cycles, `REI` takes two cycles, and the other instructions take one cycle) were evaluated.

The (estimated) ETO and CSO of the P-DETECTOR technique on the RV32I and RV32IXpdetector ISAs are shown in Figure 7.17. For the implementation with the Xpdetector extension, the difference between the best-case scenario and the more realistic scenario is also shown. As expected, both the CSO and ETO of the implementation with the Xpdetector extension are much lower than the implementation on the RV32I base ISA, both for the best-case and more realistic scenario.

Notice also that the code size overhead of the RV32I base ISA can become quite large for some case studies. This is because RISC-V does not support *store multiple* and *load multiple* instructions, where multiple registers are stored or loaded using a single instruction. Hence, the push and pop to the shadow stack was implemented using two subroutines that store or load the registers one by one. This has a large effect on the CSO for small case studies since the size of these subroutines is large compared to the size of the case study. Since the push and pop instructions are replaced by the custom instructions in the Xpdetector extension, this effect is not present in the implementation using the Xpdetector extension.

7.7 Challenges for Micro-Architectural Verification

While the development of a RISC-V extension may be relatively quick, it is not easy to formally verify it. In an interview with Semiconductor Engineering [89], Pete Hardee, group director for product management at Cadence says:

> RISC-V is an open ISA. Anyone can take it and implement a processor. But the leaders in the RISC-V market know that just because they do not need to pay license royalties, it doesn't mean RISC-V is the cheap option. There can be no shortcuts for verification if you want to be successful with RISC-V.

The complexity of verifying the micro-architecture can be a significant hurdle for the development of a new RISC-V extension. Even more so for processor verification than for ASIC verification, Hardee continues:

> It's harder. Remember, the *AS* in ASIC stands for application-specific. Fully verifying a chip for its intended application is finite and bounded. Processor verification is not. Every operation in the processor instruction set architecture must be verified to provide the specified behavior in every eventuality — every combination of instructions. In general-purpose applications, that cannot be predicted at the time of verification of the processor IP.

This challenge is recognized by the RISC-V foundation and the community, with multiple ongoing efforts to standardize, speed up and simplify the verification process. For example, Davidmann et al. [90] illustrate the approach of using a random instruction generator for RISC-V to compare implementation RTL against a reference simulation model. Other efforts include the CORE-V-Verif functional verification project for the CORE-V family of RISC-V cores [91] and the RISC-V Verification Interface (RVVI) draft open standard defining interfaces required to bring together several of the subsystems required for RISC-V processor design verification [92].

7.8 Conclusion

This chapter explored how a custom RISC-V extension can be used to support SIHFT techniques. This can be useful for improving the performance of the technique while keeping the flexible nature of SIHFT techniques.

To illustrate this concept, the Xpdetector extension was developed, which includes four custom instructions (`CPI`, `PAR`, `REI`, and `PZAL`) designed to facilitate the P-DETECTOR technique. The instructions' encoding, operation, and usage at the assembly level were described. The extension leverages two additional special-purpose registers and a duplicated register bank to provide the re-execution and assertion mechanism of P-DETECTOR.

The Imperas SDK was used to simulate the Xpdetector extension, enabling the evaluation of the P-DETECTOR technique on the RV32IXpdetector ISA. These simulation results demonstrated that the Xpdetector extension performs similarly to the base RV32I ISA in terms of error detection and SDC rates, with the base ISA showing slightly better results, mainly due to a difference in fault-space coverage. Comparing the overhead of P-DETECTOR on the base ISA and the extended ISA confirms that the Xpdetector extension significantly reduces the overhead of the technique, both in terms of ETO (from averagely ×4.5 to ×1.9) and CSO (from averagely ×6.5 to ×1.4).

The development and verification of these custom extensions pose significant challenges. The complexity of verifying the micro-architecture of a processor with these extensions is notably high, even when compared to the development of ASIC SoCs. Ongoing efforts within the RISC-V community, aim to address these challenges and streamline the verification process. This will hopefully reduce the threshold for developing custom RISC-V extensions and encourage the adoption of custom processors in the industry.

In summary, custom RISC-V extensions for SIHFT techniques may present a feasible method to enhance processor reliability. While the cost and complexity of developing and verifying these extensions are high and pose a significant hurdle, future efforts to standardize and simplify the verification process may help to overcome these challenges, making hardware-supported SIHFT techniques a viable option.

Chapter 8

A GCC Plugin to Automatically Implement SIHFT Techniques

Sections of this chapter are based on the work published in [99] and [102]. Since publication of these papers, the structure of the GCC plugin has been altered to make it more extendable and to support multiple ISA families. Hence, this chapter reflects the latest version of the GCC plugin.

This chapter discusses how the implementation of SIHFT techniques can be automated using a GCC plugin. First, Section 8.1 explains why a GCC plugin is needed to automate the implementation of SIHFT techniques, points to the two GCC plugins that have been developed by the M-Group research group, and introduces the new SIHFT plugin that combines the functionality of these two plugins and extends them to support multiple ISA families. Next, Section 8.2 explains how the plugin can be used, stipulating the plugin arguments and function attributes used to indicate which functions should be protected by which SIHFT technique. This is followed by a deeper delve into the inner workings of the GCC plugin by first discussing at which intermediate representation level the plugin operates in Section 8.3, and then explaining the structure of the plugin in Section 8.4. Finally, Section 8.5 explains how the DETECTOR and P-DETECTOR techniques are implemented in the plugin, before concluding the chapter in Section 8.6.

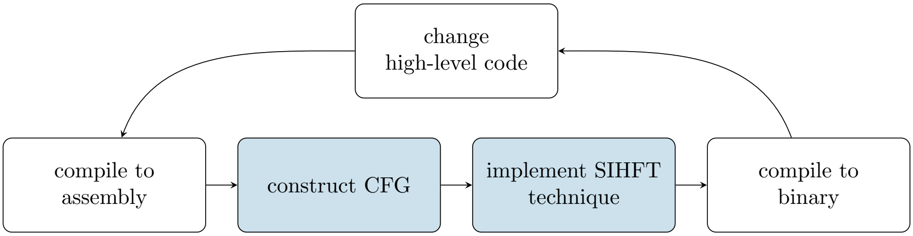


Figure 8.1: Manually implementing a SIHFT technique in assembly language is a time-consuming and error-prone process.

8.1 Introduction to the Compiler Plugin

Throughout this thesis, the different techniques have been presented using ARM and RISC-V assembly language. This is because SIHFT techniques do not work optimally when implemented in high-level programming languages like C or C++. For a start, the compiler optimizes the code, often removing the redundant operations inserted by the SIHFT techniques. Research shows that SIHFT techniques still work better when implemented in a language that is close to machine code, even when taking the appropriate actions to disable these optimization steps [59]. This is because the human-readable high-level source code does not map one-to-one to the machine code. One high-level instruction is often mapped to multiple machine instructions and the compiler often generates a different control flow graph for the machine code than what the high-level code might suggest.

While manually implementing SIHFT techniques in assembly language is possible, it is a time-consuming and error-prone process. This process is shown in Figure 8.1. First, the high-level application is compiled into assembly code. Next, this assembly code should be analyzed to construct the control flow graph of the application, after which the SIHFT technique can be implemented in the assembly code. This process is very tedious and any mistake can result in a badly implemented SIHFT technique or a broken application. Finally, the changed assembly files can be compiled into the binary for the target device. Whenever a change is made to the high-level application, the process must be repeated from the beginning. This makes the manual implementation of SIHFT techniques infeasible in practice.

To automate this process, two compiler plugins for the GCC compiler toolchain had been developed in the past. The first plugin, developed by Vankeirsbilck et al. [93] was created to automate the implementation of several CFED techniques. This plugin was the base for the DFED plugin [94], which implements several

Listing 8.1: Example showing how to compile a program using the GCC plugin.

```
gcc -fplugin=SIHFT_RISCV.so
  -fplugin-arg-SIHFT-error=error_handler foo.c
```

Listing 8.2: The developer can indicate which functions should be protected by which SIHFT technique using the `SIHFT` function attribute. Additional parameters can be provided to indicate the selective level and provide any additional options required by the technique.

```
void __attribute__((SIHFT("P-DETECTOR"))) foo() {
  // Will be protected by P-DETECTOR
}
void __attribute__((SIHFT("SWIFT", 1))) bar() {
  // Will be protected by S-SWIFT
}
void __attribute__((SIHFT("DETECTOR",1,"ra,sp"))) baz(){
  // Will be protected by S-DETECTOR with protected
     registers ra and sp
}
```

DFED techniques. Since the techniques discussed in this thesis are hybrid techniques, these plugins have been combined into a single GCC plugin, capable of implementing CFED, DFED, and hybrid SIHFT techniques.

Since the research of Vankeirsbilck and Thati focussed solely on ARM-based systems, the CFED and DFED compiler plugins were created for the ARM ISA family. This limits the use of these plugins to ARM-based systems. Therefore, the combined plugin was crafted in such a way that it can be used to support multiple ISA families [28].

8.2 Usage

The plugin can be loaded by providing the `-fplugin` argument during the compilation process, providing the path to the shared object file of the plugin. This is shown in Listing 8.1. Additionally, the function name of the error handler to be called when a SIHFT technique detects an error must be provided using the `-fplugin-arg-SIHFT-error-handler` argument.

Listing 8.3: The function to which the initialization of the SIHFT technique should be prepended can be marked using the `SIHFT_INIT` function attribute.

```
void __attribute__((SIHFT_INIT("DETECTOR"))) main() {
  // Will contain the initialization of DETECTOR at the
     start of the function
}
```

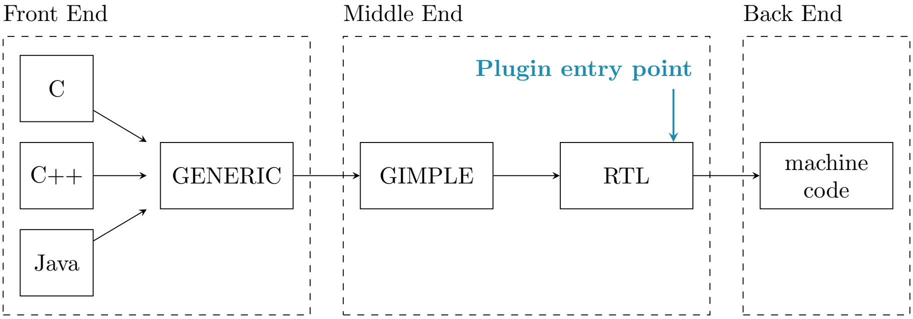


Figure 8.2: The different intermediate languages of the GCC compiler toolchain and the entry point of the GCC plugin.

The plugin uses custom function attributes to determine which functions should be protected with which SIHFT technique. To indicate that a function must be protected with a specific technique, the `SIHFT(<technique>)` attribute must be added to the function declaration, providing the name of the technique as a parameter. This is shown in Listing 8.2. When applicable, a selective implementation can be selected by providing a selective level as the second parameter. Since some techniques can require additional options, these can be provided as well. For example, `S-DETECTOR` requires the list of registers to be protected.

Finally, many techniques require an initialization process to set up the initial values of the registers used by the technique. This initialization will be inserted at the start of the function with the `SIHFT_INIT` function attribute. This is shown in Listing 8.3.

8.3 GCC Intermediate Representations

To compile a program from high-level source code to a binary executable, the GCC compiler toolchain compiles the source code through several intermediate languages, as depicted in Figure 8.2. The process starts in the front end of the compiler, where the high-level source code is parsed and translated into the GENERIC intermediate language. This is a language-independent tree structure representation that serves as an interface between the parser and optimizer. This GENERIC representation is then simplified to the GIMPLE intermediate language for use in language-independent optimization passes in the middle-end. Next, the GIMPLE representation is compiled further to the Register Transfer Level (RTL) intermediate language. This low-level language is very close to assembly language and is used for some final optimization passes before it is translated to the target machine code in the backend layer of the compiler.

GCC plugins – also called *loadable modules*, make it possible to add new features to the GCC compiler without having to modify the compiler itself. The developer of the plugin can specify after which compilation pass the plugin should be executed. In the case of the SIHFT plugin, the entry point of the plugin is located near the end of the RTL optimization pass, as shown in Figure 8.2. By choosing this entry point, the RTL language is on a near one-to-one mapping with the machine code, making it the ideal place to implement SIHFT techniques.

8.4 Structure of the GCC Plugin

The GCC plugin API provides a set of functions that can be used by the plugin to interact with the compiler. However, like different GCC compilers are needed to compile code for different ISA families, different versions of the GCC plugin API also exist for each of those targets. Therefore, a single compiler plugin can only support one single ISA family.

To circumvent this limitation, the novel SIHFT plugin is designed as a single codebase that can be compiled using different GCC plugin API versions for each ISA family. Additionally, the plugin is compartmented into four distinct components as shown in Figure 8.3.

- The **Entry Point** component contains the functionality to register the plugin module to the GCC compiler at the correct entry point and a callback function that is called by the compiler when the plugin is executed.

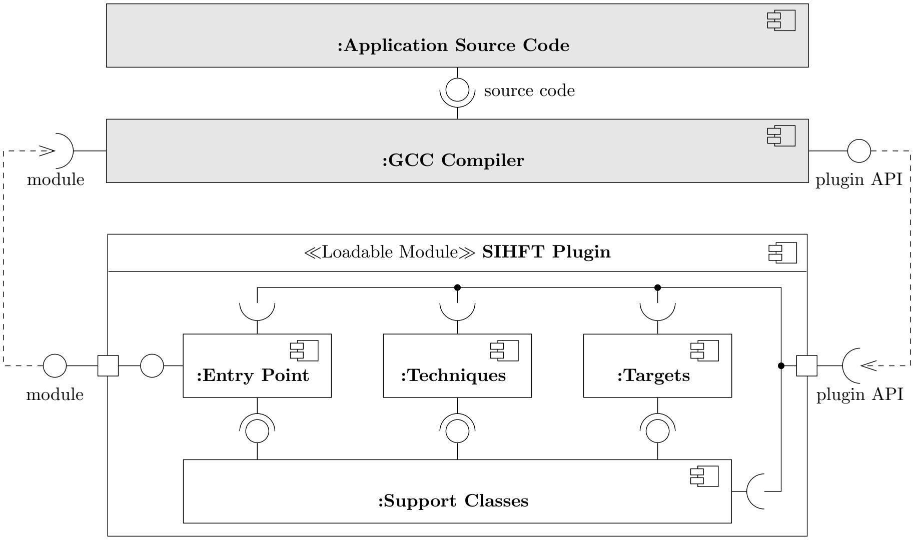


Figure 8.3: The UML component diagram of the SIHFT GCC plugin.

- The **Techniques** component contains the functionality to implement the supported SIHFT techniques.
- The **Targets** component contains the functionality to evaluate and insert instructions into the RTL code.
- The **Support Classes** component provides various utility classes that can be used by the other components.

These components will now be discussed in more detail.

8.4.1 Support Classes

The *Support Classes* component contains three static utility classes that can be used by all other components. They are shown in Figure 8.4. The `Arguments` and `Attributes` classes are used to obtain the plugin arguments and function attributes, respectively. The `ExportCfg` class is used to export the details of the CFG to a file in different formats. This can be useful for debugging purposes, as it enables the developer of the plugin to inspect the CFG of a function to verify that a SIHFT technique has been implemented correctly.

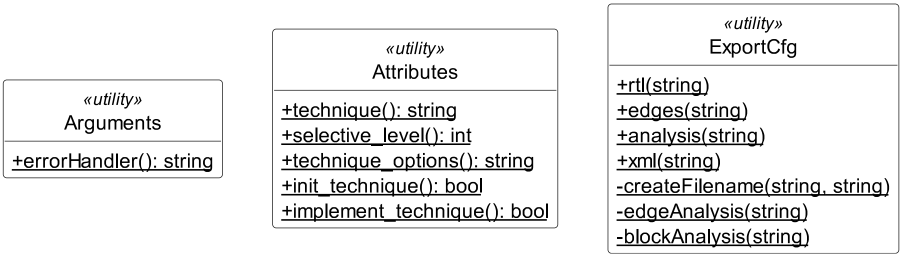


Figure 8.4: UML class diagram of the *Support Classes* component.

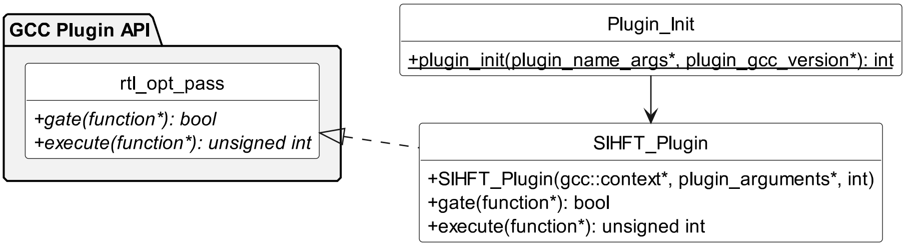

GCC Plugin API
rtl_opt_pass
+gate(function*): bool
+execute(function*): unsigned int
Plugin_Init
+plugin_init(plugin_name_args*, plugin_gcc_version*): int
SIHFT_Plugin
+SIHFT_Plugin(gcc::context*, plugin_arguments*, int)
+gate(function*): bool
+execute(function*): unsigned int

Figure 8.5: UML class diagram of the *Entry Point* component.

8.4.2 Entry Point

The *Entry Point* component is, as the name suggests, the entry point of the plugin. Its UML class diagram is shown in Figure 8.5. The `Plugin_Init` class contains a single static method `plugin_init`. This method is called by GCC at the start of the compilation process to initialize the plugin. During this initialization, the `SIHFT_Plugin` class is instantiated and a callback is registered to execute the plugin during the RTL optimization pass.

For each function, the `gate` method of `SIHFT_Plugin` class is called. This method determines whether the plugin should be executed for the current function. For this, it utilizes the `Attributes` class to determine if the function contains the required `SIHFT` or `SIHFT_INIT` function attributes.

If `gate` returned true, the `execute` method is called. This method implements the actual compilation pass of the plugin by using the functionality of the *Techniques* component.

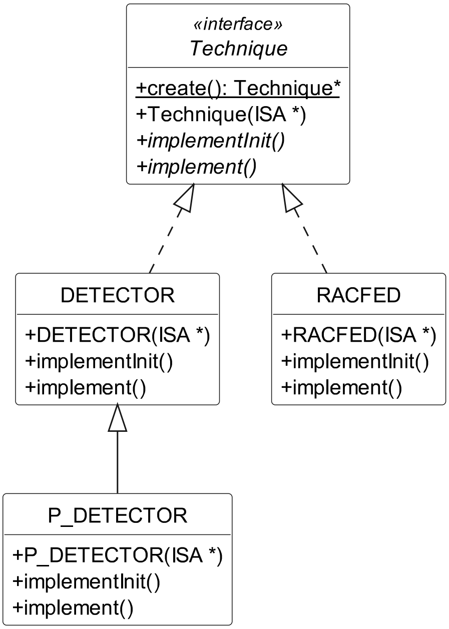


Figure 8.6: UML class diagram of the *Techniques* component.

8.4.3 Techniques

The *Targets* component is used by the *Techniques* component to implement the SIHFT techniques. Its UML class diagram is shown in Figure 8.6. Each technique is represented by a class implementing the `Technique` interface. To keep the diagram clear, only three techniques are depicted, namely `RACFED`, `DETECTOR`, and `P_DETECTOR`. Since P-DETECTOR is based on the DETECTOR technique, the `P_DETECTOR` class inherits from the `DETECTOR` class.

To instantiate a technique, the static `Technique::create()` method is called. This method instantiates an object of the correct technique class based on the technique name provided in the function attribute.

The interface defines two methods additional: `implement` and `implementInit`. The `implement` method is used to implement the SIHFT technique, while the `implementInit` method is used to implement the initialization of said technique. These methods are used by the `SIHFT_Plugin::implement()` method of the *Entry Point* component, as shown in Listing 8.4.

Listing 8.4: The methods of the `Technique` interface are used to implement and initialize the SIHFT technique in a function. Depending on the provided function attributes, the provided technique is implemented and/or initialized.

```
unsigned int SIHFT_Plugin::execute(function *fun) {
  ...
  Technique* technique = Technique::create();
  if (Attributes::implement_technique()) {
    technique->implement();
  }
  if (Attributes::init_technique()) {
    technique->implementInit();
  }
  ...
}
```

Listing 8.5: The `Emittor` object can be accessed by calling the `emit()` method of the `ISA` class. In this example, the `Emittor::addReg()` method is used to insert an addition of two registers into the RTL code. The result shows the assembly instruction that is compiled from the inserted RTL.

```
ISA isa = RV32I();
isa.emit()->
  addReg(isa.reg(5), isa.reg(6), isa.reg(7), ...);
// result: add t0, t1, t2
```

8.4.4 Targets

To implement the SIHFT techniques, the plugin must be able to insert instructions into the RTL code. However, since the RTL code is a low-level representation of the machine code, this RTL code is dependable on the ISA of the target machine. Therefore, the *Targets* component is structured as the inheritance hierarchy shown in Figure 8.7. Each target is represented by a class that implements the `ISA` interface. By calling the static `ISA::create()` method, an object of the correct `ISA` class is created.

The `ISA` interface defines various methods that can be used to get target-specific information, such as information about which registers are available on the target machine. Each ISA class also contains four helper classes that can be used for various target-specific tasks.

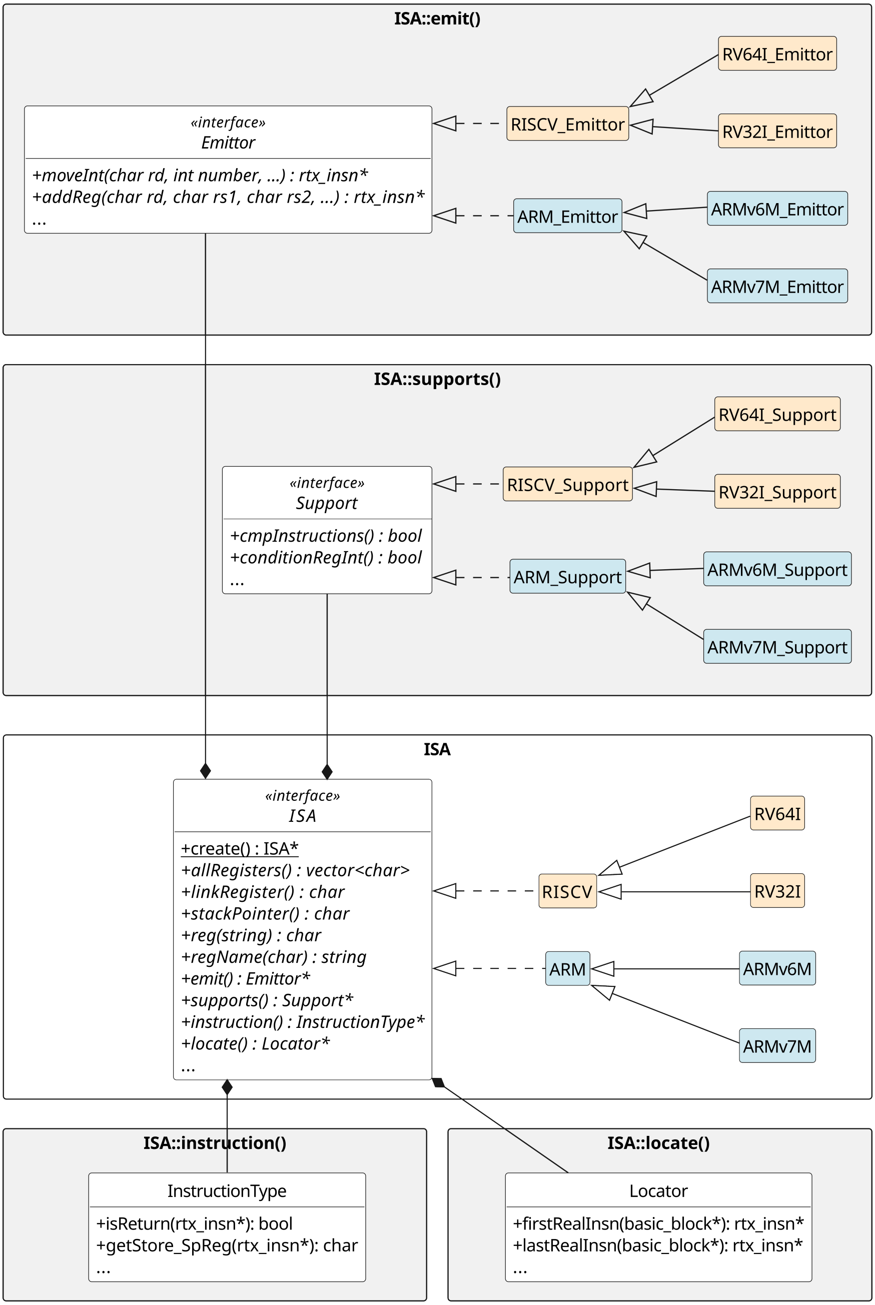


Figure 8.7: UML class diagram of the *Targets* component.

Listing 8.6: The `Support` object can be accessed by calling the `supports()` method of the `ISA` class. In this example, the `Support::conditionRegInt()` method reveals that RV32I does not support comparting a register to an integer, while ARMv7-M does.

```
ISA riscv = RV32I();
ISA arm = ARMv7M();
riscv.supports()->conditionRegInt(); // false
arm.supports()->conditionRegInt();   // true
```

Listing 8.7: The `InstructionType` object can be accessed by calling the `instruction()` method of the `ISA` class.

```
extern basic_block bb;
ISA isa = RV32I();
rtx_insn* insn = isa.locate()->lastRealInsn(bb);
isa.instruction()->isReturn(insn);
```

The `Emittor` interface defines methods to insert (emit) instructions into the RTL code. For example, the `moveInt` method will insert an instruction that inserts an integer value to a register. The `Emittor` interface is implemented by the `ARM_Emittor` and `RISCV_Emittor` classes, which specify the implementation for the ARM and RISC-V ISA families, respectively. These are in turn inherited by the target-specific Emittor classes (`RV64I`, `RV32I`, `ARMv6M`, and `ARMv7M`) to override methods where the RTL code of targets within the same ISA family differ. The `Emittor` object can be obtained by calling the `emit()` method of the `ISA` class like shown in Listing 8.5.

Not all ISAs support all types of instructions. For example, ARM targets allow for a comparison between a register and an integer, while RISC-V targets only support comparing two registers. Therefore, the `Support` interface defines methods that can be used to check if an operation is supported by the target ISA. Similarly to the `Emittor` interface, the `Support` interface is implemented by the `ARM_Support` and `RISCV_Support` classes, which are inherited by the target-specific support classes. To obtain the `Support` object, the `supports()` method of the `ISA` class can be called, resulting in the syntax shown in Listing 8.6.

To obtain details of the instructions in the RTL code, the `InstructionType` class can be used. For now, this class is not abstracted since its implementation for all currently implemented ISAs is the same. This is also the case for the `Locator` class, which can be used to locate specific instructions in the RTL code. The `InstructionType` and `Locator` objects can be obtained by calling the `instruction()` and `locate()` methods of the `ISA` class, respectively. For example, Listing 8.7 shows the code to check if the last instruction in a basic block is a return instruction.

When the plugin is compiled for an ISA family, only the classes for that ISA family (that use the GCC plugin API for that ISA family) are compiled. This means that this single inheritance tree effectively creates separate *Targets* components for each ISA family that are swapped out when compiling the plugin for different ISA families. This creates a single highly extendable codebase that can be used to support multiple ISA families.

8.5 Implementing the (P-)DETECTOR Technique

Since this thesis focuses on the novel DETECTOR and P-DETECTOR techniques, this section discusses how these techniques are implemented in the GCC plugin. As mentioned in Section 8.4.3, DETECTOR and P-DETECTOR are implemented using the similarly named classes that implement the `Technique` interface. Since the functionality of P-DETECTOR is similar to that of DETECTOR, the `P_DETECTOR` class inherits most of its functionality from the `DETECTOR` class.

The `DETECTOR::implement()` method performs three consecutive steps:

- **Locating the Vulnerable Sections:** the method searches for vulnerable instructions within the code and groups them into vulnerbale sections;
- **Finding the Checkpoints:** for each vulnerable section, the CFG is analyzed to determine from which checkpoints the section can be reached;
- **Inserting the Instructions:** the instructions to implement the DETECTOR (or P-DETECTOR) technique are inserted into the RTL code.

8.5.1 Locating the Vulnerable Sections

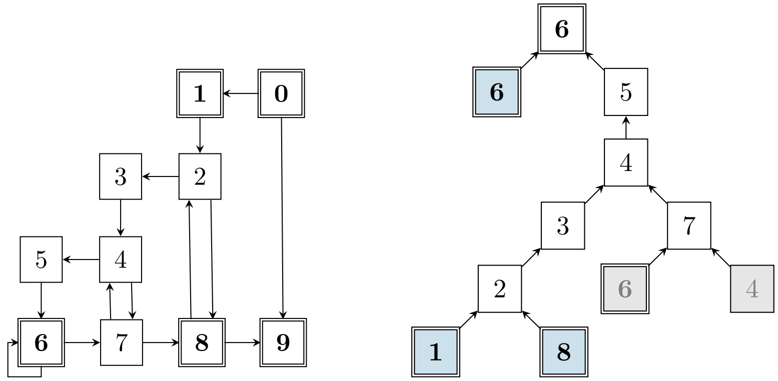
Algorithm 8.1 shows how the vulnerable sections are located in a program. The algorithm iterates over every basic block (*bb*) in the function and over every instruction in each basic block. A first instruction deemed *vulnerable* (line 6) marks the start of a new vulnerable section (lines 7 through 9). As mentioned in Chapter 4, memory write instructions, subroutine calls, and return instructions

Algorithm 8.1 The algorithm used to find the vulnerable sections in a program.

```
 1: $vulnerableSections \leftarrow [\,]$
 2: for all $bb \in function$ do
 3:     $vs, lastVs, lastCompare \leftarrow \emptyset$
 4:     $vulnerableInsn \leftarrow [\,]$
 5:     for all $instruction \in bb$ do
 6:         if $instruction$ is vulnerable then
 7:             if $vs = \emptyset$ then
 8:                 $vs \leftarrow$ new VulnerableSection($lastVs$)
 9:                 $vs$.setStart($instruction$)
10:             $vs$.addVulnerableInstruction($instruction$)
11:             if $instruction$ is conditional then
12:                 $vs$.setStart($lastCompare$)
13:             if $instruction$ is return then
14:                 $vs$.setExit(true)
15:         if $vs \neq \emptyset$ then
16:             if $instruction$ is not vulnerable or
17:                $instruction = bb$.last() then
18:                 $vulnerableSections$.add($vs$)
19:                 $lastVs \leftarrow vs$
20:                 $vs \leftarrow \emptyset$
21:         if $instruction$ is compare then
22:             $lastCompare \leftarrow instruction$
23: return $vulnerableSections$
```

are considered vulnerable instructions. This instruction and each consecutive vulnerable instruction are added to this vulnerable section (line 10). If, however, a vulnerable instruction is conditional, the instruction that sets the condition flags (the $lsatCompare$ variable set in lines 21 through 22) should be included in the vulnerable section (lines 11 through 12). Additionally, the vulnerable section gets marked as an *exit* section when an instruction in the vulnerable section is a return statement (lines 13 through 14).

The next non-vulnerable instruction or the end of the basic block marks the end of a vulnerable section (lines 15 through 20). This way, each vulnerable section in each basic block is identified. Multiple vulnerable sections in one single basic block follow a reversed linked-list data structure, where each vulnerable section points to the previous vulnerable section within the basic block ($lastVs$, line 8).



(a) A simplified control flow graph showing the vulnerable blocks.

(b) The source tree for vulnerable block 6.

Figure 8.8: Finding the source vulnerable blocks using a control flow graph.

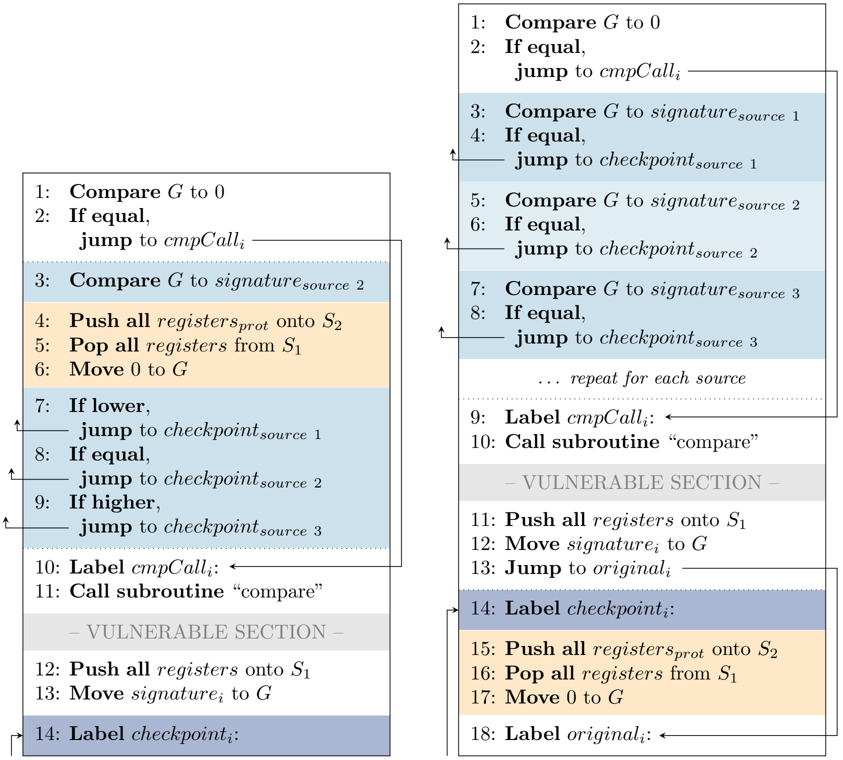
8.5.2 Finding the Checkpoints

As discussed in Chapter 4, a re-execution point is placed before each vulnerable section and a checkpoint is placed after each vulnerable section. When reaching a re-execution point during the first execution cycle, the program should jump to the last visited checkpoint to start the second execution cycle. At compile time, the plugin must therefore determine from which checkpoints each vulnerable section can be reached.

Figure 8.8a shows a simplified control flow graph of a program. In this graph, the basic blocks that contain one or more vulnerable sections are marked with a double border. From now on, these will be referred to as *vulnerable blocks*. For simplicity's sake, the first basic block is also considered a vulnerable block, although it might technically not be one.

For each vulnerable block, a recursive path analysis is performed to find all possible paths between the vulnerable blocks. During this analysis, a source tree is constructed, starting from a vulnerable block, following each incoming edge until a vulnerable block is reached or until a basic block that has already been traversed is encountered.

The source tree for vulnerable block 6 is shown in Figure 8.8b. The source checkpoints of the first vulnerable section in a vulnerable block are the source vulnerable blocks' last vulnerable sections. All subsequent vulnerable sections in the same vulnerable block have source checkpoints at the preceding vulnerable section, using the reversed linked-list structure mentioned before.



```
Compare $G$ to 0
If equal,
    jump to $cmpCall_i$
Compare $G$ to $signature_{source\ 2}$
Push all $registers_{prot}$ onto $S_2$
Pop all $registers$ from $S_1$
Move 0 to $G$
If lower,
    jump to $checkpoint_{source\ 1}$
If equal,
    jump to $checkpoint_{source\ 2}$
If higher,
    jump to $checkpoint_{source\ 3}$
Label $cmpCall_i$:
Call subroutine "compare"
– VULNERABLE SECTION –
Push all $registers$ onto $S_1$
Move $signature_i$ to $G$
Label $checkpoint_i$:
```

(a) Using separate compare and conditional branch instructions.

```
Compare $G$ to 0
If equal,
    jump to $cmpCall_i$
Compare $G$ to $signature_{source\ 1}$
If equal,
    jump to $checkpoint_{source\ 1}$
Compare $G$ to $signature_{source\ 2}$
If equal,
    jump to $checkpoint_{source\ 2}$
Compare $G$ to $signature_{source\ 3}$
If equal,
    jump to $checkpoint_{source\ 3}$
... repeat for each source
Label $cmpCall_i$:
Call subroutine "compare"
– VULNERABLE SECTION –
Push all $registers$ onto $S_1$
Move $signature_i$ to $G$
Jump to $original_i$
Label $checkpoint_i$:
Push all $registers_{prot}$ onto $S_2$
Pop all $registers$ from $S_1$
Move 0 to $G$
Label $original_i$:
```

(b) Placing part of the re-execution code after the vulnerable section.

Figure 8.9: Depending on which instructions are supported by the ISA and how the CFG of the program is structured, the instructions to implement the DETECTOR technique can be inserted in two different ways.

8.5.3 Inserting the Instructions

The final task is to insert the actual instructions to implement the DETECTOR or P-DETECTOR technique. This is done by prepending and appending the necessary instructions to each vulnerable section. There are two ways that this can be achieved, depending on which instructions are supported by the ISA and how the CFG of the program is structured. These two approaches are shown in Figure 8.9 for the DETECTOR technique.

The first approach (Figure 8.9a) is the most straightforward. This is also the version that was used in Chapter 4 to discuss how the DETECTOR technique works. All instructions for the re-execution point are prepended before each vulnerable section. Notice that the compare instruction at line 3 is separate from the conditional branch instructions at lines 7 through 9 since the instructions inserted in lines 4 through 6 change the value stored in the G register.

There are two drawbacks to this approach. Firstly, not all instruction set architectures provide separate *compare* and *conditional branch instructions*. For example, the RISC-V ISAs combine the two into conditional branch instructions with three operands: the two registers to compare and the label to jump to. Secondly, this approach is only possible when each vulnerable section in the program contains no more than three source checkpoints, since only the conditions *lower than*, *equal*, or *higher than* can be used.

When the first approach is not possible, the SIHFT plugin switches to the second approach, shown in Figure 8.9b. Here, the compare instructions can be placed together (or if the ISA allows for it, they can be combined) by reordering the instructions, placing part of the re-execution code after the checkpoint label. By inserting an unconditional jump at line 13, the re-execution code inserted after the vulnerable section is skipped during the first execution cycle.

8.6 Conclusion

This chapter described how the GCC plugin to support SIHFT techniques of the M-Group research group was extended and refactored to support multiple ISA families and how the DETECTOR, S-DETECTOR, and P-DETECTOR techniques were implemented in this plugin. The plugin is designed to be highly extendable and can be used to implement any low-level SIHFT technique. By creating a single codebase that can be compiled for different ISA families, the plugin can be used to implement SIHFT techniques for multiple ISA families. The plugin supports various ARM and RISC-V architectures and is made to be easily extendable to support other ISAs by implementing the provided interfaces.

The DETECTOR and P-DETECTOR techniques have been implemented in the plugin in three separate steps to locate the vulnerable sections, find the source checkpoints, and insert the necessary instructions. Two different approaches are used to insert the instructions, depending on the supported instructions of the ISA and the structure of the CFG of the program.

Chapter 9

Valorization

During this thesis, two components that hold direct economic value have been developed, namely the GCC SIHFT plugin described in Chapter 8 and the Fault Injection Frameworks described in Chapter 3. This chapter first discusses the open-sourced GCC SIHFT plugin in Section 9.1, which can be used to easily and quickly implement SIHFT techniques to protect their embedded systems against soft errors. Next, Section 9.2 covers the fault injection frameworks, which can be used by the industry to evaluate the resilience of their embedded systems against soft errors. Together with the GCC plugin, they create a full resilience framework that can be used by the industry to evaluate and enhance the resilience of their embedded systems. Additionally, the research on the RISC-V extension in Chapter 7 is the first step in the research of hardware-supported SIHFT techniques and is therefore leverage for new fundamental research projects. This is discussed in Section 9.3.

9.1 The GCC SIHFT Plugin

As mentioned in Chapter 8, the SIHFT plugin was developed to automate the implementation process of low-level SIHFT techniques, as manually implementing such techniques is infeasible. While the initial version of the plugin was mainly developed for research ends, the possibility of using the plugin in commercial applications was always considered. With the latest version of the SIHFT plugin, which offers support for multiple ISAs, this commercial use is more feasible than ever.

The plugin allows any embedded developer using the GCC compiler toolchain to implement a SIHFT technique, as long as their target instruction set architecture is supported. By using this plugin in their compilation process, the industry can create bitflip-resilient systems, giving them a competitive advantage in both the national, and international markets. Moreover, the use of such resilience techniques might aid companies to comply with functional safety standards to which mission and safety-critical systems have to comply. For example, the functional safety standard for road vehicles, ISO 26262, explicitly recognizes the soft error problem and recommends the *detection of data errors* and recommends *control flow monitoring* to be implemented.

The main market for SIHFT techniques is manufacturers of safety or mission-critical embedded systems that need protection against soft errors. This might be to comply with the aforementioned safety standards or to protect the systems in an environment with high radiation. As mentioned in Chapter 1, transistor-level and circuit-level mitigation against soft errors is very costly, often making them a non-viable option for many companies. In these cases, SIHFT techniques can be utilized instead. Additionally, SIHFT techniques can be implemented for a subset of the program, meaning that the most critical parts of the program can be protected, while other less critical parts can remain as-is, improving speed and/or energy efficiency. Finally, SIHFT techniques can also be an option for systems that suffer from high soft error rates but cannot be modified since they are already deployed. This could, for example, be in space applications, where a firmware update can be issued to combat errors as a result of ionizing particles in space.

The GCC SIHFT plugin has been made available as an open-source project on GitLab [28]. This repository includes a detailed guide on how to use the plugin to implement the various techniques. Fellow researchers within the SIHFT research domain are encouraged to implement their novel SIHFT techniques in the plugin by submitting merge requests to the open-source repository. This allows for greater transparency for all parties and allows for direct and unbiased comparisons between different techniques, an important factor in the research integrity of the field. This could also resolve possible erroneous interpretations of the techniques, as implementing SIHFT techniques solely based on a description in a published work can result in missed nuances and therefore less effective implementations. Additionally, this would be beneficial for the industry since this would increase the choice of SIHFT techniques.

While the latest version of the plugin can be used commercially, it is not yet validated in a real-world scenario. Active research is being conducted on methods to verify the correctness of the plugin [95], as this is an important yet non-trivial task.

Like all work derived from the GNU Compiler Collection, the SIHFT technique is licensed under the GNU General Public Licence (GPL), which guarantees end users the freedom to run, study, share, and modify the software. As per the GPL licensing, the plugin is provided “as is”, without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The license holder is in no event unless required by applicable law or agreed to in writing liable for damages, including any general, special, incidental, or consequential damages arising out of the use or inability to use the software.

While GCC is a widely used compiler toolchain, some developers use other compilers, such as the Clang/LLVM Compiler Infrastructure. If there is industry interest, a Clang plugin can be developed to further increase the reach of SIHFT techniques.

9.2 Fault Injection Frameworks

The two fault injection frameworks discussed in Chapter 3 have been mainly developed to evaluate the novel SIHFT techniques and compare them against the SOTA. However, these frameworks, together with the previously discussed GCC plugin, create a complete toolset to evaluate the resilience of embedded systems against soft errors and to determine which SIHFT technique is the best fit for a specific application. Using this toolset, along with the acquired expertise in the past decade in this field, the KU Leuven M-Group can provide a consultancy service in close cooperation with the industry. This service can take place through several distinct user scenarios.

- **Vulnerability-Only Service:** Conduct a vulnerability assessment on an embedded system and provide the results to the user.
- **Suggestion Service:** Assess the vulnerability of an embedded system and assess how various SIHFT techniques improve the resilience of the program. Provide the user with the results of this analysis, recommending which SIHFT techniques are the most suitable, taking into consideration the requirements of the end user.
- **Microcontroller or ISA Resilience Service:** For clients uncertain about which microcontroller or ISA to choose for their project, this service provides insights into the resilience of different candidates against CFEs and DFEs. It is also useful for clients setting up a hardware redundancy system, offering guidance on selecting the most resilient ISA or microcontroller.

This envisions close cooperation with the industry through service agreements. The cooperation would involve the industry providing the full embedded system, including software and hardware to the M-Group, who will first, in close contact with the client, determine the most suitable way to test the system. If a hardware-in-the-loop simulator setup is chosen, a HIL simulator would be set up in the M-Group lab, and the embedded system would be tested against the various fault injection campaigns. If the suggestive service was chosen, multiple SIHFT techniques would be applied to the target system, which would be tested using the same fault injection strategies. Additionally, the overhead analysis tools used in this research can be applied here to provide a full overview of the capabilities and impact of each SIHFT technique. Thus, by combining the toolsets developed in this research, the M-Group can provide a complete framework to the industry to evaluate and enhance the resilience of its embedded systems.

In a later phase, the M-Group's consultancy status can be further expanded by providing a cloud platform and different licensing models in the form of a Software as a Service (SaaS) model. In this model, clients would be able to upload their embedded software to the platform for resilience testing and assessment based on the scenarios outlined above. This approach has the potential to attract new clients who may not have been reached through the initial service agreement plan while ensuring that intellectual property remains in-house.

Alternatively, the developed tools could be made commercially available to be purchased or licensed. This reduces the active role of the M-Group, while still providing the industry with the tools to evaluate the resilience of their embedded systems.

The commercialization and licensing of the tools developed in this thesis, as well as the establishment of potential service agreements, will be done in discussion with the KU Leuven Research & Development department. Since all tools used in this research are developed in-house, KU Leuven owns all IP related to the fault injection frameworks. However, the simulation-based fault-injection framework uses the Imperas Instruction Set simulator, which has been licensed to KU Leuven for research use only. If the simulated fault-injection framework were to be used commercially, a commercial licensing agreement has to be made with Imperas Software Ltd.

In the context of the FIRES research project [96], seven letters of interest have been received from companies interested in the envisioned framework. These companies are from various domains, including agriculture, healthcare, IoT systems, and large machinery, showing a broad interest in the developed tools.

9.3 RISC-V Extension

While the previous two sections discussed the direct economic value of the developed tools, the valorization of the RISC-V extension is more focused on the leverage to future fundamental research projects. The RISC-V extension explored in this thesis is the first step in the research of hardware-supported SIHFT techniques. Further research in this domain could reveal the true value of such hardware-supported techniques.

The Computer Science and Electrical Engineering Departments of KU Leuven have the capacity and know-how of both SIHFT techniques and (RISC-V) hardware development to further research this domain. For example, the RISC-V extension can be implemented on the Proteus code [79] of the KU Leuven DistriNet research group, an easily configurable RISC-V CPU implemented in SpinalHDL. This would reveal the implementation effort needed to implement the ISA extension in hardware and can provide an insight into the performance overhead of the extension. Together with the partners of KU Leuven, such as the Interuniversity Micro-Electronics Center, Imec, new research projects can be set up to further research this novel domain.

The IP of a RISC-V implementation can be retained by KU Leuven if so desired. However, if the implementation of the extension is based on an existing model, the IP rights of the original model have to be respected.

Chapter 10

Directions for Future Work and General Conclusions

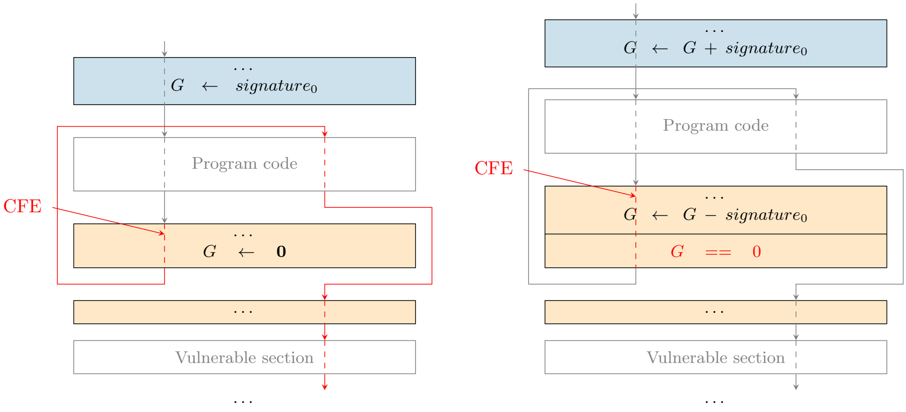
This chapter concludes this thesis by first providing some possible future work that can be done to improve the (P-)DETECTOR technique, as well as further research opportunities that can be explored regarding hardware-supported SIHFT techniques by using extendable architectures like RISC-V in Section 10.1. The chapter concludes with some closing remarks in Section 10.2, reflecting on the research objectives and the contributions made in this thesis.

10.1 Future Work

10.1.1 Borrowing Ideas from Signature Monitoring

The (P-)DETECTOR technique uses a single signature variable to keep track of the checkpoints from which the program parts must be re-executed. This variable is updated to a compile-time signature every time a checkpoint is reached and zeroed out when the program starts the re-execution of the program. It is however not used for any detection mechanism. This is where the control flow error detection capabilities of the (P-)DETECTOR technique can be improved by incorporating small signature-checking mechanisms throughout the program.

Other ideas can also be borrowed from signature monitoring techniques, like the fact that signatures are often not merely assigned, but updated throug addition [38] or exclusive or operations [31]. Because of this, a corrupted


...
$G \leftarrow signature_0$
Program code
CFE
...
$G \leftarrow \mathbf{0}$
...
Vulnerable section
...
...
$G \leftarrow G + signature_0$
Program code
CFE
...
$G \leftarrow G - signature_0$
$G == 0$
...
Vulnerable section
...

(a) The current implementation of (P-)DETECTOR updates the signature register G by assigning the new values. A CFE to the re-execution can remain undetected.

(b) A possible improvement would be to update the signature register by addition and subtraction & verifying the signature value. The previously undetected CFE is now detected by the new signature assertion.

Figure 10.1: Borrowing ideas from signature monitoring techniques can improve the (P-)DETECTOR technique.

register variable is not just overwritten, causing more errors to be caught by the signature-checking mechanism.

Figure 10.1 shows how combining a signature-checking mechanism with the signature updates trough addition could result in an improved control flow error detection mechanism for the (P-)DETECTOR technique. A CFE to a re-execution point could remain undetected in the current implementation of P-DETECTOR, like shown in Figure 10.1a. The same CFE is detected in Figure 10.1b. Here, the signature register G is updated to *zero* by subtracting the expected signature ($signature_0$). If a CFE from another part of the program (i.e. with a different signature in G) occurs, this operation will not result in G being zero. The new signature assertion will detect this, triggering the error handler.

10.1.2 From Error Detection to Error Correction

While some techniques, like SWIFT-R and TRUMP [46], have incorporated error correction mechanisms, the focus of SOTA SIHFT techniques mainly revolve around error detection, relying on the programmer's implementation of the error handler to recover from the error or to put the system in a safe state. This is because, for traditional DFED and hybrid techniques, the error correction mechanism relies on instruction triplication and utilizing a majority voting mechanism to determine the correct register values. This means that not one but two shadow registers are required for every register used in the program to store the redundant values. Therefore, the available registers for the program is reduced to a third of the original amount, making the register availability issue mentioned in Chapter 2 even more severe.

A possible error correction mechanism for (P-)DETECTOR could be to re-execute the part of the program that caused the error for a third time, therefore rewriting a corrupted register with a correct value. This mechanism would not require any additional registers and would not further impact the performance of the program, as the third re-execution would only occur when an error is detected. This mechanism could be examined in future research to determine its effectiveness in automatically correcting errors caused by bitflips.

10.1.3 Synthesizing the RV32IXpdetector Core

In Chapter 7, the possibility to create specialized cores for SIHFT techniques by using RISC-V extensions was explored by defining an Xpdetector ISA extension that introduces specialized instructions for the P-DETECTOR technique. The extension was tested by creating a simulator model for the RV32IXpdetector core, to perform fault injection experiments on five case studies, evaluating the effectiveness of the P-DETECTOR technique on the extended architecture. The next step in this research would be to synthesize the RV32IXpdetector core on an FPGA to evaluate the effort required to implement the ISA extension in hardware and to determine the real performance impact of P-DETECTOR when used with the extension.

To implement the Xpdetector extension, an extendable RISC-V FPGA implementation like the Proteus Core [79] could be used. This RV32IM SpinalHDL model uses a system of plugins to make the processor easily configurable and extensible, making the development and testing of hardware features easier.

10.1.4 Further Research on Hardware-Supported SIHFT Techniques

While Chapter 7 explored the idea of using RISC-V extensions to create specialized cores for SIHFT techniques, this has only been applied this to the P-DETECTOR technique. Further research can be performed to explore the possibilities of using RISC-V extensions to support other SIHFT techniques or categories of SIHFT techniques. For example, instruction monitoring techniques or signature monitoring techniques could be supported by introducing specialized signature registers that keep track of the program's position within the control flow graph.

10.2 Concluding Remarks

This thesis has explored low-level bare-metal software-implemented hardware fault tolerance techniques for embedded devices and how they can be used to protect systems against soft errors like single event upsets. In Chapter 1, defined three research objectives to be achieved in this thesis were defined:

RO1. Evaluate the applicability of state-of-the-art SIHFT techniques on more industrially applicable case studies;

RO2. Develop new SIHFT techniques that address the limitations discovered in RO1;

RO3. Investigate how a custom RISC-V extension can be created to support SIHFT techniques.

RO1: The first research objective was addressed in Chapter 2, where the SOTA SIHFT techniques were discussed, showing how they can be used to protect systems against control flow errors and data flow errors. However, this work also showed that these techniques require a lot of registers to store redundant values, which can be a problem for embedded systems with limited resources. This is especially true for DFED and hybrid techniques, which rely on shadow registers to store redundant register values. Many applications are not able to compile with the SOTA techniques because of this, rendering them unusable in these cases. This major limitation in the applicability of SOTA SIHFT techniques was the main motivation for the development of the techniques in this thesis.

In order to evaluate the effectiveness of the SIHFT techniques developed in this thesis, a simulation-based and a HIL-based fault injection framework was used on several data-driven and I/O-driven case studies. This was detailed in Chapter 3, which showed how the results of the fault injection campaigns were classified, bearing in mind that systems protected with different SIHFT techniques have different fault-spaces.

RO2: To address the register availability issue, Chapter 4 introduced the DETECTOR technique, which only uses three CPU registers to detect soft errors. DETECTOR introduces the concept of vulnerable sections as parts of the program where it must be ensured that the program's state is correct. At the start of the program and after each vulnerable section, a checkpoint is introduced. Before each vulnerable section, a re-execution point is inserted. When the program reaches a re-execution point, the DETECTOR mechanism stores the program's state in a shadow stack in memory and reloads the program's state from the last reached checkpoint, which is identified by a unique signature value. After re-executing the program from that checkpoint, the re-execution point is reached again, at which point the program's current state is compared to the stored state. Differences indicate that a soft error has occurred, triggering an error handler that can be implemented by the developer of the application. Fault injection experiments on the data-processing case studies showed that DETECTOR can reliably detect soft errors, reducing the average silent data corruption ratio from 54.6% to 5.7% for CFEs and from 49.0% to 4.7% for DFEs. However, the fault injection campaigns on the I/O-driven case studies also revealed that DETECTOR can struggle in some cases, even introducing more vulnerable code to the program.

Since DETECTOR only uses three CPU registers, it can be used on virtually all applications, making it a viable solution for systems where other DFE or hybrid techniques fail. DETECTOR does, however, introduce a large execution time overhead, especially for applications with a vulnerable section in a hot loop. To address this issue, Chapter 5 introduced two optimizations. The first optimization, called S-DETECTOR introduces a selective version of DETECTOR, where only the registers that are most often used to access memory are protected. This reduces the overhead of the DETECTOR technique, while still providing a good level of error detection, trading off error detection capabilities for performance. As expected, the fault injection results show a reduced error detection ratio compared to DETECTOR, with on average 10.2% of faults resulting in a silent data corruption when using S-DETECTOR, compared to the 5.7% when using DETECTOR. Meanwhile, the selective approach of S-DETECTOR reduces the ETO significantly, although the overhead is still quite high for some case studies.

The second optimization introduced in Chapter 5 is called P-DETECTOR. P-DETECTOR utilizes parity checking to verify that the two execution cycles are equal, instead of comparing each register value one by one. This does not only decrease the overhead of DETECTOR, but also improves the error detection capabilities of the technique, detecting more CFEs than DETECTOR, reducing the SDC ratio of control flow errors to a mere 3.4% on average for the data-processing case studies. The techniques developed in Chapter 4 and Chapter 5 show that it is possible to develop SIHFT techniques that do not require many registers to be reserved, thereby completing RO2.

RO3: Chapter 7 explored the possibilities of using custom RISC-V extensions to support SIHFT techniques. The idea was to use the extendable nature of the RISC-V instruction set architecture with its possibility to create custom extensions to introduce specialized instructions that can be used to implement SIHFT techniques more efficiently. As a proof-of-concept, this work describes the Xpdetector extension, which introduces specialized instructions for the P-DETECTOR technique. This extension introduced four new instructions that use special-purpose registers to store the parity value, signature value and register values at the checkpoints. Chapter 7 detailed how this extension fits within the RISC-V encoding space and how the extension was simulated using the Imperas instruction set simulator by extending an OVP RV32I model. While no exact overhead measurements can be measured when using the simulated model, the estimations show that the P-DETECTOR implementation using the Xpdetector extension can be implemented very efficiently, reducing the overhead significantly. This completes the third research objective.

Since SIHFT techniques are implemented in assembly code, implementing them manually is infeasible. To make SIHFT techniques more accessible to developers, the M-Group research group developed two GCC compiler plugins to automatically insert the necessary instructions for several CFED or DFED techniques on ARM-based systems. Chapter 8 detailed how these two existing GCC plugins were combined into one easily extendable plugin intended to support a wide range of SIHFT techniques, as well as multiple ISA families. This was done by compartmentalizing the plugin into four components that each have their own responsibility. Techniques can be easily implemented by implementing the `Technique` interface in the *Techniques* component, while support for ISAs can be added by extending the classes in the *Targets* component. This way, a single codebase can compile multiple versions of the SIHFT plugin for different ISA families.

Finally, Chapter 9 described how the research results of this thesis can be valorized. More specifically, it described how the GCC SIHFT plugin was open-sourced so that it can be used by industry to protect their systems against soft errors. Additionally, the chapter showed how the fault injection tools used to evaluate the SIHFT techniques can be combined with that GCC plugin to create a full resilience framework that can be used by industry. In this light, the M-Group at KU Leuven Bruges can offer a consultancy role to help the industry evaluate and improve the resilience of their systems. Four user scenarios were explored, ranging from a pure vulnerability assessment to a suggestion service where multiple SIHFT techniques are evaluated on the system to find the best fit. In a later phase, this solution could be marketed as a SaaS solution where most of this is automated, thereby also lowering the barrier for companies that want to explore these solutions.

To conclude, all three research objectives were successfully achieved, contributing to the research field that creates highly resilient embedded systems. The state-of-the-art techniques were evaluated, their limitations were identified, and the DETECTOR, S-DETECTOR, and P-DETECTOR techniques were introduced to overcome the limited register availibility challenge that was discovered. Moreover, the introduction of the Xpdetector RISC-V extension demonstrated the potential for hardware-supported SIHFT techniques, while the development of a flexible GCC plugin, together with the fault-injection frameworks made SIHFT techniques more accessible to developers. The valorization plan further underscores the potential impact of this research, suggesting pathways for the industry to adopt and integrate these advancements, thereby contributing to the broader goal of creating more robust and reliable embedded systems.

Appendix A

Overview of All Fault Injection Results

A.1 Fault Injection Campaigns on the Data Processing Case Studies on ARMv7-M

Table A.1: Details of the fault injection campaigns on the data processing case studies on the ARMv7-M ISA.

		instruction count	execution time (ms)	injected CFEs	injected DFEs
BC	P-DETECTOR	61	39	2239	31 583
	S-DETECTOR	38	38	1912	31 132
	DETECTOR	67	39	1912	31 424
	RACFED	41	19	2442	-
	CFCSS	35	16	1869	-
	FDSC	19	11	-	5869
	SWIFT	45	18	-	8256
	unprotected	9	7	517	11 712
BS	P-DETECTOR	96	4015	101 330	117 440
	S-DETECTOR	68	2567	100 740	113 760
	DETECTOR	94	4158	100 870	116 160
	RACFED	73	743	6025	-
	CFCSS	63	574	5730	-
	FDSC	51	938	-	12 960
	SWIFT	93	638	-	15 200
	unprotected	21	335	137 068	103 680
CRC	P-DETECTOR	71	29	100 590	161 161
	S-DETECTOR	46	31	100 410	107 840
	DETECTOR	75	31	100 410	109 440
	RACFED	63	25	5440	-
	CFCSS	45	21	5310	-
	FDSC	48	33	-	12 320
	SWIFT	56	25	-	14 560
	unprotected	19	14	90 115	102 880
CU	P-DETECTOR	1768	25	45 733	418 144
	S-DETECTOR	1261	25	32 601	418 023
	DETECTOR	1279	26	32 663	418 144
	RACFED	576	20	8891	-
	CFCSS	385	20	5523	-
	unprotected	290	20	5119	42 336

The table continues on the next page.

Table A.1 (cont.): Details of the fault injection campaigns on the data processing case studies on the ARMv7-M ISA.

		instruction count	execution time (ms)	injected CFEs	injected DFEs
DIJ	P-DETECTOR	314	10 223	103 374	176 000
	S-DETECTOR	247	9989	102 292	174 340
	DETECTOR	256	10 462	102 292	175 840
	RACFED	186	10 651	8555	-
	CFCSS	138	9749	7547	-
	unprotected	66	4210	100 590	109 056
FFT	P-DETECTOR	1763	5172	147 394	615 616
	S-DETECTOR	1285	4921	132 588	613 223
	DETECTOR	1300	5915	132 588	614 304
	RACFED	633	1984	16 722	-
	CFCSS	435	1994	12 552	-
	unprotected	288	1856	105 735	161 760
MM	P-DETECTOR	216	126	102 535	129 920
	S-DETECTOR	159	108	101 705	127 040
	DETECTOR	186	142	101 813	131 840
	RACFED	119	27	6950	-
	CFCSS	111	22	6545	-
	FDSC	189	46	-	45 920
	SWIFT	131	32	-	51 360
	unprotected	45	12	100 325	108 800
QS	P-DETECTOR	186	175	104 525	157 984
	S-DETECTOR	141	157	103 328	149 792
	DETECTOR	171	193	103 328	157 696
	RACFED	184	85	8794	-
	CFCSS	160	76	8512	-
	FDSC	256	114	-	66 912
	SWIFT	193	112	-	77 216
	unprotected	59	30	101 097	116 768

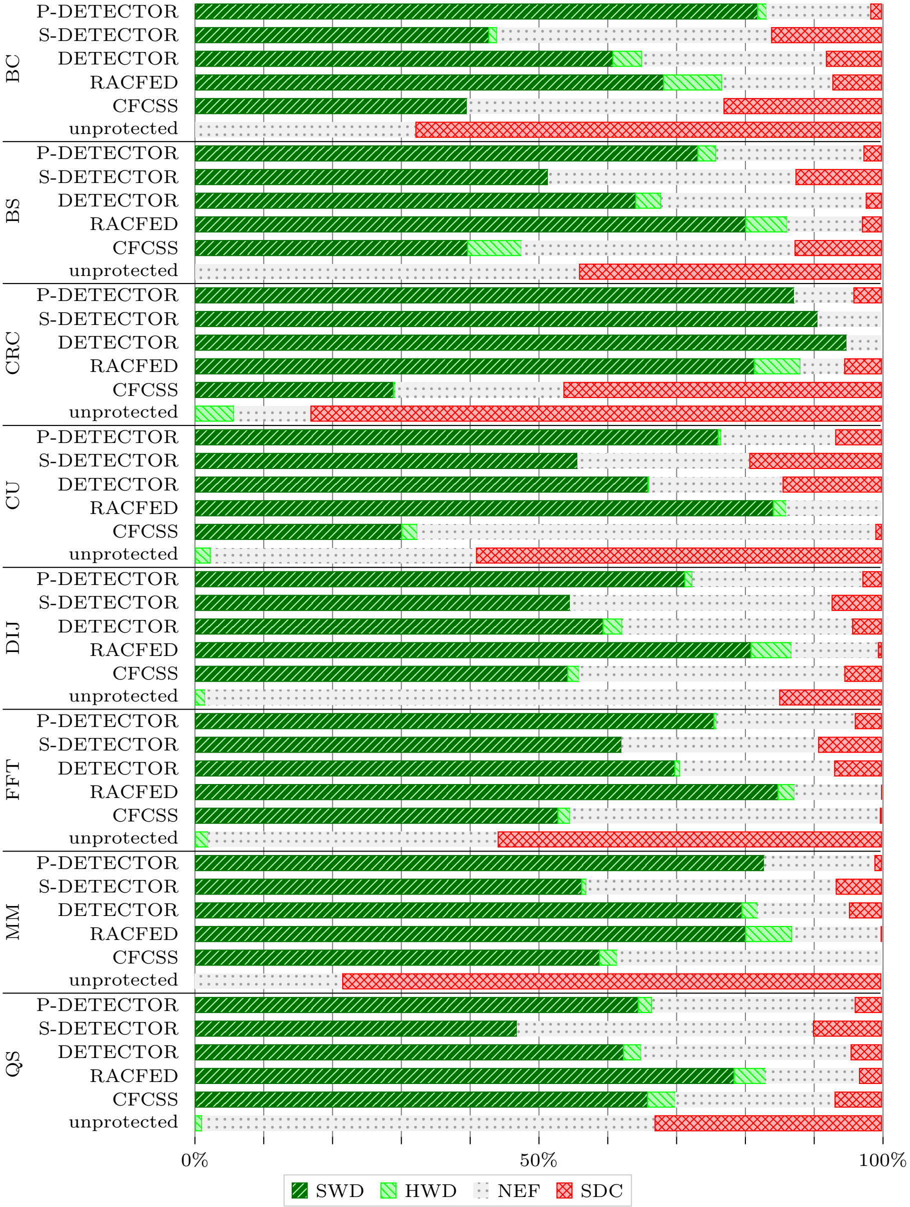


Figure A.1: The results of the single-bit CFE fault injection campaigns on the data-processing case studies protected with CFCSS, RACFED, DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA.

Table A.2: The average results of the single-bit CFE fault injection campaigns on the data-processing case studies protected with CFCSS, RACFED, DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA.

	SWD	HWD	NEF	SDC
P-DETECTOR	76.5%	1.0%	19.1%	3.4%
S-DETECTOR	57.5%	0.2%	32.1%	10.2%
DETECTOR	69.6%	2.1%	22.7%	5.7%
RACFED	79.7%	5.4%	12.4%	2.5%
CFCSS	46.2%	2.6%	39.3%	12.0%
unprotected	-	1.5%	43.9%	54.6%

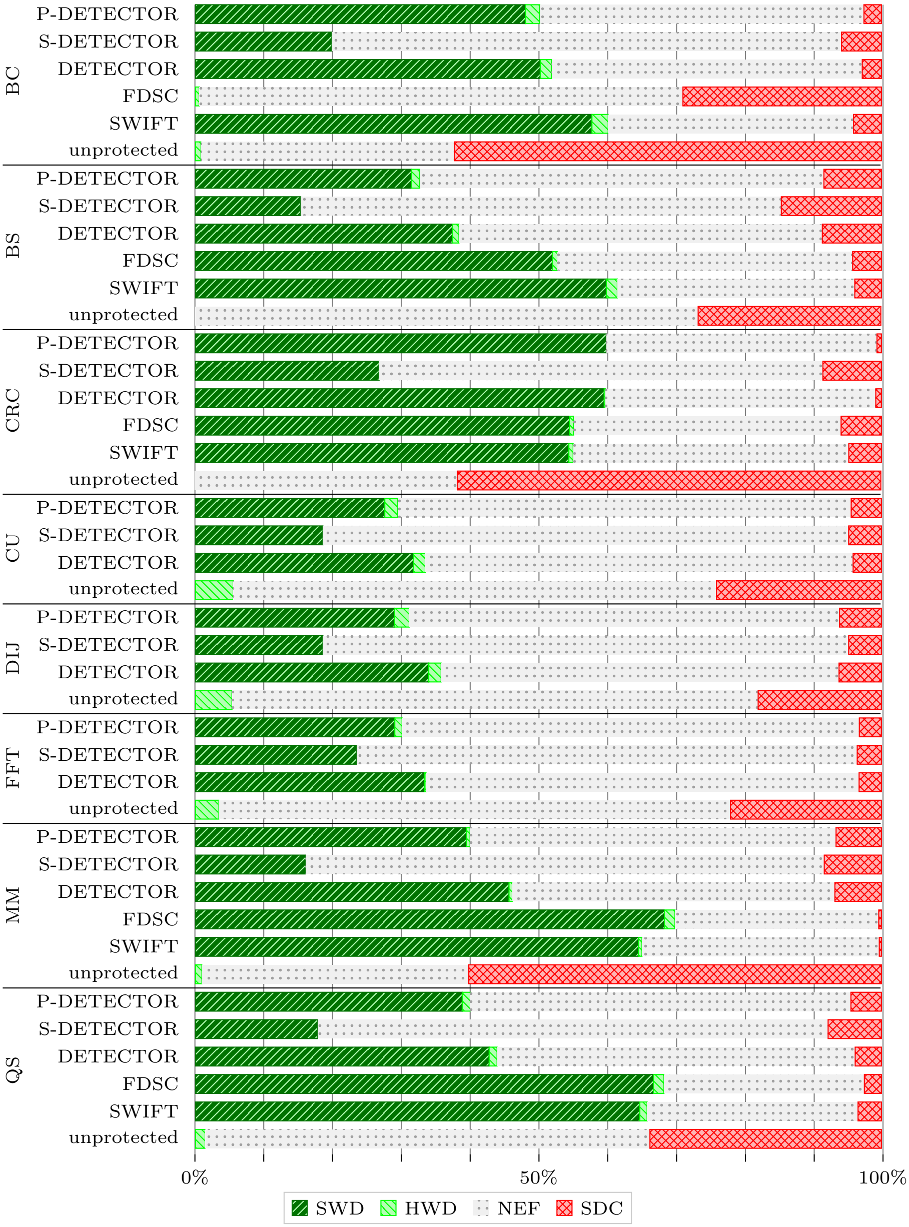


Figure A.2: The results of the single-bit DFE fault injection campaigns on the data-processing case studies protected with SWIFT, FDSC, DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA.

Table A.3: The average results of the single-bit DFE fault injection campaigns on the data-processing case studies protected with SWIFT, FDSC, DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA.

	SWD	HWD	NEF	SDC
P-DETECTOR[1]	43.5%	1.0%	50.8%	4.6%
S-DETECTOR[1]	19.1%	0.0%	71.7%	9.1%
DETECTOR[1]	47.1%	0.9%	47.3%	4.7%
FDSC[1]	48.3%	1.0%	42.2%	8.5%
SWIFT[1]	60.2%	1.2%	35.1%	3.4%
unprotected[1]	-	0.7%	50.4%	49.0%
P-DETECTOR[2]	37.9%	1.3%	56.2%	4.7%
S-DETECTOR[2]	19.5%	0.0%	73.1%	7.4%
DETECTOR[2]	41.8%	1.0%	52.5%	4.7%
unprotected[2]	-	2.2%	59.2%	38.7%

[1] Results for case studies BC, BS, CRC, MM, and QS.

[2] Results for case studies BC, BS, CRC, CU, DIJ, FFT, MM, and QS.

A.2 Fault Injection Campaigns on the I/O-Driven Case Studies on ARMv7-M

Table A.4: Details of the fault injection campaigns on the I/O-driven case studies on the ARMv7-M ISA.

		instruction count	execution time (ms)	injected CFEs	injected DFEs
Distr	P-DETECTOR	2432	8333	8401	24 000
	S-DETECTOR	1737	8352	5613	24 000
	DETECTOR	1885	8351	6308	24 000
	RACFED	1478	8344	5176	-
	CFCSS	1127	8334	4077	-
	unprotected	592	8342	2291	13 398
Test	P-DETECTOR	1893	9216	7521	18 000
	S-DETECTOR	1358	9264	5664	18 000
	DETECTOR	1443	9218	5970	18 000
	RACFED	1098	9229	4701	-
	CFCSS	848	9228	3257	-
	unprotected	452	9194	2026	11 040
Sort	P-DETECTOR	1595	6705	5730	12 000
	S-DETECTOR	1176	6883	7906	12 000
	DETECTOR	1250	6752	4597	12 000
	RACFED	1012	6951	3952	-
	CFCSS	775	6771	2989	-
	unprotected	405	7049	1724	8720

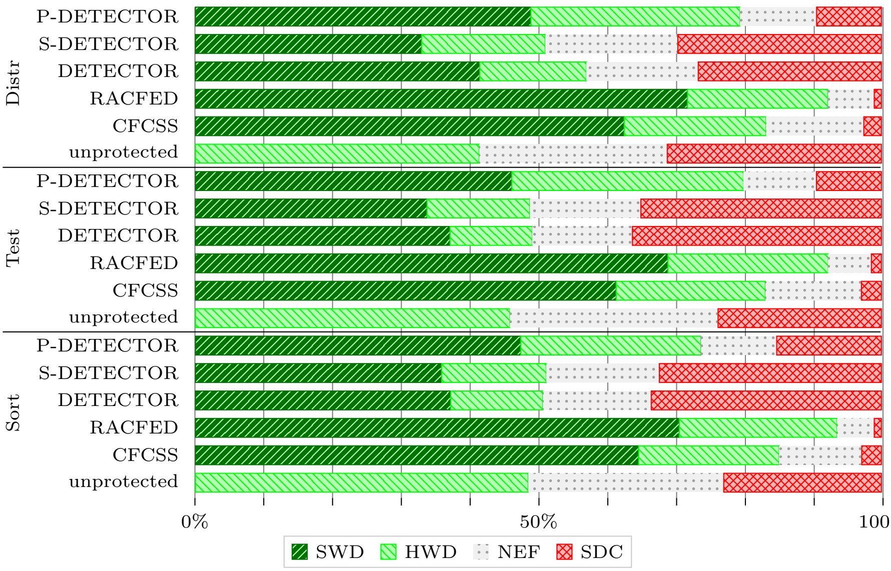


Figure A.3: The results of the single-bit CFE fault injection campaigns on the I/O-driven case studies protected with CFCSS, SWIFT, DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA.

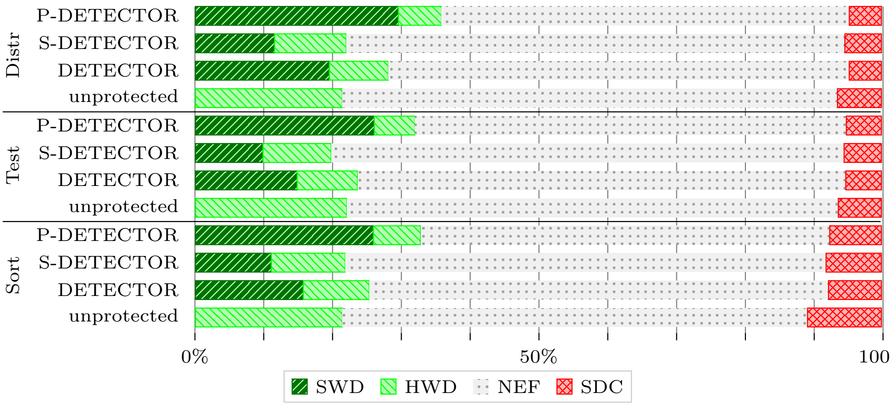


Figure A.4: The results of the single-bit DFE fault injection campaigns on the I/O-driven case studies protected with DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA.

Table A.5: The average results of the single-bit CFE fault injection campaigns on the I/O-driven case studies protected with CFCSS, RACFED, DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA.

	SWD	HWD	NEF	SDC
P-DETECTOR	47.4%	30.1%	10.9%	11.5%
S-DETECTOR	34.2%	16.1%	17.3%	32.5%
DETECTOR	38.5%	13.6%	15.5%	32.3%
RACFED	70.3%	22.2%	6.2%	1.3%
CFCSS	62.7%	20.9%	13.4%	2.9%
unprotected	-	45.2%	28.7%	26.1%

Table A.6: The average results of the single-bit DFE fault injection campaigns on the I/O-driven case studies protected with DETECTOR, S-DETECTOR, and P-DETECTOR on the ARMv7-M ISA.

	SWD	HWD	NEF	SDC
P-DETECTOR	27.2%	6.4%	60.6%	5.9%
S-DETECTOR	10.8%	10.4%	72.4%	6.4%
DETECTOR	16.7%	9.0%	68.3%	6.0%
unprotected	-	21.6%	67.7%	10.9%

A.3 Fault Injection Campaigns on RISC-V

Table A.7: Details of the fault injection campaigns on the BC, BS, CRC, MM, and QS case studies on the RV32I and RV32IXpdetector ISAs.

		inst. count	estimated instruction cycles	injected CFEs	injected DFEs
BC	Xpdetector	16	$83 + 7^{*}$	1618	7040
	P-DETECTOR	113	331	2258	4000
	unprotected	9	39	655	3424
BS	Xpdetector	29	$716\,056 + 7^{*}$	10 493	11 904
	P-DETECTOR	135	716 304	5761	7488
	unprotected	19	358 025	5249	6816
CRC	Xpdetector	24	$34\,023 + 7^{*}$	10 384	12 800
	P-DETECTOR	57	34 271	5581	7680
	unprotected	19	17 009	5196	7200
MM	Xpdetector	86	$163\,166 + 14\,007^{*}$	11 460	16 480
	P-DETECTOR	218	661 414	7305	15 680
	unprotected	66	136 935	5990	10 880
QS	Xpdetector	81	$44\,278 + 1778^{*}$	12 064	19 808
	P-DETECTOR	382	107 522	7778	16 128
	unprotected	59	21 536	6520	14 912

[*] Additional instruction cycles in a realistic scenario.

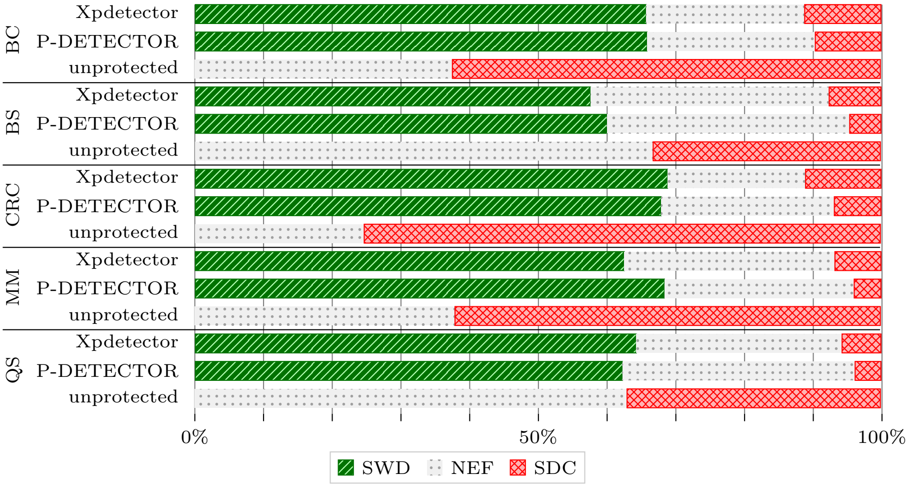


Figure A.5: The results of the single-bit CFE fault injection campaigns on the BC, BS, CRC, MM, and QS case studies protected with P-DETECTOR on the RV32I and RV32IXpdetector ISAs.

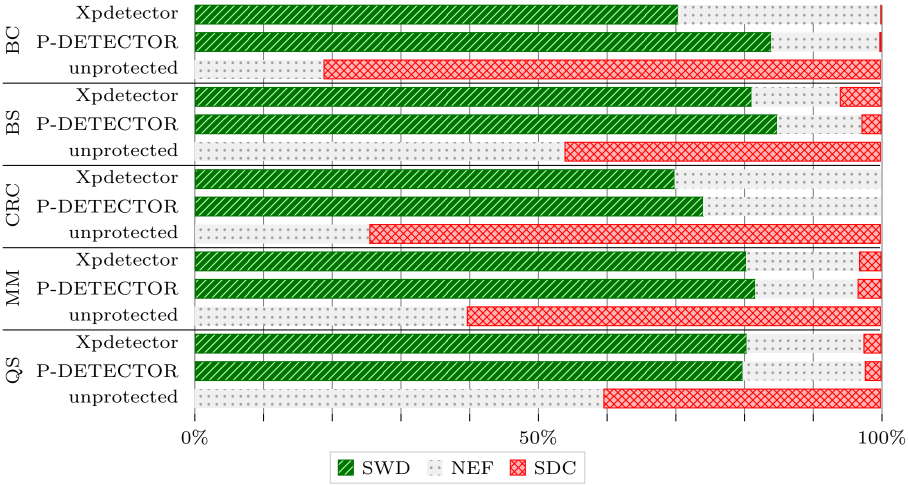


Figure A.6: The results of the single-bit DFE fault injection campaigns on the BC, BS, CRC, MM, and QS case studies protected with P-DETECTOR on the RV32I and RV32IXpdetector ISAs.

Table A.8: The average results of the single-bit CFE fault injection campaigns on the BC, BS, CRC, MM, and QS case studies protected with P-DETECTOR on the RV32I and RV32IXpdetector ISAs.

	SWD	NEF	SDC
P-DETECTOR (RV32IXpdetector)	63.7%	27.7%	8.5%
P-DETECTOR (RV32I)	64.8%	29.3%	5.9%
unprotected	-	46.0%	54.0%

Table A.9: The average results of the single-bit DFE fault injection campaigns on the BC, BS, CRC, MM, and QS case studies protected with P-DETECTOR on the RV32I and RV32IXpdetector ISAs.

	SWD	NEF	SDC
P-DETECTOR (RV32IXpdetector)	76.3%	21.3%	2.4%
P-DETECTOR (RV32I)	80.7%	17.4%	1.8%
unprotected	-	39.5%	60.5%

Appendix B

Calculating the Added Instruction Cycles per Vulnerable Section

B.1 DETECTOR

Table B.1: The added instruction cycles for each added instruction cycle per vulnerable section for DETECTOR using the ARMv7-M Cortex-M3 processor.

First execution		Second execution	
instruction	cycles	instruction	cycles
`1: cmp G, #0`	1	`12: cmp G, #0`	1
`2: beq label`	1	`13: beq label`	$P+1$
`3: cmp G, #<sig2>`	0 or 1	`14: bl compare`	$P+1$
`4: mov G, sp`	1	*compare subroutine*	$4N+P-1$
`5: stmdb S2!, {...}`	$1+N$	`15: mov lr, G`	1
`6: ldmia S1!, {...}`	$1+N$	*vulnerable section*	
`7: mov sp, G`	1	`16: mov G, sp`	1
`8: mov G, #0`	1	`17: stmdb S1!, {...}`	$1+N$
`9: blo source1`	$\left.\right\} c_{skip}$	`18: mov G, #<sigX>`	1
`10: beq source2`			
`11: bhi source3`	$P+1$		

Table B.1 shows a stack trace of the instructions added by DETECTOR for a vulnerable section executed on the Cortex-M3 processor, along with the number of cycles each dynamic instruction takes. The Technical Reference Manual of the Cortex-M3 [68] specifies that all compare and move instructions are executed in one cycle. Branch instructions take $P + 1$ cycles if the branch is taken or one cycle if the branch is not taken. In this context, P is the number of cycles required for a pipeline refill. This ranges from 1 to 3 depending on the alignment and width of the target instruction, and whether the processor manages to speculate the address early. Store multiple and load multiple instructions take $1 + N$ cycles, where N is the number of registers to be stored or loaded.

The table indicates that lines 9 and 10 take c_{skip} cycles. This c_{skip} value represents the number of branch instructions to source checkpoints that are evaluated but not taken during the first execution cycle. If a vulnerable section only has one source checkpoint, c_{skip} equals zero since the three conditional branch instructions at lines 9 through 11 are replaced by a single unconditional branch instruction (`b <checkpoint>`). Additionally, the compare instruction at line 3 is not added in this case (hence, the 0 cycles indicated in the table). If there is more than one source checkpoint, the value of c_{skip} can vary between 0 and 2, depending on if the first, second, or third branch is taken.

Table B.2 shows the added instruction cycles for the `compare` subroutine. This subroutine is called by the `bl compare` instruction at line 13 of the vulnerable

Table B.2: The added instruction cycles for the `compare` subroutine of DETECTOR using the ARMv7-M Cortex-M3 processor.

instruction	cycles	
`1: ldr G, [S2], #4`	2	$\left.\right\} 4 \cdot (N-1)$
`2: cmp r0, G`	1	
`3: bne errorHandler`	1	
`...`	...	
`21: ldr G, [S2], #4`	2	
`32: cmp r0, G`	1	
`33: bne errorHandler`	1	
`34: ldr G, [S2], #4`	2	
`35: bx lr`	$P+1$	

section, therefore adding $4N + P - 1$ cycles to the total cycle count. Therefore, the total added instruction cycles per vulnerable section for DETECTOR using the ARMv7-M Cortex-M3 processor is given by Equation (B.1).

$$I^{+}_{VS_{detector}} = \begin{cases} 7N + 4P + 14, & \text{if } \mathit{sources} = 1 \\ 7N + 4P + c_{skip} + 15, & \text{otherwise} \end{cases} \tag{B.1}$$

$$\text{where } \{N, P \in \mathbb{N} \mid P \leq 3 \mid c_{skip} \in \mathbb{N}_0 \mid c_{skip} \leq 2\}$$

This equation can be simplified by defining C, which varies between 0 and 3:

$$I^{+}_{VS_{detector}} = 7N + 4P + C + 14 \tag{B.2}$$

$$\text{where } \{N, P \in \mathbb{N} \mid P \leq 3 \mid C \in \mathbb{N}_0 \mid C \leq 3\}$$

As discussed in Chapter 8, there are two implementation variants of the DETECTOR algorithm, depending on the supported instructions of the ISA and the structure of the CFG of the program. Table B.1 and Equations (B.1) and (B.2) assume that the first version (where separate compare and conditional branch instructions are inserted) is used. When using the second variant, i.e., the variant where part of the re-execution code is placed after the vulnerable section, the same approach can be used to calculate the added instruction cycles per vulnerable section, resulting in Equation (B.3).

$$I^{+}_{VS_{detector}} = \begin{cases} 7N + 5P + 14, & \text{if } \mathit{sources} = 1 \\ 7N + 5P + 2c_{skip} + 15, & \text{otherwise} \end{cases} \tag{B.3}$$

$$\text{where } \{N, P \in \mathbb{N} \mid P \leq 3 \mid c_{skip} \in \mathbb{N}_0\}$$

B.2 S-DETECTOR

Table B.3: The added instruction cycles for each added instruction cycle per vulnerable section for S-DETECTOR using the ARMv7-M Cortex-M3 processor.

First execution		Second execution	
instruction	cycles	instruction	cycles
`1: cmp G, #0`	1	`12: cmp G, #0`	1
`2: beq label`	1	`13: beq label`	$P+1$
`3: cmp G, #<sig2>`	0 or 1	`14: bl compare`	$P+1$
`4: mov G, sp`	1	*compare subroutine*	$4M+P-1$
`5: stmdb S2!, {...}`	$1+M$	`15: mov lr, G`	1
`6: ldmia S1!, {...}`	$1+N$	*vulnerable section*	
`7: mov sp, G`	1	`16: mov G, sp`	1
`8: mov G, #0`	1	`17: stmdb S1!, {...}`	$1+N$
`9: blo source1`	$\left.\right\} c_{skip}$	`18: mov G, #<sigX>`	1
`10: beq source2`			
`11: bhi source3`	$P+1$		

Similarly to Table B.1, Table B.3 shows how many instruction cycles are added for each dynamic instruction added by S-DETECTOR per vulnerable section. Since S-DETECTOR only protects a subset of the registers, the number of registers pushed and popped onto and from the second shadow stack `S2` is denoted by M. This results in Equation (B.4).

$$I^{+}_{VS_{s-detector}} = \begin{cases} 5M + 2N + 4P + 14, & \text{if } sources = 1 \\ 5M + 2N + 4P + c_{skip} + 15, & \text{otherwise} \end{cases} \tag{B.4}$$

$$\text{where } \{N, P \in \mathbb{N} \mid P \leq 3 \mid c_{skip} \in \mathbb{N}_0 \mid c_{skip} \leq 2\}$$

This equation can again be simplified, resulting in Equation (B.5).

$$I^{+}_{VS_{s-detector}} = 5M + 2N + 4P + C + 14 \tag{B.5}$$

$$\text{where } \{M, N, P \in \mathbb{N} \mid P \leq 3 \mid M \leq N \mid C \in \mathbb{N}_0 \mid C \leq 3\}$$

Table B.4: The minimum and maximum added instruction cycles per vulnerable section added by S-DETECTOR on a Cortex-M3 processor ($N = 12$).

protected registers (M)	minimum cycles	maximum cycles
1	47	58
2	52	63
3	57	68
4	62	73
5	67	78
6	72	83
7	77	88
8	82	93
9	87	98
10	92	103
11	97	108
12	102	113

Table B.4 shows how many instruction cycles are added for every number of protected registers M when executing on the Cortex-M3 processor ($M = 12$).

Using the alternative implementation from Chapter 8, the added instruction cycles per vulnerable section for S-DETECTOR can be calculated by Equation (B.6).

$$I^{+}_{VS_{s-detector}} = \begin{cases} 5M + 2N + 5P + 14, & \text{if } sources = 1 \\ 5M + 2N + 5P + 2c_{skip} + 15, & \text{otherwise} \end{cases} \tag{B.6}$$

$$\text{where } \{N, P \in \mathbb{N} \mid P \leq 3 \mid c_{skip} \in \mathbb{N}_0\}$$

B.3 P-DETECTOR

Table B.5: The added instruction cycles for each added instruction cycle per vulnerable section for P-DETECTOR using the ARMv7-M Cortex-M3 processor.

First execution		Second execution	
instruction	cycles	instruction	cycles
`1: cmp G, #0`	1	`14: cmp G, #0`	1
`2: beq label`	1	`15: beq label`	$P+1$
`3: cmp G, #<sig2>`	0 or 1	`16: mov G, sp`	1
`4: mov G, sp`	1	`17: eor P, P, G`	1
`5: eor P, P, G`	1	`18: mov G, lr`	1
`6: mov G, lr`	1	`19: bl calculateParity`	$P+1$
`7: bl calculateParity`	$P+1$	*calculateParity subroutine*	$N+P$
calculateParity subroutine	$N+P$	`20: mov lr, G`	1
`8: ldmia S1!, {...}`	$N+1$	`21: cmp P, #0`	1
`9: mov sp, G`	1	`22: bne errorHandler`	1
`10: mov G, #0`	1	*vulnerable section*	
`11: blo source1`	$\left.\right\} c_{skip}$	`23: mov G, sp`	1
`12: beq source2`		`24: stmdb S1!, {...}`	$N+1$
`13: bhi source3`	$P+1$	`25: mov G, #<sigX>`	1

Table B.5 shows a stack trace of the instructions added by P-DETECTOR for a vulnerable section executed on the Cortex-M3 processor, along with the number of cycles needed for each instruction. The `calculateParity` subroutine, called at lines 7 and 19 takes $N + P$ cycles, as derived from Table B.6.

Table B.6: The added instruction cycles for the `calculateParity` subroutine of P-DETECTOR using the ARMv7-M Cortex-M3 processor.

instruction	cycles	
`1: eor P, P, r0`	1	$\left.\right\} N-1$
`...`	...	
`11: eor P, P, r12`	1	
`12: bx lr`	$P+1$	

Therefore, the formula to calculate the added instruction cycles per vulnerable section for P-DETECTOR using the ARMv7-M Cortex-M3 processor is given by Equation (B.7).

$$I^{+}_{VS_{p-detector}} = \begin{cases} 4N + 6P + 22 & \text{if } sources = 1 \\ 4N + 6P + c_{skip} + 23 & \text{otherwise} \end{cases} \tag{B.7}$$

$$\text{where } \{N, P \in \mathbb{N} \mid P \leq 3 \mid c_{skip} \in \mathbb{N}_0 \mid c_{skip} \leq 2\}$$

Just like before, Equation (B.7) can be simplified by using $C \leq 3$:

$$I^{+}_{VS_{p-detector}} = 4N + 6P + C + 22 \tag{B.8}$$

$$\text{where } \{N, P \in \mathbb{N} \mid P \leq 3 \mid C \in \mathbb{N}_0 \mid C \leq 3\}$$

A similar method can be used for the alternative implementation from Chapter 8, resulting in Equation (B.9).

$$I^{+}_{VS_{p-detector}} = \begin{cases} 4N + 7P + 22, & \text{if } sources = 1 \\ 4N + 7P + 2c_{skip} + 23, & \text{otherwise} \end{cases} \tag{B.9}$$

$$\text{where } \{N, P \in \mathbb{N} \mid P \leq 3 \mid c_{skip} \in \mathbb{N}_0\}$$

B.4 P-DETECTOR Using the Xpdetector RISC-V Extension

Table B.7: The added instruction cycles for each added instruction cycle per vulnerable section for P-DETECTOR using the Xpdetector RISC-V extension.

First execution		Second execution	
instruction	cycles	instruction	cycles
`1: par`	I_{PAR}	`4: par`	I_{PAR}

`2: rei <sig1> source1`
`...`
	c_{skip}	`5: rei <sig2> source2`
		`...`
		`6: rei <sigN> sourceN`
			$sources$

`3: rei <sigM> sourceM`	I_{REI}		
		`7: pzal errorHandler`	1
		vulnerable section	
		`8: cpi <sigX>`	I_{CPI}

Table B.7 shows how many instruction cycles are added for each added dynamic instruction per vulnerable section. In this table, it is assumed that all conditional instructions for which the condition is not met (lines 2, 5, 6, and 7) have an execution cycle of 1. The other `par`, `rei`, `pzal`, and `cpi` instructions have execution cycles of I_{PAR}, I_{REI}, 1, and I_{CPI}, respectively.

After the first execution cycle, the signature of the signature register `sr` determines which `rei` instruction will be executed. The number of `rei` instructions that are evaluated, but not executed during the first execution cycle is represented by c_{skip} (line 2). The value of c_{skip} can vary between 0 and $sources - 1$ as, during an error-free execution, at least one `rei` instruction will be executed. After the second execution cycle, all C `rei` instructions are traversed, but not executed, since the signature register `sr` will be zero. This results in the Equation (B.10).

$$I^{+}_{VS_{Xpdetector}} = 2I_{PAR} + I_{REI} + I_{CPI} + sources + c_{skip} + 1 \tag{B.10}$$

$$\text{where } \{I_{PAR}, I_{REI}, I_{CPI}, sources \in \mathbb{N} \mid c_{skip} \in \mathbb{N}_0 \mid c_{skip} < sources\}$$

This formula can be simplified by defining the variable $C = sources + c_{skip}$, resulting in Equation (B.11).

$$I^{+}_{VS_{Xpdetector}} = 2I_{PAR} + I_{REI} + I_{CPI} + C + 1 \tag{B.11}$$

$$\text{where } \{I_{PAR}, I_{REI}, I_{CPI}, C \in \mathbb{N}\}$$

Appendix C

Result of the Dynamic Register Analysis

Table C.1: Results of the dynamic register analysis. The table shows how often each register is used for memory access operations in each case study performed on the Cortex-M3 processor.

Register	Data-Processing Case Studies										
	BC	BS	CRC	CU	DIJ	FFT	MM	QS	Distr	Test	Sort
r0	0*	0	1000*	5	1424*	0	0	769*	0	0	0
r1	0	0	0	19	825*	0	1000*	0	4	2	1
r2	0	64 770*	0	57*	2161*	8620*	1000*	254	15*	9*	7*
r3	0	129 540*	0	56*	15 482*	2239*	0	127	20*	18*	18*
r4	0	0	0	39*	1378*	3360*	0	769*	3	3	2
r5	0	0	0	7	869	0	0	0	0	0	0
r6	0	0	0	0	0	0	0	0	0	0	0
r7	0	0	0	5	0	3072*	1100*	0	3	3	2
r8	0	0	0	10	749*	0	0	0	15*	16*	18*
r9	0	0	0	1	6	0	0	0	38*	26*	10*
r10	0	0	0	0	0	0	0	0	9	4	5
r11	0	0	0	0	0	0	0	0	0	0	0
r12	0	0	0	0	0	0	0	0	0	0	0
sp (r13)	0	0	0	82*	24 432*	36 477*	156	318	3	3	2
lr (14)	0	0	0	1	51	0	0	0	3	0	0

*Registers protected by S-DETECTOR

Appendix D

Assembly Reference

This is a summary of all the assembly instructions mentioned in this thesis. The instructions are grouped by architecture and listed in alphabetical order.

D.1 ARM Assembly

Optional Suffixes and Operands

`{addr_mode}` Addressing mode. Can be any one of:

- **`ia`** increment address after each transfer (default)
- **`ib`** increment address before each transfer
- **`da`** decrement address after each transfer
- **`db`** decrement address before each transfer

`{cond}` Condition code. Defines the conditions that must be met for the instruction to be executed (based on the condition flag). Can be any one of:

- **`eq`** equal (`Z` set)
- **`ne`** not equal (`Z` clear)
- **`cc/lo`** unsigned lower (`C` clear)
- **`hi`** unsigned higher (`C` set and `Z` clear)
- **`cs/hs`** unsigned higher or same (`C` set)

`{Rd}` Destination register. If not specified, the result is stored in the first operand register.

`{s}` If suffixed with `s`, update the condition flags on the result of the operation.

`{type}` Type of load or store operation. Can be any one of:

`b` unsigned byte

`sb` signed byte

`h` unsigned half-word

`sh` signed half-word

or can be omitted for a word operation.

Instructions

`add{s} {Rd}, Rn, Operand2` Add without carry
Add the values in `Rn` and `Operand2`.

`and{s} {Rd}, Rn, Operand2` Logical AND
Perform a bitwise AND operation on the values `Rn` and `Operand2`.

`b{cond} Dest` Branch
Branch to the destination address `Dest`.

`bl Dest` Branch with Link
Branch to the destination address `Dest` and copy the address of the next instruction into the link register `lr`.

`bx Rm` Branch (and exchange instruction set)
Branch to address contained in register `Rm` and (exchanges the instruction set, if required).

`cbz Rn, Dest` Compare and Branch on Zero
Compare register `Rn` with zero. If equal, branch to destination address `Dest`.

`cmp Rn, Operand2` Compare
Compare the value in register `Rn` with `Operand2` and update the condition flags on the result.

`eor Rd, Rn, Operand2` Logical Exclusive Or
Perform a bitwise exclusive or operation on the values in `Rn` and `Operand2`.

`itt cond` If-Then
Apply the `cond` condition to the next instructions.

`ldm{addr_mode} Rn{!}, regList` Load Multiple Registers
Load the values at the address in register `Rn` into the registers in `regList`. If `!` is added, write the final address back into `Rn`.

`ldr{type}{cond} Rt, [Rn, {#offset}]{!}` Load with (pre-indexed) immediate offset
Load the value located `#offset` bytes above the address in register `Rn` into register `Rt`. If `!` is added, increment `Rn` by `#offset`.

`ldr{type}{cond} Rt, [Rn] #offset` Load with post-indexed immediate offset
Load the value located in the address in register `Rn` into register `Rt` and increment `Rn` by `#offset`.

`mov{s} Rd, Operand2` Move
Copy the value of `Operand2` to register `Rd`.

`pop regList` Pop registers off a full descending stack
Synonym for `ldmia sp! regList`.

`push regList` Push registers onto a full descending stack
Synonym for `stmdb sp!, reglist`.

`stm{addr_mode} Rn{!}, regList` Store Multiple Registers
Store the values in the registers in `regList` at the address in register `Rn`. If `!` is added, write the final address back into `Rn`.

`str{type}{cond} Rt, [Rn, {#offset}]{!}` Store with (pre-indexed) immediate offset
Store the value in register `Rt` at the address `#offset` bytes above the address in register `Rn`. If `!` is added, increment `Rn` by `#offset`.

`str{type}{cond} Rt, [Rn] #offset` Store with post-indexed immediate offset
Store the value in register `Rt` in the address in register `Rn` and increment `Rn` by `#offset`.

`sub{s} {Rd}, Rn, Operand2` Subtract without carry
Subtracts the value of `Operand2` from register `Rn`.

Source: [97]

D.2 RISC-V Assembly

Instructions

`add rd, rs1, rs2` Add
Add the values in registers `rs1` and `rs2` and store the result in register `rd`.

`addi rd, rs1, imm` Add Immediate
Add the value `imm` to the value in register `rs1` and store the result in register `rd`.

`beq, rs1, rs2, dest` Branch if Equal
Branch to `dest` if the values in registers `rs1` and `rs2` are equal.

`bge, rs1, rs2, dest` Branch if Greater or Equal
Branch to `dest` if the value in register `rs1` is greater than or equal to the value in register `rs2`.

`bgeu rs1, rs2, dest` Branch if Greater or Equal, Unsigned
Branch to `dest` if the value in register `rs1` is greater than or equal to the value in register `rs2` (unsigned).

`jal, rd, dest` Jump and Link
Jump to `dest` and store the address of the next instruction in register `rd`.

`lbu, rd, offset(rs1)` Load Byte Unsigned
Load an 8-bit value from the memory address at `rs1 + offset`, zero-extend it and store it in register `rd`.

`lui rd, imm` Load Upper Immediate
Place the U-immediate value in the top 20 bits of the destination register `rd`, filling in the lowest 12 bits with zeros.

`sb rs2, offset(rs1)` Store Byte
Store the 8-bit value from the low bits of register `rs2` to the memory address at `rs1 + offset`.

`sw rs2, offset(rs1)` Store Word
Store the 32-bit value from the low bits of register `rs2` to the memory address at `rs1 + offset`.

`xor rd, rs1, rs2` Exclusive OR
Perform a bitwise exclusive OR operation on the values in registers `rs1` and `rs2` and store the result in register `rd`.

`xori rd, rs1, imm` Exclusive OR Immediate
Perform a bitwise exclusive OR operation on the value in register `rs1` and the immediate value `imm` and store the result in register `rd`.

Pseudo-instructions

`bleu rs, rt, dest` Branch if Lower or Equal, Unsigned
`bgeu rt, rs, dest`

`blez rs, dest` Branch if Lower or Equal to Zero
`bge zero, rs, dest`

`li, rd, imm` Load Immediate
Place the immediate `imm`, into register `rd`. This instruction is realized through `myriad sequences`: the compiler can generate different instruction sequences to recreate this behavior.

`j, dest` Jump
`jal zero, dest.`

`jal, dest` Jump and Link (without `rd`)
`jal ra, dest.`

`mv, rd, rs` Move
`addi rd, rs, 0.`

Source: [98, 86]

Bibliography

[1] Censtry Electronics, Feb 2024. [Online]. Available: https://www.censtry.com/blog/how-many-transistor-in-a-cpu.html (accessed June 11, 2024).

[2] Wikipedia contributors, “Transistor count,” Jun 2024. [Online]. Available: https://en.wikipedia.org/wiki/Transistor_count (accessed June 11, 2024).

[3] B. D. Sierawski, R. A. Reed, M. H. Mendenhall, R. A. Weller, R. D. Schrimpf, S.-J. Wen, R. Wong, N. Tam, and R. C. Baumann, “Effects of scaling on muon-induced soft errors,” in *2011 International Reliability Physics Symposium*. IEEE, Apr 2011, pp. 3C.3.1–3C.3.6. [Online]. Available: http://ieeexplore.ieee.org/document/5784484/

[4] R. Harada, Y. Mitsuyama, M. Hashimoto, and T. Onoye, “Neutron induced single event multiple transients with voltage scaling and body biasing,” in *2011 International Reliability Physics Symposium*. IEEE, Apr 2011, pp. 3C.4.1–3C.4.5. [Online]. Available: http://ieeexplore.ieee.org/document/5784485/

[5] R. Baumann, “Soft errors in advanced semiconductor devices-part I: the three radiation sources,” *IEEE Transactions on Device and Materials Reliability*, vol. 1, no. 1, pp. 17–22, Mar 2001. [Online]. Available: http://ieeexplore.ieee.org/document/946456/

[6] R. Baumann, “Radiation-induced soft errors in advanced semiconductor technologies,” *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, Sep 2005. [Online]. Available: http://ieeexplore.ieee.org/document/1545891/

[7] M. O’Bryan, “Single Event Effects,” Oct 2021. [Online]. Available: https://radhome.gsfc.nasa.gov/radhome/see.htm (accessed June 11, 2024).

[8] E. H. Ibe, S. Yoshimoto, M. Yoshimoto, H. Kawaguchi, K. Kobayashi, J. Furuta, Y. Mitsuyama, M. Hashimoto, T. Onoye, H. Kanbara, H. Ochi,

K. Wakabayashi, H. Onodera, and M. Sugihara, *VLSI Design and Test for Systems Dependability*, S. Asai, Ed. Tokyo: Springer Japan, 2019. [Online]. Available: http://link.springer.com/10.1007/978-4-431-56594-9

[9] "ATSB transport safety report: In-flight upset 154 km west of Learmonth, WA 7 October 2008 VH-QPA Airbus A330-303," Australian Transport Safety Bureau, Tech. Rep. October, 2008. [Online]. Available: https://www.atsb.gov.au/publications/investigation_reports/2008/aair/ao-2008-070

[10] "International Roadmap for Devices and Systems 2022 Update - More Moore," International Roadmap for Devices and Systems, Tech. Rep., 2022. [Online]. Available: https://irds.ieee.org/editions/2022/more-moore

[11] S. Baffreau, S. Bendhia, M. Ramdani, and E. Sicard, "Characterisation of microcontroller susceptibility to radio frequency interference," in *Proceedings of the Fourth IEEE International Caracas Conference on Devices, Circuits and Systems (Cat. No.02TH8611).* IEEE, 2002, pp. I031–1–I031–5. [Online]. Available: http://ieeexplore.ieee.org/document/1004088/

[12] K. Kim and A. A. Iliadis, "Critical Bit Errors in CMOS Digital Inverters due to Pulsed Electromagnetic Interference," in *2007 International Conference on Electromagnetics in Advanced Applications.* IEEE, Sep 2007, pp. 217–220. [Online]. Available: http://ieeexplore.ieee.org/document/4387276/

[13] N. A. Estep, J. C. Petrosky, J. W. McClory, Y. Kim, and A. J. Terzuoli, "Electromagnetic Interference and Ionizing Radiation Effects on CMOS Devices," *IEEE Transactions on Plasma Science*, vol. 40, no. 6, pp. 1495–1501, jun 2012. [Online]. Available: http://ieeexplore.ieee.org/document/6204100/

[14] S. Jagannathan, Z. Diggins, N. Mahatme, T. D. Loveless, B. L. Bhuva, S.-J. Wen, R. Wong, and L. W. Massengill, "Temperature dependence of soft error rate in flip-flop designs," in *2012 IEEE International Reliability Physics Symposium (IRPS).* IEEE, Apr 2012, pp. SE.2.1–SE.2.6. [Online]. Available: http://ieeexplore.ieee.org/document/6241927/

[15] J. G. van Woudenberg, M. F. Witteman, and F. Menarini, "Practical Optical Fault Injection on Secure Microcontrollers," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography.* IEEE, Sep 2011, pp. 91–99. [Online]. Available: http://ieeexplore.ieee.org/document/6076471/

[16] J. Van Waes, D. Vanoost, J. Vankeirsbilck, J. Lannoo, D. Pissoort, and J. Boydens, "Resilience of Error Correction Codes Against Harsh Electromagnetic Disturbances: Fault Mechanisms," *IEEE Transactions on*

Electromagnetic Compatibility, vol. 62, no. 4, pp. 1017–1027, Aug 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8795581/

[17] T. Li, J. A. Ambrose, R. Ragel, and S. Parameswaran, “Processor Design for Soft Errors,” *ACM Computing Surveys*, vol. 49, no. 3, pp. 1–44, Sep 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/2996357

[18] National Aeronautics and Space Administration (NASA), “Perseverance rover components.” [Online]. Available: https://mars.nasa.gov/mars2020/spacecraft/rover/brains/ (accessed October 17, 2023).

[19] R. Ginosar, “Survey of processors for space,” *Data Systems in Aerospace (DASIA). Eurospace*, pp. 1–5, 2012.

[20] N. G. Leveson, “Using cots components in safety-critical systems,” Apr 2000. [Online]. Available: http://sunnyday.mit.edu/papers/cots.pdf

[21] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, “Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance,” in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN’07)*. IEEE, jun 2007, pp. 297–306. [Online]. Available: http://ieeexplore.ieee.org/document/4272981/

[22] C. Wang, H.-s. Kim, Y. Wu, and V. Ying, “Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection,” in *International Symposium on Code Generation and Optimization (CGO’07)*. IEEE, Mar 2007, pp. 244–258. [Online]. Available: http://ieeexplore.ieee.org/document/4145119/

[23] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August, “DAFT: Decoupled Acyclic Fault Tolerance,” *International Journal of Parallel Programming*, vol. 40, no. 1, pp. 118–140, Feb 2012. [Online]. Available: http://link.springer.com/10.1007/s10766-011-0183-4

[24] Aspencore, “The Current State of Embedded Development,” May 2023. [Online]. Available: https://www.embedded.com/wp-content/uploads/2023/05/Embedded-Market-Study-For-Webinar-Recording-April-2023.pdf (accessed June 13, 2024).

[25] N. Oh and E. McCluskey, “Error detection by selective procedure call duplication for low energy consumption,” *IEEE Transactions on Reliability*, vol. 51, no. 4, pp. 392–402, Dec 2002. [Online]. Available: http://ieeexplore.ieee.org/document/1044337/

[26] J. Ma, D. Yu, Y. Wang, Z. Cai, Q. Zhang, and C. Hu, “Detecting Silent Data Corruptions in Aerospace-Based Computing

Using Program Invariants," *International Journal of Aerospace Engineering*, vol. 2016, pp. 1–10, 2016. [Online]. Available: https://www.hindawi.com/journals/ijae/2016/8213638/

[27] B. Nicolescu and R. Velazco, "Detecting soft errors by a purely software approach: method, tools and experimental results," in *2003 Design, Automation and Test in Europe Conference and Exhibition.* IEEE Comput. Soc, 2003, pp. 57–62. [Online]. Available: http://ieeexplore.ieee.org/document/1253806/

[28] B. De Blaere, J. Vankeirsbilck, V. B. Thati, and J. Boydens, "SIHFT GCC Plugin," Oct 2024. [Online]. Available: https://gitlab.kuleuven.be/m-group-campus-brugge/distrinet_public/sihft-gcc-plugin (accessed October 29, 2024).

[29] M. Guthaus, J. Ringenberg, T. Austin, T. Mudge, and R. Brown, "MiBench Version 1.0," Nov 2001. [Online]. Available: https://vhosts.eecs.umich.edu/mibench/ (accessed July 4, 2024).

[30] Z. Alkhalifa, V. Nair, N. Krishnamurthy, and J. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 627–641, jun 1999. [Online]. Available: http://ieeexplore.ieee.org/document/774911/

[31] N. Oh, P. Shirvani, and E. McCluskey, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, Mar 2002. [Online]. Available: http://ieeexplore.ieee.org/document/994926/

[32] B. Nicolescu, Y. Savaria, and R. Velazco, "SIED: software implemented error detection," in *Proceedings. 16th IEEE Symposium on Computer Arithmetic.* IEEE Comput. Soc, Mar 2004, pp. 589–596. [Online]. Available: http://ieeexplore.ieee.org/document/1250159/

[33] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Proceedings. 16th IEEE Symposium on Computer Arithmetic.* IEEE Comput. Soc, Mar 2004, pp. 581–588. [Online]. Available: http://ieeexplore.ieee.org/document/1250158/

[34] A. Li and B. Hong, "Software implemented transient fault detection in space computer," *Aerospace Science and Technology*, vol. 11, no. 2-3, pp. 245–252, Mar 2007. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1270963806000800

[35] S. Amir Asghari, A. Abdi, H. Taheri, H. Pedram, and S. Pourmozaffari, "SEDSR: Soft Error Detection Using Software Redundancy," *Journal of Software Engineering and Applications*, vol. 05, no. 09, pp. 664–670, 2012. [Online]. Available: http://www.scirp.org/journal/doi.aspx?DOI=10.4236/jsea.2012.59078

[36] S. A. Asghari, H. Taheri, H. Pedram, and O. Kaynak, "Software-Based Control Flow Checking Against Transient Faults in Industrial Environments," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 1, pp. 481–490, Feb 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6469216/

[37] S. A. Asghari, M. Binesh Marvasti, and M. Daneshtalab, "A software implemented comprehensive soft error detection method for embedded systems," *Microprocessors and Microsystems*, vol. 77, p. 103161, Sep 2020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0141933120303288

[38] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random Additive Signature Monitoring for Control Flow Error Detection," *IEEE Transactions on Reliability*, vol. 66, no. 4, pp. 1178–1192, Dec 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8067656/

[39] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random Additive Control Flow Error Detection," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer Verlag, Sep 2018, vol. 11093 LNCS, pp. 220–234. [Online]. Available: http://link.springer.com/10.1007/978-3-319-99130-6_15

[40] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, Mar 2002. [Online]. Available: http://ieeexplore.ieee.org/document/994913/

[41] N. Oh, S. Mitra, and E. McCluskey, "ED/sup 4/I: error detection by diverse data and duplicated instructions," *IEEE Transactions on Computers*, vol. 51, no. 2, pp. 180–199, Feb 2002. [Online]. Available: http://ieeexplore.ieee.org/document/980007/

[42] V. B. Thati, J. Vankeirsbilck, D. Pissoort, and J. Boydens, "Instruction Level Duplication and Comparison for Data Error Detection: a First Experiment," in *2018 IEEE XXVII International Scientific Conference Electronics - ET*. IEEE, Sep 2018, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/8549589/

[43] A. Abi, S. A. Asghari, S. P. Mozaffari, H. Taheri, and H. Pedram, "An Optimum Instruction Level Method for Soft Error Detection," *International Review on Computers and Software (I.RE.CO.S.)*, vol. 7, no. 2, pp. 637–641, 2012. [Online]. Available: https://www.researchgate.net/publication/235743309_An_Optimum_Instruction_Level_Method_for_Soft_Error_Detection

[44] B. Arasteh, A. Bouyer, and S. Pirahesh, "An efficient vulnerability-driven method for hardening a program against soft-error using genetic algorithm," *Computers & Electrical Engineering*, vol. 48, pp. 25–43, Nov 2015. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0045790615003419

[45] V. B. Thati, J. Vankeirsbilck, N. Penneman, D. Pissoort, and J. Boydens, "An Improved Data Error Detection Technique for Dependable Embedded Software," in *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, Dec 2018, pp. 213–220. [Online]. Available: https://ieeexplore.ieee.org/document/8639681/

[46] J. Chang, G. Reis, and D. August, "Automatic Instruction-Level Software-Only Recovery," in *International Conference on Dependable Systems and Networks (DSN'06)*, vol. 2006. IEEE, 2006, pp. 83–92. [Online]. Available: http://ieeexplore.ieee.org/document/1633498/

[47] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: Software Implemented Fault Tolerance," in *International Symposium on Code Generation and Optimization*, vol. 2005. IEEE, 2005, pp. 243–254. [Online]. Available: http://ieeexplore.ieee.org/document/1402092/

[48] M. Didehban and A. Shrivastava, "nZDC," in *Proceedings of the 53rd Annual Design Automation Conference*, vol. 05-09-June. New York, NY, USA: ACM, Jun 2016, pp. 1–6. [Online]. Available: https://dl.acm.org/doi/10.1145/2897937.2898054

[49] X. Guang and Z. Zhang, *Linear Network Error Correction Coding*, ser. SpringerBriefs in Computer Science. New York, NY: Springer New York, 2014. [Online]. Available: https://link.springer.com/10.1007/978-1-4939-0588-1

[50] R. Zhu and Y. Ma, Eds., *Information Engineering and Applications*, ser. Lecture Notes in Electrical Engineering. London: Springer London, 2012, vol. 154. [Online]. Available: http://link.springer.com/10.1007/978-1-4471-2386-6

[51] H. Wu, *Moving Broadband Mobile Communications Forward - Intelligent Technologies for 5G and Beyond*. IntechOpen, Aug 2021, ch. A Brief

Overview of CRC Implementation for 5G NR, pp. 69–79. [Online]. Available: https://www.intechopen.com/books/moving-broadband-mobile-communications-forward-intelligent-technologies-for-5g-and-beyond

[52] G. Charlot, "Utilité de la définition de brönsted des acides et des bases en chimie analytique," *Analytica Chimica Acta*, vol. 1, pp. 59–68, 1947. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0003267000897214

[53] J. D. Van Der Waals, "The equation of state for gases and liquids," *The Nobel Prize in Physics*, 1910. [Online]. Available: https://www.nobelprize.org/prizes/physics/1910/waals/lecture/

[54] D. Pal, M. S. Deepak, N. Sireesha, and S. Sadiya, "Implementation of CSMA/CD and Link State Routing Protocols for Dijkstra Algorithm," in *2023 1st International Conference on Circuits, Power and Intelligent Systems (CCPIS)*. IEEE, Sep 2023, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/10292084/

[55] R. J. Burgess, *The history of music production.* New York, NY: Oxford University Press, 2014.

[56] D. C. Lay, S. R. Lay, and J. J. McDonald, *Linear Algebra and its Applications*, 5th ed. Harlow: Pearson, 2016.

[57] V. B. Thati, J. Vankeirsbilck, and J. Boydens, "Comparative study on data error detection techniques in embedded systems," in *2016 XXV International Scientific Conference Electronics (ET)*. IEEE, sep 2016, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/document/7753517/

[58] V. B. Thati, J. Vankeirsbilck, N. Penneman, D. Pissoort, and J. Boydens, "CDFEDT - Comparison of Data Flow Error Detection Techniques in Embedded Systems: an Empirical Study," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*. New York, NY, USA: ACM, Aug 2018, pp. 1–9. [Online]. Available: https://dl.acm.org/doi/10.1145/3230833.3230854

[59] J. Vankeirsbilck, "Advancing Control Flow Error Detection Techniques for Embedded Software using Automated Implementation and Fault Injection," Ph.D. dissertation, KU Leuven, Jan 2020. [Online]. Available: https://lirias.kuleuven.be/2861069

[60] Imperas Software Limited, "ISS - The Imperas Instruction Set Simulator." [Online]. Available: https://www.imperas.com/iss-imperas-instruction-set-simulator (accessed July 6, 2024).

[61] Festo, "MPS." [Online]. Available: https://ip.festo-didactic.com/Infoportal/MPS/Overview/EN/index.html (accessed July 6, 2024).

[62] V. Khuat, J.-M. Dutertre, and J.-L. Danger, “Software countermeasures against the multiple instructions skip fault model,” *Microelectronics Reliability*, vol. 155, p. 115370, Apr 2024. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0026271424000507

[63] B. E. Forlin, W. Van Huffelen, C. Cazzaniga, P. Rech, N. Alachiotis, and M. Ottavi, “An unprotected RISC-V Soft-core processor on an SRAM FPGA: Is it as bad as it sounds?” *Proceedings of the European Test Workshop*, vol. 2023-May, 2023.

[64] H. Schirmeier, C. Borchert, and O. Spinczyk, “Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors,” in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* IEEE, Jun 2015, pp. 319–330. [Online]. Available: https://ieeexplore.ieee.org/document/7266861

[65] J. Vankeirsbilck, T. Cauwelier, J. Van Waes, H. Hallez, and J. Boydens, “Software-Implemented Fault Injection for Physical and Simulated Embedded CPUs,” in *2018 IEEE XXVII International Scientific Conference Electronics - ET.* IEEE, sep 2018, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/8549630/

[66] V. B. Thati, “Software Strategies to Improve Immunity of Programmable Embedded Systems against Disturbances,” Ph.D. dissertation, KU Leuven, April 2020. [Online]. Available: https://lirias.kuleuven.be/2935817

[67] OVP, “Open Virtual Platforms - the source of Fast Processor Models & Platforms.” [Online]. Available: https://www.ovpworld.org/ (accessed July 6, 2024).

[68] ARM, “Cortex-M3 Technical Reference Manual,” 2010. [Online]. Available: https://developer.arm.com/documentation/ddi0337/ (accessed June 11, 2024).

[69] E. Chielle, J. R. Azambuja, R. S. Barth, F. Almeida, and F. L. Kastensmidt, “Evaluating Selective Redundancy in Data-Flow Software-Based Techniques,” *IEEE Transactions on Nuclear Science*, vol. 60, no. 4, pp. 2768–2775, Aug 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6560441/

[70] M. Nikseresht, J. Vankeirsbilck, D. Pissoort, and J. Boydens, “A Selective Soft Error Protection Method for COTS Processor-based Systems,” in *2021 30th International Scientific Conference Electronics, ET 2021 - Proceedings.* IEEE, Sep 2021, pp. 1–5. [Online]. Available: https://ieeexplore.ieee.org/document/9579862/

[71] RISCV International, “The Top 10 RISC-V Milestones & Highlights from 2023,” Dec 2023. [Online]. Available: https://riscv.org/blog/2023/12/the-top-10-risc-v-milestones-highlights-from-2023/ (accessed July 1, 2024).

[72] OpenHW Group, “CORE-V Family of Open-Source RISC-V Cores,” Feb 2020. [Online]. Available: https://github.com/openhwgroup/core-v-cores (accessed June 27, 2024).

[73] “Pulp Platform.” [Online]. Available: https://pulp-platform.org/ (accessed June 27, 2024).

[74] SiFive, “A winning processor portfolio.” [Online]. Available: https://www.sifive.com/risc-v-core-ip (accessed June 27, 2024).

[75] Espressif, “ESP32-C3.” [Online]. Available: https://www.espressif.com/en/products/socs/esp32-c3 (accessed June 27, 2024).

[76] “RISC-V Market Report: Application Forecasts in a Heterogeneous World,” SDH Group, Tech. Rep., Jan 2024. [Online]. Available: https://theshdgroup.com/2023risc-v_download/

[77] RISC-V International, “The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture,” Apr 2024. [Online]. Available: https://riscv.org/technical/specifications/

[78] S. Sharma, “RISC-V Architecture: A Comprehensive Guide to the Open-Source ISA,” Mar 2024. [Online]. Available: https://www.wevolver.com/article/risc-v-architecture (accessed August 28, 2024).

[79] M. Bognar, J. Noorman, and F. Piessens, “Proteus: An extensible risc-v core for hardware extensions,” in *RISC-V Summit Europe ’23*, Jun. 2023. [Online]. Available: https://lirias.kuleuven.be/retrieve/716676

[80] Antmicro Ltd, “Triple-Modular-Redundancy RISC-V Demonstrator,” Apr 2018. [Online]. Available: https://github.com/ThalesGroup/TMR/blob/master/RISC-V-demonstrator--docs.pdf (accessed June 25, 2024).

[81] C. Rodrigues, I. Marques, S. Pinto, T. Gomes, and A. Tavares, “Towards a Heterogeneous Fault-Tolerance Architecture based on Arm and RISC-V Processors,” in *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, Oct 2019, pp. 3112–3117. [Online]. Available: https://ieeexplore.ieee.org/document/8926844/

[82] L. Blasi and F. Vigli, “The first space-qualified KlessydraRISC-V microcontroller to be launched on a satellite,” in *RISC-V*

Workshop Zurich Proceedings. RISC-V International, Jun 2019. [Online]. Available: https://riscv.org/wp-content/uploads/2019/06/16.30-Vigli-Blasi-Olivieri-riscv-zurich-2019_NEW.pdf

[83] PULP Platform, “PULPino: An open-source microcontroller system based on RISC-V,” Nov 2015. [Online]. Available: https://github.com/pulp-platform/pulpino

[84] A. Dorflinger, B. Kleinbeck, M. Albers, H. Michalik, and M. Moya, “A Framework for Fault Tolerance in RISC-V,” in *2022 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech).* IEEE, Sep 2022, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/9927800/

[85] S. Park, D. Kang, J. Kang, and D. Kwon, “Bratter: An Instruction Set Extension for Forward Control-Flow Integrity in RISC-V,” *Sensors*, vol. 22, no. 4, p. 1392, Feb 2022. [Online]. Available: https://www.mdpi.com/1424-8220/22/4/1392

[86] RISCV International, “RISC-V Assembly Programmer’s Manua,” May 2017. [Online]. Available: https://github.com/riscv-non-isa/riscv-asm-manual/ (accessed July 16, 2024).

[87] Imperas Software Limited, “OVP VMI Morph Time Function Reference,” Sep 2021. [Online]. Available: https://www.ovpworld.org/vmi-morph-time-vmi-mt-api-reference-guide (accessed July 17, 2024).

[88] Imperas Software Limited, “OVP Fast Processor Model Variant: RISC-V / RV32I,” Jun 2023. [Online]. Available: https://www.ovpworld.org/processor-model-variant-risc-v-riscv32-rv32i (accessed July 17, 2024).

[89] B. Bailey, “RISC-V Micro-Architectural Verification,” Dec 2023. [Online]. Available: https://semiengineering.com/risc-v-micro-architectural-verification/ (accessed July 17, 2024).

[90] S. Davidmann, L. Moore, R. Ho, T. Liu, D. Letcher, and A. Sutton, “Rolling the dice with random instructions is the safe bet on risc-v verification,” in *Design and Verification Conference and Exhibition (DVCon)*, 2020. [Online]. Available: https://dvcon-proceedings.org/wp-content/uploads/rolling-the-dice-with-random-instructions-is-the-safe-bet-on-risc-v-verification.pdf

[91] OpenHW Group, “CORE-V-Verif,” Nov 2023. [Online]. Available: https://github.com/openhwgroup/core-v-verif (accessed July 17, 2024).

[92] Imperas Software Limited, "RISC-V Verification Interface (RVVI)," Sep 2023. [Online]. Available: https://github.com/riscv-verification/RVVI (accessed July 17, 2024).

[93] J. Vankeirsbilck, H. Hallez, and J. Boydens, "Automatic Implementation of Control Flow Error Detection Techniques," in *Proceedings of the 2019 3rd International Symposium on Computer Science and Intelligent Control*. New York, NY, USA: ACM, Sep 2019, pp. 1–8. [Online]. Available: https://dl.acm.org/doi/10.1145/3386164.3389106

[94] B. De Blaere, M. Nikseresht, J. Vankeirsbilck, V. B. Thati, and J. Boydens, "Public_DFED_Plugin," Dec 2023. [Online]. Available: https://gitlab.kuleuven.be/m-group-campus-brugge/distrinet_public/public_dfed_plugin (accessed July 14, 2024).

[95] V. S. N. Kariyakarawana and T. Holvoet, "Logging - the missing component for gcc plugin testing," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering*, Jan 2024-01-08. [Online]. Available: https://lirias.kuleuven.be/4145342&lang=en

[96] J. Boydens, T. Holvoet, J. Vankeirsbilck, B. De Blaere, and V. S. N. Kariyakarawana, "FIRES - A Framework to Increase the Resilience of Embedded Systems," 2021. [Online]. Available: https://iiw.kuleuven.be/onderzoek/m-group/research-and-projects/fires (accessed July 17, 2024).

[97] ARM, "ARM Compiler armasm User Guide," 2016. [Online]. Available: https://developer.arm.com/documentation/ddi0337/ (accessed August 9, 2024).

[98] Msyksphinz, "RISC-V Instruction Set Specifications," 2019. [Online]. Available: https://msyksphinz-self.github.io/riscv-isadoc/ (accessed August 9, 2024).

Publications

[99] B. De Blaere, E. Verstappe, J. Vankeirsbilck, and J. Boydens, "A Compiler Extension to Protect Embedded Systems Against Data Flow Errors," in *2021 XXX International Scientific Conference Electronics (ET).* IEEE, sep 2021, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/9580074/

[100] B. De Blaere, J. Vankeirsbilck, and J. Boydens, "Soft Error Detection through Low-level Re-execution," in *2021 5th International Conference on System Reliability and Safety (ICSRS).* IEEE, Nov 2021, pp. 181–189. [Online]. Available: https://ieeexplore.ieee.org/document/9660636/

[101] M. Nikscresht, B. De Blaere, J. Vankeirsbilck, D. Pissoort, and J. Boydens, "Impact of Selective Implementation on Soft Error Detection Through Low-level Re-execution," in *2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech).* IEEE, Oct 2021, pp. 112–117. [Online]. Available: https://ieeexplore.ieee.org/document/9730291/

[102] B. De Blaere, J. Vankeirsbilck, and J. Boydens, "Utilizing Parity Checking to Optimize Soft Error Detection Through Low-Level Reexecution," *IEEE Transactions on Reliability*, vol. 72, no. 4, pp. 1355–1366, Dec 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10197575/

Curriculum Vitae

Brent De Blaere
PhD researcher | Software Engineer
brent.deblaere@kuleuven.be
https://brent-deblaere.web.app/
Zedelgem, Belgium

Brent obtained a Master's degree in Electronics and ICT Engineering Technology in 2020 at KU Leuven, Belgium. Following his graduation, he obtained a PhD position at the Department of Computer Science of his alma mater, where he is currently conducting research at the DistriNet Research Group. His research focuses on increasing the reliability of modern embedded systems against transient errors using software-implemented fault tolerance.

Work Experience

KU Leuven | Nov 2020 – Oct 2024
RESEARCHER – TEACHING ASSISTANT

- Researching soft error resilience techniques (C/C++, Assembly, Python)
- Collaboration on the FIRES project, working in a team of five people
- Presenting in seminaries to audiences with both academic and industry backgrounds
- Teaching programming-related lab sessions (C, Python)
- Supervising four Master's theses

Televic Healthcare | Feb 2020 – Mar 2020
INTERNSHIP R&D

- Full stack development and prototyping on the mobile Aqura nurse call system (linked to Master's thesis) (Java/Android, C#/.NET)

KU Leuven | Sep 2019 – Dec 2019
STUDENT-ASSISTANT

- Teaching exercise courses *Computational Thinking* (Python)

Jeugdraad Deinze | Sep 2018 – Aug 2023
VOLUNTEER

- Creating & maintaining the website www.jeugdraaddeinze.be (HTML/CSS)
- Member of the advisory board (city of Deinze)

Education

Nov 2020 – Nov 2024 (expected)
DOCTOR OF ENGINEERING TECHNOLOGY PhD
KU Leuven Arenberg Doctoral School

Sep 2016 – Jul 2020
MASTER OF ELECTRONICS & ICT ENGINEERING TECHNOLOGY cum laude
KU Leuven, Faculty of Engineering Technology

Languages

Dutch (C2)

English (C1)

French (A2)

German (A1)

Connect with me on LinkedIn via www.linkedin.com/in/brent-de-blaere.

References and a more detailed CV are available upon request.

FACULTY OF ENGINEERING TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE
DISTRINET - M-GROUP
Spoorwegstraat 12
B-8200 Brugge
m-group@kuleuven.be