



ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE. TAKES YOU FAR



Doctoral Dissertation

Doctoral Program in Computer and Control Engineering (36th cycle)

Simulation Techniques For Rapid Software Development and Validation

By

Mohammadreza Amel Solouki

Supervisor(s):

Prof. Massimo Violante

Doctoral Examination Committee:

Prof. Alberto Bosio, Referee, École centrale de Lyon

Prof. Andrea Acquaviva, Referee, University of Bologna

Prof. Maksim Jenihhin, Tallinn University of Technology

Prof. Maurizio Rebaudengo, Politecnico di Torino

Prof. Ernesto Sanchez, Politecnico di Torino

Politecnico di Torino

2024

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Mohammadreza Amel Solouki
2024

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my loving parents

Acknowledgements

I would first like to express my sincere gratitude to my advisor, Prof. Massimo Violante, whose knowledge and expertise were invaluable in formulating my Ph.D. research direction based on my interests. Your kindness, patience, insightful advice, and feedback enlightened me, polished my thinking, and steered my work to a higher level.

I am deeply grateful to my committee members, Prof. Alberto Bosio and Prof. Andrea Acquaviva, for their invaluable help, constructive comments, and insightful suggestions that significantly enriched my dissertation. I also extend my thanks to Prof. Maksim Jenihhin, Prof. Maurizio Rebaudengo, and Prof. Ernesto Sanchez for their role in my supervisory committee.

Besides, I would like to extend my sincere acknowledgment to my kind and intelligent colleagues, Jacopo Sini, Juan David Guerrero Balanguera, Josie Esteban Rodriguez Condia, Luigi Pugliese, and all of my colleagues in the Electronic CAD & Reliability Group, for their inspiring discussions and help with our collaborative work.

On a personal note, I want to thank my family, especially my parents, for their unwavering support and understanding throughout this process. I would also like to express my appreciation to my sister, Leila, and my brother, Alireza, who have always supported me and my decisions throughout my life.

Abstract

Random hardware failures (RHF) pose a significant risk, potentially leading to data corruption and Control Flow Errors (CFEs) within embedded systems. To counteract these vulnerabilities, hardening strategies are employed, leveraging either specialized hardware or Software-Implemented Hardware Fault Tolerance (SIHFT) methods. This thesis introduces a novel approach, focusing on the C-level implementation of SIHFT techniques to detect CFEs, alongside the development of simulation techniques for rapid software development and validation. Our proposed approach centers on applying SIHFT methods to detect CFEs within C language-based application code preceding compilation.

However, evaluating these methods presents challenges, notably in terms of the introduced overhead to code size and, critically, real-time application execution. The majority of these methods in the literature are implemented using low-level languages like Assembly. Unfortunately, the development flow for embedded systems applications prefers high-level programming languages like C, aligning with functional safety standards.

Nevertheless, a portion of code persists in Assembly language, where the compiler can automatically insert SIHFT methods, albeit typically limited to highly optimized routines or device drivers. An alternative approach, compiling the application code and then hardening the resultant assembly code, introduces more substantial overhead compared to protecting individual statements in a high-level programming language before compilation.

Therefore, our proposed approach in this thesis centers on applying SIHFT methods to detect CFEs, also recognized as Control Flow Checking (CFC), within C language-based application code preceding compilation. To illustrate this approach, we conducted a comparative analysis of two established software-based control flow error detection methods—Yet Another Control-Flow Checking us-

ing Assertions (YACCA) and Random Additive Control Flow Error Detection (RACFED)—implemented in the C programming language. We also assessed the impacts of compiler optimizations.

In the contemporary automotive industry, there is a prevailing trend toward adopting the model-based software design approach. This involves automatically translating executable algorithm models into C or C++ source code. In this context, CFC methods have been integrated into the application behavioral model, and off-the-shelf code generators seamlessly produce the fortified source code for the application. It is worth noting that the majority of SIHFT methods primarily target soft errors, such as single-event upsets typically manifesting as bit flips.

Consequently, the diagnostic metrics commonly provided in the literature, such as error detection latency, fault coverage, and mean time to failure (MTTF), fall short of effectively characterizing these methods when considering the broader spectrum of faults, especially permanent random hardware faults like stuck-at faults. To bridge this gap, our thesis addresses a scenario pertinent to the automotive industry, where the primary concern revolves around permanent random hardware faults, particularly stuck-at faults. Furthermore, we propose a classification scheme aligned with ISO26262 compliance. This classification aims to benefit developers within the automotive sector, where cost and safety considerations often drive the adoption of software-only strategies.

Our results indicated that the diagnostic coverage (DC) for the YACCA method was highest with the **01** optimization level, showing a marked improvement over the unoptimized version (**00**). For RACFED, a similar trend was observed, with the detection rate increasing significantly from **00** to **01**. However, at higher optimization levels (**02** and **03**), while the code size was reduced, some intra-block detection capabilities were lost.

Additionally, our experiments quantified the overheads introduced by these methods. For the TS benchmark, YACCA imposed a text segment size (TSS) overhead ranging from 43.8% at **00** to -28.8% at **03**, while RACFED's TSS overhead ranged from 261.23% at **00** to 100.69% at **03**. Execution time overheads, measured as the increase in the number of executed instructions, showed that YACCA imposed a 318.17% overhead at **00**, decreasing to 90.55% at **03**. RACFED exhibited a 44.58% overhead at **00**, which turned into a slight reduction of 5.06% at **03**. These results underscore the trade-offs between different levels of compiler optimizations

and the effectiveness of CFC methods, providing crucial insights for developers optimizing embedded systems for reliability and performance.

Contents

List of Figures	xii
List of Tables	xiv
Abbreviations	xvi
1 Introduction	1
1.1 Research Objectives	3
1.2 Contributions	16
1.3 Structure	19
2 Background And State-Of-The-Art	21
2.1 Setting the Scene	22
2.2 Software-Implemented Detection Techniques	24
2.3 Control Flow Error	26
2.4 In-Depth Examination of Control Flow Checking Methods	29
2.4.1 CFCSS	29
2.4.2 YACCA	30
2.4.3 ECCA	30
2.4.4 RSCFC	30
2.4.5 CEDA	31

2.4.6	ACFC	31
2.4.7	SCFC	31
2.4.8	HETA	32
2.4.9	SEDSR	32
2.4.10	SIED	33
2.4.11	RASM	34
2.4.12	RACFED	36
2.4.13	In Closing	37
2.5	Design Diversity Based Software Fault Tolerance	38
2.6	Single-Design Software Fault Tolerance Approach	42
2.7	Hardware-Based Fault Tolerance Techniques	45
2.7.1	Redundancy in Hardware-Based Fault Tolerance Techniques	46
2.8	Hybrid methods	47
2.9	Using the C language in automotive industry applications	48
2.10	Functional Safety in the Automotive Industry	49
2.11	ISO26262-compliant classification	50
2.12	A Note on Control-flow Integrity Techniques for Soft Errors-security	54
2.12.1	Data integrity	55
3	Experiment Prerequisites	57
3.1	Fault models	57
3.2	Implemented Software-Based Hardening Technique	58
3.2.1	YACCA	60
3.2.2	RACFED	62
3.2.3	Experimentation with Compiler Optimizations	65
4	Experimental Study on CFC Detection Techniques	66

4.1	Target platform	66
4.2	Hardening technique performance assessment	67
4.3	Fault injection results	68
4.4	C programming language Fault injection results	69
4.4.1	Diagnostic coverage	71
4.4.2	Overheads	75
4.5	Model-Based Software Design Fault injection results	78
4.5.1	Diagnostic Coverage	79
4.5.2	Overhead	80
5	Conclusion	82
5.1	Summary	82
5.2	Main contributions	82
5.3	Future Work	84
	References	86
	Appendix A Guidelines	95
A.1	Introduction	95
A.2	Functions or macros needed in C language	96
A.3	Switch-case construct	96
A.4	If-else construct	98
A.5	Function calls	99
A.6	For loops	103
A.7	Conclusions	103
	Appendix B My publications	105
B.1	Journals	105

B.2 Conferences and Workshops 106

List of Figures

1.1	Automotive SW and E/E content per car [1, 2].	4
1.2	Automotive SW and E/E market and Split of SW market into SW development, integration, and validation/verification [1, 2].	5
1.3	The automotive sensor market is projected to surpass automotive sales, primarily due to robust growth in ADAS sensors [1, 2].	5
1.4	The bathtub curve (red, upper solid line) is a combination of a decreasing rate of early-life failures and an increasing rate of wear-out failures, plus a constant level of random (latent) failures)[3].	7
2.1	Sample code and program CFG example. The execution from basic block BB_1 to BB_2 or from BB_1 to BB_3 are legal, but a jump from BB_1 to BB_4 is illegal and called Control Flow Error (CFE).	28
2.2	The classifier FSM. The transition from <i>Latent after injection</i> or any of the <i>Dangerous/Residual</i> group to a state of the <i>Detected</i> side is allowed only before the FTTI elapses. The transition from the state (<i>detected</i>) by <i>software hardening</i> to the state <i>As golden</i> is performed, when one last line of the log file has been read, only if the behavior of the software components remains the same of the golden run for the entire log file.	53
3.1	Indication, inside the model-based flow, indicating when the CFC is applied using high-level programming languages. The source code is obtained automatically via the Mathworks Embedded Coder from a Simulink semi-formal model, then the obtained source code is manually hardened.	60

3.2	Model-based approach for implementing the CFC methods. The benchmarks are obtained by hardening them in the model, then generating the source code automatically thanks to the Embedded Coder.	61
3.3	Mapping between the signature updates instructions in C and the relative Assembly (RISC-V RV32I) translation. It is possible to see that GCC, configured with O0 optimization settings, keeps the instructions in order.	64
4.1	The proposed test bench architecture. GCC compiles also the classifier, whose source code is generated by the FIM.	67
A.1	Instructions on how to read the Control Flow Graphs represented in this chapter.	96
A.2	Positions of the TEST and SET operations for inter-block CFE detection inside the <code>switch-case</code> constructs for the entry block (indicated as 0, in blue.)	98
A.3	Positions of the TEST and SET operations for inter-block CFE detection inside the <code>switch-case</code> constructs for the exit block (indicated as 5, in green.)	99
A.4	Positions of the TEST and SET operations for inter-block CFE detection inside the <code>if-else</code> constructs for the entry block (indicated as 0, in blue.)	100
A.5	Positions of the TEST and SET operations for inter-block CFE detection inside the <code>if-else</code> constructs for the exit block (indicated as 5, in green.)	100
A.6	Positions of the TEST and SET operations for inter-block CFE detection for a <code>function call</code> .	102
A.7	Positions of TEST and SET operations to detect inter-block CFEs for a <code>for loop</code> .	103
A.8	Positions of TEST and SET operations to detect inter-block CFEs for a <code>for loop</code> containing a <code>break</code> instruction inside it.	104

List of Tables

2.1	Compare Control Flow Control techniques.	38
2.2	Overview of the techniques classification. [4]	49
4.1	Cumulative classifier results obtained from the 7 fault injection campaigns evaluating the YACCA and RACFED methods without compiler optimizations, manually implemented directly in C code, on benchmarks. The "As Golden", "False Positive", "Undefined", and "Error" results are all zero for all columns, so they are not reported in the table.	71
4.2	Classifier results obtained from the fault injection campaign assessing the YACCA implemented manually directly within the C code on TS benchmark with different compiler optimizations. "As golden", "False positive", "Undefined", and "Error" outcomes are all zero for all the columns, so they are not reported in the table.	72
4.3	Classifier results obtained from the fault injection campaign assessing the RACFED implemented manually directly within the C code on TS benchmark with different compiler optimizations. "As golden", "False positive", "Undefined", and "Error" outcomes are all zero for all the columns, so they are not reported in the table.	72
4.4	ISO 26262-compliant classification of the cumulative results obtained from the fault injection campaigns on the benchmarks compiled with almost no optimization (O0).	76

-
- 4.5 ISO 26262-compliant classification of the results obtained from the fault injection campaigns on the TS benchmark compiled with different compiler optimization levels. The results obtained with almost no optimizations (O0) are also reported for ease of reading. . 76
- 4.6 Data regarding memory occupation and executed instruction. T = Tank Level, TS = Timeline Scheduler, and TSS = Text Segment Size. Vanilla refers to the application that is not hardened from its original form. For TS, are reported the overheads with the different optimization levels. All the differences are computed in comparison to the Vanilla version compiled with almost no optimizations (O0). . 77
- 4.7 Classifier results obtained from fault injection campaigns evaluating the YACCA and RACFED methods without compiler optimizations, hardened in MBSD, on benchmarks. The "As Golden", "False Positive", "Undefined", and "Error" results are all zero for all columns, so they are not reported in the table. 79
- 4.8 ISO 26262-compliant classification of the cumulative results obtained from the fault injection campaigns on the benchmarks compiled with almost no optimization (O0). 79
- 4.9 Data regarding memory occupation and executed instruction. T = Tank Level, TS = Timeline Scheduler, and TSS = Text Segment Size. Vanilla refers to the application that is not hardened from its original form. For TS, are reported the overheads with the different optimization levels. All the differences are computed in comparison to the Vanilla version compiled with almost no optimizations (O0). . 81

Abbreviations

ACFC	Assertions for Control Flow Checking
AD	Autonomous Driving
ADAS	Advanced Driver Assistance Systems
ASIL	Automotive Safety Integrity Level
BB	Basic Block
CAGR	Compound Annual Growth Rate
CEDA	Control-flow Error Detection using Assertions
CFC	Control Flow Checking
CFCET	Control Flow Checking by Execution Tracing
CFCSS	Control Flow Checking by Software Signatures
CFE	Control Flow Error
CFG	Control Flow Graph
COTS	Commercial Off-The-Shelf
DC	Diagnostic Coverage
DCU	Domain Control Unit
DFT	Design for Testability
DWC	Duplication With Comparison
E/E	Electrical and Electronic
ECC	Error Correcting Code
ECCA	Enhanced Control Flow Checking using Assertions
ECU	Electronic Control Unit
EV	Electric Vehicle
FI	Fault Injection
FIM	Fault Injection Manager
FM	Failure Mode
FMEDA	Failure Mode, Effects, and Diagnostic Analysis
FSC	Functional Safety Concept
FSM	Finite State Machine
FTA	Fault Tree Analysis
FTC	Fault Tolerant Control
FTTI	Fault Tolerance Time Interval

FuSa	Functional safety
GDB	Gnu DeBugger
HSI	Hardware/Software Interfaces
IC	Integrated Circuit
I-IP	Infrastructure Intellectual Properties
IP	Intellectual Property
ISA	Instruction Set Architecture
ISO	International Standard Organization
MBSD	Model-Based Software Design
NMR	N modular redundancy
OSLC	Online Signature learning and Checking
PC	Program Counter
PCB	Printed Circuit Board
PLD	Programmable Logic Device
PPM	Parts Per Million
RACFED	Random Additive Control Flow Error Detection
RAID	Redundant Arrays of Independent Disks
RHF	Random Hardware Failure
RISC	Reduced Instruction Set Computing
RSCFC	Relationship Signatures for Control Flow Checking
SEooC	Safety Element out of Context
SET	Single Event Transient
SETA	Software-only Error-detection Technique using Assertions
SEU	Single Event Upset
SG	Safety Goal
SIED	Software implemented error detection
SIHFT	Software Implemented Hardware Fault Tolerance
SS	Standby-sparing
TLC	Target Language Compiler Tool
TMR	Triple Modular Redundancy
TSC	Technical Safety Concept
TSR	Technical Safety Requirements
USD	United States dollar
WdM	(AUTOSAR) Watchdog Manager
WDP	Watchdog Direct Processing
YACCA	Yet Another Control-Flow Checking using Assertions

Chapter 1

Introduction

Emerging technologies significantly enhance various aspects of our quality of life, concurrently bolstering societal productivity and efficiency. Exemplary instances include pioneering environmentally friendly transportation systems and advanced production methodologies, which not only diminish human exertion but also optimize the manufacturing of appliances and services.

In the realm of automotive systems, contemporary technological trends primarily revolve around the integration of new functionalities, thereby augmenting the number of on-board embedded systems and processors [5]. Within the automotive domain, these systems are designed to enhance energy efficiency, elevate user experiences through the incorporation of infotainment support, and facilitate autonomous and semi-autonomous control mechanisms, encompassing strategies for cruise control and autonomous piloting [6]. Additionally, in the context of industrial production, the current trajectory of automation fosters collaborative work environments involving human-robot interactions, thereby enhancing overall production. This automation paradigm seeks to mitigate human risks, particularly in scenarios involving hazardous conditions. Both automotive and industrial production scenarios exemplify contemporary instances of safety-critical applications, where any functional failure in the equipment, machinery, or devices supporting the application can lead to severe consequences, including critical injuries, fatalities, substantial property damage, or extensive environmental harm [7]. Consequently, the intricate electronic devices integrated into these systems must adhere to rigorous safety, reliability, and security constraints to ensure the flawless operation of the entire system. Furthermore, it is

essential to acknowledge that the integration of cutting-edge technologies in these domains not only amplifies the potential benefits but also introduces new challenges. The complexity of these systems demands continuous advancements in safety measures, reliability protocols, and security frameworks to stay ahead of potential risks and vulnerabilities. Researchers and practitioners alike must remain vigilant in addressing these challenges to uphold the integrity of technological innovations in the automotive and industrial production sectors.

The automotive industry is currently experiencing a transformative phase marked by the synergistic evolution of autonomous driving (AD), connected vehicles, electrification of the powertrain, and shared mobility (also called the ACES trends). These developments not only disrupt the traditional automotive value chain but also exert a significant influence on stakeholders, contributing to the anticipated 7 percent compound annual growth rate (CAGR) in the automotive software (SW) and electrical and electronic components (E/E) market. This growth is projected to escalate from USD 238 billion to USD 469 billion between 2020 and 2030, surpassing the 3 percent CAGR expected for the overall automotive market during the same period.

The collective impact of ACES, further accelerated by the COVID-19 pandemic, is reshaping the future of mobility, affecting customer preferences, technology adoption, and regulatory frameworks [8]. The electric vehicle (EV) market is witnessing an influx of new players with higher valuations than established OEMs, prompting substantial investments in software and electrification by automotive companies and their suppliers. By 2030, the global automotive software and electronics market is forecasted to reach USD 462 billion, growing at a 5.5 percent CAGR from 2019 to 2030, while the overall automotive market for passenger cars and light commercial vehicles (LCVs) is expected to expand at a 1 percent CAGR during the same period. Figure 1.1 details the breakdown of automotive software (SW) development, highlighting diverse domains and tech-stack elements. While the overall electrical and electronic (E/E) market is expected to outpace the automotive market, the specific content of electronics and software per car varies significantly based on segment, powertrain, and AV level.

Despite a modest increase in passenger car and LCV sales, the automotive software and electronics market is poised to experience nearly four times the growth rate, with electronic control unit (ECU) and domain control unit (DCU) sales projected to reach USD 144 billion by 2030. Software development, particularly integration,

verification, and validation, is anticipated to be the second-largest market segment with a revenue potential of USD 83 billion. Power electronics emerges as the fastest-growing component market, driven by a 23 percent CAGR through 2030, fueled by the adoption of electric vehicles. Sensors, especially those for AD or Advanced Driver Assistance Systems (ADAS), are expected to grow at a 6 percent CAGR, driven by LiDAR, cameras, and radars. Beyond core SW development, subsequent processes include customization, validation, verification, and integration, with post-production maintenance adding to development costs, as shown in Figure 1.2.

The automotive software market is set to more than double from USD 31 billion in 2019 to around USD 80 billion in 2030, with ADAS and AD software accounting for nearly half of the market. The sensor market is also projected to grow, reaching USD 46 billion in 2030, primarily due to increasing demand for ADAS and AD sensors. As the industry transitions towards software-defined vehicles, strategic and operational actions are imperative for automotive companies to harness the potential of this transformative shift. Most sensors align with automotive market growth, but those linked to ADAS and autonomous driving drive an anticipated 8 percent growth in automotive sensors, detailed in Figure 1.3.

In 2023, the automotive industry served as an economic powerhouse for Europe, contributing approximately 10 percent to the region's exports. With over 17,300 companies forming a comprehensive network of OEMs and suppliers, the sector directly or indirectly employed more than 6 percent of the region's workforce, generating positive spillover effects beyond its boundaries.

1.1 Research Objectives

A key facet of fault tolerance is to ensure the continued correct operation of modern computing systems despite internal faults. The primary objective underlying fault tolerance endeavors is to increase system dependability. In a fault-tolerant system, the aim is to facilitate seamless transitions to alternative modules and thereby sustain service provision in the face of faults, by either concealing faults or detecting errors. To fulfill this aim, fault-tolerant systems must uphold specified service delivery, even amidst component faults.

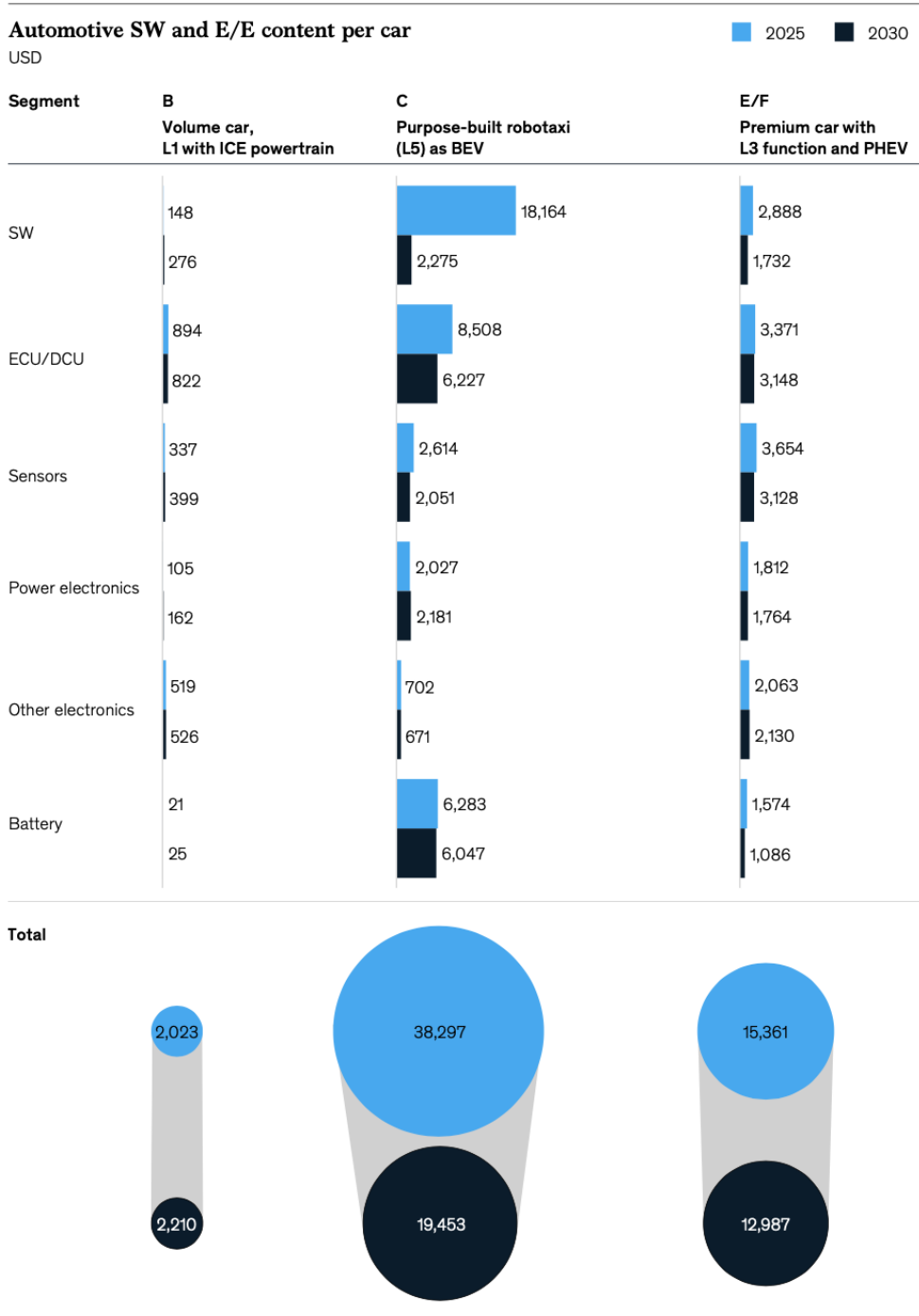


Fig. 1.1 Automotive SW and E/E content per car [1, 2].

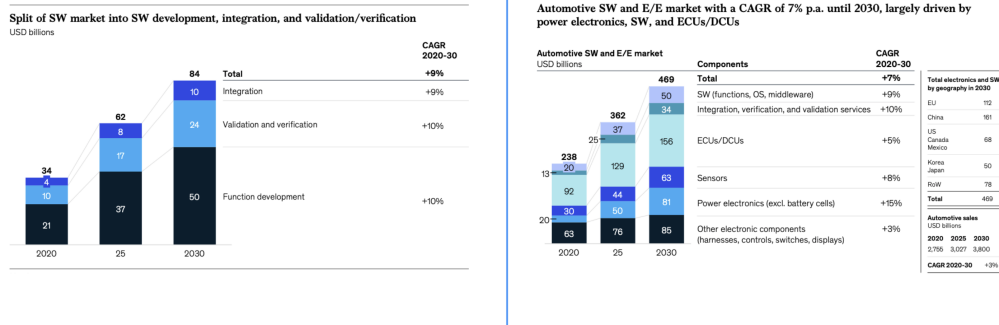


Fig. 1.2 Automotive SW and E/E market and Split of SW market into SW development, integration, and validation/verification [1, 2].

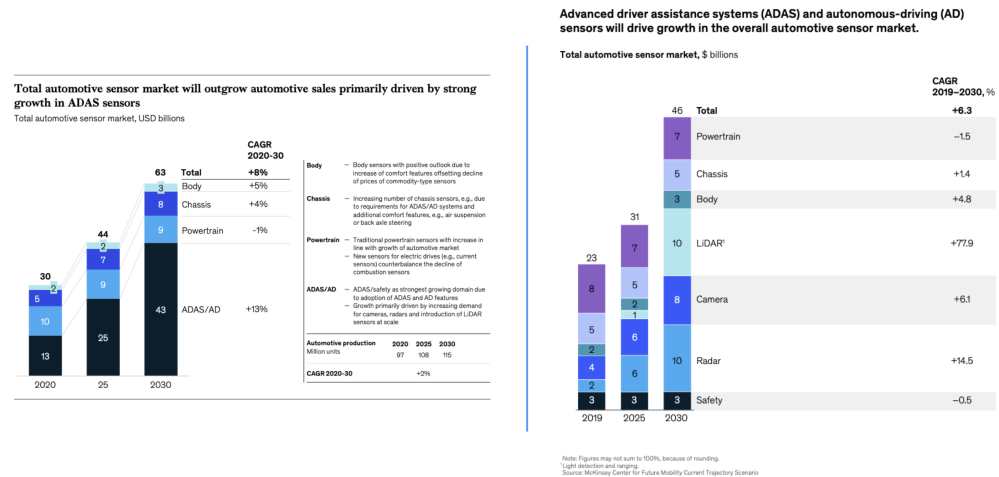


Fig. 1.3 The automotive sensor market is projected to surpass automotive sales, primarily due to robust growth in ADAS sensors [1, 2].

Failures arise when the behavior of a running system diverges from the system's expected behavior. Failures are caused by errors, while faults are the underlying cause of errors. Yet, it is noteworthy that not all faults necessarily lead to errors, and a single fault can precipitate multiple errors. Similarly, a solitary error can culminate in multiple failures. Redundancy, in some form, is an essential component across all fault tolerance approaches to ensure the system's capacity to withstand faults. Redundant devices, networks, data, or applications are leveraged based on the fault class at hand.

As of now, novel technologies elevate various facets of our quality of life while concurrently bolstering societal productivity and efficiency. Illustrative instances include innovative environmentally conscious transportation systems and advanced production methodologies, streamlining human effort and optimizing the generation of appliances and services. In the realm of automotive systems, the trajectory of emerging technological trends accentuates the introduction of novel features, expanding the array of onboard embedded systems and processors [9]. Within the automotive domain, these systems are engineered to optimize energy consumption, enrich user experiences through infotainment support, and institute autonomous and semi-autonomous control mechanisms encompassing methods like cruise control and autonomous piloting [10]. Furthermore, within the production sphere, burgeoning automation trends foster collaborative work environments uniting human workers and autonomous robots, thus amplifying production. This automation paradigm additionally seeks to mitigate human risk in scenarios involving hazardous conditions.

Central to the silicon lifecycle within automotive industries are Design for Testability (DFT) tasks, spanning from initial DFT insertion and high-quality test pattern generation to stress tests accelerating aging. Traditional structural patterns, along with increasingly prevalent system-level tests, are then applied. Volume diagnosis supplements these steps within the manufacturing test framework, historically the primary role of DFT beyond manufacturing. However, the contemporary landscape demands an extension of DFT to encompass in-system and in-field operations (see Figure 1.4).

While manufacturing stress tests effectively filter out chips prone to early-life failures, subsequent phases in the integrated circuit (IC) lifecycle introduce random latent defects. As aging takes hold, wear-out failures become a concern, leading to an escalating failure rate over time. Consequently, in-system testing and monitoring

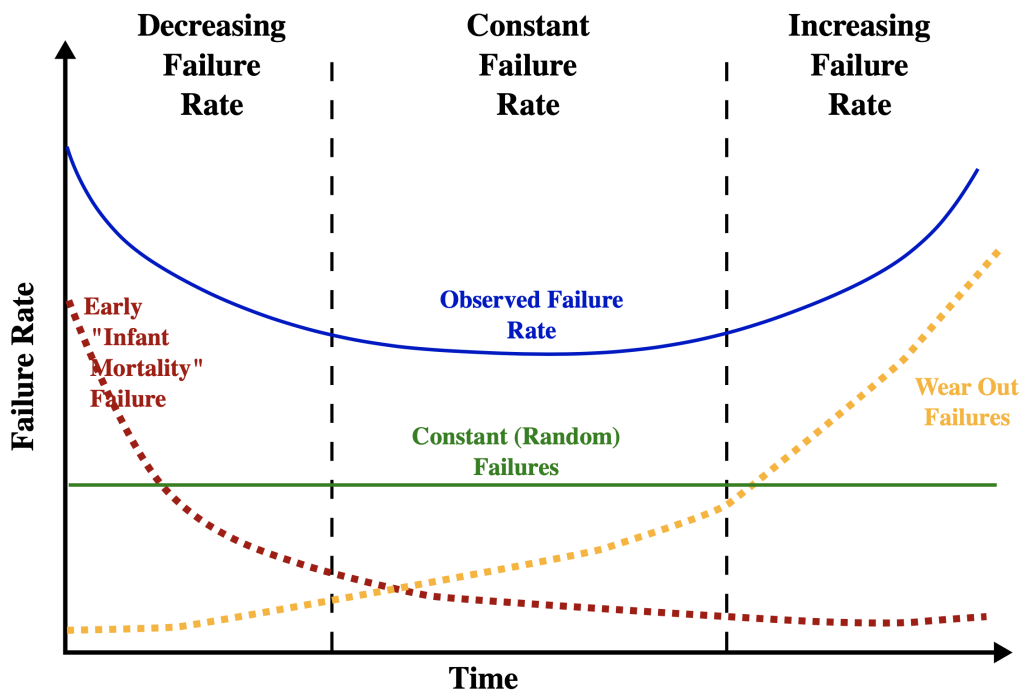


Fig. 1.4 The bathtub curve (red, upper solid line) is a combination of a decreasing rate of early-life failures and an increasing rate of wear-out failures, plus a constant level of random (latent) failures)[3].

during these lifecycle stages are imperative. In specific domains, remote diagnosis proves valuable for retrieving fail logs and computing defectivity profiles. This data, in turn, contributes to refining tests and preempting similar issues in future deployments.

Both automotive and industrial production domains represent paradigmatic instances of safety-critical applications, wherein any functional malfunction of the supporting equipment, machinery, or devices could trigger dire repercussions, spanning critical injuries, fatalities, substantial property damage, or extensive environmental harm [11]. Consequently, the intricate electronic devices now integrated within these systems must rigorously adhere to safety, reliability, and security imperatives to ensure the seamless operation of the entire system.

Within the automotive domain, prominent corporations have invested, and are poised to continue investing, substantial capital in new technologies to not only implement but also broaden their applicability across various automotive functions. These applications encompass the development of diverse levels of vehicular autonomy

driven by the attendant benefits to user safety, security, traffic latency reduction, and energy efficiency. Nevertheless, these technological advantages concurrently present various challenges yet to be definitively resolved. In principle, well-established methodologies for designing and developing secure and safe devices could be repurposed for use in these novel applications. However, both the automotive and autonomous machinery domains presently exploit a medley of innovative technologies, including Artificial Intelligence (AI) and computer vision, furnishing a distinct advantage in effecting more streamlined procedures. It's worth noting, though, that this trend equally introduces the ability for contemporary devices to integrate intricate algorithms, thereby augmenting application complexity and imposing substantial constraints concerning real-time operation, available power resources, and performance thresholds.

In practice, the development of modern safety-critical applications hinges upon three core elements: *i*) robust high-performance operation and power efficiency, *ii*) cost-effectiveness, and *iii*) unwavering safety and reliability [12]. In numerous instances, manufacturers and designers confront these demands by harnessing the latest transistor technology and scaling methods, thereby pushing the boundaries of Moore's law to incorporate an elevated transistor count within the same device. This endeavor yields appreciable enhancements in execution performance, power consumption, and practical production expenses.

However, various studies [13], [14], [15], [16], [17] have demonstrated that devices constructed using these cutting-edge technologies are inherently susceptible to an array of faults manifesting during initial operational stages and, with greater frequency, throughout their active lifespan. These faults may arise from two primary sources: *(i)* inherent defects stemming from manufacturing processes or component fatigue, and *(ii)* environmental or external influences [18]. In the former case, device faults might emanate from manufacturing anomalies that evade detection during end-of-production testing, thereby precipitating unforeseen behaviors during operational life-cycles. Furthermore, components within a device are predisposed to degradation (e.g., electro-migration or gate-oxide effects) following prolonged operation or even during periods of idleness (e.g., idle operational mode) [19], thereby potentially generating intermittent or permanent faults. In such scenarios, the faults arise due to aging or wear-and-tear effects [20], [21], [22]. Conversely, external influences also exert sway over device operation. Environmental factors temporarily or permanently alter electrical parameters, resulting in transient fault effects that impinge upon

ongoing device applications. These fault effects propagate across the device as soft errors, which solely emerge when applications are executing on afflicted devices. Exposure to high-energy particles (triggering radiation effects) or electromagnetic interference (EMI) increases device vulnerability to transient faults, disrupting the electronic charge of one or more storage components within the device and toggling the state of transistors employed for data storage. As this data courses through the circuitry, multiple errors can arise within the application. In the most extreme instances, external interventions can lead to permanent damage to the device.

fault-tolerance methods focus on detecting and recovering from faults, regardless of their types, in order to ensure the correct functioning of the system. To achieve a given reliability target, one commonly used fault-tolerance technique is the utilization of redundancy, in terms of hardware, software, information, and time, exceeding what is normally required for system operation. Hardware redundancy techniques involve adding extra hardware components to detect or tolerate faults. For example, multiple cores or processors can be utilized instead of a single one, with each application being executed on a separate core/processor, enabling fault detection and even correction. Another technique, time redundancy, allocates extra time to perform system functions and detect faults, without violating the timing constraints of real-time systems. The re-execution technique is an example of time redundancy, where a faulty task is repetitively executed on the same hardware until the correct output is obtained. Information redundancy techniques, such as error detection and correction coding, are commonly used in memory units, storage devices, and data communication to ensure reliability. Redundant Arrays of Independent Disks (RAIDs) are another example of information redundancy, where data is organized and stored in multiple configurations to enhance reliability. Additionally, software redundancy involves adding extra software to detect and tolerate faults. For example, N-version programming involves separate groups of programmers designing and coding a software module multiple times, reducing the likelihood of the same mistake occurring in all versions. Checkpointing, on the other hand, stores the last fault-free state of a process in stable memory, allowing the system to roll back to that state and re-execute the application in case of a fault. By employing these fault-tolerance methods, systems can ensure reliable functioning despite the occurrence of faults [23].

The proliferation of semiconductors in automotive applications is further propelled by the increasing importance of vehicle software. Current legal mandates

mandate the integration of diverse software-driven safety features, such as emergency braking assistance. In premium-tier vehicles, software-based functionalities, ranging from infotainment systems to advanced ADAS and autonomous driving capabilities, are progressively shaping customers' purchasing preferences. This dynamic landscape highlights the growing significance of software-driven functions, influencing the overall appeal and market competitiveness of vehicles.

Several of these new devices and technologies, initially integrated into vehicles as enablers for ADAS, have been instrumental in enhancing vehicle safety. Despite providing simple and partial autonomous features at low levels of autonomy, ADAS, inclusive of lane departure warning systems, adaptive cruise control, blind spot monitors, and automatic parking, has demonstrated its value. These systems operate within the conventional vehicle Electrical/Electronic (E/E) architecture, requiring no major modifications. ADAS has found extensive adoption in today's commercial vehicles due to its cost-effectiveness.

Embedded systems are employed in various industries, such as aerospace, automotive, and defense, to implement safety or mission-critical applications. Functional safety (FuSa) is a part of the product safety process that focuses mainly on the absence of unreasonable risks. For this purpose, FuSa standards provide reference life cycles to implement embedded systems. In other words, it is required to guarantee that the system can perform tasks correctly within a defined time or, at least, to bring the controlled physical process into a safe state.

The incorporation of FuSa testing introduces an additional layer of complexity to semiconductor testing, a dimension further intensified by the autonomy inherent in advanced systems. The absence of human intervention in autonomous processes amplifies the necessity for stringent safety measures, as there is no human presence to improvise or rectify potential issues in real-time. Consequently, in scenarios where a 10nm chip functions as the neural core of an autonomous vehicle's artificial intelligence, the conventional benchmark of testing 95% or more of the transistors on the chip is no longer deemed acceptable.

In the landscape of semiconductor testing, the advent of autonomy poses distinctive challenges that demand a reevaluation of established testing paradigms. The intricate interplay of autonomous systems and semiconductor components underscores the criticality of comprehensive testing strategies. Merely adhering to traditional testing thresholds may prove insufficient in ensuring the robustness and

reliability of semiconductor devices, particularly when deployed in autonomous applications.

This paradigm shift necessitates a nuanced approach to testing methodologies, wherein the focus extends beyond conventional benchmarks to encompass the intricate interactions within autonomous systems. Researchers and practitioners must delve into innovative testing frameworks that address the unique requirements posed by the autonomy factor, ensuring that safety and reliability are upheld at levels commensurate with the heightened complexities of modern semiconductor devices. As semiconductor technology continues to evolve, the scientific community must remain proactive in developing and refining testing protocols to match the intricacies introduced by autonomous functionalities in semiconductor-based applications.

Most current standards derive from IEC 61508 on functional safety for electrical, electronic, and programmable electronic safety-related systems. In particular, the standard for automotive industry applications is ISO 26262, targeting applications in charge of safety-critical tasks. It was first released in 2011 and then updated in 2018 [24].

In compliance with the latest safety standard, which mandates an exceptionally stringent norm, the imperative requirement is to limit defects to as small as zero. This strict requirement arises from the increasing shift towards automation in the electric and electronic control of vehicles. Consequently, there is a heightened significance in evaluating the margin of error in component performance and reliability. To illustrate the gravity of a defect, consider a Printed Circuit Board (PCB) featuring 40 Integrated Circuits (ICs), each boasting 90% fault coverage and 90% yield. This configuration results in a reject rate of 34.4% or 344,000 defective parts per million (PPM). This underscores the critical need for precision in adhering to stringent safety standards in the context of automated control systems in vehicles [25].

Widely acknowledged as a cornerstone in the realm of automotive electronics design and formalized in ISO 26262 standards, the practice of designing for functional safety entails a meticulous examination and integration of countermeasures against potential vulnerabilities within the System-on-Chip (SoC) design. These vulnerabilities encompass facets of the design that render it susceptible to mission failure arising from transient errors occurring during routine vehicle operation. In the context of ADAS applications, the repercussions of such failures could be catastrophic for vehicle occupants, posing a significant threat to human safety. Furthermore, such

failures have the potential to erode consumer trust precisely at a juncture when technological adoption is on the ascent, thereby impeding the widespread acceptance of the technology.

The Automotive Safety Integrity Level (ASIL) systematically addresses the mitigation of risks in automotive systems, as defined by the ISO 26262 standard. This classification system evaluates the severity of hazards arising from potential malfunctions in automotive components, assigning ASILs ranging from A (lowest) to D (highest). Conforming to ISO 26262 involves a comprehensive process that guides identifying safety goals and requirements, encompassing internal and external components within and beyond the system boundary. Automotive Original Equipment Manufacturers (OEMs) and their suppliers are diligently aligning their practices with ISO 26262 standards. Alignment with ASIL is integral to contemporary vehicle development, representing a pivotal aspect of ISO 26262 compliance. Safeguarding vehicle systems in automotive software development extends to managing fail-safe conditions, including hardware malfunctions, environmental stress, and software bugs, with ISO 26262 providing a structured framework for analysis. In the realm of ADAS, some manufacturers openly disclose data regarding the failure rates of their components, contributing valuable insights into the relative reliability of diverse technologies. For instance, NXP Semiconductors provides failure rate data for its automotive microcontrollers, shedding light on the robustness of its offerings. Similarly, Infineon Technologies specifies a failure rate of less than 10^{-9} per hour for its automotive-specific microcontrollers [26, 27].

Under adverse conditions, ADAS systems typically transition into a fail-safe state, a strategic design approach that aims to mitigate the consequences of a failure rather than preventing it outright. This proactive strategy aims to reduce or eliminate potential harm resulting from system malfunctions, emphasizing the commitment to safety inherent in ASIL and ISO 26262 compliance.

ISO 26262 addresses two distinct categories of faults within the context of functional safety [28]:

- **Systematic faults:** Arising from specification or design anomalies, systematic faults manifest in a deterministic manner. These faults can affect both software and hardware components and necessitate corrective measures to enhance the development process, including safety analyses and thorough verification. A prevalent example of a systematic fault is a development bug.

- Random hardware faults: Unpredictably occurring during the operational lifespan of a hardware component, random hardware faults result from physical processes such as wear-out, physical degradation, or environmental stress. While reliability engineering practices can mitigate the occurrence of random hardware faults, their complete elimination remains unattainable. This category further branches into two subtypes:
 - a.* Permanent faults: These persist until addressed or repaired and encompass examples like stuck-at faults and bridging faults.
 - b.* Transient faults: Occurring momentarily and then dissipating, transient faults may stem from causes like electromagnetic interference or alpha particles. With the reduction in technology node scale, memory elements like flip-flops and memory arrays are increasingly susceptible to transient faults, exemplified by occurrences such as Single Event Upset (SEU) and Single Event Transient (SET).

SEU involves a transient change in a digital memory element's state caused by a single ionizing particle, leading to temporary errors. Typically associated with cosmic rays, SEUs are critical concerns in semiconductor devices. In contrast, SET occurs when a high-energy particle disturbs a semiconductor device, resulting in transient changes in its electrical characteristics. Unlike SEUs affecting memory states, SETs manifest as temporary glitches in logic gates or circuits, causing momentary disruptions in electronic systems[29].

The susceptibility of memory elements to transient faults, exemplified by SEU and SET, intensifies with shrinking technology nodes. Consequently, addressing and mitigating these faults become pivotal for ensuring the reliability and functional safety of semiconductor devices, especially in safety-critical applications adhering to ISO 26262 standards [28].

Moving to permanent faults, achieving the required safety targets clearly mandates the adoption of special techniques to minimize the chances that possible faults created by the manufacturing process or by other mechanisms (e.g., aging) escape the different test procedures applied at the device, board and system level. Moreover, given the very high safety targets required, the advanced semiconductor technology used to manufacture current automotive devices, and the relatively short life-time of these technologies in safety-critical applications, including ADAS systems, it is

mandatory to develop efficient techniques to detect permanent faults (in-field test) before they cause critical failures.

In this concept, the designers aim to prevent systematic design errors and ensure hardening against Random Hardware Failures (RHF). An RHF is a failure that affects a physical component of a computation platform, especially in this work, a central processing unit register or a (random access) memory location. While systematic errors can be avoided with a properly implemented life cycle, RHFs are unavoidable due to the physical nature of the electronic components. For example, consider platooning vehicles, which is the linking of two or more trucks in convoy using connectivity technology and automated driving support systems. In this example, sensor faults have become a common fault problem in fault tolerant control (FTC) research, and the frequency of fault occurrence in the actual system cannot be ignored. The faults of single or multiple vehicles lead to the breakdown of the entire platooning system, so the FTC is a critical issue [30].

Hardening the system generally means adding redundancy. It can be implemented in two ways: (i) adding extra hardware components or (ii) adding software instructions in the application code. The first strategy requires adding special hardware modules to the system architecture like watchdogs [31], checkers [32], or Infrastructure Intellectual Properties (I-IP) [33]. On the other hand, software redundancy techniques are much more flexible and cost-effective in error detection compared to hardware methods. This is because they perform extra instructions without any hardware component changes and allow for monitoring of the application's correct execution.

Software-Implemented Hardware Fault Tolerance (SIHFT) methods are software redundancy techniques that are especially helpful when other hardening methods result in hardened hardware components with high computation power or high cost per unit. The component cost per unit is particularly critical for automotive applications, where a design is produced in tens of thousands of units. On the other hand, software techniques allow the implementation of dependable systems without the high cost of hardened hardware but at the expense of higher development costs. Nonetheless, this cost can be split over the produced units, making them economically convenient. Various SIHFT methods have been proposed over the years, such as Control Flow Checking (CFC) [34–48].

Selecting among the various CFC methods proposed in the literature is challenging. Hence, in this paper, we propose a comparison methodology that consists of selecting a set of representative applications, hardening them with chosen methods, and finally performing the fault injection. To evaluate our approach, two established CFC methods were selected and applied to two benchmarks representing typical applications used in the Automotive Industry.

In this work, we are mainly concerned with RHF's caused by permanent stuck-at faults. Most of the proposed approaches for SIHFT methods target single-event upsets (soft errors), such as bit flips. As a result, the diagnostic figures provided in the literature are insufficient to characterize the techniques effectively. Common diagnostic metrics in the literature include:

- **Error Detection Latency:** Measures the time taken to detect an error after it has occurred. This metric is useful for understanding the response time of a fault-tolerance method but does not provide insights into the method's overall effectiveness against various types of faults.
- **Fault Coverage:** Indicates the proportion of faults that a method can detect and/or correct. While important, this metric often focuses on specific fault models (e.g., single-event upsets) and may not account for more complex or permanent faults like stuck-at faults.
- **Mean Time to Failure (MTTF):** Represents the average time between failures for a system. MTTF provides a general sense of reliability but does not capture the nuances of how different faults impact system performance and safety.
- **Performance Overhead:** Assesses the additional computational resources (e.g., processing time, memory usage) required by a fault-tolerance method. Although crucial for real-time applications, this metric alone cannot determine the method's efficacy in various fault scenarios.

Therefore, targeting the most suitable fault models is essential, as we propose in this work.

In this work, we implemented the CFC methods in the C programming language, though in the literature, the CFC methods are usually implemented in assembly. There is still a non-negligible portion of code written in the assembly language. Also,

implementing SIHFT methods in assembly lets the compiler automatically insert most SIHFT methods.

One of the reasons for our choice is that implementing in C, compared to other programming languages, significantly outperforms execution time, energy consumption, and peak memory usage for selected benchmarks [49]. In addition, implementing the CFC method in a high-level programming language reduces the developers' challenges in comparison to using low-level programming languages. Moreover, writing Assembly code is not the preferred development flow for embedded systems since the functional safety standards mandate adopting high-level programming languages such as C whenever possible (as requested by part 6 of ISO 26262 Standard).

For application codes written in high-level programming languages, such as C, it is possible to compile and then harden the obtained assembly code. However, this approach introduces more significant overhead, especially in terms of execution time, which is a primary concern for real-time applications compared to our approach, which includes protecting single statements in the high-level programming language before compiling the code. The drawback of our approach is that the compiler may remove the extra instructions or change the order of all the instructions to optimize the code. We also investigated this aspect by repeating the fault injection campaigns with all four optimization levels offered by the Gnu Compiler Collection (GCC).

The simulation results were expressed in compliance with ISO 26262 automotive functional safety standards. These results are obtained by assessing the efficiency of the CFC methods based on the RHF's detection, defined by the Standard as Detection Coverage (DC).

1.2 Contributions

In this chapter, we present the contributions of our research, which encompass several novel aspects in the field of fault tolerance and safety-critical systems.

Our research focuses on enhancing the reliability of embedded systems through Control Flow Checking (CFC) methods. At the core of our investigation is the assumption that faults affecting the Program Counter (PC) are critical. These faults can disrupt the control flow of a program, leading to significant operational failures. Thus, we consider PC faults as our primary fault model, introducing this assumption

at the outset of our study to establish a clear context for our contributions. To mitigate the effects of PC faults, we propose a robust software-based hardening technique. This technique is designed to counteract the disruptions caused by these faults, ensuring the integrity of the program's execution flow. Our approach is a key contribution to the field, demonstrating significant improvements in fault tolerance for embedded systems.

To validate our methodology, we implemented and evaluated two distinct CFC techniques: "Yet Another Control-Flow Checking using Assertions" (YACCA) and "Random Additive Control Flow Error Detection" (RACFED). These techniques were chosen for their contrasting philosophies—YACCA utilizes bit masking and focuses on inter-block detection, while RACFED employs random numbers for both inter-block and intra-block detection capabilities. YACCA was selected for its simplicity of implementation, whereas RACFED was chosen for its recent advancements and comprehensive detection capabilities.

In our evaluation, we used two benchmark applications to represent real-world scenarios similar to automotive applications. The first benchmark, a timeline scheduler (TS), is essential for operating systems managing periodic tasks. The second benchmark, a tank level controller (T), maintains the liquid level in a tank using on-off logic, analogous to battery charge level control algorithms in electric and hybrid vehicles. These benchmarks were selected due to their relevance and representativeness, despite not using proprietary automotive applications for intellectual property reasons.

Both benchmarks were automatically generated using Simulink Coder, based on the Model-Based System Development (MBSD) methodology. This high-level approach contrasts with traditional assembly language implementations of CFC techniques. Implementing CFC methods in C programming language, as we have done, offers numerous advantages: it is more developer-friendly, reduces implementation time, and minimizes errors compared to assembly language.

Furthermore, we investigated the impact of compiler optimizations on the effectiveness of CFC methods by conducting experiments with different optimization levels (00, 01, 02, and 03), offered by GCC for RISC-V architecture. Maintaining the correct instruction order and avoiding optimizations that could interfere with signature updates are crucial aspects addressed in our methodology.

The simulation results obtained from the benchmark applications running on a RISC-V-based target platform provide insights into the performance of the proposed hardening techniques. We analyzed fault injection, diagnostic coverage, and overhead for both manually implemented CFC methods. These analyses were conducted considering different optimization levels for manual hardening in the C programming language and no optimization levels for the Model-Based Software Development (MBSD) approach.

Additionally, we present a comprehensive set of guidelines for implementing CFC methods using the C programming language. These guidelines are designed to assist developers in the development of safe and reliable embedded systems. While following these guidelines is not mandatory, they offer a practical approach to implementing critical safety embedded systems.

The effectiveness of our proposed approach and guidelines is demonstrated through two case studies, showcasing the successful deployment of reliable embedded systems in safety-critical scenarios, particularly in the automotive industry context. These case studies validate the applicability of our methodology in real-world settings.

We acknowledge several assumptions and limitations inherent in our approach. Firstly, our reliance on manual implementation of CFC techniques in C code introduces potential challenges such as time consumption and error-proneness, particularly for large and complex codebases, suggesting a need for future exploration into automated insertion methods. Moreover, our evaluation's focus on the GCC compiler for the RISC-V architecture may limit generalizability, necessitating investigation into the behavior of other compilers and architectures. While our approach effectively targets faults affecting the Program Counter (PC), it may not be as efficient for other fault types, warranting further research into broader fault model coverage. Additionally, our findings and guidelines are tailored to the C programming language, requiring careful consideration when adapting to other languages like C++. Furthermore, while model-level implementation using Simulink simplifies complexity, it may impose limitations based on model fidelity and tool capabilities. Assumptions regarding a homogeneous fault model, consistent compiler behavior, and model accuracy are made, acknowledging potential variations and deviations that could affect the validity of our results. Manual implementation overhead, limited fault coverage, performance overhead, dependency on target architecture, and the assump-

tion of an adversarial fault introduction environment are further considerations that shape the scope and applicability of our approach. Understanding and addressing these assumptions and limitations provide a holistic perspective on the context and interpretation of our research findings.

In conclusion, our contributions extend beyond the development of novel fault-tolerant techniques. We provide a systematic methodology, supported by guidelines, for implementing CFC methods in high-level programming languages, thereby offering a practical solution for enhancing the safety and reliability of embedded systems. Future research directions could explore the applicability of our approach to C++ compilers, considering the prevalence of embedded systems using C++ code.

1.3 Structure

The subsequent sections of this manuscript are organized as follows:

chapter 2 provides additional background information about the contemplated embedded systems, specifically focusing on the Control Flow Error, along with an examination of Software-Implemented Hardware Fault Tolerance techniques. The discussion delves into the utilization of the C programming language within the context of functional safety in the automotive industry, including a consideration of ISO 26262-compliant classification. Concluding this chapter is a dedicated section titled "A Note on Soft Errors in Security."

chapter 3 explains the proposed fault models and outlines our methodological approach for the integration of established software-based hardening techniques into high-level programming languages.

chapter 4 outlines the experimental configuration and presents the results obtained through simulation. It is followed by an extensive theoretical analysis of the efficacy of the high-level programming language implementation. This analysis encompasses not only an evaluation of the techniques' effectiveness but also the essential examination of the associated overheads, particularly in the context of real-time systems.

Finally, chapter 5 unveils forthcoming research directions and provides a summation of the study's key conclusions.

Additionally, Appendix A delivers a comprehensive guideline for the application of a subset of CFC methods to application code coded in the C programming language. This appendix serves as a valuable resource for practitioners seeking to enhance the reliability and robustness of their software systems.

Chapter 2

Background And State-Of-The-Art

This section embarks on an exploration of fault-tolerant systems, focusing on both hardware and software methodologies. We begin by providing a clear and comprehensive explanation of Control Flow Errors (CFEs) and Control Flow Checking (CFC) methodologies, which form the cornerstone of our investigation. As outlined in the introduction, our research is dedicated to enhancing the reliability of embedded systems against CFEs induced by external perturbations. Before delving deeper into the intricacies of CFE, it is crucial to delineate the scope of our research.

We commence with an extensive review of state-of-the-art (SOTA) software-implemented detection methods in section 2.2, clarifying the concept of CFE in section 2.3: Control Flow Error. Subsequently, we conduct an in-depth examination of Control Flow Checking Methods in section 2.4, elucidating their role in bolstering system resilience against CFEs.

Design Diversity-Based or Multiple-Version-Based software fault tolerance, which involves using multiple versions or variants of software, either executed sequentially or in parallel, is discussed in section 2.5: Design Diversity Based Software Fault Tolerance.

Complementary to this, section 2.6 delves into a detailed analysis of software-based fault tolerance methods, particularly focusing on the Single-Design Software Fault Tolerance Approach.

Various hardware-based fault tolerance methods are delineated in section 2.7 : Hardware-Based Fault Tolerance, shedding light on their efficacy in fortifying system robustness.

In section 2.8: Hybrid-Based Approaches, we explore hybrid methodologies that integrate both hardware and software techniques, presenting a comprehensive overview of their benefits and limitations.

Furthermore, section 2.9: Using the C Language in Automotive Industry Applications, addresses the utilization of the C language in automotive industry applications, highlighting its significance in developing fault-tolerant systems.

To augment understanding, a comprehensive explanation of ISO 26262-compliant classification is provided in section 2.10 and section 2.11: Functional Safety Standards and ISO 26262 Compliance.

Additionally, a note on Control-flow Integrity Techniques for Soft Errors-security is presented in section 2.12: A Note on Control-flow Integrity Techniques for Soft Errors-Security, underscoring their relevance in mitigating security vulnerabilities arising from soft errors.

2.1 Setting the Scene

Embedded systems oriented toward safety-critical tasks play a crucial role in facilitating the implementation of sensor fusion and deep learning algorithms for AI applications. These embedded systems find particular relevance in Advanced Driver-Assistance Systems (ADAS) within the automotive domain, where they are integrated into systems designed to perform various functions, such as Automatic Cruise Control, Pedestrian Recognition and Protection, Forward Collision Warning, Automatic Parking, and Automatic Pilot [50] [51]. ADAS systems heavily rely on sensor inputs, including cameras, radars, and LiDARs [52], which generate a continuous stream of data that necessitates real-time processing and decision-making [53]. Embedded systems are well-suited for handling the data-intensive processing requirements of these ADAS applications and are increasingly adopted by manufacturers. Additionally, ADAS serves as an intermediary step towards the realization of semi-autonomous and self-driving vehicles [54].

In contemporary safety-critical embedded system design, adherence to industrial standards, such as ISO26262 in the automotive sector, is imperative. These standards impose specific conditions to ensure correct execution, a high level of functional safety, and reliability of GPU devices throughout their production and operational life. Compliance with these regulations is vital due to several key factors: i) operational constraints inherent to safety-critical applications, ii) the technological advancements within embedded systems, and iii) the architectural intricacies and complexities of embedded systems.

Regarding the first factor, any proposed fault-tolerance solutions must account for real-time operation limitations, constrained power budgets, and the high data-intensive processing nature of safety-critical applications. Furthermore, these solutions should evaluate performance limitations during in-field testing and potential mitigation strategies. The availability of system modules and other resources must also be considered during the operation of safety-critical applications.

On the other hand, technology scaling approaches aim to boost performance and reduce the physical size of embedded systems while maintaining cost-effectiveness. However, these cutting-edge scaling methods increase susceptibility to faults, making new devices more prone to errors stemming from both internal and external factors, such as radiation. Notably, these faults can manifest after prolonged device operation, as a result of aging or wear-and-tear, or due to external factors like radiation effects, electromagnetic interference, and extreme variations in temperature and power supply [55], [56], [57], [58], [59]. Consequently, fault-tolerance solutions and reliability assessments assume a pivotal role in the safety-critical domain, particularly in the context of GPUs used for such applications.

An embedded system comprises hardware and software components that interact with the physical environment through sensors and actuators, serving a dedicated function [60]. The software component typically consists of two subcomponents: the application and, in some cases, a real-time operating system. The need for an operating system depends on the specific applications the embedded system is intended to execute. Embedded systems without an operating system are referred to as "bare-metal" systems, indicating that the application directly interfaces with the hardware. The hardware component of an embedded system can be a microcontroller or an application processor, each tailored for distinct use cases. Application processors are employed when a general-purpose operating system is required to support the desired

application(s), whereas microcontrollers are designed for executing bare-metal or real-time applications with the assistance of a real-time operating system.

Within the hardware components of a microcontroller, several elements are susceptible to erroneous bit-flips, including memory, peripherals, and buses. Such bit-flips can lead to incorrect data, instructions, or interrupt signals being supplied to the CPU core, resulting in erroneous outputs from the application. The application of error correction codes (ECC) to safeguard memory and peripheral registers against bit-flips is a well-established practice. ECCs introduce redundant bits to stored data, enabling data correction in the event of a bit-flip. The data transmitted between hardware components via buses is also susceptible to external disturbances, warranting ongoing research into the application and improvement of ECCs to protect transmitted data.

Additionally, bit-flips can disrupt the program's execution flow and data manipulation, leading to Control Flow Errors (CFEs) or Data Flow Errors (DFEs). A Data Flow Error (DFE) occurs when a bit-flip corrupts the data being processed by the CPU. This can happen within the register bank, the arithmetic logic unit (ALU), or during data transfer through the data interface. The corrupted data then leads to incorrect calculations or manipulations.

On the other hand, a Control Flow Error (CFE) arises when a bit-flip alters the program's execution sequence. This can occur if the bit-flip affects the instruction register, modifying the instruction type or encoded values. A corrupted program counter (PC) register, which stores the address of the next instruction, can also lead to a CFE by directing the CPU to an unintended instruction

2.2 Software-Implemented Detection Techniques

Various techniques have been proposed in the literature to address transient and permanent faults in different parts of a system, targeting both hardware and software components and relying on different forms of redundancy. Among these techniques, CFC stands out as it can cover faults affecting memory components containing the executable program, as well as the hardware components handling the program and its flow [61]. CFC has been suggested to handle reliability issues for both transient and permanent faults ([62], [63], and more recently, it has been applied to address

security issues caused by the injection of malicious faults ([64], [65]). Malicious faults, within the context of fault tolerance, refer to deliberate and intentional actions taken by malicious actors to disrupt or compromise the normal functioning of a computer system, network, or software application. These actions are aimed at exploiting vulnerabilities in order to compromise the system's integrity, availability, or confidentiality [66]. Unlike transient and permanent faults, which often arise from natural hardware failures or environmental factors, malicious faults are caused by human intent and typically involve actions such as hacking, malware deployment, or unauthorized access.

In a cost-effective method proposed in [67], transient faults are detected through coarse-grain CFC, achieving efficiency by simplifying signature calculations within BBs and conducting checks at a coarse-grain level. To assess the effectiveness of this approach, a comprehensive fault injection campaign was conducted, using single bit-flips to model transient faults. Transient faults may not cause permanent damage to the hardware, but they can silently corrupt an application's correctness during runtime or even lead to system crashes. For instance, HP [68] reported frequent failures in their 2048-CPU system at the Los Alamos National Laboratory due to high-energy cosmic rays. A study [69] revealed that the BlueGene/L machine installed in Lawrence Livermore National Labs experienced soft errors approximately every four hours. Considering the estimated reliability drop per bit with each generation of processors [70], it becomes essential to provide transient fault protection schemes for both current and future systems. Transient fault detection techniques rely on different forms of redundant checking, either in hardware or software. Hardware solutions like DMR, TMR, and watchdog processors [71] are employed in systems like IBM Z-Series servers [72], HP NonStop systems [73], and Boeing 777 airplanes [74]. However, hardware-based solutions introduce unavoidable area and energy costs, making them unsuitable for commodity embedded systems. Software-based redundant checking, on the other hand, is more appealing for transient fault detection due to its lower production costs and higher flexibility. Securing control flows is crucial for transient fault protection, as CFEs are more likely to cause programs to behave incorrectly. While traditional software methods [35, 36] provide high fault coverage, they inject a significant number of validating instructions into programs, resulting in moderate to large performance overhead. Recent studies [75, 76] attempt to reduce this validation overhead by injecting fewer instructions, but they may sacrifice fault coverage due to their heuristic approaches. Software-based transient fault detection

techniques are categorized into data flow protection and control flow protection. Although data flow errors can be masked during program executions, CFEs are more challenging to hide. This work focuses on detecting illegal control flows since they can lead to incorrect program behavior. Researchers from industry and academia have been actively seeking solutions to counter the threat of transient faults in both hardware and software. Hardware-only solutions, with sufficient resources, are more efficient for a single, fixed reliability policy, while software-only solutions offer flexibility and lower costs. Software-only solutions can be deployed immediately on existing hardware by recompiling the application. However, devising correct software solutions for transient faults is a challenging task due to the numerous fault scenarios. Various techniques are suggested in the literature for detecting transient faults, falling into two general classes: hardware or software redundancy. Hardware-based methods provide better fault coverage but impose higher costs and overhead on the system, making them less suitable for some general-purpose applications. On the other hand, software-based techniques offer less fault coverage and large

2.3 Control Flow Error

There are three ways to implement data redundancy: (i) passive, (ii) active, and (iii) hybrid.

The first one is based on a voting mechanism in such that passive redundancy, i.e., obtaining/computing the data from multiple independent sources) allows isolating the error and avoids propagating wrong copy.

The second one, active redundancy, usually is founded by dividing the error handling into three phases: fault detection, isolation, and recovery (FDIR). In this approach, if an error has been detected, the faulty module is replaced with another one.

The third one, hybrid redundancy, is a combination of the two previous methods. Namely, it uses error masking to prevent the system from producing erroneous outputs by determining, thanks to voting based on FDIR mechanisms, the fault-free modules of which the output has to be propagated. The monitoring system can detect timing errors by working in two phases because of watchdogs. The watchdogs are configured in the system startup with the expected timing information. Then, at

run-time, the watchdogs are reset. The system is working correctly if the resets occur with the scheduled timings. In the other case, a time-out error is raised, triggering proper recovery or mitigation strategies.

Control Flow Checking (CFC) is chosen for this purpose among the various RHF's protection techniques available in the literature. It should be mentioned that our case study only considers permanent faults.

In the automotive industry, the use of high-level programming languages is recommended. Moreover, using CFC techniques perfectly suits the automotive industry's needs. Usually, the production-grade embedded software is developed by adopting the Model-Based Software Design (MBSD) [77]. With this approach, the software is not developed by a traditional high-level programming language (like C, C++, or ADA) but resorting a graphical representation of its functionality in the form of a physical/control model or a Finite State Machine (FSM). The adopted tools for developing these models are developed by the company MathWorks [78].

Their popular tool for describing behavior models is Simulink, while its package Stateflow is used to develop FSMs. Since these are commonly implemented software units, CFC is perfect for hardening them against RHF's.

The main idea of CFC is to verify that the program performs in the correct order. Before diving into the specific implementation adopted to develop the benchmark application mentioned in this paper, we summarize the CFC techniques.

Various CFC techniques have been proposed to detect faults that modify the execution flow. A common way to implement this approach is through signature monitoring. It does not require special hardware or operating system requirements and is based on inserting some redundant instructions into the software unit source code. Thanks to this characteristic, it is adaptable to any COTS microcontroller, including low-power consumption units. Moreover, CFC does not interfere with hardware-based hardening techniques, like watchdogs, and can be accelerated if external hardware support is available to execute run-time signatures from the instructions and compare them with the expected ones.

At the bottom of CFC, the main idea is the concept of a Control Flow Graph (CFG). CFG is a methodology to divide the program code within basic blocks (BBs).

BBs are maximal sets of ordered instructions that run sequentially from the beginning to the end. So a BB cannot contain branching instructions, such as jumps

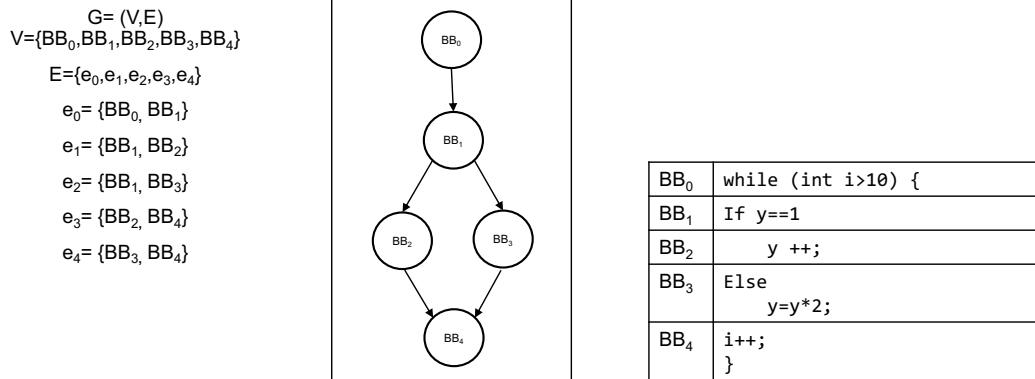


Fig. 2.1 Sample code and program CFG example. The execution from basic block BB_1 to BB_2 or from BB_1 to BB_3 are legal, but a jump from BB_1 to BB_4 is illegal and called Control Flow Error (CFE).

or calls to functions, since they change the execution flow. The only exception is the last instruction of the block, which can jump to the first instruction of another BB.

More formally, by defining an oriented graph composed of a set of vertices denoting basic blocks $V = \{v_1, v_2, \dots, v_n\}$ and the set of edges $E = \{br_{ij} | br_{ij} \text{ is a branch from } v_i \text{ to } v_j\}$ denoting the legal set of possible jumps between the basic blocks, a program can be represented by the graph, $G = \{V, E\}$. Any branch not present in E is illegal and hence denotes a Control Flow Error (CFE). Please note that the legal branches, represented as edges contained in E , are not necessarily defined by explicit branch instructions but may be implicit through execution paths, jumps, subroutine calls, and returns. A graphical representation of CFG for a sample source code developed in the C language can be found in Figure 2.1.

Software-based CFC methods use CFG alongside signatures computed by redundant instructions to detect illegal branches. The basic idea of signature-monitoring techniques is to have a static signature for each BB of a given program and a dynamic global signature. In all CFE detection methods, a unique static signature is associated with each basic block. CFC methods should be able to detect three types of CFEs :

1. CFEs due to unwanted jump of the program flow from a legal BB to an illegal BB (so a jump not present in the E set). These are called inter-block CFEs.
2. CFEs that represent an unwanted jump of the program flow from a legal BB to an unused memory space.

3. CFEs that manage an unwanted jump of the program flow from the BB to another space in the same BB by BB partitioning (e.g., the BB is split into partial-BBs even if they are not present in the CFG).

CFC methods trigger all approaches of detection action in an exact way. First, the CFG is generated by the high-level language source code, then the BB signatures and their computation methods are defined. While the hardened software component is executing, the signature values computed at run-time will be compared with the predetermined signature. Finally, an error signal that triggers the detection will be activated in case of a mismatch

2.4 In-Depth Examination of Control Flow Checking Methods

Several widely adopted techniques for Control Flow Checking (CFC) revolve around the assessment of runtime signatures in contrast to the predetermined values allocated to individual program blocks during the design or compilation stages. This approach aims to fortify software systems by ensuring the integrity and authenticity of their control flow. In this scholarly exploration, we delve into the elucidation of these techniques, offering comprehensive insights into their mechanisms, applications, and significance within the domain of software reliability. Our objective is to enhance the understanding of CFC methodologies and their role in safeguarding software systems against unauthorized or malicious control flow alterations.

2.4.1 CFCSS

In Control Flow Checking by Software Signatures (CFCSS), which is discussed in [35], instead of their sources, are evaluated at the destinations of all branches and then jump. During execution, a global variable G is initialized with the signature of the first BB of a program. When transitioning from one BB to another, CFCSS calculates the target block signature from the source block's signature by using the XOR function to determine the difference between the signatures of the source and target blocks. Control flow will be checked by comparing the computed signature with the expected one. The method described in [35] inserts control flow checking assertions

manually. This will be done by adding a few instructions at the beginning of each BB. First, check the incoming signature variable and then set its outgoing signature. This way, it is possible to guarantee that the execution flow is working accurately. It needs no dedicated hardware, such as a watchdog for CFC. It implies that CFCSS can be used even when the operating system does not support multitasking. CFCSS is not able to detect errors if multiple BBs transition, at their ends, to a common BB.

2.4.2 YACCA

"Yet Another Control-Flow Checking using Assertions" (YACCA) technique is one of the most powerful (in terms of detection capabilities) among the methods explained in [38]. This method assigns a unique signature to each BB entry and exit point. The advantage of this method is it makes it possible to detect CFEs that happened when the program flow jumped from one BB's inside to one of its legal successors, even if the successive BB gives back the control to the BB affected by the wrong jump. This is possible since the signature is re-assessed before each branch instruction to drop the wrong-successor CFE. The YACCA has fewer undetected errors and higher performance overhead compared to CFCSS.

2.4.3 ECCA

Enhanced Control Flow Checking Using Assertions (ECCA) method was proposed by [34]. The idea is to allocate a unique numerical identifier to each BB of a program. When the processor executes a new BB, particular assertions check the control flow using the involved BBs identifiers. ECCA methods, extending the CCA technique, can detect all CFEs between diverse BBs but can neither detect errors inside the same BB nor faults that cause incorrect decisions on a conditional branch.

2.4.4 RSCFC

The Relationship Signatures for Control Flow Checking (RSCFC) was proposed in paper [42]. The method encodes the control flow relations between different BBs into specially formatted signatures and then inserts CFC instructions into every BB's head and end. This technique detects inter-block CFEs with three variables:

a compile-time signature s_i , the CFG locator L_i , and the cumulative signature m_i . RSCFC has a higher fault detecting rate than CFCSS. The main drawback of this method is a higher performance overhead w.r.t. the previously described methods.

2.4.5 CEDA

The authors of [36] proposed Control-flow error detection using assertions (CEDA) by assigning a signature verification at the beginning and end of each BB, detecting the *aliasing errors* by maintaining unique signatures for each one of the *aliased* blocks. CEDA uses run-time signatures to efficiently detect faults in the control flow by inserting them during compilation. By doing so, CEDA can detect all faults that violate the program flow graph but cannot detect incorrect but legal jumps (according to the program flow graph). Therefore, CEDA cannot achieve complete fault detection.

2.4.6 ACFC

Assertions for Control Flow Checking (ACFC), mentioned in [37], is a classification design for control flow faults and the control checking method that does not depend on the predecessor-successor relationships between BBs. The technique inserts fewer instructions than previous methods. Therefore, the method has less memory overhead than the previous technique but worsening its detection performance.

2.4.7 SCFC

Software-Based Control Flow Checking (SCFC) was proposed by [39]. The technique uses two run-time variables: A variable containing the BBs' run-time values ID and a variable containing the run-time signature S. The compile-time signature is constructed as in SEDSR. A CFE can be detected at two places in the basic block; in the run-time ID or the run-time signature S that contains a wrong value. The ID should contain the compile-time value of the BB, and the S should contain a signature that indicates the predecessor BB. ID and S are updated at different places in the basic block. The S is updated in the middle of the BB after verifying it, while

the ID is updated to the compile-time id of the successor block at the end of the BB.

2.4.8 HETA

Another approach is Hybrid Error-detection Technique using Assertions (HETA) [40]. By using HETA we can detect incorrect jumps during the program execution. HETA develops CEDA techniques and associates them with hardware resources, a watchdog, for achieving complete fault detection. Using HETA methods cannot detect 100% faults in the control flow because it only detects errors that violate the CFG: an incorrect instruction that branches to a BB that is a legal successor will not be detected since it does not feature mechanisms to reveal data errors.

Software-only Error-detection Technique using Assertions (SETA) is another approach. It was proposed in [41] for recognizing CFEs in processors without hardware-implemented hardening techniques. By utilizing this method, they can reduce the computation units' costs. SETA is based on two previously described techniques: HETA and CEDA. These techniques use run-time signatures to identify errors related to the control flow. Signatures are calculated a priori and compared with the signature computed at run-time. The application code is divided into BBs. Two Basic Block Types (BBTs) are defined: A and X. Type A is the BB with multiple predecessors, and at least one of its predecessors has multiple successors. BBs without these conditions are called type X. Then, defined BBs are grouped into networks, and BBs sharing a common predecessor refer to the same network.

Every BB has two different signatures. The first is called Node Ingress Signature (NIS), compared when entering the BB. The other is called Node Exit Signature (NES), which is checked when exiting the basic block. The NIS describes the current basic blocks, and the NES is used to identify the successor network and its legal successor BBs subsequently.

2.4.9 SEDSR

Soft Error Detection using Software Redundancy (SEDSR) is an inter-block CFE detection technique proposed by Asghari et al. [79]. SEDSR assigns just one variable to each basic block at compile time, i.e. the compile-time signature s_i . It is a bit

sequence that shows the valid successor basic blocks of the current basic blocks. If there are n basic blocks in the CFG of the program, s_i is n bits wide and the bits of the successor blocks are set to 1. The run-time signature S is verified at the beginning of each basic block. The verification checks whether or not the current basic block is a valid successor of the previous basic block. In case of an error-free run, the bit on the position associated with the current basic block should be set. If that bit is zero, a CFE has occurred. The run-time signature S is updated in the middle of each basic block and assigns the compile-time signature s_i of the current basic block to S . Fig. 2.11 shows our implementation of SEDSR. The run-time signature is stored in register $r11$. We implemented the run-time signature verification with a bitwise and operation (AND), between the run-time signature and the expected set bit. Next, the result of the bitwise and operation is verified (CMP). In an error-free run, the bit at the wanted position is set, so control is transferred to the error handler located at address $0x246$ if the run-time signature is zero (BEQ). The signature update in the middle of each basic block is implemented using a move instruction (MOV), except in exit basic blocks.

2.4.10 SIED

The final SOTA technique we discuss in this chapter is Software-Implemented Error Detection (SIED) proposed by Nicolescu [80]. SIED is capable of detecting both inter-block and intra-block CFEs. At compile time, each basic block is assigned a unique identifier IDB , a list containing the compile-time signatures of all successor basic blocks and a variable n that indicates how many instructions must be executed in the basic block. The run-time signature X is updated and verified at the beginning of each basic block. The update consists of storing the result of the addition between the unique identifier of the current basic block and the status condition branch (SCB) in X . In other words, the update is the following $X = IDB_i + SCB$. The status condition branch is updated each time a conditional branch is taken and indicates whether the false or true path should have been taken. Next, the run-time signature verification compares X with a second run-time variable Y . In an error-free run, both should hold the same value, thus a mismatch indicates a CFE has occurred. Next, the intra-block updates are inserted. To detect intra-block CFEs, SIED uses the run-time variable checkpass. Once the run-time signature has been verified, checkpass is updated with the n_i variable of the current basic block. After each

original program instruction, checkpass is decremented. At the end of the basic block, a verification instruction is inserted to validate whether or not checkpass is now zero. If that is not the case, a CFE has occurred. Finally, the run-time variables SCB and Y are assigned their new values, after the intra-block verification. Y is updated with the IDBi of the successor basic block and SCB is updated to 1 if the true path of a conditional branch has to be taken, otherwise it is updated to 0. For our implementation of SIED, register r10 stores X or SCB depending on the location in the basic block, register r9 stores Y and register r11 stores checkpass. Fig. 2.13 shows that the implementation is rather straightforward. The update $X = IDBi + SCB$ is implemented with the addition instruction (ADD) and the according verification, i.e. $X == Y$, is implemented using the comparison instruction (CMP). When a mismatch is detected, control is transferred to the error handler located at address 0x28a (BNE). Next, the intra-block updates are inserted. The initialization of checkpass, i.e. $checkpass = ni$, is implemented using the simple move instruction (MOV). Then a decrement is implemented after each original original program instruction (SUB). The last thing each basic block does, is update SCB and Y. We implemented this using the MOV instruction. When a basic block ends with a conditional branch, these last two updates are executed conditionally. As with all our implementation, exit basic blocks do not perform these last updates. The verification $checkpass == 0$ is inserted at the beginning of each basic block, except for the first one. We implemented it at the start of the basic block, instead of at the end of each basic block as proposed by Nicolescu et al., to assure the correct control flow through the program. When a comparison instruction is inserted at the end of a basic block, it might overwrite the system flags needed to take the correct path of a conditional branch. To avoid this problem, we insert the verification and branch to the error handler at the beginning of each basic block.

2.4.11 RASM

In this section, we delve into the implementation process of the Run-time Assurance Signature Monitoring (RASM) technique, which involves several steps aimed at ensuring the reliable detection of Control Flow Errors (CFEs). The overarching goal of RASM is to maintain the integrity of program execution paths by utilizing a combination of gradual signature updates and signature verifications.

The implementation journey begins with a global step that assigns essential variables to all basic blocks within the program. Two random values are allocated to each basic block. The first value, known as the compile-time signature, is unique to each basic block. Simultaneously, the second value, referred to as `subRanPrevVal`, is employed in the subsequent step of the implementation process for updating the run-time signature. To ensure the uniqueness of each basic block, the sum of the compile-time signature and `subRanPrevVal` must be distinct. Consequently, `subRanPrevVal` is continually assigned until this criterion is met.

Following the assignment of random values to basic blocks, protective instructions are inserted into all basic blocks. The first instruction that a protected basic block must execute is an update to the run-time signature. This update involves subtracting the signature from the `subRanPrevVal` of the current basic block. Once executed, the run-time signature should align with the compile-time signature of the current basic block. Subsequently, a verification instruction is introduced to validate this result. Any divergence in the run-time signature from the expected compile-time signature is reported as a CFE occurrence.

The concluding phase of RASM's implementation process entails inserting the second run-time signature update at the end of each basic block. This final instruction ensures that all intentional paths within the Control Flow Graph (CFG) remain accessible during error-free runs, without triggering false positive CFE detections. To sustain the integrity of intentional paths, the signature is updated with an adjustment value. This value is computed as the disparity between the compile-time signature of the current basic block and the sum of the compile-time signature and `subRanPrevVal` of the successor block. If a basic block culminates with a conditional branch, this last update is conditionally executed to guarantee that the run-time signature retains the correct value for the corresponding successor block.

The implementation strategy diverges slightly when dealing with exit blocks. Depending on the number of instructions within such a block, either an additional verification is introduced or no further instructions are added. For exit blocks with more than one instruction, an extra verification is incorporated before the return instruction. This verification cross-checks whether the run-time signature matches the randomly chosen `subRanPrevVal`. This additional safeguard allows the detection of certain CFEs that might lead to premature program termination. In this scenario, the

adjustment value corresponds to the difference between the compile-time signature of the current basic block and `subRanPrevVal`. However, for basic blocks containing only a return instruction and no other instructions, no extra instructions are added in this step to minimize the overhead imposed by RASM.

2.4.12 RACFED

In this section, we present the implementation process for the Random Additive Control Flow Error Detection (RACFED), an extended version of RASM designed to detect intra-block Control Flow Errors (CFEs) in addition to inter-block CFEs, thereby enhancing the overall CFE detection capabilities. The implementation of RACFED involves four fundamental steps, building upon the foundation of the RASM process.

In the first step, akin to RASM, we initiate the process by assigning two random values to each basic block. These values include the compile-time signature and `subRanPrevVal`, with the latter being utilized in the subsequent step to update the signature. To ensure each basic block's uniqueness, the sum of the compile-time signature and `subRanPrevVal` is required to be distinct, and we continue assigning `subRanPrevVal` until this condition is met.

Step 2 introduces the critical component of instruction monitoring within RACFED. Instruction monitoring entails the incorporation of run-time signature updates with random values after each original instruction. This countermeasure is selectively implemented, targeting basic blocks with three or more original instructions, as they are susceptible to intra-block CFEs. Basic blocks with only one instruction are exempt from this countermeasure, as intra-block CFEs do not apply to them, and inter-block CFEs are detected through RACFED's signature monitoring instructions. For basic blocks with two instructions, intra-block CFEs cannot be effectively detected via instruction monitoring due to the absence of skipped updates.

Step 3 closely mirrors the third step of the RASM technique, involving the insertion of the first run-time signature update and the sole verification per basic block. This update consists of subtracting the signature from the `subRanPrevVal` of the current basic block, aiming to align the run-time signature with the compile-time signature. Any deviation from this alignment is indicative of a CFE, triggering detection.

Step 4 culminates in the insertion of the final signature update within each basic block, ensuring that intentional paths in the Control Flow Graph (CFG) remain accessible during error-free runs without yielding false positive CFE detections. The adjustment value for this update is computed as the difference between the run-time signature updates of the current block and the first update of the next block. The process entails determining the current run-time value, factoring in the cumulative impact of inserted intra-block updates. Additionally, it computes the expected value for each successor by summing `subRanPrevVal` and the compile-time signature of the respective successor. The adjustment value is then integrated into the run-time signature. For basic blocks concluding with a conditional branch, this update is conditionally executed to ensure the run-time signature's accuracy for the corresponding successor.

The procedure slightly varies for basic blocks ending with a return instruction, as these instructions signal the exit from the current function. Depending on the number of instructions within such blocks, an additional verification is introduced or no further instructions are added. For multi-instruction basic blocks, an extra verification precedes the return instruction, scrutinizing whether the run-time signature aligns with the randomly selected `returnVal`. This safeguard helps identify CFEs that could prematurely exit the program. The adjustment value for this scenario is computed as the difference between the run-time signature updates of the current block and `returnVal`. For basic blocks containing solely a return instruction without additional instructions, no extra instructions are introduced in this step, effectively reducing the execution time overhead of RACFED.

In summary, the RACFED implementation process comprises four comprehensive steps, building upon the principles of RASM while extending its capabilities to detect intra-block CFEs, thus fortifying control flow error detection within programs.

2.4.13 In Closing

To summarize, signature monitoring methods like, for instance, YACCA [38], CFCSS [35], CEDA [36], RASM [43], SEDSR [48], and ECCA [34], exclusively addressed illegal inter-block jumps during application execution by monitoring run-time signatures with compile-time signatures at the basic block level. The essential

difference among these techniques is how signatures are computed and checks are performed.

To improve the aforementioned methods providing covering illegal intra-block jumps, instruction monitoring techniques, such as the previously described RSCFC [42], Software implemented error detection (SIED) [44], and Random Additive Control Flow Error Detection (RACFED) [45] were developed to inspect whether instruction executed in the correct order. Moreover, in [46], a software behavior-based technique is presented to detect CFEs in multi-core architectures.

[81] has presented the Software Implemented Hardware Fault Tolerance (SIHFR) approach to CFEs online detection, which is considered an appropriate method for safety-critical applications implemented by low-cost embedded systems in which availability and execution speed are minor issues.

As a final point, it is essential to remark that there is a trade-off among the aforementioned methods regarding achieved detection rate and computed time overhead, depending on the number of additional statements inserted in the various proposals.

Table 2.1 Compare Control Flow Control techniques.

Algorithm	Used Variables	Signatures	intra-block	detection performance [%]	Code size overhead [%]	Execution time overhead[%]
ECCA	4	prime-numbers	χ	73.5	36.0	244.8
CFCSS	2	randomized-bit	χ	75.8	15.2	76.6
YACCA	2	bit-field	χ	82.8	30.0	203.2
RSCFC	2	bit-field	✓	49.4	17.5	86.8
SEDSR	3	bit-field	✓	46.8	12.3	67.1
SCFC	3	bit-field	✓	60.4	22.9	115.7
SIED	2	random numbers	✓	52.4	14	115.7
RACFED	3	random numbers	✓	N.A.	N.A.	81.5

2.5 Design Diversity Based Software Fault Tolerance

Design Diversity-Based or Multiple-Version-Based software fault tolerance involves using multiple versions or variants of software, either executed sequentially or in parallel. These versions are used as alternatives, with separate means of error detection, and can be implemented in pairs or larger groups for replication checks or masking through voting. The main idea is that components built differently should fail differently, so if one version fails on a specific input, at least one alternate version should be able to produce the correct output. This section explores various approaches

to software reliability and safety through design diversity. However, ensuring the independence of failure among multiple versions and developing effective output selection algorithms are critical challenges in deploying multi-version software fault tolerance techniques.

Design diversity serves as a means of protection against uncertainty, specifically, design faults and their associated failure modes in software design. The objective of applying design diversity techniques to software design is to build program versions that fail independently and with a low probability of coincidental failures. Achieving this objective greatly reduces or eliminates the probability of encountering incorrect outputs during program execution. However, due to the complexity of software, the application of design diversity for software fault tolerance is currently more of an art than a science.

The concept of multiple-version software design was pioneered by Algirdas Avizienis and his team at UCLA in the 1970s, primarily focusing on software. Their research also explored the application of design diversity concepts to other system aspects such as the operating system, hardware, and user interfaces. Even with rigorous development and proper application of design diversity, there is still the issue of identical input profiles leading to common errors. Experiments have shown that error manifestations are not equally distributed across the input space, and the probability of coincident errors is influenced by the chosen inputs. Data diversity techniques can potentially mitigate this issue, but quantifying their effectiveness remains a challenge.

An important consideration in using multi-version software is the cost involved. Replicating the entire development effort, including testing, would be expensive. In some cases, where only certain parts of the functionality are safety-critical, applying design diversity only to those critical parts can reduce development and production costs. [82] highlights the need to address the problem of identical input profiles as a common source of errors, highlighting that experiments have indicated unequal distribution of error manifestations across the input space. While data diversity techniques may reduce the impact of this error source, quantifying their effectiveness remains a challenge.

In summary, Design Diversity-Based or Multiple-Version-Based software fault tolerance offers a means of enhancing software reliability and safety by using multiple versions of software with independent failure properties. However, challenges

exist in ensuring independence from failure and developing suitable output selection algorithms. The concept of design diversity has evolved as an art in software fault tolerance, with applications extending beyond software to other system aspects. The issue of identical input profiles leading to common errors requires attention, and while data diversity techniques may mitigate this, quantifying their effectiveness remains a challenge. The cost of using multi-version software is an important consideration, and selectively applying design diversity to critical parts can help reduce development and production costs.

In this study, we explore various fault-tolerance approaches in software that incorporate design diversity, both with multiple versions and a single design. The approaches we focus on are as follows:

- **The Recovery Block Scheme:** The Recovery Block Scheme (RBS) combines the checkpoint and restart approach with multiple versions of a software component [83]. Before execution, checkpoints are created to allow for recovery after detecting errors. This ensures a valid operational starting point for the next version if an error is detected. Additionally, embedded checks are used to enhance error detection. The primary version executes more frequently compared to alternates, which are designed for degraded performance. Multiple versions can be executed sequentially or in parallel, depending on processing capability and desired performance. In the event that all alternates fail, the component must raise an exception to communicate its failure to the system.
- **The N-Version Programming Scheme:** The N-Version Programming Scheme (NVPS) is a multiple-version technique where all versions fulfill the same basic requirements, and the correctness of output decisions relies on comparing all outputs [84]. A voter selects the correct output, eliminating the need for an acceptance test based on the application. Developing NVPS requires considerable effort as all versions must adhere to the same conditions, resulting in complexity comparable to creating a single version. Designing the voter can be challenging and may involve inexact voting. Different voters, such as the Formalized Majority Voter, Generalized Median Voter, Formalized Plurality Voter, and Weighted Averaging Techniques, can be used, with weights based on the application and individual versions' features.

- **The N Self-Checking Programming Scheme:** The N Self-Checking Programming Scheme (NSCPS) combines various structural variations of Recovery Blocks and N-Version Programming using multiple software versions [85]. Independent development of versions and acceptance tests based on shared requirements are used in this technique. NSCPS utilizes separate acceptance tests for each version, distinguishing it from the Recovery Blocks approach. The technique benefits from using an application-independent decision algorithm for selecting the correct output.
- **The Consensus Recovery Blocks Scheme:** The Consensus Recovery Blocks Scheme (CRBS) combines N-Version Programming and Recovery Blocks to achieve higher reliability compared to either approach individually [86]. The acceptance test in Recovery Blocks techniques lacks guidance and may have design faults, whereas voters in N-Version Programming can be unsuitable in certain cases. CRBS incorporates the first layer of decision-making using a similar algorithm to that of N-Version Programming. If the first layer declares a failure, the second layer, which utilizes acceptance tests similar to Recovery Blocks, is invoked. Although more complex than the individual techniques, CRBS has the potential to deliver a more reliable result.
- **The $t/(n-1)$ -Variant Programming Scheme:** The $t/(n-1)$ -Variant Programming Scheme (VPS) involves n variants and the $t/(n-1)$ diagnosability measure to restrict faulty units to a subset of size at most $(n-1)$, assuming a maximum of t faulty units. This approach differs from the previous methods in terms of the method used to isolate faulty units [87].

In summary, the utilization of Design Diversity-Based or Multiple-Version-Based software fault tolerance techniques offers promising avenues to enhance software reliability and safety. These approaches leverage multiple versions of software, designed to fail independently, thereby reducing the likelihood of encountering erroneous outputs during program execution. However, the practical implementation of design diversity in software fault tolerance remains more of an art than a science due to the complexity of software and the challenges in ensuring independence from failure. Additionally, addressing the issue of identical input profiles leading to common errors and quantifying the effectiveness of data diversity techniques remain significant challenges. Furthermore, the cost implications of employing multi-version software must be carefully considered, and selective application of design diversity

to critical components can help mitigate development and production expenses. The various fault-tolerance approaches explored, such as the Recovery Block Scheme, N-Version Programming Scheme, N Self-Checking Programming Scheme, Consensus Recovery Blocks Scheme, and $t/(n-1)$ -Variant Programming Scheme, provide diverse strategies to implement design diversity effectively in software fault tolerance.

2.6 Single-Design Software Fault Tolerance Approach

Single-design fault tolerance is a method that involves introducing redundancy to a single version of the software in order to detect and recover from faults. In the context of single-version software fault tolerance techniques, various factors need to be considered, including program structure, error detection, exception handling, checkpoint and restart, process pairs, and data diversity.

In terms of software engineering aspects, the use of modularizing techniques is crucial for implementing fault tolerance effectively. Modular decomposition should include built-in protections to prevent abnormal behavior from propagating to other modules. Control hierarchy issues, such as visibility and connectivity, should also be taken into account to minimize the risk of uncontrolled corruption of the system state. Partitioning can provide isolation between functionally independent modules, leading to simplified testing, easier maintenance, and lower propagation of side effects. System closure, which states that no action is allowed unless explicitly authorized, is another important principle of fault tolerance. Atomic actions, which are activities in which components exclusively interact with each other without any interaction with the rest of the system, offer error confinement and recovery capabilities. If an atomic action terminates normally, its results are complete and committed. If a failure occurs during an atomic action, it only affects the participating components [88].

To ensure the effective application of fault tolerance techniques in single version systems, structural modules should possess two basic properties: self-protection and self-checking. Self-protection means that a component can detect errors in the information passed to it by other interacting components. Self-checking means that a component can detect internal errors and take appropriate actions to prevent error propagation. The extent to which error detection mechanisms are used in a design depends on the cost of additional redundancy and the run time overhead. It's

important to note that fault tolerance redundancy is not intended to contribute to system functionality but rather to the quality of the product. Similarly, detection mechanisms can affect system performance. The utilization of fault tolerance in a design involves trade-offs between functionality, performance, complexity, and safety.

Assertions, which are logical statements inserted at different points in a program reflecting relationships between program variables, can also be used for fault tolerance. However, their effectiveness depends on the nature of the application and the programmer's ability. CFC involves partitioning the application program into basic blocks (BBs) and computing deterministic signatures for each block. Faults can be detected by comparing the run time signature with a precomputed one.

Replication checks involve matching components with error detection based on the comparison of their outputs, making them suitable for multi-version software fault tolerance. Timing checks are applicable to systems and modules with timing constraints and can look for deviations from acceptable module behavior. Watchdog timers, a type of timing check, can be used to monitor system behavior and detect "lost or locked out" components. Reversal checks use the output of a module to compute the corresponding inputs and detect errors if the computed inputs do not match the actual inputs. Coding checks utilize redundancy in the representation of information and check relationships between actual and redundant information before and after operations. Reasonableness checks rely on semantic properties of data, such as range, rate of change, and sequence, to detect errors. Data structural checks involve inspecting known properties of data structures, such as number of elements, links, and pointers. Augmenting data structures with redundant structural data can enhance the effectiveness of structural checks. Runtime checks are standard error detection mechanisms in hardware systems and can be used as fault detection tools [87]. Fault trees, top-down graphical representations of failures and triggering conditions can aid in the development of fault detection methods by identifying failure classes and triggering conditions.

Exception handling involves interrupting normal operations to handle abnormal responses. Exceptions are signaled by error detection mechanisms, and the design of exception handlers requires consideration of possible triggering events, their effects on the system, and appropriate recovery actions [87].

Checkpoint and restart is a common recovery method for single-design software. Most software faults that occur after development are unanticipated, state-dependent faults. Restarting a module is usually sufficient to complete its execution successfully. Restart recovery can be static or dynamic. Static restart returns the module to a pre-determined state, while dynamic restart uses dynamically created checkpoints [87].

Process pairs utilize two identical versions of software running on separate processors. The recovery method is a checkpoint and restart. The primary processor actively processes input and creates output while generating checkpoint information for the backup processor. Upon error detection, the secondary processor loads the last checkpoint and takes over the primary processor's role. The faulty processor goes offline for diagnostic checks. This technique ensures uninterrupted delivery of services after a failure [87].

Data diversity is an effective defense method against design faults, especially when combined with checkpoint and restart methods. By implementing "input sequence workarounds" and using different input re-expressions on each retry, data diversity enhances the success rate of checkpoint and restart procedures. The desired outcome of each retry is to generate output results that are either exactly the same or semantically equivalent, although the definition of equivalence may vary depending on the application. In [89], three fundamental data diversity models are presented: (i) Input Data Re-Expression, which focuses on modifying the input; (ii) Input Re-Expression with Post-Execution Adjustment, which involves processing the output to achieve the desired value or format; and (iii) Re-Expression via Decomposition and Recombination, where the input is broken down into smaller elements and then recombined after processing to obtain the desired output. It is worth noting that data diversity works hand in hand with the Process Pairs technique, allowing for different re-expressions of the input in the primary and secondary.

In the context of operating systems, software fault tolerance is crucial to ensure the proper functioning of any application-level software. While designing and building operating systems can be complex, time-consuming, and costly, it may be necessary to develop custom operating systems with highly structured design processes involving experienced programmers and advanced verification techniques for safety-critical applications. Another approach to achieving fault tolerance in operating systems for mission-critical applications is to use wrappers on off-the-shelf operating systems to enhance their robustness against faults. However, utilizing

off-the-shelf software on dependable systems poses the challenge of ensuring the reliability of the components for the intended application. It is known that the development process for commercial off-the-shelf software lacks consideration for safety or mission-critical standards, resulting in weak documentation for design and validation activities. On the other hand, commercial operating systems offer advantages such as incorporating the latest developments in operating system technology and potentially having fewer bugs overall due to continuous bug-fixing efforts driven by user complaints. In order to minimize the risk of introducing design faults, it is preferable to adopt techniques that utilize the operating system as is, without internal modifications. Wrappers serve as middleware between the operating system and application software, monitoring the flow of information to prevent undesirable values from propagating. By limiting the input and output spaces of a component, wrappers provide application-transparent fault tolerance functionality. In [90], wrappers referred to as "sentries" encapsulate operating system services and can modify the characteristics of these services as perceived by the application layer. Through wrappers, fault-tolerance methods can be dynamically assigned to specific applications based on their individual needs in terms of fault tolerance, cost, and performance. Authors proposed using wrappers at the micro-kernel level for off-the-shelf operating systems, aiming to verify semantic consistency constraints using abstractions or models of the expected component functionality.

In conclusion, Software based fault tolerance methods offer several advantages, including the absence of additional auxiliary devices, no specific operating system requirements, good expansibility, and support for continuous exploration and repeated experiments. However, these methods come with significant time and space overhead due to the inclusion of numerous redundant instructions, which can significantly impact program performance.

2.7 Hardware-Based Fault Tolerance Techniques

Hardware-based techniques have two main groups: *(i)* redundancy-based and *(ii)* hardware monitors. The first group relies on hardware or time redundancy. In contrast, the second group adds special hardware modules to the system's architecture to monitor the control flow of the programs inside the processors and memory accesses performed by them, such as watchdog processors [31], checkers [32], or infras-

structure intellectual properties (I-IP) [33]. Hardware-based techniques have a high cost, verification and testing time, and area overhead, leading to a higher power consumption.

2.7.1 Redundancy in Hardware-Based Fault Tolerance Techniques

Hardware redundancy is the most common technique, which is the addition of extra hardware components for detecting or tolerating faults [29, 91]. For example, instead of using a single core/processor, more cores/processors can be exploited so that each application is executed on each core/processor; then, the fault can be detected or even corrected. Hardware redundancy can be applied through (i) passive, (ii) active, and (iii) hybrid methods.

(i) Passive Redundancy

Examples of this redundancy are N Modular Redundancy (NMR), such as Triple Modular Redundancy (TMR), and using voting techniques. These techniques are referred to as M -of- N systems, which means that the system consists of N components, and the correct operation of this system is achieved when at least M components correctly work. The TMR system is a 2-of-3 system with $M = 2$ and $N = 3$, which is realized by three components performing the same action, and the result is voted on [29, 91].

(ii) Active Redundancy

This type of active hardware redundancy includes duplication with comparison (DWC), standby-sparing (SS), a pair-and-a-spare technique, and watchdog timers. In DWC, two identical hardware components perform the exact computation in parallel, and their output is compared. Therefore, the DWC technique can only detect faults but cannot tolerate them because the faulty component cannot be determined [29, 91]. In standby-sparing, one module is operational, and one or more modules are standby or spares. If the fault is detected in the main component, it will be omitted from the operation, and the spare component will continue the execution [29, 91]. Meanwhile, pair-and-a-spare is a combination of DWC and SS techniques. For instance, two modules are executed in parallel, and their results are compared to detect the fault [29, 91].

(iii) Hybrid Redundancy

The basic concept of this method is integrating the main features of both active and passive hardware redundancies. N modular redundancy with spare, sift-out modular redundancy, self-purging redundancy, and triple duplex architecture are examples of hybrid hardware redundancy [29, 91]. The basic concept of self-purging is based on NMR with spare techniques. All modules are active and participate in the function of the system. In sift-out modular redundancy, there are N identical modules. However, they are configured in the system through special circuits (comparators, detectors, and collectors). The triple duplex architecture combines the DWC technique with TMR, which helps to detect the faulty module and remove it from the system.

2.8 Hybrid methods

Hybrid fault-tolerance methods typically involve the integration of a Software Implemented Hardware Fault Tolerance (SIHFT) method with a hardware module designed to perform consistency checks within the processor. In a study by [92], SIHFT techniques are combined with a Control Flow Checking (CFC) module, which is responsible for monitoring the trace port of the processor. Another hybrid approach, proposed by [40], is known as Hybrid Error-detection Technique using Assertions (HETA). This method utilizes a watchdog module and assertions (or signatures) to address control-flow errors.

Lockstep is another hybrid fault-tolerance technique that utilizes both software and hardware redundancy for error detection and correction ([93], [94], [95], [96]). Lockstep involves executing the same application simultaneously and symmetrically in two identical processors. These processors are initialized to the same state and receive identical inputs during system start-up. During normal operation, the state of both processors should be identical at each clock cycle. By monitoring the processor's data, addressing, and controlling buses ([97]), a checker module periodically compares the outputs of the processors to check for inconsistencies. To enforce verification, specific points are inserted in the program to indicate when the application execution should be locked and the outputs compared. If any discrepancies are found, the lockstep system leverages a rollback method to restore the processors

to a safe state. In the absence of errors, a checkpoint operation is performed, which stores the context of the processor (including registers and main memory) in a secure memory location. Memories can be protected using Error Correction Code (ECC) to prevent data corruption. ECC is capable of detecting and correcting single-bit errors and detecting double-bit errors. To recover from errors, the fault-free copy of the processor's context is retrieved from memory using the rollback method. The processor is then recovered to a state without errors and restarts the application execution from this point.

In summary, hybrid fault-tolerance methods combine software and hardware approaches to enhance error detection and correction. One approach integrates Software Implemented Hardware Fault Tolerance (SIHFT) with Control Flow Checking (CFC) or Hybrid Error-detection Technique using Assertions (HETA) to monitor and address control-flow errors. Another hybrid method, known as Lockstep, executes applications in parallel on identical processors, comparing outputs and employing rollback and checkpoint mechanisms to ensure system reliability and error recovery. These hybrid approaches provide robust fault tolerance in critical systems.

Table 2.2 provides an overview of the classification of hardware-based, software-based, and hybrid-based techniques.

2.9 Using the C language in automotive industry applications

C programming language is extensively utilized in the automotive industry due to its flexibility, support, and portability, making it suitable for high-speed, low-level input/output operations and complex applications that require high efficiency.

However, programming errors are relatively easy to make, and the language lacks proper support for error detection, posing a potential danger to safety-critical systems. Consequently, several constraints, such as the MISRA C guidelines [98], have been developed, limiting the use of problematic language features. In addition, various tools and techniques like static analysis tools, code reviews, and unit testings are available to enhance the security of C codes. Nonetheless, C language's weaknesses, like incomplete type checking, lack of exception handling mechanisms, and limited

Table 2.2 Overview of the techniques classification. [4]

Technique Classification	Pros	Cons
Hardware	<ul style="list-style-type: none"> -High fault detection -Fast detection -No software modification 	<ul style="list-style-type: none"> -Most does not correct errors -Mainly single fault model -High area and power overhead -Implemented only in physical level -Can be expensive
Software	<ul style="list-style-type: none"> -High fault detection -High flexibility -No hardware modification -Small area overhead -Some can correct errors 	<ul style="list-style-type: none"> -High performance overhead -Mainly single fault model -Focuses only on data or control flow, but not both
Hybrid	<ul style="list-style-type: none"> -High fault detection -High efficiency -Can achieve small area overhead -Some can detect both SDC and SEFI -Some can correct errors 	<ul style="list-style-type: none"> -Can also achieve high performance or area overhead -Software and hardware modification

run-time error checking, increase the need to incorporate SIHFT to the code written in C to guarantee safety even after eliminating programming defects.

2.10 Functional Safety in the Automotive Industry

In safety-critical embedded systems, ensuring adequate safety levels represents the primary factor in the success of these systems. ISO 26262 is an international standard for the functional safety of electrical and electronic systems titled “Road vehicles—Functional safety.” It was released in 2011, and the current edition is the second one, released in 2018 [24]. ISO 26262 was derived from the generic functional safety standard IEC 61508 to address the specific needs of electrical and electronic systems, and focuses on malfunctioning behaviors.

The standard is divided into eleven parts, covering all activities during the safety life cycle of safety-related systems, including electrical, electronic, and software elements that provide safety-related functions. The process prescribed in ISO 26262 uses a top-down approach in which, first, hazard analysis is conducted to identify potential hazards and system-level requirements. The most important parts related to our paper are the third (concept phase), fifth (development at the hardware level), and sixth (development at the software level).

The third is the “concept phase”, where the item is defined. From the definition, it is possible to perform the hazard analysis and risk assessment needed to define the risk level associated with its functionality (automotive safety integrated level, ASIL), the safety goals (SGs) to be achieved, and its functional safety concept (FSC).

Based on the obtained SGs and their ASILs, actions that prevent the presence of systematic failures or mitigate random hardware failures (RHF) have to be taken in phases five and six. The fifth one is about product development at the hardware level. An essential result of this phase is the list of the possible failure modes (FMs) that can affect the designed item and, in particular, its computation unit. A detailed description of the application of the safety life cycle to semiconductors is given in part 11 [24].

The design group develops embedded software in parallel to the hardware design. It shall be developed by following part six of the standard to avoid the presence of defects (also known as bugs) in the code (prevention against systematic errors). The possibility of unavoidable RHF shall be taken into account, hence the need to implement hardening techniques such as CFC.

2.11 ISO26262-compliant classification

The ISO26262 standard, tailored for automotive applications, was initially released in 2011 and subsequently updated in its second edition in 2018 [24]. This standard mandates the implementation of detection and mitigation systems capable of responding effectively to Random Hardware Failures (RHF). A critical aspect of evaluating a design, outlined in Part 5 of the standard, titled "Product Development at the Hardware Level," involves conducting Failure Mode, Effects, and Diagnostic Analysis (FMEDA). Within this analysis, a key focus lies in ascertaining the de-

tectability of RHF's and evaluating the efficiency of adopted mitigation strategies. For this reason, we concentrate solely on the detection mechanisms.

It is worth noting that failure modes associated with semiconductor components are comprehensively described in Part 11 of the standard, titled "Guidelines on the Application of ISO 26262 to Semiconductors," which was added in the second update of the standard.

The central premise of this thesis is to assess, in accordance with ISO26262 guidelines, the effectiveness of CFC (Code Fault Coverage) algorithms implemented in the C programming language for applications developed using Model-Based System Development (MBSD). To achieve this objective, the process involves the injection of faults followed by a classification of the outcomes.

The classification hinges on describing an application's behavior through a comparison of outputs with a fault-free execution (referred to as the "golden run"), as well as analyzing the flow of the Program Counter (PC) register after a fault has been introduced. Seven distinct outcomes have been defined:

- "Latent after injection": A fault is injected, and the behavior remains identical to the fault-free run.
- "Erratic behavior": The behavior deviates from the fault-free run.
- "Infinite loop": The PC enters an infinite loop not present in the original program flow, resulting from the interaction between the source code and the faulty PC register.
- "Stuck at some instruction": The PC remains fixed, pointing to a valid instruction. This occurs when the injected fault impedes the PC from incrementing its value, especially when the 3rd bit is involved.
- "(Detected) by SW hardening": Detection occurs through the CFC.
- "(Detected) by HW (mechanism)": The PC points outside the FLASH/RAM addressing space or triggers other hardware traps.
- "As golden": Detection with an output identical to the golden run. This classification differs from "Latent after injection" in that it signifies detection of a fault that has no impact on the application's output. Furthermore, four additional outcomes are provided: "Latent," "Error," "Undefined," and

- "False Positives." These outcomes do not directly relate to the application itself but serve to monitor the classifier, with "False Positives" indicating flaws in the CFC implementation.

The classification process is vital in determining Diagnostic Coverage (DC), as defined by ISO26262, which categorizes detection as "detected" if an embedded mechanism identifies the presence of the considered RHF, or "undetected" if not. In the "detected" category, two subclasses are possible: "safe" when the RHF poses no significant danger to the user or the environment, and simply "detected" when making such an assumption is not feasible (as is the case in this paper, where mitigation strategies are not considered).

Conversely, within the "undetected" class, two subclasses can be defined: "latent" when the RHF has no effect on the item's behavior and "residual." Additionally, a third subclass, "false positive," not formally defined by ISO26262, is introduced to describe instances where the detection mechanism is erroneously triggered. It's important to note that in an effective detection mechanism, the frequency of the "false positive" subclass should ideally be 0%.

The ISO26262-compliant classifications are computed using the following formulas, taking into account:

- N : The total number of injections.
- L : The number of "latent after injection" outcomes.
- D_{HW} : The number of simulations where a hardware mechanism has detected the RHF.
- D_{SW} : The number of simulations where the RHF has been detected by the CFC.
- U : The experiments in which the application entered an "infinite loop," remained "stuck at some instruction," or exhibited an "erratic behavior." The formulas are as follows:

$$\text{Safe} = \frac{\text{As golden}}{N}$$

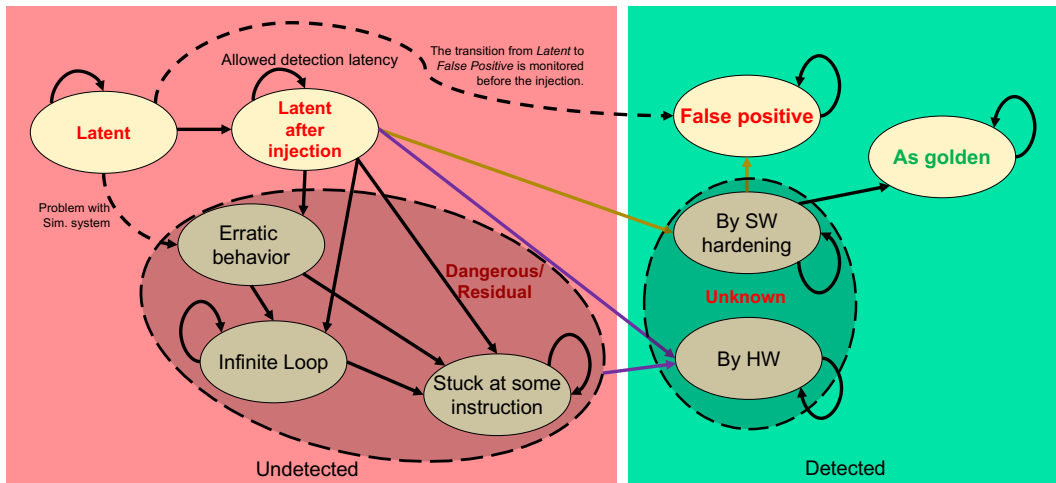


Fig. 2.2 The classifier FSM. The transition from *Latent after injection* or any of the *Dangerous/Residual* group to a state of the *Detected* side is allowed only before the FTTI elapses. The transition from the state (*detected*) by software hardening to the state *As golden* is performed, when one last line of the log file has been read, only if the behavior of the software components remains the same of the golden run for the entire log file.

$$\text{Detected} = \frac{D_{HW} + D_{SW}}{N}$$

$$\text{Latent} = \frac{L}{N}$$

$$\text{Residual} = \frac{U}{N}$$

$$\text{False positive} = \frac{\text{false positive}}{N}$$

It is important to consider the RHF as "detected," adhering to the concept of the Fault Tolerance Time Interval (FTTI) outlined in the Standard, only if detection occurs within a specific number of machine instructions. For all simulations in this paper, this value has been set to 200 assembly instructions, as it allows for the execution of multiple Basic Blocks (BB).

2.12 A Note on Control-flow Integrity Techniques for Soft Errors-security

Control-flow integrity (CFI) techniques are employed to ensure that a program functions as intended without being affected by soft errors, which can arise from external factors like radiation, power surges, or electromagnetic disturbances. These errors have the potential to cause unintended consequences, such as data loss, diminished system reliability, and even security breaches. [99] The primary purpose of CFI techniques is to mitigate the risk of security breaches resulting from soft error-induced deviations by implementing a set of rules on the program's control-flow graph (CFG). This graph represents the program's control flow and the relationships between its various components. These rules dictate the permissible execution paths and prevent any unauthorized or malicious alterations to the control flow. One commonly used CFI technique is "strict control-flow integrity" (SCFI), which enforces rules to maintain the integrity of the program's control-flow graph during execution. Any attempt to deviate from this graph is detected and prevented, thus safeguarding the program's integrity. Additional CFI techniques include "shadow-stack-based CFI," "implicit CFI," and "hybrid CFI," each with its own specific rules and requirements. Soft errors can affect the direct as well as indirect branches and hence CFI, as is, is not directly applicable for soft errors. Though direct branches can also be protected in a manner similar to dynamic branches, but the already high overhead (20%-60% for dynamic branches only) would become prohibitive [100].

Synergy with CFC Methods:

CFI restricts execution paths: CFI constrains the program's execution to authorized paths within the CFG, mitigating the risk of control flow alterations by malicious actors. This reduction in the attack surface enhances system security against control flow deviations.

CFC methods detect deviations: CFC methods actively monitor the program's execution flow during runtime. In the event of a deviation, whether intentional or due to soft errors, CFC can promptly detect it and trigger appropriate recovery mechanisms, such as error logging or attempting to restore to a known good state.

Complementary Approaches:

CFI and CFC methods complement each other in ensuring control flow integrity. While CFI focuses on preventing unauthorized deviations, CFC diligently detects and reacts to any deviations that may occur. This collaborative approach strengthens the overall control flow security and reliability of the program, offering robust protection against both intentional attacks and inadvertent soft errors.

In summary, CFI techniques serve as a set of measures to protect software systems from security breaches caused by soft errors. By enforcing strict rules on the program's control-flow graph, these techniques can identify and thwart any unauthorized or malicious changes to the program's execution, thereby bolstering its security and reliability.

2.12.1 Data integrity

Data integrity refers to the concept of ensuring that data remains accurate, consistent, and reliable throughout its entire life cycle. In the context of soft error security, data integrity becomes especially crucial in protecting against potential vulnerabilities and risks posed by transient faults or soft errors. These errors can be caused by various factors, such as cosmic radiation, electrical noise, or electromagnetic interference, and can adversely impact the integrity of stored data. To mitigate such risks, data integrity measures involve implementing error detection and correction techniques, such as checksums and parity bits, to detect and correct any errors that may occur.

Maintaining data integrity is relatively straightforward in a standalone system with a single database. This is achieved through the use of database constraints and transactions, typically managed by a database management system (DBMS). Transactions should adhere to the ACID principles (atomicity, consistency, isolation, and durability) to ensure data integrity. Most databases support ACID transactions, which aids in preserving data integrity. However, data integrity in cloud-based systems refers to the preservation of data accuracy. It is crucial to ensure that data remains unchanged and is not lost due to unauthorized user actions. Data integrity forms the foundation for cloud computing services like Software as a service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) [101]. In addition to storing large volumes of data, cloud environments typically offer data processing services. methods such as RAID-like methods and digital signatures can be employed to maintain data integrity in cloud systems.

Remote verification of data integrity in the cloud is a prerequisite for deploying applications. Bowers et al. introduced the "Proofs of Retrievability" theoretical framework, which combines error correction codes and spot-checking to facilitate remote data integrity checks [102]. The High-Availability and Integrity Layer (HAIL) system utilizes the Proofs of Retrievability (POR) method to verify data storage across different clouds, ensuring redundancy of copies and enabling availability and integrity checks [103]. Schiffman et al. proposed the use of Trusted Platform Modules (TPM) for remote data integrity checks [104].

Due to numerous entities and access points in a cloud environment, authorization plays a vital role in ensuring that only authorized entities interact with data. By preventing unauthorized access, organizations can have greater confidence in data integrity. Monitoring mechanisms provide increased visibility, enabling the identification of any alterations made to data or system information that may affect its integrity. While cloud computing providers are entrusted with maintaining data integrity and accuracy, it is important to establish a third-party supervision method alongside users and cloud service providers.

In summary, data integrity is paramount for safeguarding data accuracy and consistency, particularly in the context of transient faults or soft errors. Techniques like checksums and parity bits are used to detect and correct errors. While standalone systems can ensure data integrity through database constraints and ACID transactions, cloud-based systems require remote verification methods, such as Proofs of Retrievability and Trusted Platform Modules, to maintain data accuracy across various access points. Authorization and monitoring mechanisms also play crucial roles in preserving data integrity in the cloud, requiring collaboration among users, providers, and third-party oversight for effective security.

Chapter 3

Experiment Prerequisites

This chapter introduces a fault model, as CFCs are capable of detecting only those fault models (FMs) that directly or indirectly modify the instruction flow. Our focus was specifically on fault models affecting the Program Counter (PC). We then applied a software-based hardening technique to address these fault models. To investigate the effectiveness of this approach, we manually implemented two CFC techniques within the C listing of two benchmark applications. These benchmarks were generated automatically using Simulink Coder, based on the Model-Based System Development (MBSD) methodology. This allowed us to compare the process of implementing CFC methods in high-level programming with the conventional approach of implementing CFC techniques in assembly language. However, it's important to note that the latter approach can be significantly more time-consuming and prone to errors.

3.1 Fault models

In the context of this study, since CFCs can detect only fault models (FMs) directly or indirectly modify the instructions flow, we considered only those affecting the Program Counter (PC). We chose to inject faults into the PC register since it directly affects the instructions flow. Considering the scope of CFCs, we know without any need for simulation results that failures affecting data or making the program follow a wrong but legal (present in the CFG) path are not detected. For example, choosing

a wrong path on conditional assertion (e.g., if-else) due to corruption on the variable to which the condition will be applied cannot be detected.

The Fault Injection Manager (FIM) described in [105] mentioned above features two fault models: (i) "Permanent" and (ii) "PermanentStuckAt". "Permanent" affects only one bit of the target register. It remains, from the injection time on, fixed to 0 or 1. The "PermanentStuckAt" affects the entire register globally, making it stuck to a fixed value.

For the simulation campaigns, we decided to use only the "Permanent" fault representing, coherently with the definition commonly found in literature, a condition when a bit inside the affected register remains permanently stuck at 0 or 1 from the moment of injection till the end of the simulation. Injection time, the affected bit, and its state are randomly chosen. To increase injection efficiency, it is possible to choose a subset of bits that can be affected thanks to a bitmask, allowing to avoid injection on higher positions, causing the PC immediately to move outside the "text segment size," triggering hardware failure detection mechanisms. The positions that can be affected by the fault are indicated by the bits set to 1 of a bit field called `bitPosMaks`. This mask will be clearly indicated for each campaign in the simulation results.

3.2 Implemented Software-Based Hardening Technique

Usually, CFC techniques are implemented in assembly language, which can be time-consuming and error-prone. Moreover, international standards like ISO26262 prescribe that code should be written in a structured manner using high-level programming languages, with minimal exceptions. Therefore, we decided to harden the code by implementing CFC methods in the C programming language.

The two CFC techniques were manually implemented in C and directly on the Model-Based System Development (MBSD) listing of the two benchmarks. These benchmarks were generated using Simulink Coder, which is based on the MBSD approach.

Simulink also provides the capability for code generation starting from models through the use of Embedded Coder. This process facilitates the generation of C, C++, optimized MEX functions, and HDL code. Simulink includes the Target Language Compiler tool (TLC), allowing developers to customize the generated code to suit the target platform based on any model. TLC is used for converting the model into C code.

For CFC methods, maintaining the instruction order is crucial to allow the correct signature update. If the update is based on arithmetic computations, the compiler might merge all the partial sums to obtain the correct numerical results before the correctness verification. These optimizations can affect the detection capabilities of the CFC method. To investigate this possibility, we designed experiments with each of the four optimization levels (00, 01, 02, and 03) offered by GCC for RISC-V, implementing CFC methods manually in the C programming language.

The first benchmark is a Finite State Machine (FSM) that implements a timeline scheduler (TS). A timeline scheduler is responsible for executing periodic tasks triggered by a timer interrupt. These tasks are executed in a fixed order defined by the system designer. In our benchmark, we have 15 tasks that are scheduled to run in a fixed order, each allocated a 200 ms time slot.

The second benchmark is a software-based controller designed to maintain the liquid level in a tank at a desired height using an on-off logic (T). It monitors the liquid level within the tank and the current absorbed by the pumps. Based on this data, it decides when to activate the pump and generates an alarm if over-current is detected. In such cases, it shuts down the pump to prevent motor damage. This controller is implemented as an FSM and includes decision logic and an independent monitoring checker to ensure the physical plant executes the commands correctly.

We selected these two benchmarks as they are representative of applications similar to those in the automotive industry, which cannot be used due to intellectual property protection reasons. The first benchmark, referred to as "TS," is essential for implementing any operating system that handles periodic tasks. The latter, referred to as "T," is commonly found in industrial applications and is analogous to algorithms that control battery charge levels in electric and hybrid vehicles within the automotive domain.

Figure 3.1 and Figure 3.2 illustrates the model-based development process, indicating when the CFC methods were applied in the high-level programming language, as per our proposed approach.

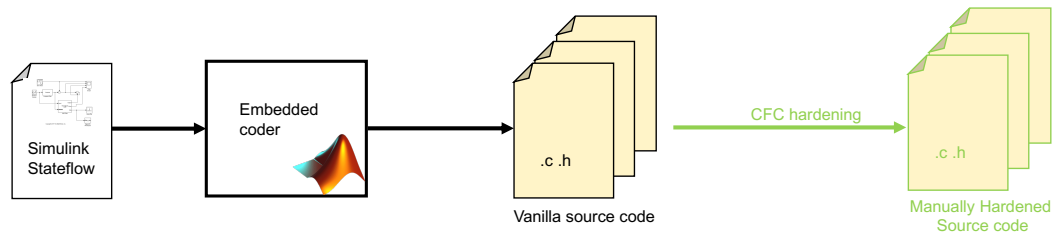


Fig. 3.1 Indication, inside the model-based flow, indicating when the CFC is applied using high-level programming languages. The source code is obtained automatically via the Mathworks Embedded Coder from a Simulink semi-formal model, then the obtained source code is manually hardened.

We chose YACCA and RACFED due to their different underlying philosophies (bit mask vs. random numbers, inter-block vs. intra-block detection capabilities). YACCA was selected for its simplicity of implementation, while RACFED was chosen because it represents one of the most recent approaches in this context.

3.2.1 YACCA

We adopted the methods described in [81] and [38], where YACCA was proposed as a software-implemented RHF detection mechanism suitable for safety-critical applications.

The program has been divided, into BBs. A vertex in the CFG represents one BB. Each one of the vertices are associated two different random numbers (signatures) embedded into the C code at compile-time. The first signatures represent the ID of the BB, while the second one is the mask of its predecessors. Since signatures have been assigned to each BB at compile-time, it is possible to compute them independently at run-time and then compare the latter with the assigned one. In this case, the algorithm makes use of two variables: `ERR_CODE` and ID_s . A unique ID corresponding to a power of 2 (to have only 1-bit, assigns 1 in binary representation) is assigned to each BB. At the program start-up, `ERR_CODE` is set to equal 0, and ID_s is equal to the ID of the first BB that will be executed. When the program enters a BB, it checks if the content of ID_s is equal to the ID of the current BB. If this

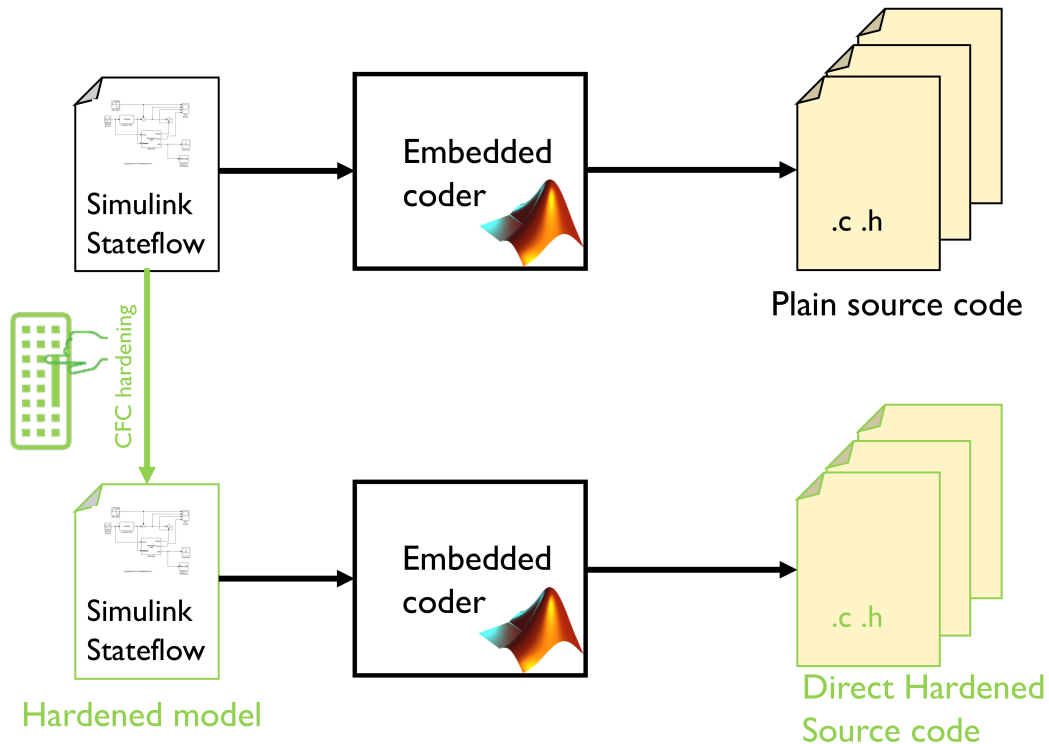


Fig. 3.2 Model-based approach for implementing the CFC methods. The benchmarks are obtained by hardened them in the model, then generating the source code automatically thanks to the Embedded Coder.

condition is not verified, the `ERR_CODE` variable is increased by 1. When the BB ends, before jumping to the next BB, it resets the ID_s content by performing an AND operation between ID_s itself and a mask corresponding to the bit-wise NOT of its ID, then OR it with the ID of its legal successor. If the program flow is correct, the first comparison is verified, so `ERR_CODE` remains 0. Then ID_s is set to all zeros by the AND operation (if ID_s is the correct one, the bit-wise NOT of the current state ID is also the bit-wise NOT of ID_s , hence $ID_s \text{ NOT } ID = 0$) and can be set to the ID of its legal successor by the OR operation. In the case a CFE happens, the AND followed by the OR operations sets two different bits to 1, so none of the BBs can successfully pass the comparison between ID_s and its ID, causing `ERR_CODE` to increase.

YACCA Functions or macros needed in C language:

YACCA uses the following TEST and SET operations to check and update the control flow signatures:

Algorithm 1 TEST operation (YACCA)

```

1: TEST(RTS, predecessors_mask)
2: if RTS  $\wedge$  ( $\neg$  predecessors_mask) then
|   CFE detected
|   end
3: Continue normal execution

```

Algorithm 2 Set operation (YACCA)

```

1: SET(RTS, predecessors_mask, BB_ID)
2: RTS = RTS  $\wedge$   $\neg$  predecessors_mask
3: RTS = RTS  $\vee$  ( $1 \ll \text{BB\_ID}$ )

```

The TEST function checks if the runtime signature (RTS) matches the expected signature based on the predecessors' mask. A mismatch indicates a control flow error. The SET operation updates the RTS to reflect the execution of a basic block (BB), identified by BB_ID.

3.2.2 RACFED

Another technique we opted to implement in our benchmark is RACFED [45]. RACFED was developed based on Random Additive Signature Monitoring (RASM) technique [43] to detect both inter-block and intra-block CFEs. RASM is a signature monitoring technique that uses two gradual signature updates and one signature verification per BB. Using gradual updates means that all updates on a specific, intentional path are linked together, acting as one update. Skipping one gradual update implies that the run-time signature can never hold the correct value again. Of course, compiler optimization can also affect these gradual signature updates, making it act as a single update in the compiled application. However, RACFED extends this functionality by inserting gradual signature updates after each instruction inside the run time signature (RTS) variable.

Below its implementation steps will be discussed in detail.

1. Firstly, for each BB there are two signatures needed at compile time `compile time signature (CTS)` and `subRanPrevVal`. The CTS is a random number defining the expected signature value. In the case that there are more than two payload instructions inside the considered BB, a random number is

assigned for each payload instruction. `subRanPrevVal` is the sum of all the chosen compile time random numbers previously assigned to the payload instructions. It should be noticed that `subRanPrevVal` is equal to zero if BB has less than two instructions.

2. Consider now the execution of a BB after the signature check (see Figure 3.3 for an example). After each payload instruction is executed, **run time signature (RTS)** is increased by the random value assigned in the previous step to each payload instruction. This process allows for the detection of intra-block CFEs.
3. Next, at the end of the considered BB (all payload instructions have been executed), an **adjustment value** is computed as the sum of its CTS and the sum of all the random numbers assigned to its payload instructions (numerically equal to `subRanPrevVal` but independently computed at run time), then by subtracting the CTS and `subRanPrevVal` of its successor BB. At the end of considered BB, the RTS is increased by the yet computed **adjustment value** (starting the two-phases RTS update).
4. Finally, RTS is updated at the beginning of the successor BB, (concluding the two-phases RTS update) by subtracting the `subRanPrevVal` of its predecessor. At this point RTS shall equal CTS. If not, CFEs happened, otherwise CTS equals RTS and the process repeats from step 2.

RACFED Functions or macros needed in C language:

RACFED uses random numbers to update and check the RTS, providing more granular intra-block detection:

Algorithm 3 TEST operation (RACFED). *bb* represents the ID of the BB which is calling TEST(). RTS is an array containing the compile-time signature of every BB.

```

1: TEST(bb)
2: if RTS  $\neq$  CTS[bb] then
|   CFE detected
|   end
3: Continue normal execution

```

Here, CTS is an array containing compile-time signatures for each BB.

To simplify the implementation of the algorithm, considering that each state of our FSMs contain a branch instruction, and that the branches have a different number of C code statements (in this specific case, 4 and 1, respectively), we decided to choose random numbers such that the sum is the same regardless the chosen branch. The reader can find the branch instruction at line 162 of Figure 3.3, where it is possible to see that the sums in both the execution paths are the same: the sum obtained by executing the statements at lines 165, 169, and 173 (respectively adding 25, 35, and 67 to the signature) is the same as executing the statement at line 177, which adds $25+35+67=127$.

Figure 3.3 illustrates the mapping between signature update instructions written in C and their corresponding assembly instructions (RISC-V RV32I). The left side of the figure shows the C code, where runtime signature updates are performed. The right side displays the assembly translation generated by the GCC compiler with optimization level O0. The arrows highlight the correspondence between specific instructions in C and their assembly counterparts, demonstrating that the compiler preserves the order of instructions as intended. This ensures the integrity of the CFC techniques implemented in the C code.

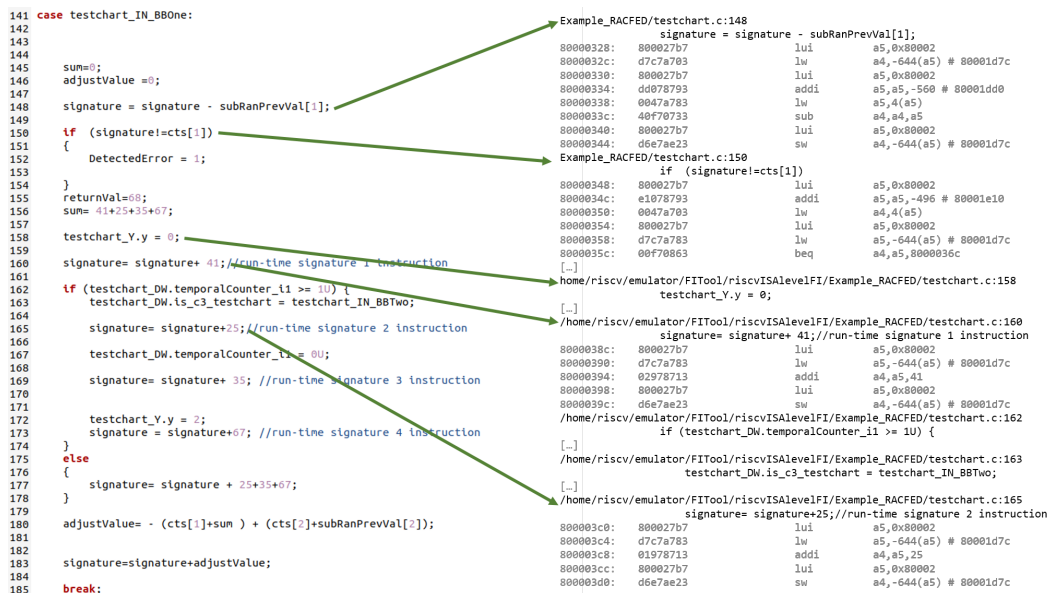


Fig. 3.3 Mapping between the signature updates instructions in C and the relative Assembly (RISC-V RV32I) translation. It is possible to see that GCC, configured with O0 optimization settings, keeps the instructions in order.

Appendix A presents a set of guidelines for implementing CFC methods in our case study YACCA and RACFED using the C programming language.

3.2.3 Experimentation with Compiler Optimizations

To verify whether the hardened code is correctly translated into assembly, we first compiled the code of a BB hardened with RACFED with no optimizations and investigated the obtained assembly code. This step ensured that the compiler did not alter the order of instructions. The results of these experiments are discussed in chapter 4.

Additionally, we compiled the code of a BB hardened with RACFED with optimizations to verify if the compiler impacts RACFED's effectiveness by filtering out its instructions. The results of these experiments are also discussed in chapter 4.

Chapter 4

Experimental Study on CFC Detection Techniques

This chapter presents the simulation results obtained and provides an overview of the selected RISC-V environment. The benchmark application runs on a RISC-V-based target platform, enabling the assessment of the performance of the hardening techniques. Additionally, it delves into the results of fault injection, diagnostic coverage, and overhead analysis for both CFC methods. These analyses consider different optimization levels for manual hardening in the C programming language and no optimization levels for the Model-Based Software Development (MBSD) approach.

4.1 Target platform

The target platform on which we run the benchmark application is based on RISC-V. It is a free and open Instruction Set Architecture (ISA) introduced by the University of California, Berkeley [106]. RISC-V is based on Reduced Instruction Set Computing (RISC) theory to decrease hardware implementation costs, improve performance, and simplify instruction specifications. Developers can take advantage of RISC-V to modify the architecture to suit specific applications or to remain open to applications made by programmers unaware of the underlying hardware. It allows developers to combine the advantages of both worlds [107], providing flexibility to both hardware

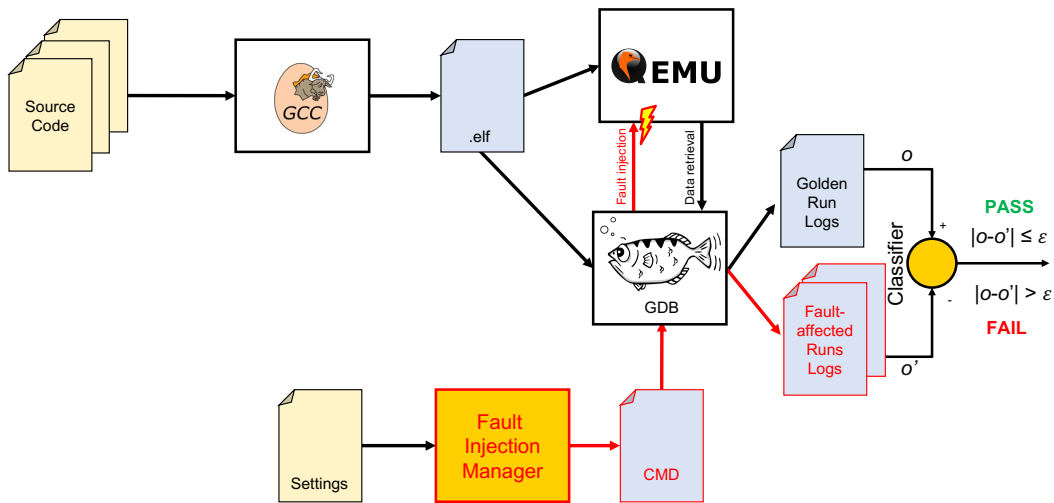


Fig. 4.1 The proposed test bench architecture. GCC compiles also the classifier, whose source code is generated by the FIM.

and software. The benchmark applications considered in this thesis were compiled using the GNU RISC-V Toolchain [108].

As the target platform, as described in [105], we choose RISC-V RV32I, simulated at the ISA level thanks to the QEMU (Quick Emulator) [109].

QEMU is an open-source machine emulator and virtualizer written by Fabrice Bellard. Most parts are licensed under GNU General Public License (GPL), others under different GPL-compatible licenses. The main reason QEMU was used in our proposal is to make the test bench agnostic to ISAs, allowing for application on different architectures.

The GNU Debugger (GDB) [110] is used to interact with QEMU. The fault injection is managed by a Fault Injection Manager that writes the GDB scripts needed to inject the faults and log the simulation results. The classifier uses these results to assess the fault injection results.

4.2 Hardening technique performance assessment

As described in section chapter 3, the source codes were generated directly from the Simulink StateFlow chart via the Embedded Coder and then were hardened manually.

To verify whether the hardened code is correctly translated in Assembly, the code of a BB hardened with RACFED was compiled with no optimizations (of course, with the O0 optimization flag of GCC). In Figure 3.3, it is possible to verify that the compiler with the aforementioned settings did not change the order of the instructions.

When the benchmarks are compiled with optimization flags enabled, the execution time overhead decreases. Still, as expected, this reduction in execution time overhead is counterbalanced by a reduction in error detection.

In addition to the compilation process, compiler optimizations can also affect RACFED. To verify if the compiler impacts RACFED effectiveness by filtering out its instructions, the code of a BB hardened with RACFED was compiled with optimizations. We found that the compiler filters out the mechanism's operation and makes it less effective, as better described in the following sections.

Before starting with the fault injections, a golden execution was performed for each campaign. A golden execution runs when the target system is simulated without injecting any faults. It is needed to obtain a log file representing the benchmark applications' nominal behaviors and gather information on the target system and the simulator's state. Moreover, it is a way to guarantee no false positive detections.

4.3 Fault injection results

In the initial phase, we need to articulate the simulation results in terms of "detected," signifying whether an embedded mechanism can discern the presence of the considered Random Hardware Fault (RHF), or "undetected" if not.

Within the "detected" category, we can further categorize it into two subgroups:

- "Safe," where it can be reasonably assumed that the RHF does not pose significant risks to the user or the surrounding environment.
- "Detected," for situations where making such an assumption is not feasible. This is particularly relevant in our context, as we do not consider mitigation strategies in this paper.

Similarly, in the "undetected" category, we establish two subcategories:

- "Latent," when the RHF does not affect the item's behavior.
- "Residual," encompassing situations where the RHF does impact the item's behavior.

To enhance the clarity of our analysis, we introduce a third subclass, "false positive," which is not formally defined by ISO 26262. This subclass helps quantify the likelihood of the detection mechanism erroneously triggering. Notably, in the case of a robust detection mechanism, the occurrence of this subclass should be minimal, ideally 0

The outcomes expressed in these ISO 26262-compliant terms are tabulated in Table 4.5. This table sets the stage for the subsequent step, which involves computing the Diagnostic Coverage (DC).

Within our definition of "detected," we include:

- All faults that resulted in detection by the software.
- Those faults that led to a timeout following the last SET instruction, mimicking the behavior of a windowed watchdog.
- The fault detected by the hardware, where the Program Counter (PC) points beyond the boundaries of the instruction memory.

These delineations are essential for a comprehensive evaluation of the diagnostic coverage.

4.4 C programming language Fault injection results

We conducted 13 campaigns, each one of 1000 injections of "Permanent" faults affecting the Program Counter (PC) of the target. There needs to be more than this number of injections to provide statistical results, but the CFC methods have already been proven effective in the literature. The purpose of this work is not to assess their effectiveness again but to provide data about the Diagnostic Coverage to application developers in a realistic scenario, taking into account also the effect of the optimization introduced by the compilers.

The 13 campaigns are organized as follows:

- 7 campaigns with 00 optimizations
 - 5 have been performed on the timeline scheduler (TS) benchmark;
 - 2 on the Tank level controller (T).
- Moreover, other 6 campaigns have been conducted on the TS benchmark:
 - 3 for the TS hardened with YACCA;
 - 3 for the TS hardened with RACFED.

Each campaign has been performed by compiling the application with the remaining 3 optimization levels (01, 02, and 03), obtaining all the possible combinations.

In all the campaigns, the injected "Permanent" faults are of the stuck-at type, described as one of the bits composing the registers remaining stuck at 0 or 1 from the moment of the injection up to the end of the simulation. When injecting stuck-at faults on the PC, the expectation of an unwanted instruction only happens if the stuck-at fault changes the PC value. Since the PC is 32-bit long, injecting a stuck-at fault on its most significant bits will lead to a considerable jump in the instruction memory. It is noteworthy that all the stuck-at faults were injected in random positions of the PC bits at random times.

The same faults have been injected on both YACCA and RACFED. Considering the campaigns with 01, 02, and 03 optimization levels, we injected faults only on TS since the T benchmark seemed unsuitable to be hardened with CFC (very low detection rate).

The results obtained from the classifier for YACCA and RACFED are available in Table 4.1. In both tables, columns show the benchmarks on cumulative results which the fault injection campaign was conducted for different random fault injection masks. TS stands for the timeline scheduler benchmark, and T stands for the tank level controller benchmark. In each row, the number of occurrences of 5 different outcomes is reported.

Analyzing the experimental results where the CFC is not able to detect the failures (rows for "infinite loop" or "Stuck at some instructions"), it is possible to observe a known limitation of the selected CFC methods. Since the application and the CFC code are executed on the same computation unit, no detection is possible

Table 4.1 Cumulative classifier results obtained from the 7 fault injection campaigns evaluating the YACCA and RACFED methods without compiler optimizations, manually implemented directly in C code, on benchmarks. The "As Golden", "False Positive", "Undefined", and "Error" results are all zero for all columns, so they are not reported in the table.

Classification results	YACCA		RACFED	
	TS Benchmark	T benchmark	TS benchmark	T benchmark
Latent after injection	226	883	230	945
Erratic behavior	0	29	0	0
Infinite loop or Stuck at some instruction	408	20	1066	0
(Detected) by SW hardening + Safe	253	66	255	52
(Detected) by HW mechanism	1063	2	1449	3

if the error prevents the CFC test instructions from executing. ISO26262 indicates these cases as "not free from interference" since the same cause can affect both the benchmark and the CFC code.

In Table Table 4.1, there is no "Infinite loop" or "Stuck at some instruction" outcomes for the T benchmark with hardening with RACFED. This observation, alongside the very high rate of "Latent after the injection" outcomes for the TS benchmark, shows how much the effectiveness of the hardening method is application-dependent. Suppose no decisions (and hence transitions between BBs) are performed in the time window between the injection and the end of the simulation. In that case, the injected faults remain latent due to the impossibility of executing the CFC's test instructions. In the case of sufficient spare execution time, a possible proposal to solve this issue can be to add a dummy control flow to check if the computation unit is working correctly. This solution can be adopted when an online test is unavailable for the target platform, or the application should not depend on any specific platform for commercial or intellectual property protection reasons.

The detailed experimental results for the campaigns are reported in Table Table 4.1 for the experiments without almost all compiler optimizations (O0), and in Table Table 4.2 and Table Table 4.3 for the three levels of optimization.

4.4.1 Diagnostic coverage

The results shown in Section section 4.3 were transposed into ISO 26262-compliant classifications, which requires computing the Diagnostic Coverage (DC) of the pro-

Table 4.2 Classifier results obtained from the fault injection campaign assessing the YACCA implemented manually directly within the C code on TS benchmark with different compiler optimizations. "As golden", "False positive", "Undefined", and "Error" outcomes are all zero for all the columns, so they are not reported in the table.

Classification results	O0	O1	O2	O3
Latent after injection	110	393	433	521
Erratic behavior	0	0	0	0
Infinite loop or Stuck at some instruction	266	0	0	0
(Detected) by SW hardening + Safe	112+0	266 + 0	118+0	132+0
(Detected) by HW mechanism	512	341	449	347

Table 4.3 Classifier results obtained from the fault injection campaign assessing the RACFED implemented manually directly within the C code on TS benchmark with different compiler optimizations. "As golden", "False positive", "Undefined", and "Error" outcomes are all zero for all the columns, so they are not reported in the table.

Classification results	O0	O1	O2	O3
Latent after injection	86	229	181	199
Erratic behavior	0	0	0	0
Infinite loop or Stuck at some instruction	385	0	266	0
(Detected) by SW hardening + Safe	93+0	183 + 0	129+7	156+0
(Detected) by HW mechanism	436	558	673	561

posed CFC methods. The obtained results are presented in Table 4.5 and Table 4.5. It is important to remark that the "Detected" column is calculated by taking into account both hardware and software-detected failures; hence, in the following analysis, this sum is considered. More specifically, the "Detected" column in Table 4.5 is the sum of the last two rows of Table Table 4.1 for YACCA and RACFED methods.

We can observe no "safe" detected failures for the TS benchmark, while the "safe" detected failures are predominant for the T benchmark. The state in the Time scheduler (TS) benchmark FSM is changed continuously. Since the FSM is implemented by `switch-case` structures, every time the state is updated, the BBs are changed accordingly. On the other hand, the states of the Tank level Controller (T) benchmark FSM are changed only in reaction to input changes. However, considering

the tank level inertia with respect to its controller update time (10 milliseconds), the controller usually keeps the current state, avoiding BB changes.

Table 4.5 considers the codes compiled with no optimizations (O0). Starting with YACCA, its DC for TS benchmark is 67.49% and for T benchmark is 2.80%. It is important to note that YACCA does not feature intra-block detection mechanisms, so skipping only one instruction results in a high probability of remaining unnoticed.

As shown in Table Table 4.4, hardening the TS benchmark and T benchmark with RACFED, its DC respectively 56.80% and 0.3%. This observation is due to the fact that RACFED features intra-block detection mechanisms.

Considering the "Undetected failures" in Table 4.4, there are 11.0% and 88.30% of "latent" undetected failures for the hardening TS benchmark and T benchmark hardening with YACCA, respectively. The "latent" undetected failures for hardening benchmarks with RACFED are 8.60% for TS benchmark and 94.50% for the T benchmark. This outcome can be explained considering that the fault injection can affect a higher significant bit. If the affected bit is stuck at a value equal to the expected one, the PC is the same as expected. Hence the fault does not affect the code execution. For the T benchmark, the number of "latent" undetected failures is greater in comparison to TS benchmark since it changes states less frequently compared to the TS benchmark, so a situation where its output remains stuck can remain unnoticed for a longer time.

The "Residual" undetected failures for the TS benchmark hardening with YACCA for TS benchmark is 20.92% and for T benchmark is 4.90% and using RACFED and from 38.5% and 0.0% for TS and T benchmarks. These failures are not detectable by the CFC itself but can be detected as timeout errors thanks to external hardware components like watchdogs. Considering the T benchmark, this case is rare: 4.9% occurrences for YACCA and 0.0% for RACFED.

In conclusion, considering Tables Table 4.1 as observed in the rows titled "Detected by SW hardening + safe," the CFC methods can increase the system DC by an average of 15,65% in a realistic scenario. These results may not appear impressive. However, considering that these SIHFT methods should be employed alongside other hardening methods like watchdogs and memory error detection and correction, they can be useful to reach a high diagnostic rate (usually $\geq 99\%$) required by the Standard.

Table 4.2 and Table 4.3 show results with different compiler optimizations obtained on the TS benchmark, Table 4.5 the obtained DCs, and finally Table 4.6 the corresponding overheads.

Starting from the results in Table 4.2, the diagnostic coverage (SW only) for YACCA is the best at **01**, while the DC is similar for **00** and **02**, with **02** being slightly better than **00**. Overall with all three different compiler optimizations, the DC is improved compared to the case where no compiler optimizations were used (**00**). This can be explained, considering that YACCA does not feature intra-block detection. Hence the code will be shorter in each optimization, and the probability that the failure inside the PC triggers a detectable CFE increases. Moreover, the transition between BBs is due to transitions inside the application algorithm, so the optimization cannot strongly affect them.

A similar story can be seen in Table Table 4.3 for RACFED, which also features intra-block detection capability. From **00** to **01**, the detections by software increased from 93 to 183. The same pattern, even if less evident, is observed from **02** (136) to **03** (156). To explain this phenomenon, we analyzed the generated Assembly code. Between **00** and **01**, the intra-block signature update instructions are almost kept in the correct order, but the occupied program memory is shrunken (causing the CFC test function calls to be closest to each other) of about 20%. Similar to YACCA, this leads to an increase in the probability that the failure causes a detectable CFE. Repeating the analysis for **02** and **03**, we observed that the intra-block updates are merged, completely losing the intra-block detection offered by RACFED. But again, shrunken occupied program memory section increases the probability of a detectable CFE between **02** and **03**.

In Table Table 4.5 the DCs for different compiler optimizations are reported. We can observe that, for YACCA, the DC decreases as the optimization levels increase (shortened code counterbalances **Detected** with the **Undetected** ones). While for RACFED, the results are less intuitive. From **00** to **01**, the DC increases by about 24% as the occupied program memory is shrunken by about 20%. The DC for **02** and **03** remains approximately the same as the **01** optimizations.

For RACFED, all compiler optimizations result in zero "residual undetected" failures. This means that all undetected failures are "latent." Hence, no wrong program execution is expected due to the fact that the failure is either detected or is a latent undetected that does not change the behavior of the program. However,

for YACCA, all compiler optimizations result in zero "residual undetected" failures except the 02 optimization.

It is essential to highlight that, under the hypothesis of a multi-core system or external hardware (like companion chips or dedicated custom peripherals for CFC), two HW mechanisms can aid the detection of these failures: (i) a trap raised when the injected fault results in the PC pointing outside the program memory to a non-valid memory address. (ii) the injected fault results in the PC pointing to a valid but undesirable memory address. This can be explained by considering the differences in the program memory size for the two benchmarks. The occupied program memory is composed of 9012 instructions for the T benchmark and only 1736 instructions for the TS benchmark. Since the number of instructions for the T benchmark is almost five times the number of instructions for the TS benchmark, and the probability of finding jump/branch instructions increases when the number of instructions increases, the likelihood of the occurrence of case (ii) increases for the T benchmark compared to the TS benchmark. For the same reason, the probability of case (i) decreases for the T benchmark.

In conclusion, YACCA and RACFED, two different CFC methods, react similarly to the compiler optimizations: for both, the number of "detected" by only software is better with 01 optimization, then 02 and 00 are similar, while 03 performs better compared to 02. The "Residual Undetected" failures, with 00 optimization, are relatively high (26.6% for YACCA, 38.50% for RACFED). Then, the "Residual Undetected" failures for all compiler optimizations are zero, except for the 02 optimization with YACCA, which is equal to the case of no compiler optimization. The "latent undetected" failure for YACCA increases with the optimization level, with a huge step between 00 and 01, explainable as the occupied program memory is reduced to half of its size, leading to freeing one bit of the PC to be used for representing a valid instruction address. At the same time, for RACFED, we have 8.6% for 00 and about 20% for the optimized versions.

4.4.2 Overheads

There are two types of overheads considered in this work: (i) the increases in Text Segment Size (TSS), which shows the increase in the size of the occupied program memory due to the CFC instructions added to the program instructions

Table 4.4 ISO 26262-compliant classification of the cumulative results obtained from the fault injection campaigns on the benchmarks compiled with almost no optimization (O0).

CFC method	Benchmark	Detected		Undetected		False Pos.
		<i>Safe</i>	<i>Detected</i>	<i>Latent</i>	<i>Residual</i>	
YACCA	TS	0.00%	67.49%	11.59%	20.92%	0.00%
YACCA	T	4.00%	2.80%	88.30%	4.90%	0.00%
RACFED	TS	0.00%	56.80%	7.67%	35.53%	0.00%
RACFED	T	5.2%	0.3%	94.50%	0.00%	0.00%

Table 4.5 ISO 26262-compliant classification of the results obtained from the fault injection campaigns on the TS benchmark compiled with different compiler optimization levels. The results obtained with almost no optimizations (O0) are also reported for ease of reading.

CFC method	Compiler Optimization	Detected		Undetected		False Pos.
		<i>Safe</i>	<i>Detected</i>	<i>Latent</i>	<i>Residual</i>	
YACCA	O0	0.00%	62.40%	11.00%	26.60%	0.00%
YACCA	O1	0.00%	60.70%	39.30%	0.00%	0.00%
YACCA	O2	0.00%	56.70%	43.30%	26.60%	0.00%
YACCA	O3	0.00%	47.90%	52.10%	0.00%	0.00%
RACFED	O0	0.00%	52.90%	8.60%	38.50%	0.00%
RACFED	O1	0.00%	77.10%	22.90%	0.00%	0.00%
RACFED	O2	0.71%	81.01%	18.28%	0.00%	0.00%
RACFED	O3	0.00%	78.28%	21.72%	0.00%	0.00%

after compiling the hardened program. This leads to requiring more space in the flash memory of the embedded system. (ii) Execution time overhead, measured, given the ISA-level simulation adopted to run the campaigns, as the extra number of machine instructions (# exec. instr.) it takes for the hardened program to execute. The overhead has been computed with respect to the non-optimized version without the hardening.

Considering both overheads is essential for embedded applications. The concerns are the code size for applications running on low-cost micro-controllers with a minimal amount of embedded flash memory and the number of executed instructions (as a figure of the execution time) for real-time applications.

Table Table 4.6 reports the overhead on the program memory represented as "TSS" and the overhead on executed instructions represented as "# exec. instr." The

Table 4.6 Data regarding memory occupation and executed instruction. T = Tank Level, TS = Timeline Scheduler, and TSS = Text Segment Size. Vanilla refers to the application that is not hardened from its original form. For TS, are reported the overheads with the different optimization levels. All the differences are computed in comparison to the Vanilla version compiled with almost no optimizations (O0).

CFC method	Benchmark	Compiler Optimization	TSS Overhead	# exec. instr. Overhead
Vanilla	T	O0	9012	42593
YACCA	T	O0	10512 (+16.6%)	44668 (+4.9%)
RACFED	T	O0	10966 (+21.7%)	43864 (+3.0%)
Vanilla	TS	O0	1736	3991
YACCA	TS	O0	2496 (+43.8%)	16689 (+318.17%)
YACCA	TS	O1	1620 (-6.68%)	8761 (+119.52%)
YACCA	TS	O2	1480 (-14.75%)	8363 (+109.55%)
YACCA	TS	O3	1236 (-28.80%)	7605 (+90.55%)
RACFED	TS	O0	6271 (261.23%)	5770 (+44.58%)
RACFED	TS	O1	5404 (+211.29%)	4680 (+17.26%)
RACFED	TS	O2	3980 (+129.26%)	3972 (-0.48%)
RACFED	TS	O3	3484 (+100.69%)	3789 (-5.06%)

overhead on the number of executed instructions is obtained from the increase in the ISA-level simulator counts when running the simulation of the fault-injected program compared to the simulation of the fault-free program.

For both benchmarks, YACCA imposes less TSS overhead compared to RACFED. The difference between TSS overheads imposed by these two CFC methods is insignificant for the T benchmark, while it is much more significant for the TS benchmark.

The YACCA method is implemented in the TS benchmark by inserting the CFC instructions at each BB entry and exit point. While in the T benchmark, the YACCA method is implemented by calling functions for the set and test operations. For the TS benchmark, the TSS overheads are more significant than the TSS overheads in the T benchmark. This is due to the strategy to harden the code without using function calls. In RACFED implementation, there are many duplicated instructions (similar to the inserting strategy adopted for YACCA implementation). On the contrary, in the T benchmark, in which we chose to use functions, the TSS overheads are around 20%.

The discussion is more complicated regarding the number of executed instructions overhead, considering the compiler optimization's importance. For both bench-

marks, YACCA imposes a greater overhead in terms of executed instructions than RACFED; hence RACFED outperforms YACCA in this comparison. For the T benchmark, the difference between the executed instructions overhead of by the two CFC methods is negligible, while this difference is more noticeable for the TS benchmark. This can be explained considering that almost every BB of the TS benchmark has more than two instructions, while the T benchmark has fewer. This is important since, for those BBs containing more than two instructions, RACFED sums to the signature a random number after each instruction. At the same time, YACCA does not perform any different operations. Again, considering the other optimization levels, we can observe that with optimization 02 and 03, the number of executed instructions is less than that of the vanilla version without optimization. Still, it is important to remark that, in these two latter cases, the intra-block detection is lost.

For YACCA, with any compiler optimization level, the number of executed instructions is greater than the number of executed instructions in the Vanilla 00. However, the executed instructions overhead decreases drastically from 00 to 01, while this overhead decreases slowly while changing the optimization level from 01 to 02 and from 02 to 03. For RACFED, with 01 compiler optimization level, the number of executed instructions is greater than the number of executed instructions in the Vanilla 00. On the contrary, with 02 and 03 optimization levels, the executed instructions become negative. The executed instructions overhead decreases drastically from 00 to 01, and also from 01 to 02. While this overhead decreases slowly while changing the optimization level from 02 to 03. In conclusion, for both benchmarks, YACCA imposes less TSS overheads while RACFED imposes less executed instructions overheads.

4.5 Model-Based Software Design Fault injection results

This section shows the obtained simulation results, describes MBSD approach, and provides how the performances of the hardening techniques have been assessed. Finally, it covers the fault injection results, diagnostic coverage, and overheads for both CFC methods when compiled with without optimization levels.

4.5.1 Diagnostic Coverage

The results obtained from the classifier for YACCA and RACFED implemented on the timeline scheduler and tank level are available in Table 4.7.

Table 4.7 classifier results obtained from the fault injection campaign assessing the YACCA and RACFED CFC methods implemented on the timeline schedule and tank level.

Table 4.7 Classifier results obtained from fault injection campaigns evaluating the YACCA and RACFED methods without compiler optimizations, hardened in MBSD, on benchmarks. The "As Golden", "False Positive", "Undefined", and "Error" results are all zero for all columns, so they are not reported in the table.

Classification results	YACCA		RACFED	
	TS Benchmark	T benchmark	TS benchmark	T benchmark
Latent after injection	110	791	113	771
Erratic behavior	0	0	0	0
Infinite loop or Stuck at some instruction	261	0	167	0
(Detected) by SW hardening + Safe	112+0	0+13	305+0	1+34
(Detected) by HW mechanism	512	2	395	0

Table 4.8 ISO 26262-compliant classification of the cumulative results obtained from the fault injection campaigns on the benchmarks compiled with almost no optimization (O0).

CFC method	Benchmark	Detected		Undetected		False Pos.
		<i>Safe</i>	<i>Detected</i>	<i>Latent</i>	<i>Residual</i>	
YACCA	TS	0.00%	51.80%	9.10%	39.10%	0.00%
YACCA	T	1.61%	0.25%	98.14%	0.00%	0.00%
RACFED	TS	0.00%	70.00%	13.30%	16.70%	0.00%
RACFED	T	4.22%	0.12%	95.66%	0.00%	0.00%

For YACCA, we obtained a DC of 51.8% for the timeline scheduler (TS) and 1.86% for the tank level (T).

For RACFED, we obtained a DC of 70.0% for the TS and 4.34% for T.

Regarding the CFC algorithms that we opted for, we can say that RACFED is more effective than YACCA. This is an expected result since it also provides, alongside intra-block detection not exploited in our benchmark, a two-phase signature update.

For both cases, it is evident how the DC lowers for the T benchmark compared with the one obtained on the TS due to both algorithms' different natures.

The TS, due to its scheduler nature, performs a state transition every time its `step()` function (the function that is called at a fixed rate to make it behave as a periodic task) is called, whereas this is not true for the T, since its transitions depend on the level of the liquid inside a slow-changing physical system (a tank), leading to a relatively lower number of transitions.

Analyzing the experiments where the CFC is not able to detect the injected failure (infinite loop or stuck at some instructions results in Table 4.8), it is possible to observe a known limitation of those algorithms experimentally. Since the application and the CFC are executed on the same computation unit, no detections are possible if the error prevents the TEST instructions from executing. In the ISO26262, these cases are indicated as not "free from interferences", since the same root cause can affect both. This is an important point when designers choose to adopt such solutions, since CFC by itself is incapable of detecting 100% of faults: this highlights the need to combine this type of algorithm with other techniques, such as watchdog or the execution of tests on other cores if they are available.

The last observation is that there are no "Infinite loop" or "Stuck at some instruction" outcomes for the T benchmark. This, alongside the very high rate of "Latent after the injection" with respect to the TS results, shows how the nature of the application to be hardened is essential in a real use case: if no change in decision is needed in the time window between the injection and the end of the simulation, the behavior remains accidentally correct.

A possible proposal for such applications, in the case of sufficient spare execution time, can be to add a dummy control flow to check if the computation unit is working correctly. This solution can be adopted when an online test is unavailable for the target platform, or the application should not depend on any specific platform for commercial or intellectual property protection reasons.

4.5.2 Overhead

Table 4.9 presents the data on the overhead of the hardening techniques considering two different aspects: the increase in the program size (evaluated in terms of the

increase in its text segment) and the number of actually executed machine-code instructions (a reliable metric to estimate its effect on its execution time).

It is possible to see that the explanation on the lower DC for the T benchmarks is confirmed considering the overhead in terms of the executed instructions number: for both the hardening techniques, we have a stronger impact. This result is expected: the more the signature is tested, the greater the probability that it detects CFEs. Considering the T benchmark, the impact of the CFCs in terms of executed instructions is almost negligible (a small number of BBs transitions happen).

The huge difference in terms of text segment sizes between the two benchmarks can be explained in terms of the number of BBs present: the functions related to the hardening techniques are implemented with the *inline* option of Embedded Coder, so a greater number of BBs requires, in a proportional way, more C language statements and then more machine code ones.

Regarding the main memory occupation, our implementation of the YACCA algorithm requires three unsigned 64-bit variables for a total of 24 bytes for each execution flow. The TS has only one execution flow, whereas T has two execution flows (one for the decision based on the tank level, the other to implement the overcurrent protection) with an occupation of 48 bytes. Considering RACFED, it needs four 64-bit variables for the execution flow. Hence, the overhead is 32 bytes for TS and 64 bytes for T.

Table 4.9 Data regarding memory occupation and executed instruction. T = Tank Level, TS = Timeline Scheduler, and TSS = Text Segment Size. Vanilla refers to the application that is not hardened from its original form. For TS, are reported the overheads with the different optimization levels. All the differences are computed in comparison to the Vanilla version compiled with almost no optimizations (O0).

CFC method	Benchmark	Compiler Optimization	TSS Overhead	# exec. instr. Overhead
Vanilla	T	O0	9012	33460
YACCA	T	O0	110432 (+15.7%)	33498 (+0.1%)
RACFED	T	O0	12804 (+42.0%)	33534 (0.2%)
Vanilla	TS	O0	1736	3991
YACCA	TS	O0	6056 (+249%)	10771 (+170%)
RACFED	TS	O0	17320 (+322%)	7492 (+87.7%)

Chapter 5

Conclusion

5.1 Summary

The pivotal outcomes and contributions of the research, as outlined in this thesis, are synthesized within this section. Furthermore, we delve into potential avenues for future research and advancements in the context of addressing random hardware failures (RHF) in embedded systems, emphasizing the importance of fault testing and mitigation in GPU architectures.

5.2 Main contributions

In this thesis, we conducted an extensive exploration of various fault-tolerance methods tailored to mitigate random hardware failures (RHF) in embedded systems, with a particular emphasis on real-time embedded systems. The growing prevalence of embedded systems in safety-critical and mission-critical applications underscores the need for robust fault tolerance techniques, ensuring seamless automation and operational efficiency across commercial and industrial sectors.

We began by discussing the fundamental importance of fault tolerance in contemporary computing systems, elucidating how these techniques bolster system dependability by concealing faults and identifying errors, thereby enabling uninterrupted service delivery even in the presence of internal failures. We highlighted the

distinctions between hardware and software redundancy as mechanisms for achieving fault tolerance, guaranteeing the reliability of system operations.

Special consideration was given to software fault tolerance, acknowledging that software faults are a leading cause of system failures. While software engineering endeavors to eliminate most deterministic design faults, the inherent complexity of software design makes it virtually impossible to ensure their complete absence. Consequently, software fault tolerance techniques serve as an additional layer of protection, ensuring the continued provision of services at an acceptable level of performance and safety.

Furthermore, we delved into the growing challenge posed by soft errors or transient bit-errors in modern computing systems, emphasizing the critical role of fault tolerance in mitigating these errors to prevent system malfunctions.

Our exploration encompassed a wide array of fault tolerance techniques, including hardware, software, and hybrid redundancy, providing valuable insights into their advantages and suitability across different scenarios. Additionally, we addressed fault-tolerance strategies tailored specifically for resource-constrained embedded systems, acknowledging the importance of accounting for limited memory and low-end computational environments in such systems.

Nonetheless, it is imperative to highlight that the realm of real-time embedded systems demands further research and development in fault-tolerance methods to ensure the reliable and resilient operation of interconnected computing systems. In conclusion, this thesis offers valuable insights into fault mitigation techniques and underscores the critical role of fault tolerance in guaranteeing the dependability and functionality of modern computing systems. The methods presented, including CFC, redundancy approaches, optimized resource management, and security-oriented measures, pave the way for continued advancements in the field of fault tolerance and its application in critical computing systems.

Moving on to another facet of the research, numerous Software-Implemented Hardware Fault Tolerance (SIHFT) methods have been proposed in the literature to enhance the reliability of embedded systems against Random Hardware Failures (RHF). However, selecting a specific SIHFT method presents a challenge due to the multitude of available options, necessitating an objective comparison methodology.

We introduced a comprehensive comparison methodology, comprising the selection of representative applications that were hardened with chosen SIHFT methods, followed by fault injection experiments. Specifically, two well-established CFC methods were selected and applied to two benchmark applications to evaluate our approach.

We then reported simulation results in alignment with the automotive functional safety standard ISO 26262, assessing the efficacy of the CFC methods in detecting RHF. Additionally, we examined the impact of compiler optimization on their effectiveness by conducting experiments at each of the four optimization levels offered by GCC.

It's worth noting that our approach can be extended to various industrial domains, such as unmanned aerial vehicles, as it transcends the confines of the automotive industry. For this study, we opted for automotive industry benchmarks to align with the requirements of automotive functional safety standards, particularly for the use of high-level programming languages.

Adhering to a model-based software design approach, we implemented hardening against RHF at a high level of abstraction, directly within behavioral models, and then automatically translated these models into a high-level language, namely C. This streamlined the implementation of CFC within the Simulink Stateflow environment.

5.3 Future Work

As a natural extension of this research, we propose conducting a more extensive fault injection campaign within the context of the small-scale factory environment. This expanded initiative aims to further validate the research findings, encompassing the efficacy of the developed techniques, the GCC plugin, and the fault injection tool, all of which were instrumental in our investigation.

Up to this point, our exploration has been primarily preliminary in nature, focusing on the small-scale case study. While this preliminary study has allowed us to draw initial conclusions regarding the effectiveness of implementing Control Flow Checking (CFC) methods through high-level programming languages, it represents just the tip of the iceberg. A comprehensive fault injection campaign holds several key objectives:

Firstly, it offers the opportunity to gain a more profound understanding of the potential ramifications of Control Flow Errors (CFEs) within a more extensive industrial setup. Our thesis identified three primary CFE types based on data analysis. However, a more extensive investigation could unveil additional CFE types or shed light on further causal factors contributing to the observed CFE types.

Secondly, it allows for a more thorough assessment of whether CFC methods genuinely avert all hazardous scenarios. While our preliminary study demonstrated that CFC methods can effectively detect a majority of errors, empirical evidence from other experiments suggests that they may not capture all CFEs. By extending the fault injection campaign to cover a wider range of code segments, we can ascertain whether potentially dangerous situations persist within the small-scale factory environment.

Lastly, it would be prudent to explore the feasibility and applicability of these techniques to C++ compilers, considering the prevalent use of C++ in embedded systems. This avenue presents a promising direction for future research endeavors in the realm of embedded systems, addressing the evolving landscape of programming languages and their relevance to fault tolerance strategies.

Our approach can be extended to other industrial landscapes, e.g., unmanned aerial vehicles, since it is more comprehensive than the automotive industry application domain. Here, we selected automotive industry benchmarks to address the needs of automotive functional safety standards for the use of high-level programming languages.

References

- [1] McKinsey & Company. Outlook on the automotive software and electronics market through 2030. <https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/mapping-the-automotive-software-and-electronics-landscape-through-2030?cid=eml-web>.
- [2] McKinsey & Company. Automotive software and electronics 2030. <https://www.mckinsey.com/~media/mckinsey/industries/automotive%20and%20assembly/our%20insights/mapping%20the%20automotive%20software%20and%20electronics%20landscape%20through%202030/outlook%20on%20the%20auto%20C2%ADmotive%20software%20and%20electronics%20market%20through%202030/automotive-software-and-electronics-2030-full-report.pdf>.
- [3] Jens Lienig and Hans Bruemmer. *Fundamentals of electronic systems design*. Springer, 2017.
- [4] ADRIA BARROS DE OLIVEIRA. Applying dual-core lockstep in embedded processors to mitigate radiation-induced soft errors, Number 2017. Available at <https://www.lume.ufrgs.br/bitstream/handle/10183/173785/001061371.pdf?sequence=1>.
- [5] B. Fleming. Microcontroller units in automobiles [automotive electronics]. *IEEE Vehicular Technology Magazine*, 6(3):4–8, 2011.
- [6] J. P. Trovao. Trends in automotive electronics [automotive electronics]. *IEEE Vehicular Technology Magazine*, 14(4):100–109, 2019.
- [7] J. C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 547–550, 2002.
- [8] Research European Parliament’s committee on Industry and Energy (ITRE). The future of the eu automotive sector. [https://www.europarl.europa.eu/RegData/etudes/STUD/2021/695457/IPOL_STU\(2021\)695457_EN.pdf](https://www.europarl.europa.eu/RegData/etudes/STUD/2021/695457/IPOL_STU(2021)695457_EN.pdf), 2021.
- [9] Bill Fleming. Microcontroller units in automobiles [automotive electronics]. *IEEE Vehicular Technology Magazine*, 6(3):4–8, 2011.

- [10] João P. Trovao. Trends in automotive electronics [automotive electronics]. *IEEE Vehicular Technology Magazine*, 14(4):100–109, 2019.
- [11] John C Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th international conference on software engineering*, pages 547–550, 2002.
- [12] Haleh Ardebili, Jiawei Zhang, and Michael G. Pecht. 10 - trends and challenges. In Haleh Ardebili, Jiawei Zhang, and Michael G. Pecht, editors, *Encapsulation Technologies for Electronic Applications (Second Edition)*, Materials and Processes for Electronic Applications, pages 431–479. William Andrew Publishing, second edition edition, 2019.
- [13] Seyab Khan, Said Hamdioui, Halil Kukner, Praveen Raghavan, and Francky Catthoor. Bti impact on logical gates in nano-scale cmos technology. In *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 348–353, 2012.
- [14] Said Hamdioui, Dimitris Gizopoulos, Groeseneken Guido, Michael Nicolaidis, Arnaud Gasset, and Philippe Bonnot. Reliability challenges of real-time systems in forthcoming technology nodes. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 129–134, 2013.
- [15] Innocent Agbo, Mottaqiallah Taouil, Said Hamdioui, Pieter Weckx, Stefan Cosemans, Francky Catthoor, and Wim Dehaene. Read path degradation analysis in sram. In *2016 21th IEEE European Test Symposium (ETS)*, pages 1–2, 2016.
- [16] Sangwoo Pae, Jose Maiz, Chetan Prasad, and Bruce Woolery. Effect of bti degradation on transistor variability in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 8(3):519–525, 2008.
- [17] Krzysztof Iniewski. *Radiation effects in semiconductors*. CRC press, 2018.
- [18] Daniel Oliveira, Sean Blanchard, Nathan Debardeleben, Fernando F. Dos Santos, Gabriel Piscoya Dávila, Philippe Navaux, Carlo Cazzaniga, Christopher Frost, Robert C. Baumann, and Paolo Rech. Thermal neutrons: a possible threat for supercomputers and safety critical applications. In *2020 IEEE European Test Symposium (ETS)*, pages 1–6, 2020.
- [19] Hans G. Kerkhoff and H. Ebrahimi. Intermittent resistive faults in digital cmos circuits. In *2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, pages 211–216, 2015.
- [20] Daniel Gil-Tomás, Joaquín Gracia-Morán, J.-Carlos Baraza-Calvo, Luis-J. Saiz-Adalid, and Pedro-J. Gil-Vicente. Studying the effects of intermittent faults on a microcontroller. *Microelectronics Reliability*, 52(11):2837–2846, 2012.

- [21] Martin Radetzki, Chaochao Feng, Xueqian Zhao, and Axel Jantsch. Methods for fault tolerance in networks-on-chip. *ACM Comput. Surv.*, 46(1), jul 2013.
- [22] Shekhar Borkar, Norman P. Jouppi, and Per Stenstrom. Microprocessors in the era of terascale integration. In *2007 Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6, 2007.
- [23] Mohammadreza Amel Solouki, Shaahin Angizi, and Massimo Violante. Dependability in embedded systems: A survey of fault tolerance methods and software-based mitigation techniques. *arXiv preprint arXiv:2404.10509*, 2024.
- [24] ISO 26262:2018 Road vehicles – functional safety, 2018.
- [25] Afaq Ahmad. Automotive semiconductor industry-trends, safety and security challenges. In *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*, pages 1373–1377. IEEE, 2020.
- [26] Infineon. Aurixtm tc3xx functional safety (fusa) in a nutshell. https://www.infineon.com/dgdl/Infineon-AN0001-AURIX_TC3xx_functional_safety_FUSA_in_a_nutshell-ApplicationNotes-v01_00-EN.pdf?fileId=8ac78c8c8c3de074018c823bd97f1cbc.
- [27] Kavya Prabha Divakarla. Iso26262 and iec61508 functional safety overview. <https://community.nxp.com/pwmxy87654/attachments/pwmxy87654/tech-days/160/1/AMF-AUT-T2713.pdf>, 2017.
- [28] Wen Chen and Jayanta Bhadra. Practices and challenges for achieving functional safety of modern automotive socs. *IEEE Design & Test*, 36(4):31–47, 2019.
- [29] Elena Dubrova. *Fault-tolerant design*. Springer, 2013.
- [30] Bai-Fan Yue and Wei-Wei Che. Data-driven dynamic event-triggered fault-tolerant platooning control. *IEEE Transactions on Industrial Informatics*, 2022.
- [31] Aamer Mahmood and Edward J McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37(2):160–174, 1988.
- [32] Todd M Austin. Diva: A reliable substrate for deep submicron microarchitecture design. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 196–207. IEEE, 1999.
- [33] CA Lisboa, Marcelo Ienczszak Erigson, and Luigi Carro. System level approaches for mitigation of long duration transient faults in future technologies. In *12th IEEE European test symposium (ETS'07)*, pages 165–172. IEEE, 2007.

- [34] Zeyad Alkhalifa, VS Sukumaran Nair, Narayanan Krishnamurthy, and Jacob A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):627–641, 1999.
- [35] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. Control-flow checking by software signatures. *IEEE transactions on Reliability*, 51(1):111–122, 2002.
- [36] Ramtilak Vemu and Jacob Abraham. Ceda: Control-flow error detection using assertions. *IEEE Transactions on Computers*, 60(9):1233–1245, 2011.
- [37] Rajesh Venkatasubramanian, John P Hayes, and Brian T Murray. Low-cost on-line fault detection using control flow assertions. In *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003.*, pages 137–143. IEEE, 2003.
- [38] Olga Goloubeva, Maurizio Rebaudengo, M Sonza Reorda, and Massimo Violante. Improved software-based processor control-flow errors detection technique. In *Annual Reliability and Maintainability Symposium, 2005. Proceedings.*, pages 583–589. IEEE, 2005.
- [39] Seyyed Amir Asghari, Hassan Taheri, Hossein Pedram, and Okyay Kaynak. Software-based control flow checking against transient faults in industrial environments. *IEEE Transactions on Industrial Informatics*, 10(1):481–490, 2013.
- [40] Jose Rodrigo Azambuja, Mauricio Altieri, Jürgen Becker, and Fernanda Lima Kastensmidt. Heta: Hybrid error-detection technique using assertions. *IEEE Transactions on Nuclear Science*, 60(4):2805–2812, 2013.
- [41] Eduardo Chielle, Gennaro S Rodrigues, Fernanda L Kastensmidt, Sergio Cuenca-Asensi, Lucas A Tambara, Paolo Rech, and Heather Quinn. S-seta: Selective software-only error-detection technique using assertions. *IEEE transactions on Nuclear Science*, 62(6):3088–3095, 2015.
- [42] Aiguo Li and Bingrong Hong. Software implemented transient fault detection in space computer. *Aerospace science and technology*, 11(2-3):245–252, 2007.
- [43] Jens Vankeirsbilck, Niels Penneman, Hans Hallez, and Jeroen Boydens. Random additive signature monitoring for control flow error detection. *IEEE transactions on Reliability*, 66(4):1178–1192, 2017.
- [44] Bogdan Nicolescu, Yvon Savaria, and Raoul Velazco. Sied: Software implemented error detection. In *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 589–596. IEEE, 2003.
- [45] Jens Vankeirsbilck, Niels Penneman, Hans Hallez, and Jeroen Boydens. Random additive control flow error detection. In *International Conference on Computer Safety, Reliability, and Security*, pages 220–234. Springer, 2018.

- [46] Mohammad Maghsoudloo, Hamid R Zarandi, and Navid Khoshavi. An efficient adaptive software-implemented technique to detect control-flow errors in multi-core architectures. *Microelectronics Reliability*, 52(11):2812–2828, 2012.
- [47] Jens Vankeirsbilck, Niels Penneman, Hans Hallez, and Jeroen Boydens. Random additive signature monitoring for control flow error detection. *IEEE transactions on Reliability*, 66(4):1178–1192, 2017.
- [48] Seyyed Amir Asghari, Hassan Taheri, Hossein Pedram, and Okyay Kaynak. Software-based control flow checking against transient faults in industrial environments. *IEEE Transactions on Industrial Informatics*, 10(1):481–490, 2013.
- [49] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, pages 256–267, 2017.
- [50] J. Borrego-Carazo, D. Castells-Rufas, E. Biempica, and J. Carrabina. Resource-constrained machine learning for adas: A systematic review. *IEEE Access*, 8:40573–40598, 2020.
- [51] Weijing Shi, Mohamed Baker Alawieh, Xin Li, and Huafeng Yu. Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey. *Integration*, 59:148–156, 2017.
- [52] Jelena Kocić, Nenad Jovičić, and Vujo Drndarević. Sensors and sensor fusion in autonomous vehicles. In *2018 26th Telecommunications Forum (TELFOR)*, pages 420–425. IEEE, 2018.
- [53] J. Janai, Fatma Güney, Aseem Behl, and Andreas Geiger. Computer vision for autonomous vehicles: Problems, datasets and state of the art. *Foundations and Trends® in Computer Graphics and Vision*, 12(1–3):1–308, 2020.
- [54] R. Hussain et al. Autonomous cars: Research results, issues, and future challenges. *IEEE Communications Surveys Tutorials*, 21(2):1275–1313, 2019.
- [55] S. Khan, S. Hamdioui, H. Kukner, P. Raghavan, and F. Catthoor. Bti impact on logical gates in nano-scale cmos technology. In *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 348–353, 2012.
- [56] S. Pae, J. Maiz, C. Prasad, and B. Woolery. Effect of bti degradation on transistor variability in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 8(3):519–525, 2008.
- [57] M. Dumont, M. Lisart, and P. Maurine. Electromagnetic fault injection : How faults occur. In *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 9–16, 2019.

- [58] C. Constantinescu. Intermittent faults and effects on reliability of integrated circuits. In *2008 Annual Reliability and Maintainability Symposium*, pages 370–374, 2008.
- [59] C. Sandionigi, O. Heron, C. Bertolini, and R. David. When processors get old: Evaluation of bti and hci effects on performance and reliability. In *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*, pages 185–186, 2013.
- [60] Ashraf Armoush. *Design patterns for safety-critical embedded systems*. PhD thesis, RWTH Aachen University Aachen, Germany, 2010.
- [61] Abhishek Rhisheekesan, Reiley Jeyapaul, and Aviral Shrivastava. Control flow checking or not?(for soft errors). *ACM Transactions on Embedded Computing Systems (TECS)*, 18(1):1–25, 2019.
- [62] Aviral Shrivastava, Abhishek Rhisheekesan, Reiley Jeyapaul, and Carole-Jean Wu. Quantitative analysis of control flow checking mechanisms for soft errors. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6, 2014.
- [63] Alfredo Benso, Stefano Di Carlo, Giorgio Di Natale, and Paolo Prinetto. A watchdog processor to detect data and control flow errors. In *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003.*, pages 144–148. IEEE, 2003.
- [64] Ameya Chaudhari, Junyoung Park, and Jacob Abraham. A framework for low overhead hardware based runtime control flow error detection and recovery. In *2013 IEEE 31st VLSI Test Symposium (VTS)*, pages 1–6. IEEE, 2013.
- [65] Jacob A Abraham and Ramtilak Vemu. Control flow deviation detection for software security, December 31 2009. US Patent App. 12/484,839.
- [66] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.
- [67] Ze Zhang, Sunghyun Park, and Scott Mahlke. Path sensitive signatures for control flow error detection. In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 62–73, 2020.
- [68] Shubu Mukherjee. *Architecture design for soft errors*. Morgan Kaufmann, 2011.
- [69] Greg Bronevetsky, B de Supinski, and Martin Schulz. A foundation for the accurate prediction of the soft error vulnerability of scientific applications. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2009.
- [70] Shekhar Borkar et al. Microarchitecture and design challenges for gigascale integration. In *MICRO*, volume 37, pages 3–3, 2004.

- [71] Aamer Mahmood and Edward J McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37(2):160–174, 1988.
- [72] Wendy Bartlett and Lisa Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on dependable and secure computing*, 1(1):87–96, 2004.
- [73] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. Nonstop/spl reg/advanced architecture. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 12–21. IEEE, 2005.
- [74] Ying C Yeh. Triple-triple redundant 777 primary flight computer. In *1996 IEEE Aerospace Applications Conference. Proceedings*, volume 1, pages 293–307. IEEE, 1996.
- [75] Daya Shanker Khudia and Scott Mahlke. Low cost control flow protection using abstract control signatures. In *Proceedings of the 14th ACM SIG-PLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, pages 3–12, 2013.
- [76] Zhiqi Zhu, Joseph Callenes-Sloan, and Benjamin Carrion Schafer. Control flow checking optimization based on regular patterns analysis. In *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 203–212. IEEE, 2018.
- [77] K Vinoth Kannan. Model-based automotive software development. In *Automotive Embedded Systems: Key Technologies, Innovations, and Applications*, pages 71–87. Springer, 2021.
- [78] The MathWorks Inc. Matlab version: 9.13.0 (r2022b), 2022.
- [79] Seyyed Amir Asghari, Atena Abdi, Hassan Taheri, Hossein Pedram, Saadat Pourmzaffari, et al. Sedsr: Soft error detection using software redundancy. *Journal of Software Engineering and Applications*, 5(09):664, 2012.
- [80] Bogdan Nicolescu, Yvon Savaria, and Raoul Velazco. Sied: Software implemented error detection. In *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 589–596. IEEE, 2003.
- [81] Olga Goloubeva, Maurizio Rebaudengo, M Sonza Reorda, and Massimo Violante. Soft-error detection using control flow assertions. In *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 581–588. IEEE, 2003.
- [82] F Saglietti. Strategies for the achievement and assessment of software fault-tolerance. *IFAC Proceedings Volumes*, 23(8):303–308, 1990.
- [83] Brian Randell and Jie Xu. The evolution of the recovery block concept. *Software fault tolerance*, 3:1–22, 1995.

- [84] Algirdas Avizienis. The methodology of n-version programming. *Software fault tolerance*, 3:23–46, 1995.
- [85] Jean-Claude Laprie, Jean Arlat, Christian Beounes, and Karama Kanoun. Definition and analysis of hardware-and-software fault-tolerant architectures. In *Predictably Dependable Computing Systems*, pages 103–122. Springer, 1995.
- [86] R. Keith Scott, James W. Gault, and David F. McAllister. Fault-tolerant software reliability modeling. *IEEE transactions on Software Engineering*, (5):582–592, 1987.
- [87] Dhiraj K Pradhan et al. *Fault-tolerant computer system design*, volume 132. Prentice-Hall Englewood Cliffs, 1996.
- [88] Torres Wilfredo. Software fault tolerance: A tutorial. 2000.
- [89] Victor F Nicola. *Checkpointing and the modeling of program execution time*. University of Twente, Department of Computer Science and Department of . . . , 1994.
- [90] Mark Russinovich and Zary Segall. Fault-tolerance for off-the-shelf applications and hardware. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 67–71. IEEE, 1995.
- [91] Israel Koren and C Mani Krishna. *Fault-tolerant systems*. Morgan Kaufmann, 2020.
- [92] Eduardo Chielle, Boyang Du, Fernanda L. Kastensmidt, Sergio Cuenca-Asensi, Luca Sterpone, and Matteo Sonza Reorda. Hybrid soft error mitigation techniques for cots processor-based systems. In *2016 17th Latin-American Test Symposium (LATS)*, pages 99–104, 2016.
- [93] F. Abate, L. Sterpone, and M. Violante. A new mitigation approach for soft errors in embedded processors. In *2007 9th European Conference on Radiation and Its Effects on Components and Systems*, pages 1–6, 2007.
- [94] Massimo Violante, Cristina Meinhardt, Ricardo Reis, and Matteo Sonza Reorda. A low-cost solution for deploying processor cores in harsh environments. *IEEE Transactions on Industrial Electronics*, 58(7):2617–2626, 2011.
- [95] Julen Gomez-Cornejo, Aitzol Zuloaga, Uli Kretzschmar, Unai Bidarte, and Jaime Jimenez. Fast context reloading lockstep approach for seus mitigation in a fpga soft core processor. In *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*, pages 2261–2266, 2013.
- [96] Hung-Manh Pham, Sébastien Pillement, and Stanisław J. Piestrak. Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor. *IEEE Transactions on Computers*, 62(6):1179–1192, 2013.

- [97] N.S. Bowen and D.K. Pradham. Processor- and memory-based checkpoint and rollback recovery. *Computer*, 26(2):22–31, 1993.
- [98] The MISRA Consortium Limited. Misra c:2023 guidelines for the use of the c language in critical systems, 2023.
- [99] Gilad Dar, Giorgio Di Natale, and Osnat Keren. Nonlinear code-based low-overhead fine-grained control flow checking. *IEEE Transactions on Computers*, 71(3):658–669, 2021.
- [100] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.
- [101] K Chandrasekaran. *Essentials of cloud computing*. CrC Press, 2014.
- [102] Kevin D Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 43–54, 2009.
- [103] Kevin D. Bowers, Ari Juels, and Alina Oprea. Hail: A high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS '09*, page 187–198, New York, NY, USA, 2009. Association for Computing Machinery.
- [104] Joshua Schiffman, Thomas Moyer, Hayawardh Vijayakumar, Trent Jaeger, and Patrick McDaniel. Seeding clouds with trust anchors. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, pages 43–46, 2010.
- [105] Jacopo Sini, Massimo Violante, and Fabrizio Tronci. A novel iso 26262-compliant test bench to assess the diagnostic coverage of software hardening techniques against digital components random hardware failures. *Electronics*, 11(6):901, 2022.
- [106] The risc-v instruction set manual volume i: Unprivileged isa document version 20191213.
- [107] Stefano Di Mascio, Alessandra Menicucci, Gianluca Furano, Claudio Monteleone, and Marco Ottavi. The case for risc-v in space. In *International Conference on Applications in Electronics Pervading Industry, Environment and Society*, pages 319–325. Springer, 2019.
- [108] Gnu risc-v toolchain [online] available. <https://github.com/johnwinans/riscv-toolchain-install-guide>, 2022.
- [109] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. California, USA, 2005.
- [110] The gnu debugger [online] available. <https://www.gnu.org/software/gdb/>, 2022.

Appendix A

Guidelines

A.1 Introduction

This chapter presents a guideline for applying a subset of SIHFT methods called Control Flow Checking (CFC) methods to application code written in C language. The motivation is that in the literature few guidelines can be found that provide insight on implementing CFC methods with high-level programming languages. Most proposals implement CFC methods in low-level languages such as assembly. The rationale behind developing high-level language implementations lies in the pursuit of architecture independence as well as the inadequacy of a certified compiler for the target platform that can conveniently incorporate Certified Functionally Correct into the compiled assembly/machine language code.

This chapter presents a set of guidelines for implementing CFC methods using C programming language. We evaluate the effectiveness of the proposed approach in two case studies. Our results show that our approach regarding the implementation of the CFC in C language is maintaining the effectiveness of the CFC method in detecting RHF in the embedded systems. Hence, our approach is a viable alternative to implementing traditional hardening techniques.

To understand this guideline, Figure A.1 explains how to read Control Flow Graphs. The control graph is a visual representation of the control flow of a program. The nodes in the graph represent the statements in the program, and the edges represent the control flow between the statements.







Graphic representation	Explanation
	Statement/statements to be added to perform a test/set operation needed by the CFC technique. (Regardless of the color).
	A basic block of the algorithm to be hardened. (Regardless of the color).
	A legal transition between two basic blocks.
	Function call. (Regardless of the color).
	A body of statements inside a basic block to indicate if the statement of CFC algorithm has to be put before or after it.
	Horizontal bars indicate borders between basic blocks (regardless of the color).

Fig. A.1 Instructions on how to read the Control Flow Graphs represented in this chapter.

A.2 Functions or macros needed in C language

The YACCA and RACFED algorithms use different methods for performing the TEST and SET operations, with algorithm 4 used for TEST in YACCA and algorithm 5 used for RACFED, and algorithm 6 used for SET in YACCA and algorithm 7 used for RACFED. In RACFED, the Run Time Signature (RTS) is updated after each basic block statement by summing a random number, and the total of these random numbers is subtracted before signature checking to allow for intra-block detection capabilities. The SET operation is conducted in two phases within the actual algorithm. To optimize YACCA, the predecessor's mask is retrieved from the last TEST call in the implementation, as TEST always occurs before SET.

A.3 Switch-case construct

For a switch-case construct, Figure A.2 shows the positions of the TEST and SET statements for the entry BB which is indicated with the `switch(...)` statement.

Algorithm 4 TEST operation (YACCA)

```

1: TEST(RTS, predecessors_mask)
2: if RTS  $\wedge$  ( $\neg$  predecessors_mask) then
|   CFE detected
|   end
3: Continue normal execution

```

Algorithm 5 TEST operation (RACFED). *bb* represents the ID of the BB which is calling TEST(). *RTS* is an array containing the compile-time signature of every BB.

```

1: TEST(bb)
2: if RTS  $\neq$  CTS[bb] then
|   CFE detected
|   end
3: Continue normal execution

```

Algorithm 6 Set operation (YACCA)

```

1: SET(RTS, predecessors_mask, BB_ID)
2: RTS = RTS  $\wedge$   $\neg$  predecessors_mask
3: RTS = RTS  $\vee$  (1  $\ll$  BB_ID)

```

Algorithm 7 Set operation (RACFED). *bb* represents the current BB, while *bb*₊₁ its expected successor. subRanPrevVal and *CTS* are arrays (with lengths equal to the number of BBs) containing respectively the random number sums and the compile-time signature for every BB.

```

1: SET(bb, bb+1)
2: RTS = RTS - subRanPrevVal[bb]
3: adjVal = (CTS[bb] + subRanPrevVal[bb]) + (CTS[bb+1] + subRanPrevVal[bb+1])
4: RTS = RTS + adjVal

```

Meanwhile, Figure A.3 shows the positions of the TEST and SET statements for the exit BB which is indicated with a } character.

It is important to note that in cases where there is no default case for the entry BB, its inclusion is necessary. We address this by adding a default case to the original algorithm used in implementing the CFC. Lastly, an optimized diagnostic coverage for exit BB can be achieved by testing the signature of its only legal predecessors for each switch case. This is due to the presence of diverse paths in this construct.

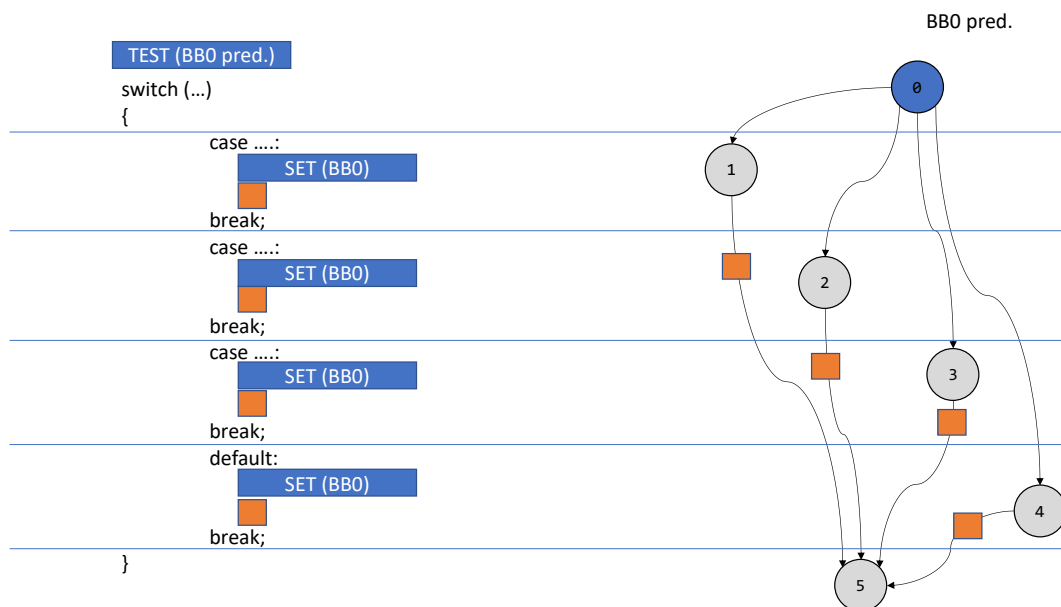


Fig. A.2 Positions of the TEST and SET operations for inter-block CFE detection inside the switch-case constructs for the entry block (indicated as 0, in blue.)

A.4 If-else construct

For the if-else construct, Figure A.4 illustrates the positions of the TEST and SET statements for the entry BB, which is indicated with the if (...) statement. While Figure A.5 displays the positions of the TEST and SET statements for the exit block which is indicated with a } character.

If there is no else statement, an else statement should be included. In this situation, it is strongly recommended to add a comment clarifying that the else

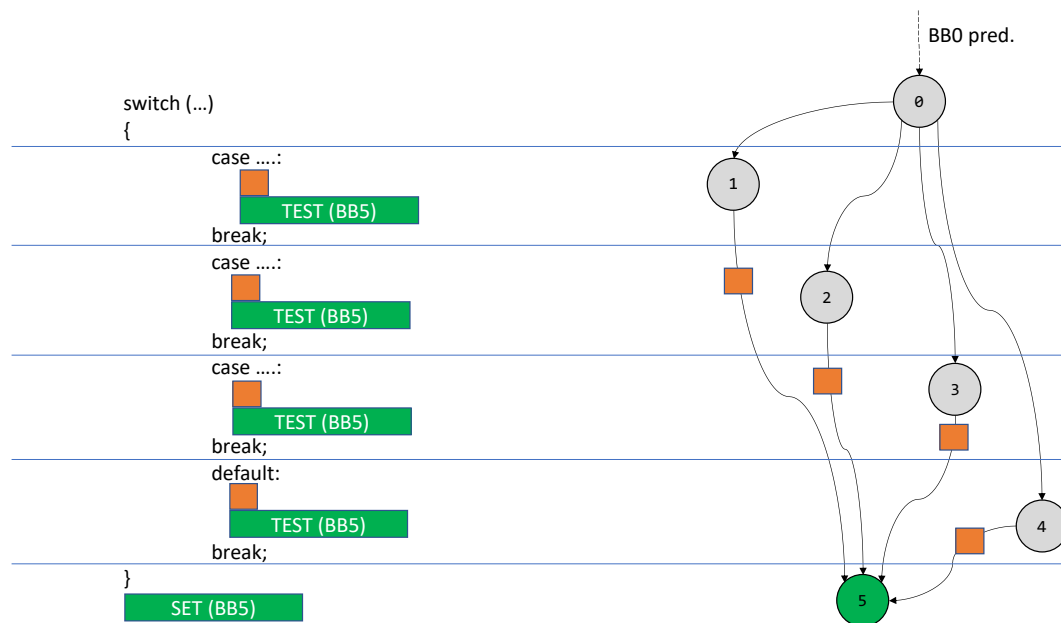


Fig. A.3 Positions of the TEST and SET operations for inter-block CFE detection inside the `switch-case` constructs for the exit block (indicated as 5, in green.)

statement was inserted to the initial algorithm to facilitate the implementation of the CFC.

A.5 Function calls

Figure A.6 illustrates that a function call is considered a basic block (BB) since the call and return statements act as jumps. Each function has its own return to subroutine Run Time Signature (RTS) so its TEST and SET functions operate on two different signatures: one for the caller and another for the called function. The blue operations refer to operations on the caller's Run Time Signature (RTS), while the yellow operations refer to operations on the function's Run Time Signature (RTS). The BBs are also color-coded. Within a function call, there are generally three BBs, including:

- the previous BB of the caller (p_c),
- the function call (f),
- and the BB following the function call (f_c).

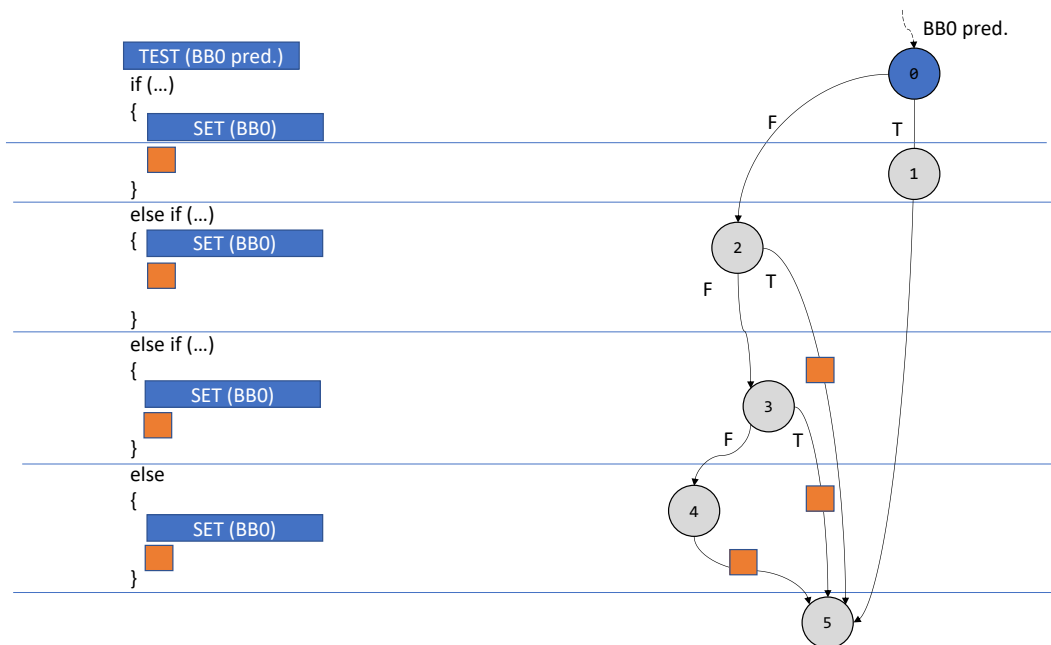


Fig. A.4 Positions of the TEST and SET operations for inter-block CFE detection inside the `if-else` constructs for the entry block (indicated as 0, in blue.)

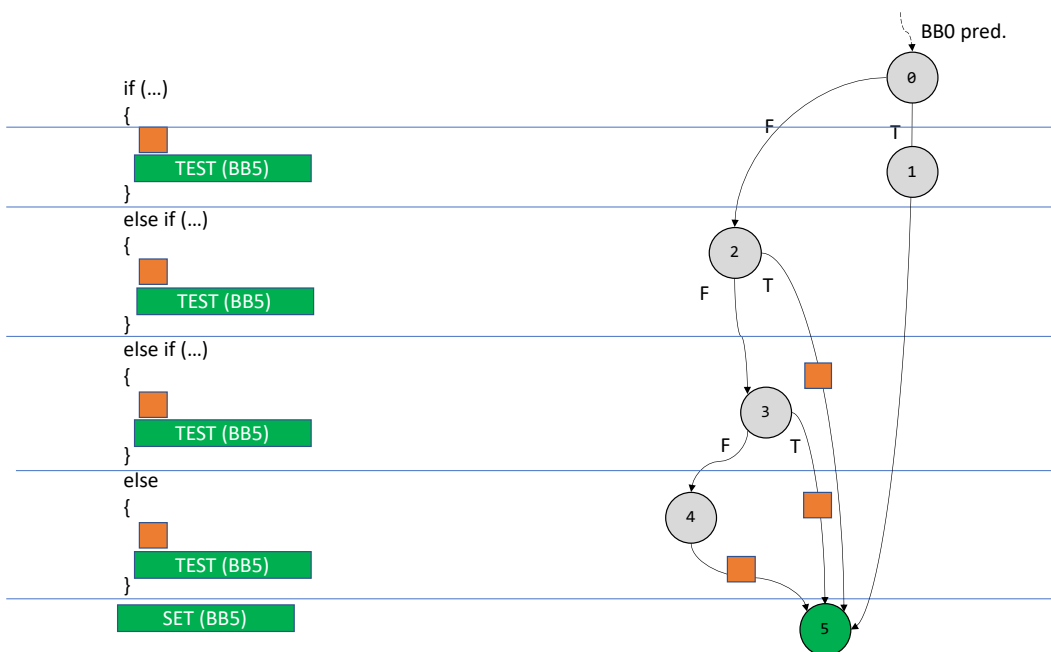


Fig. A.5 Positions of the TEST and SET operations for inter-block CFE detection inside the `if-else` constructs for the exit block (indicated as 5, in green.)

Meanwhile, in a called function, we can define at least two BBs, including the (i) initial BB (i_f) and (ii) final BB (f_f). The two BBs may be merged if the called function has only one BB or if its source code is unavailable.

The following implementation steps are taken:

1. Before the function call, we insert the SET operation for the caller BB (p_c) signature.
2. Then, the function call BB starts. As usual, the signature of the predecessor BB is tested with the TEST operation for the caller BB (p_c) signature.
3. Now, the SET operation is called for the signature of the i_f BB. If the function has been already called in other points of the program, the signature remains set to the signature of the f_f BB, generating a false CFE. The signature of the i_f BB is the signature of the function wrapper, while the signature of the f_f BB is the signature expected at the end of the function call. Hence, the signatures of the i_f and f_f BBs are different if the function is hardened, equal otherwise.
4. The function is executed (with possible hardening) and terminates. The signature of the f_f BB is tested with the TEST operation.
5. The wrapper sets the signature to the signature of the f BB. (since it is a normal BB of the function call) as its last instruction.
6. The BB following the function call tests the signature against the signature of the caller BB (f).

If the compiler decides to inline the function, the proposed strategy behaves correctly (since the TESTs/SETs order is kept). If a function is impossible to harden (for example, using standard libraries or Application Programming Interfaces) or the function contains only a single BB, it is treated as a normal statement.

If a function is called from different locations within the code, separate wrappers must be utilized, each with the appropriate TEST and SET operations. Finally, if the application is multi-threaded, it is critical to avoid having the Run Time Signature (RTS) and predecessor masks of the function as `static` variables to avoid conflicts. `Local` variables are used in this case.

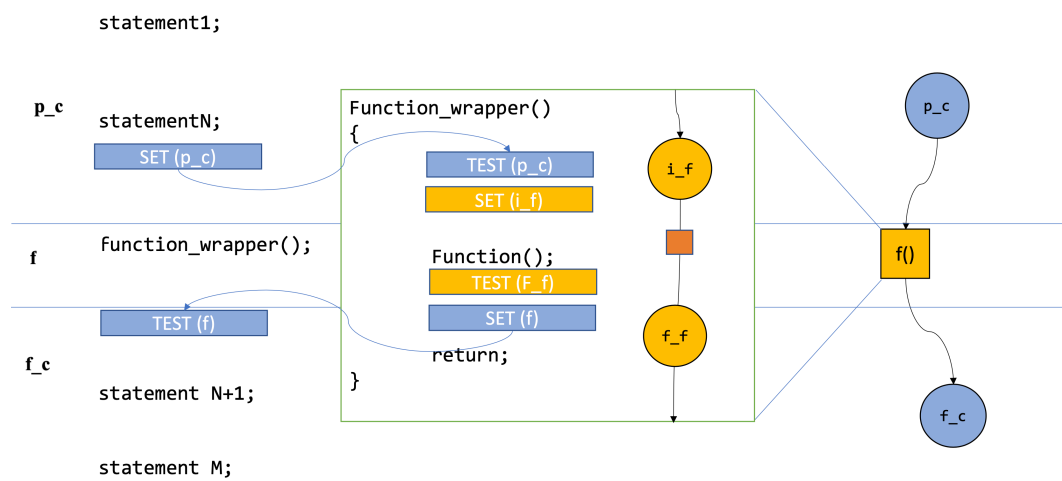


Fig. A.6 Positions of the TEST and SET operations for inter-block CFE detection for a function call.

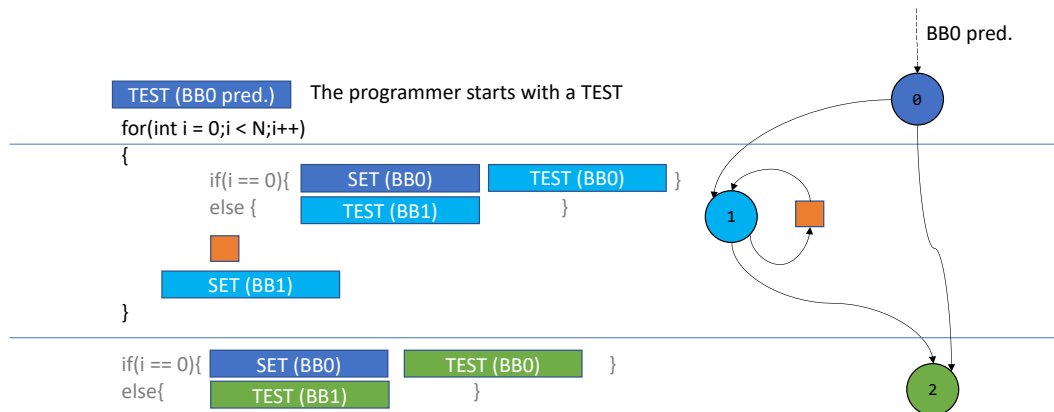


Fig. A.7 Positions of TEST and SET operations to detect inter-block CFEs for a for loop.

A.6 For loops

To implement a hardened `for` loop, the structure depicted in Figure A.7 must be employed, along with the `if . . . else` structures shown in gray. It should be noted that these structures are not separate basic blocks (BBs), as they are not present in the original algorithm but are necessary to properly call the TEST/SET functions in the correct order. If the body of the `for` loop statement (denoted by the orange square in the figure) contains more than one BB, they should be hardened as usual, ensuring that the last BB places the same tested signature in the `else` statements (identified as BB1 in the figure). For handling a break statement within a `for` loop, as demonstrated in Figure A.8, it is impossible to know if the break has been executed. Therefore, the only way is to refrain from performing the TEST and SET operations on BB2 (which is disregarded in the implementation). The reason is that the break is the sole means of reaching BB5 without executing BB4; hence, BB1 is a legal predecessor of BB5.

A.7 Conclusions

This chapter presents a comprehensive set of guidelines to assist in the development of safe and reliable embedded systems written in C while employing CFC hardening methods.

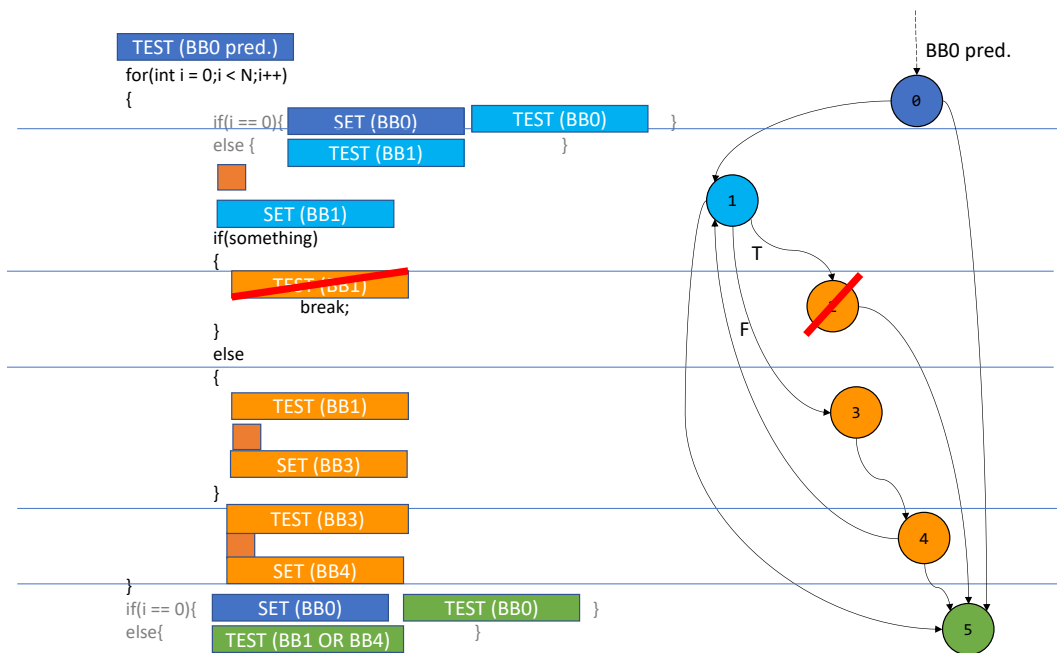


Fig. A.8 Positions of TEST and SET operations to detect inter-block CFEs for a for loop containing a break instruction inside it.

Following the proposed guideline is not mandatory, but it is designed to offer a practical approach to developing critical safety embedded systems. The effectiveness of this guideline has been demonstrated through a series of case studies, where reliable embedded systems were developed and deployed in safety-critical situations. The experimental results emphasize the applicability of the guidelines in automotive industry contexts due to their successful employment in this automotive industry scenario.

In conclusion, it would be valuable to investigate the applicability of this method to C++ compilers, especially given the current prevalence of embedded systems using C++ code. This could be a potential avenue for future research in the field of embedded systems.

Appendix B

My publications

B.1 Journals

1. Dependability in Embedded Systems: A Survey of Fault Tolerance Methods and Software-Based Mitigation Techniques
M. Amel Solouki, S. Angizi and M. Violante.
to be submitted
arXiv preprint arXiv:2404.10509
2. An Experimental Evaluation of Control Flow Checking for Automotive Embedded Applications Compliant with ISO 26262
M. Amel Solouki, J. Sini and M. Violante.
IEEE Access, doi: 10.1109/ACCESS.2023.3279731
3. Implementation of Control-Flow Checking - A New Perspective Adopting a Model-Based Software Design Approach
M. Amel Solouki, J. Sini and M. Violante.
Electronics 2022, 11(19), 3074; DOI= 10.3390/electronics11193074
4. A Novel Redundant Validation IoT System for Affective Learning based on Facial Expressions and Biological Signals
A. Marceddu Costantino, L. Pugliese, J. Sini, G. Ramirez Espinosa, **M. Amel Solouki**, P. Chiavassa, E. Giusto, B. Montrucchio, M. Violante, and F. De Pace
Sensors (ISSN: 1424-8220) Vol 22, 2022-DOI = 10.3390/s22072773

B.2 Conferences and Workshops

1. Enhancing Automotive Embedded Applications: A Comprehensive Evaluation of Control Flow Checking Methods
M. Amel Solouki, J. Sini and M. Violante
IEEE 2nd International conference on Design, Test & Technology of Integrated Systems (DTTIS).
Accepted
2. Guidelines for Implementing Control Flow Checking into Automotive Embedded Applications Developed with C Language
J. Sini, **M. Amel Solouki** and M. Violante
IEEE Nordic Circuits and Systems Conference (NorCAS), 2023, pp. 1-6, doi: 10.1109/NorCAS58970.2023.10305466.‘
3. A New Approach to Selectively Control Flow Checking Methods Compliant with ISO 26262
M. Amel Solouki, J. Sini and M. Violante
20th ACM International Conference on Computing Frontiers (CF ’23), 215–216.
<https://doi.org/10.1145/3587135.3592185>
4. Assessing Effectiveness of Software-Implemented Control Flow Checking Methods for Automotive Embedded Applications : a New Approach
M. Amel Solouki, J. Sini and M. Violante
The 6th Conference on PhD Research in Microelectronics and Electronics in Latin America (PRIME-LA) was held in conjunction with the 2023 IEEE 14th Latin American Symposium on Circuits and Systems (LASCAS) in Quito, Ecuador -*Presented*
5. An Empirical Study on the Effectiveness of Control Flow Checking Algorithms Implemented by the Model-Based Software Design Approach
M. Amel Solouki, J. Sini and M. Violante
29th IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2022, pp. 1-4, doi: 10.1109/ICECS202256217.2022.9970849.
6. Control Flow Error Detection Techniques Assessment for Embedded Software Development and Validation

M. Amel Solouki, J. Sini and M. Violante

27th IEEE European Test Symposium (ETS)-*Poster*