POLITECNICO DI TORINO

Master Degree course in Mechatronic Engineering

Master Degree Thesis

Rust4Safety: Comparison of Software-Implemented Hardware Fault Tolerance Techniques between C and Rust Programming Languages

Supervisors
Prof. Massimo VIOLANTE
Dr. Jacopo SINI
Dr. Mohammadreza AMEL SOLOUKI

Candidate
Luana CUCCHIARA

Academic Year 2024-2025

Abstract

In recent years, embedded system have faced a rise of their application in different fields, from industrial to automotive and avionic, increasing also their utilization in safety-critical application.

Random Hardware Failures (RHFs) are unavoidable phenomenon, which can lead to various effect such as data corruption that can lead to disruption of the application instructions execution leading to Control Flow Errors (CFEs), potentially resulting in unpredictable and catastrophic consequences. Ensuring the elimination of systematic errors can be achieved by adhering to the guidelines and a well-designed code but these measures does not prevent the application to be prone to RHFs. To mitigate Random Hardware Failures it is possible to apply Software-Implemented Hardware Fault Tolerance, like the Control Flow Checking (CFC). Control Flow Checking adds extra instruction inside the code, assuring the correct control flow of the application and the correct function.

This work of thesis is inserted in a bigger study aimed at evaluating and validating the use of Rust as a programming language in automotive application, as a valid alternative to C. C language is currently the industry standard and is widely accepted within the automotive functional safety standard ISO 26262, which mandates the use of high-level programming languages for developing safety-relevant software components. In any case, concerns on security of embedded applications are raising. This, together with the increasing complexity of the applications, makes development of robust applications more challenging, leading to the need for a new programming language that can address some of these issues. Rust was selected for its features that provides a robust memory protection, addressing a critical vulnerability encountered in C-based application.

In this work, an application code is developed using a model-software design approach, which is widely adopted in the automotive field. This approach involves generating C/C++ code directly from the model developed with graphical programming environment. The generated code is then hardened with two distinct CFCs: Yet Another Control-flow Checking using Assertion (YACCA) and Random Additive Control Flow Error Detection (RACFED). These two CFC methods add in the code extra instruction. They assign an unique signature to each Basic Block of the code and then check if the code has been executed sequentially, following the signatures previously assigned. Then, the application is translated into Rust for a direct comparison. The application in Rust has been hardened with the same CFC methods of the C-based one.

The objective of this work of thesis is to plan a fault injection providing also the coverage metrics to evaluate the effectivity of the two Control Flow Checking methods back-to-back with respect also to the two different programming language.

It is proposed to submit the hardened code to a campaign of fault injection in an Instruction Set Architecture (ISA) emulation environment. The target processor for this study is 32 bit RISC-V, and the campaigns perform fault injection of the type stuck-at-bit targeting the program counter register.

Moreover, to present the results in compliance with ISO 26262, in the last part of this thesis work all the metrics for evaluating the efficacy of the two methods applied to the two different programming languages, Rust and C are presented.

Contents

List of Figures

List of Tables

Listings

Acronyms

ADAS Advanced Driver Assistance Systems.

ASIL Automotive Safety Integrity Level.

BB Basic Block.

CFC Control Flow Checking.

CFE Control Flow Error.

CFG Control Flow Graph.

E/E Electrical and Electronic.

ECU Electronic Control Unit.

FI Fault Injection.

FIM Fault Injection Manager.

FMEDA Failure Modes Effect and Diagnostic Analysis.

FSC Functional Safety Concept.

FSM Finite State Machine.

FTTI Fault Tolerance Time Interval.

GDB Gnu Debugger.

HARA Hazard Analysis and Risk Assessment.

HW Hardware.

IEC International Electrotechnical Commission.

ISA Instruction Set Architecture.

ISO International Organization for Standardization.

MBSD Model-Based Software Design.

MISRA Motor Industry Software Reliability Association.

MTTF Mean Time to Failure.

PC Program Counter.

QEMU Quick EMUlator.

RACFED Random Additive Control Flow Error Detection.

RHF Random Hardware Failures.

RISC Reduced Instruction Set Computer.

SEE Single Event Effect.

SEU Single Event Upset.

SG Safety Goal.

SIHFT Software-Implemented Hardware Fault Tolerance.

SW Software.

YACCA Yet Another Control-flow Checking using Assertion.

Chapter 1

Introduction

1.1 Motivation and Context

Every day more vehicles are sold all over the world. McKinsey's research [1] on the growth of the global automotive market defined an increase of 3% per year in sales, going from USD 2,755 billion in 2020 to USD 3,800 billion. A bigger growth is expected for Software (SW) and Electrical and Electronic (E/E) as shown in Figure 1.1. Their market is expected to grow at a rate of 7% per year, going from a total market size of USD 238 billion to USD 469 billion.

The automotive industry has undergone significant transformation in the past few decades, driven by the integration of embedded systems. Modern vehicles are no longer purely mechanical machines, but integrate software and digital technologies to provide a wide range of functions that increase the overall drive experience. This evolution has been fueled by advances in microelectronics, software engineering, and digital communication, resulting in vehicles that offer improved safety, efficiency, and user experience. This trend is continuing to grow: in the last 60 years, the electronic components within a car have grown from 3% of the total value of a car to 50%, which is expected to be reached in 2030. [9]

From a historical point of view, the first use of an embedded system in a car was developed by Bosch (Gerlingen, Germany) to implement an electronic fuel injection system for a Volkswagen in 1968. As this system was directly related to engine control, it became known as the Electronic Control Unit (ECU) [10]. From that moment, embedded systems have played a crucial role in optimizing vehicle performance, managing a wide range of functions, from engine operation and fuel efficiency to infotainment, connectivity and Advanced Driver Assistance Systems (ADAS), all of which contribute to enhanced driver comfort and safety.

The increasing reliance on software-based applications in vehicles has facilitated numerous innovations, including systems such as the Anti-lock Braking System (ABS) or the Electronic Stability Control (ESC), as well as the rise of Advanced Driver Assistance Systems (ADAS) features, such as adaptive cruise control, collision avoidance, or automatic braking. These vehicle upgrades serve as a bridge between the old way of driving and the innovative Autonomous Driving (AD). However, the rise of software-defined vehicles, which integrate connectivity and electrification, requires the use of more software and so it is relevant to create standards to introduce robust testing, validation, and good practices for the development of software-based vehicles. [9]

Embedded systems are widely employed in safety-critical applications, especially in autonomous vehicles, where systematic and random errors can pose serious risks. Systematic errors arise due to flaws in design or development and can be mitigated through adherence to standards and best practices in software engineering. However, hardware faults can be random and caused by some errors due to the manufacturing process of the component, internal and external noise sources, and aging. The software is immune to aging, so it can contain onlu systematic defects.

Random Hardware Faults (RHFs) affect the hardware component and can be caused by different reasons and factors, such as the power supply, external noise or, for space application, alpha radiations or cosmic rays. The faults generated by the RHFs can be soft errors, which generates computational errors, but usually they do not cause damage to the hardware, or permanent errors [11] [12]. These factors produce faults that can undermine the good performance of the software.

Detecting and mitigating RHFs is a significant challenge for the developer due to their sporadic nature. Despite the inherent unpredictability, it is essential for automotive designers to acknowledge that such faults will inevitably occur in deployed systems. Consequently, robust design strategies are crucial for ensuring safety developing an application that allows a fault tolerance, so a way for the application to continue working even if there is a fault detected. Fault tolerance is typically achieved through redundancy, which can be applied at the hardware, software, information, and temporal levels, or through hybrid approaches combining multiple redundancy strategies [12] [4].

To ensure the reliability and safety of embedded systems, fault-tolerance techniques must be integrated with industry standards to mitigate systematic and random faults. Measures of fault tolerance are necessary in every field, not only in the automotive field but also in the aerospace field, where a failure can produce catastrophic outcomes, and also in the medical field. In this work of thesis, the main focus is to plan a fault injection campaign for testing the performances and the effectiveness of control flow checking to detect random hardware failures in an application developed for embedded system.

1.2 Problem Statement and Objectives

Safety-critical applications must account for the possibility of faults throughout their operational lifetime. These applications must comply with international functional safety standards to ensure reliability despite the occurrence of Random Hardware Failures (RHFs).

In digital systems, the terms fault and failure have distinct meanings. A failure occurs when system behavior deviates from expectations, resulting from an error, which itself is the manifestation of a fault.

Faults can be systematic, originating from design defects, or induced by external factors such as cosmic rays, noise, or power supply fluctuations. Faults may remain

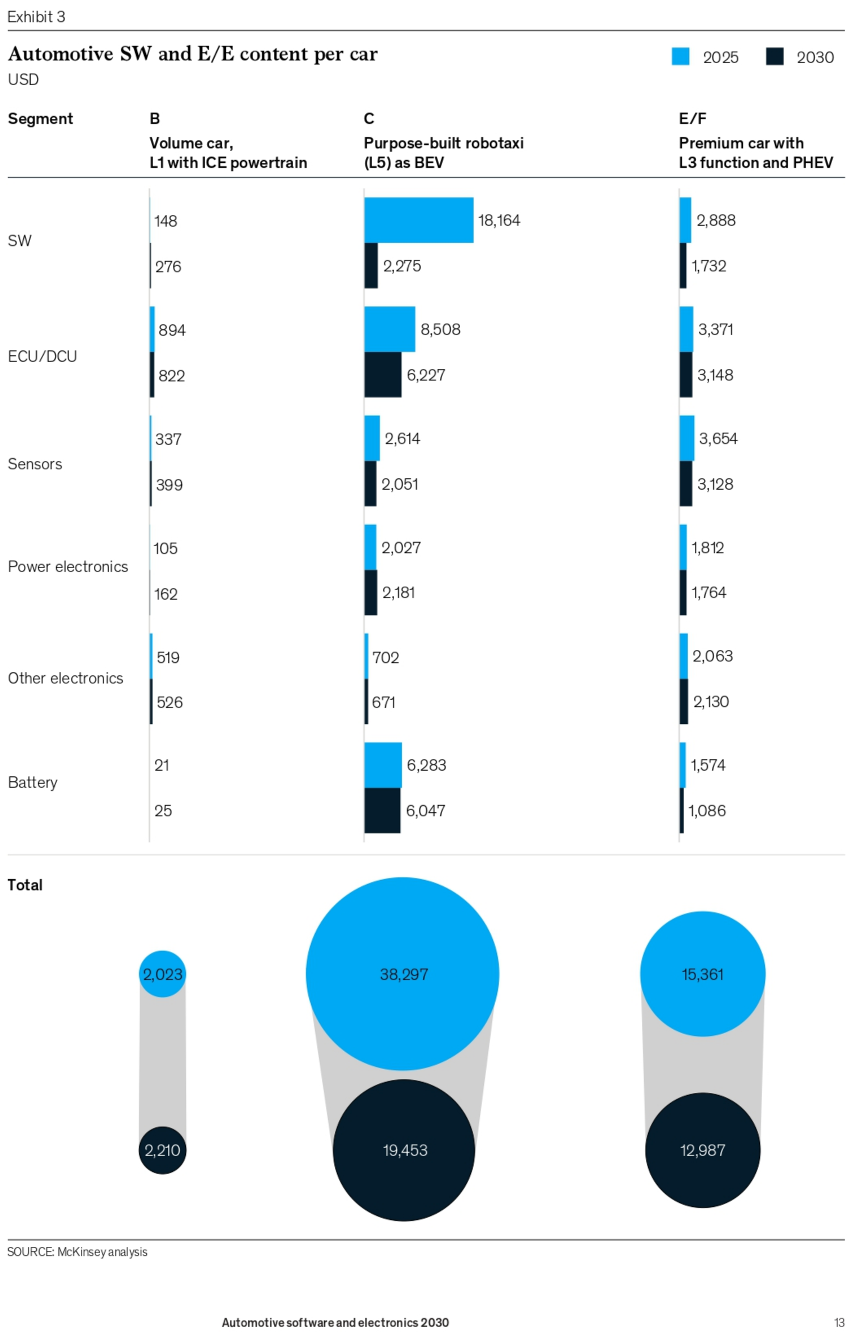


Figure 1.1. Comparison between 2025 and 2030 trend of automotive SW and E/E content per car [1].

latent, causing no immediate effects, or they may trigger errors [13]. A single fault can lead to multiple errors, potentially culminating in a failure that propagates further errors.

The impact of a fault varies depending on the affected processor component and the type of fault involved.

Modern software applications are designed with a fault tolerance approach, contrasting with traditional fault avoidance techniques that focus on eliminating fault occurrence. The fault tolerance approach ensures system safety even in the presence of faults applying two types of philosophies: a fail-safe, where the system maintains a minimum set of operations to allow the user a minimum set of controllability, or a fail-operational, where the system maintains the same functionalities even in the event of fault.

Fault tolerance techniques enhance system robustness by introducing redundancy, which can be implemented at the hardware, software, or temporal levels, or through hybrid methods.

Hardware redundancy involves additional components such as watchdogs, checkers, or Infrastructure Intellectual Properties (I-IPs), increasing system cost. Conversely, software redundancy, which adds extra code, is more flexible and cost-effective but may introduce execution overhead [14].

Time redundancy extends processing time to enhance fault tolerance, while hybrid methods combine multiple strategies to achieve resilience.

However, all the fault tolerance techniques incorporate one or more of the following elements [15]:

- Masking: Correcting errors during execution.
- Detection: Identifying errors within the application.
- Containment: Preventing error propagation.
- Repair: Bypassing faulty components to deliver correct results.
- Recovery: Restoring the system to an operational state.

Given the unpredictability of RHFs, mitigating their effects requires implementing fault tolerance measures. A Software-Implemented Hardware Fault Tolerance (SIHFT) technique utilized in this study is Control Flow Checking, which employs signature verification to ensure correct execution paths within the application.

To evaluate the performances of the techniques adopted in this work of thesis the parameters proposed by the literature are:

- **Error Detection Latency**: the time needed by the system to detect and mitigate the fault.
- **Fault Coverage**: determine how many faults can be detected by the hardening system
- **Mean Time to Failure (MTTF)**: provides the average time between the failures
- **Performance Overhead**: determine the additional computational resourced required by the fault tolerant system.

In this work were implemented CFC techniques into both the C and the Rust programming language, to be compliant with the mandatory rules described in the international standards ISO 26262.

Then it was performed fault injection campaigns to verify the efficacy of the hardening implementation into the application. The result are presented in compliance with the standard ISO 26262 classification.

1.3 Thesis Outline

This work of thesis is structured as follows:

Chapter 2 reviews the role of embedded systems in safety-critical domains, highlighting its criticalities and focusing on a detailed description of the Random Hardware Failures that can affect them. It also presented a comparison back to back of the two programming languages used in this thesis work. Additionally, an overview of the international standards in the automotive field is presented.

Chapter 3 is dedicated to the description of the Model-Based Software Design approach and the implementation of the Control Flow Checking techniques to the application. It also describes the setup of the test bench used to implement the fault injection campaigns into the software.

Chapter 4 presents the planning of the fault injection campaign, describing the metrics in compliance with international standards to provide a comparison between the two implemented Control Flow Checking methods in different programming languages.

And then, lastly, in chapter 5 provides a summary of this manuscript with some proposal for future works.

Chapter 2

Background and Related Work

This chapter explores the role of embedded systems in safety-critical domains and its criticalities.

A key focus is the management of Random Hardware Failures (RHFs), which arise from various factors related to the environment and hardware. The chapter presents the common classification of these failures, into transient and permanent error, and reviews different fault-tolerance techniques that can be applied following a fault-tolerance approach.

Additionally, the chapter examines the two programming languages used in this thesis work, comparing C and Rust, presenting their strengths and their criticalities in embedded system applications.

In the end, the ISO 26262 standard is described, which provides safety guidelines for automotive embedded systems throughout their lifecycle. The discussion includes risk assessment methodologies, the Automotive Safety Integrity Level (ASIL) classification, and the role of Failure Modes Effect and Diagnostic Analysis (FMEDA) in ensuring compliance with safety requirements.

This chapter sets the foundation for the thesis by analyzing key safety concerns and mitigation strategies in embedded systems, with a particular focus on fault tolerance and programming language selection for reliability and security.

2.1 Embedded System in Safety-Critical Domains

In the recent years embedded system noted an increase in applications in various fields due to their contribution to the safety-criticality of the system and in terms of cost.

Embedded systems play a crucial role in various industries; in the automotive sector, they power Advanced Driver-Assistance Systems (ADAS), enhancing vehicle safety, efficiency, and the progression toward autonomous driving. The aerospace industry relies on these systems for both safety and mission-critical operations, demanding exceptional fault tolerance and precision. In healthcare, embedded systems are integral to diagnostic and patient monitoring devices, ensuring compliance with strict safety standards to protect patient well-being. Similarly, in consumer electronics, their reliability influences user experience, supporting applications from smart home technology to personal wearables.

A defining characteristic of embedded systems in safety-critical applications is their strict adherence to real-time constraints. These systems must respond promptly to external input and generate accurate outputs within a predefined time frame. Any delay or incorrect response can lead to catastrophic consequences, such as delayed airbag deployment in a car accident or failure in an aircraft's control system during flight. The margin of error is minimal, making fault-tolerant design and real-time processing fundamental to their operation.

To mitigate potential risks, embedded systems in safety critical domains incorporate fail-safe mechanisms to detect, isolate, and recover from faults before they escalate to system-wide failures. These hardened items include redundancy, watchdog timers, and self-diagnostic capabilities, ensuring continued operation even under adverse conditions.

To prevent this outcome, the automotive industry applied strict standards as ISO 26262 which follows the entire product life of E/E components in the automotive field, covering everything from initial design and development to maintenance and decommissioning. In addition, the other fields present similar standards, with norms related to their respective fields.

2.2 Random Hardware Failures and their Implications

Electronic components during their lifetime are prone to failures with a rate described in the graph in figure 2.1. The graph is called *bathtub graph*, because it remembers the longitudinal section of a bathtub.

The graph represent the overall life cycle of an item and failure rate over time, giving a graphical representation of reliability of the item. The red line represents the early life mortality, errors that can occur when the item starts its operative life. These failures can be produced by manufacturing defects, installation issues, from design or material defects. Tests as the burn-in test can reduce the possibility of early life faults, removing faulty components before they reach the market.

The yellow line represents the rising rate of wear-out failures rate, given by the aging of the component.

In green there is the representation of failure rate during the operation life, it is constant and with a low rate, usually these faults depend on the functioning of the item in critical environments.

The union of these three graph returns failure rate over time of the component, this work of thesis will consider mostly the possibility of faults during the normal life of the component, represented in the green line.

Random Hardware Failures (RHF) are faults that can affect the hardware part of the embedded system. They can be found in each part of the life-time of the system and be a result of various causes, so their probability can be represented by the green line of the graph, the constant (random) failure rate.

The most common causes are the aging of the hardware component with its degradation, differences in the power supply, internal or external noise, and for space applications the exposure to cosmic rays and alpha-radiation can also be a source of faults [12]. All of these factors during the lifetime of the application can be causes of RHFs.

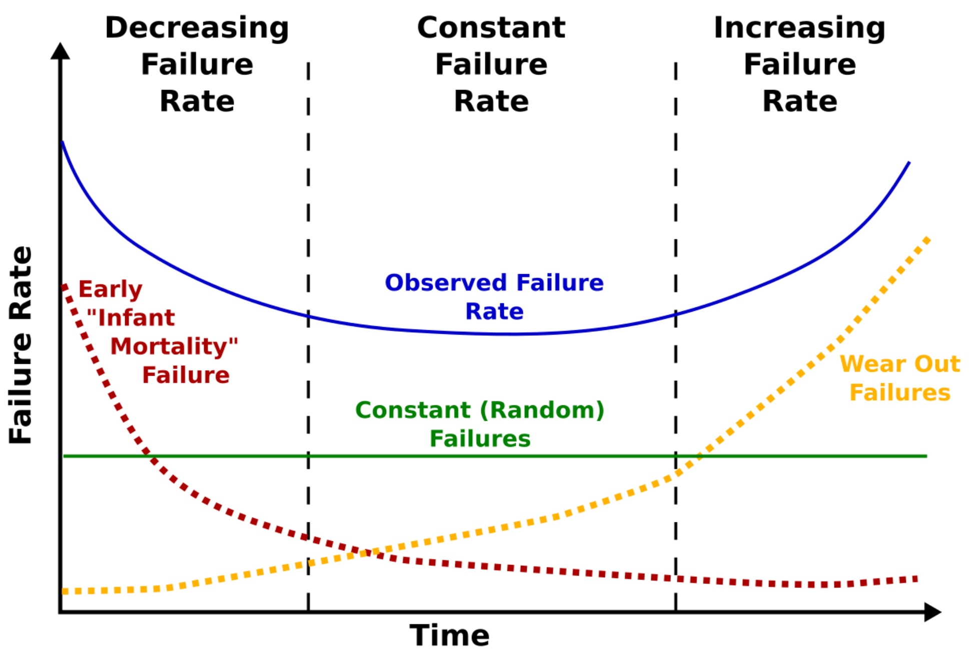


Figure 2.1. Bathtub curve, in red the rate of failure in early stage of life of the component, in yellow the rate of failure produced by aging while in green the constant failure, random. The union of these three lines produce the blue one [2].

The type of errors that can affect the hardware can be classified into two main groups: transient error, intermittent errors, caused by signal interferences, and permanent error as can be seen in Figure 2.2.

Transient errors, or "*soft error*", such as bit flip, are caused by the environment in which the embedded system operates. They are caused by a change of a bit value in a transient time without any permanent damage to the silicon component.

Permanent errors, or "*hard error*", such as stuck-at, present a permanent change in the value of the bit from the moment the fault happens and for the entire lifetime of the application.

To better explain what type of fault can affect the item a briefly explanation of what a Single Event Effect (SEE) is provided; SEE can be seen as the result of the interaction between the item and a radiation that alters the system. It can be a transient event or a permanent one, in which the fault produces a permanent alteration of the hardware structure of the item.

Transient SEE, instead, does not change permanently the structure but only the value of a bit for a transient time. However this can produce a fault.

Single Event Upset (SEU) are a subset of SEE, where the value stored in the memory cell or in the register changes its value and it can affect multiple parts of the processor, as shown in Figure 2.3.

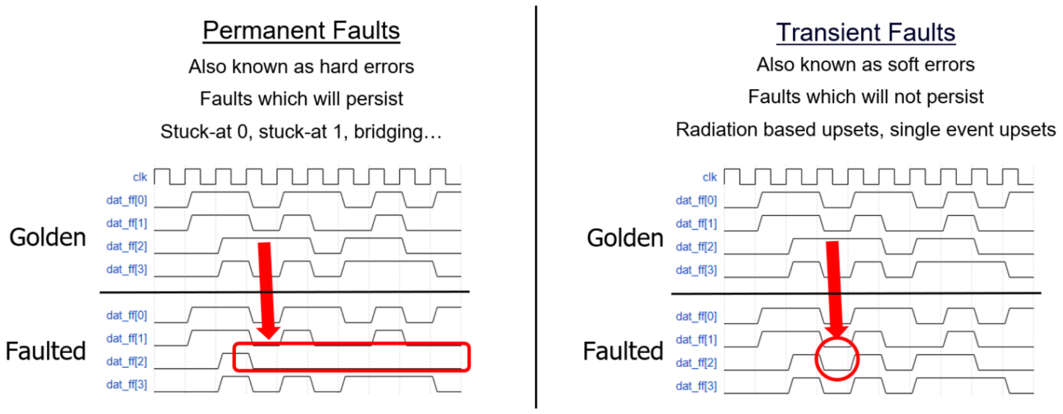

Permanent Faults
Also known as hard errors
Faults which will persist
Stuck-at 0, stuck-at 1, bridging…
Transient Faults
Also known as soft errors
Faults which will not persist
Radiation based upsets, single event upsets
Golden
Faulted
clk
dat_ff[0]
dat_ff[1]
dat_ff[2]
dat_ff[3]

Figure 2.2. Visual example for a permanent fault versus a transient fault [3].

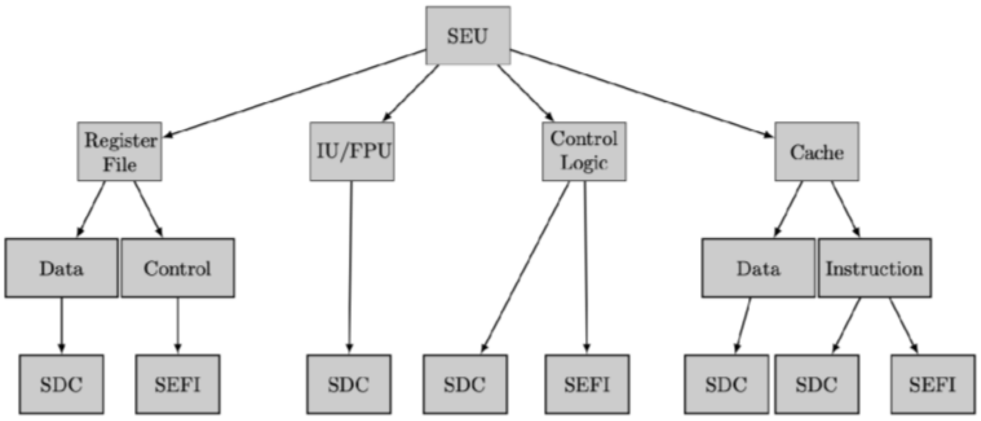


Figure 2.3. All the part of processor that can be affected by SEU [4]

The impact of faults can be various [4]:

- It can induce crash of the program and an incorrect termination of it.
- It can produce incorrect output and results.
- It can undermine the performance of the system, producing a degradation.
- It can increase security vulnerability and produce data corruption.

To overcome the possibility of failures that can prevent the right outcome of the application, hardening techniques are applied to the item with a fault tolerance approach.

When developing the application with a fault tolerance approach, the possibility of faults happening during the lifetime is taken into account, and the system is designed so that it delivers the correct operation of the system despite the faults.

Modern safety critical application are developed around three main focus points: (i) robust high-performance with power efficiency, (ii) cost effectiveness and (iii) safety and reliability. The development of the application is a trade-off between these three characteristics to obtain the best outcome applied to the field of application.

The fault tolerance approach crosses all these core points, because it provides a higher safety standard but can undermine performances, and some types of fault tolerance techniques that involves hardware redundancy can increase the cost of the system. The type of fault tolerance applied to the system depends on the criticalities that can happen to it, usually hardware solution are chosen for high critical system that can result in loss of life or damaging high value pay-load.

To select the most suitable fault tolerance method to apply to the item, plenty factors must be taken into account, such as:

- *Application criticality*: for component with application in safety-critical application, such as aerospace, medical field, or components in the automotive, with high relevance for system safety, the choice falls on hardware redundancy.
- *Resource availability*: embedded systems are often selected for their ability to perform even with restrictive constraints on memory and computational performance. For these reasons, usually it is not possible to apply some fault-tolerant techniques to them because these constraints won't be satisfied.
- *Performance overhead*: The embedded system in a safety-critical application must provide the result in a specified time, deadline. The introduction of a fault tolerance system can increase the time needed by the application to provide a result, increasing the performance overhead.
- *Cost*: Embedded systems are often chosen for they low cost, but some types of fault tolerance techniques can increase their cost.
- *Scalability and Flexibility*: software-based fault tolerant techniques are preferred with respect to the hardware-based ones, especially in system that are upgraded or modified often. This because it is possible to modify them more easily.

The selection of the best technique to obtain a fault-tolerant system depend on many of factors.

To obtain fault tolerant system it is possible to use a Design-Diversity approach. It can be applied both by hardware or software. For hardware, the same component or calculation is executed on different hardware and then the results obtained are compared to check if they are equal between them. For software solution, usually it is obtained by writing the same program code by different teams with different developers to ensure diversification of the item. However, this method can be costly, both for hardware and software, and not suitable for some of the embedded system applications, especially when the cost cap is relevant.

Instead of relying on diversification, Single-Design fault tolerance approach is based on redundancy. It can be applied to the hardware, to the software, to the execution time, or in hybrid methods that take into account more solutions.

Hardware redundancy is the most common and consists of adding more hardware components that can detect and prevent the expansion of the failure in the system. It usually consists of three stages: detection of the fault, isolation of the faulty component, and recovery of the application. Hardware redundancy can be done using an active, passive, or a hybrid redundancy.

Active hardware redundancy presents the same software applied to different hardware to have a result ready in case an error occurs. The most common methods are [16]: duplication with comparison, where the same software is applied to more hardware and the results are compared between them to check if they are consistent and equal. If this does not happen, the system alerts the user of a possible fault. Another active redundancy is reserve ready, where another hardware module is ready to take the place of the faulty component. It can be more or less fast depending on the type of standby that is chosen for the reserve component.

The most common passive hardware redundancy technique is the N-modular or M-of-N system, where the mechanism is based on voting and masking the fault with a polling system. This passive redundancy method strongly depends on the voting system, which can be done by hardware or software. If the voting system fails, all the fault tolerance mechanisms do not return the expected outcome.

Moreover, a hardware redundancy system produces hardware overhead, which also increases the cost of the embedded system, which is not advised in some cases.

Another way to introduce redundancy for fault tolerance technique is time redundancy, where the same code is repeated on the same piece of hardware and the two results are compared between them. This method is not recommended for real-time applications, where output deadlines are stringent and are the most relevant feature of the system.

Lastly, it is possible to apply redundancy also with information, by software. This is the case of Control Flow Checking (CFC) [4], a method that cross-checks the run-time control flow to see if there is any violation of the application path.

CFC can handle both transient and permanent errors that affect the Program Counter (PC), which can result in an alteration of the normal flow of instructions. The main drawback about using CFC is the introduction of more instructions in the code, increasing the overhead of code size and execution time. Moreover, almost all CFC methods are less effective in detecting intra-block faults or wrong but legal transitions, such as taking the wrong "if-else" branch.

The most common CFC method is signature checking that can be integrated with hardware hardening methods such as watchdogs to perform a more reliable check. Usually, CFC methods are applied directly to the code in Assembly programming language, but this method is not compliant with the standard ISO 26262 that requires the use of high-level programming language. Additionally, applying the hardening method directly in assembly removes the portability of the application between different target hardware.

For this thesis work,two different methods of control flow checking applied to a software implemented created following a Model-Based approach will be analyzed. The description of the two techniques applied can be found in paragraph 3.2.1 and 3.2.2.

2.3 Rust vs C in Safety-Critical Applications

The evolution of embedded systems followed an increase in the use of high-level languages such as C. Historically, embedded applications were primarily developed using assembly language, which tightly coupled software to specific hardware architectures. However, to improve portability and maintainability, high-level languages such as C have become the standard for the development of embedded systems.

The C programming language offers significant advantages in terms of performance and extensive software support. At the same time, it presents critical challenges in relation to fault modes that can lead to failures. The Motor Industry Software Reliability Association (MISRA) introduced MISRA C, a list of procedures and rules aimed at standardized and facilitating code safety, security, portability, and reliability in the context of embedded systems. MISRA C introduces 93 mandatory required rules, and 34 advisory rules, recommendations, with the aim of standardizing the use of the C language in automotive applications. This standard, moreover, avoids programming techniques that can produce a source of possible faults; as not initializing a variable, incomplete type checking, or lack of an exception handling mechanism [17].

Despite these standards, C remains susceptible to critical memory management issues, including concurrency-related resource access problems and null pointer dereferences. These vulnerabilities arise due to C's manual memory management approach, which places a significant burden on developers to ensure memory safety.

To overcome the insurgence of memory issues in the application, there are many studies to validate the use of Rust as a programming language for the embedded system to replace C in applications where memory issues are relevant.

Rust is a robust programming language created by Mozilla Fundation that implements features to avoid concurrency and memory safety bugs, preventing common programming language errors. To manage the memory programmers need to use features as scope, borrowing, single ownership, and lifetimes. All of these measures applied to the program ensure the developer the avoidance of program's part with vulnerabilities in memory [18] [7] [19]. Rust takes into account that for some applications, its requisites are too stringent, so it is possible to have unsafe regions where the management of the memory is left more to the developer. However, embedded crates, software applications for embedded systems, have a higher use of unsafe regions compared to standard applications developed in Rust.

2.4 Standard ISO 26262

The ISO 26262 standard has its origin in IEC 61508, the international standard published by the International Electrotechnical Commission (IEC) on the life of electronic components. ISO 26262: "Road vehicles - Functional safety" has two editions, the most recent was published in 2018 and presents 12 sections [20]. The standard presents guidelines for the entire life cycle of the electronic/electric component in a vehicle, from the concept phase to the delivery in the market. The main focus is to prevent dangerous situations and systematic errors that can arise during the development of the item. In addition, the guidelines guarantee safety even with the insurgence of RHFs.

The *first part* introduces the vocabulary used in the guidelines, as the terms item to refer to what is in focus to develop, but also the definitions of Hazard, potential source of harm, and Failure, the wrong and/or unexpected outcome of the item. The *third part* introduces the concept phase, where at the end a document is produced that contains all the requirements for safety of the item, the Functional Safety Concept (FSC). The main focus of the concept phase is Hazard Analysis and Risk Assessment, where all possible hazard causes are analyzed, and all Safety Goals (SGs) are defined with their Automotive Safety Integrity Level (ASIL). The ASIL level results from a matrix composed of other indexes related to the SG analysis and can go from the minimum level QM (quality measure) to a scale from A to D where A is the minimum and D is the most important. The definition of ASIL level starts with the definition of the other three indicators:

- *Severity*, from 0, no injuries caused, to 3, where life-threatening or fatal injuries can occur.
- *Controllability*, from 0, controllable in general, to 3, the item is completely out of control.
- *Exposure*, from 0, where the probability of the event are very low, to 4, where the operational situation is always happening.

These indices combined return the ASIL related to the hazardous event.

The *fifth part* is focused on the development at the hardware level and returns the Failure Modes, Effects, and Diagnostic Analysis (FMEDA). In this part, it is possible to find all the hardware safety requirements and how to design the hardware to prevent systematic faults that can affect it. Moreover, it takes into account the safety goals violation due to the RHFs. The FMEDA starts from the Failure Mode (FM), and if a failure is detected in every circumstance, it is possible to define the Diagnostic Coverage (DC).

The Diagnostic Coverage compliant with the ISO 26262 request to define two classes, "detected" where the fault is detected and "undetected". In addition to the two ISO 26262 classes, a third is added, "false positive", where the system detects a fault where it is not injected. This third class, for a well-designed application, should be around 0%.

The two classes, "detected" and "undetected", can be split in two more subclasses; for the "detected" one, the outcome can be "safe" if the RHF cannot have a dangerous effect on the user or the application environment, or "detected" when it is not possible to make any assumption, like when no mitigation strategies are applied.

For the "undetected" class, the two subclasses are "latent" if the RHF does not affect the result of the application and "residual", where the fault has been injected but not detected by the system and it is possible to have the wrong result. More information on diagnostic coverage matrices is in the section 4.1.

Chapter 3

Methodology

In this chapter, it will be analyzed all the previous steps to obtain the code of the application used as benchmark for this work of thesis. In the first section, the Model-Based Software Design approach will be described. It was used for developing application for its compliance with the international standards.

Then, there is an insight of the Control Flow Checking techniques, the main features of the Control Flow Graph (CFG) and the algorithm to develop the two CFC techniques employed in this thesis work.

The last part is dedicated to the experimental setup, describing the testing bench, the type of faults that is possible to inject, and the procedure to perform the fault injection campaign.

3.1 Model-Based Software Design (MBSD)

Model-Based Software Design (MBSD) is a well established approach for developing an application of a complex system using a simplification of it, called a model. This approach uses a graphical language to develop the application instead of relying on a traditional high-level programming language to obtain the same result.

The model is an abstraction of the original system and retains the main characteristic and functionalities that are useful for the development process. The graphical representation of the model enhances the functionalities of the item or its Finite State Machine (FSM).

For this work of thesis, Mathworks environment was employed as the graphical environment used for developing the benchmark, especially the graphical programming environment Simulink. To develop the FSM, the control logic tool Stateflow was used, also provided by Mathworks. Mathworks environment was selected for its compliance with the MBSD paradigm.

3.1.1 Case Study

For this work of thesis, an application that simulates the behavior of an emergency post-collision brake system was developed.

This system was introduced by Ford as standard in 2019 and this system can be found in many other vehicles' from different car manufacturers, such as KIA, Volvo, SEAT and Ford itself. [21] [22] [23] [24].

The Post-Collision Braking system increases the pressure applied to the brake pedal to stop the vehicle when a collision is detected, using the already preexistent airbags and fuel cutoff sensors. With this method, it is possible to avoid secondary collisions with other vehicles, objects such as a light pole, structures, or pedestrians, reducing the total damage caused by the crash.

This system was selected as benchmark because it could be developed with a Finite State Machine approach, for its simplicity and also because it is safety relevant. In fact, after performing Hazard Analysis and Risk Assessment (HARA) over the item, it was classified as ASIL B, so it is compliant to a fail-safe approach and to the implementation of hardening techniques.

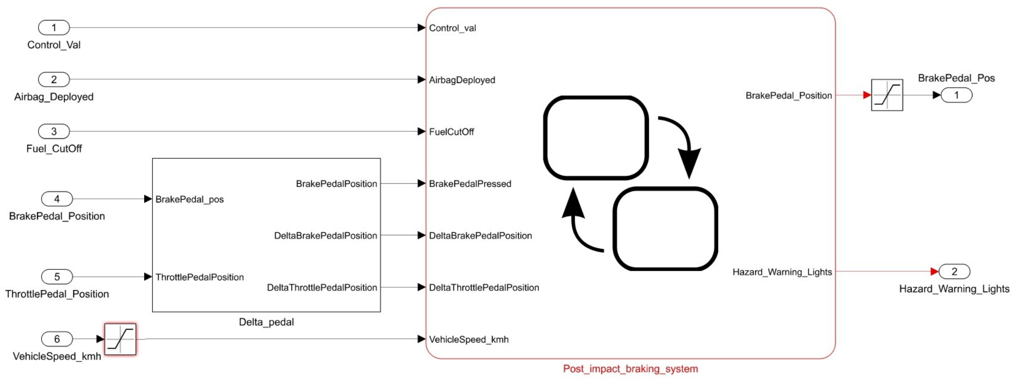
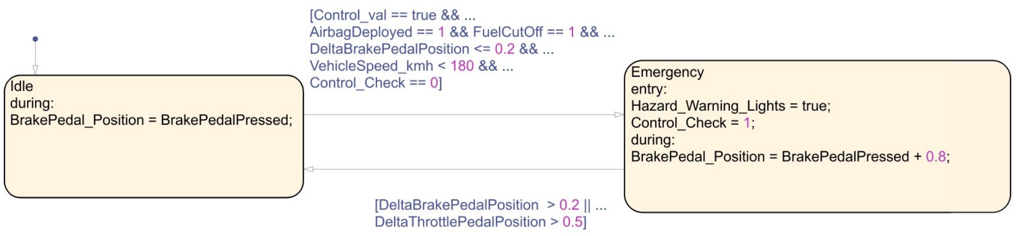


Figure 3.1. The system developed in Simulink environment



```
Idle
during:
BrakePedal_Position = BrakePedalPressed;
```

```
[Control_val == true && ...
AirbagDeployed == 1 && FuelCutOff == 1 && ...
DeltaBrakePedalPosition <= 0.2 && ...
VehicleSpeed_kmh < 180 && ...
Control_Check == 0]
```

```
Emergency
entry:
Hazard_Warning_Lights = true;
Control_Check = 1;
during:
BrakePedal_Position = BrakePedalPressed + 0.8;
```

```
[DeltaBrakePedalPosition  > 0.2 || ...
DeltaThrottlePedalPosition > 0.5]
```

Figure 3.2. The Stateflow chart with the two states of the system

The application was developed employing the Mathworks Simulink graphical language.

The solver is set as fixed-step with size of 0.05 seconds.

The main system contains as input airbag and fuel cutoff sensors, represented as Boolean variables. Moreover, there is a variable `Control_Val` that represents a check

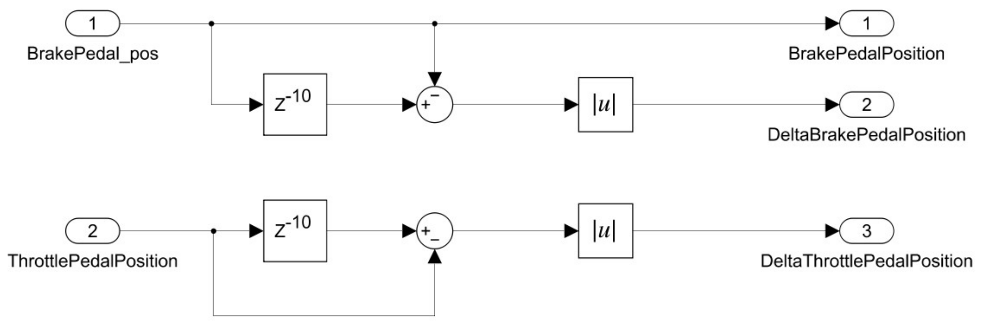


Figure 3.3. The subsystem that calculates the variation of the pedal

if the other systems on which it depends are functioning. The other inputs are the `Vehicle_Speed` in [Km/h], it is necessary because the system does not activate if the speed is higher than a certain threshold, in this case fixed at 180 Km/h. The other three inputs are variables of type single, that contains the position of the brake pedal and the variation of the brake and throttle pedal. The first step of the system in represented in figure 3.1.

The delay, represented in the subsystem 3.3, calculates the difference in position of the throttle pedal and brake pedal. It was selected a delay of 10 time-steps, so every 0.5 seconds. This delay in the system is necessary because the item returns the control of the vehicle to the driver if it detects a variation in the brake pedal or throttle pedal position when the crash is detected.

Inside the main subsystem block there is the Stateflow chart, in figure 3.2. It has two states, one "idle" where stays there almost all the time and switches to an active state when the conditions are satisfied, so in this case, when sensors perceive a crash and the vehicle speed is below the threshold.

This system was then tested inside the Simulink environment simulating a driver and the vehicle plant with the longitudinal model of it, to see if the system behaves as required.

3.1.2 Generating C and Rust Code

After developing the system on Simulink as described in the previous section, the code was generated from the Simulink model using the add-on provided by Mathworks environment as Embedded Coder and Simulink Coder.

The Embedded Coder, an extension provided by Mathworks, was used to obtain the source code directly from the Simulink model in the C programming language, specifying both the programming language to be translated and the target platform, in this case RISC-V R32I.

The same procedure was applied to obtain the Rust code. Starting from the baseline of the C code generated from the Embedded Coder, it was translated into Rust programming language keeping the characteristic of the code generated through the Embedded Coder and integrating the restriction about memory, characteristic of Rust programming language.

The translation from C into Rust programming language was made by hand, even if it is possible to find some valid items to translate the code such as C2Rust [25] by Galois and Immunant and Corrode [26] by Jamey Sharp. It is possible to find the repositories of these two translating applications on GitHub, and in the community of Rust users are the ones used more to translate preexistent C application in Rust but, however, these methods are not perfect and requires the human intervention to fix the code.

For this reason and because the code to be exported in Rust was not too long, it was preferred to translate it by hand, avoiding errors produced by an erroneous use of the translating mechanisms.

3.2 Control Flow Checking Techniques

Control Flow Checking (CFC) is a SIHFT method that applies information redundancy to cross-check the runtime control flow of the code. To perform any CFC methods beforehand, it is needed to obtain a Control Flow Graph (CFG).

Control Flow Graph is an oriented graph obtained from the high-level language source code. The two main components are: Basic Block (BB) and the associated transitions.

Basic Blocks [14] are the maximal set of ordered instruction that runs sequentially. They do not have inside any jump or transition, the only exception is the last line of instruction that indicates the next Basic Block in execution. BBs correspond to the vertices of the graph, $V = \{v_1, v_2, ..., v_n\}$. The transitions between Basic Blocks that correspond to the edges of the graph, $E = \{br_{ij} | br_{ij}$ correspond to the flow from v_i to $v_j\}$. The Control Flow Graph (CFG) of a program can be defined as $G = \{V, E\}$. Each transition not included in E is considered illegal, so, if detected, it corresponds to a Control Flow Error (CFE).

An example of Control Flow Graph for a small C code is represented in figure 3.4.

Below are listed six different situations where the program, for an error, executes a jump from a BB to another that result in a Control Flow Error:

- An illegal jump from the end of one BB to the beginning of another BB.
- a legal but incorrect jump from the end of one BB to the beginning of another BB.
- a jump from the end of one BB to any point within another BB.
- a jump from any point within one BB to any point within another BB.
- a jump from any point within a BB to another point within the same block.
- an illegal jump from a BB to the unused space of memory, which refers to the space between BBs

The most widely used CFC methods are those that implement signature checking. There can be different types of signature, given by a single bit or random number, but overall the logic behind this type of hardening implementation is the same and consists of two phases: SET and TEST. The SET phase updates the signature of the Basic Block, while the TEST checks that the signature is the one expected by the control graph, so if the predecessor is a legal one included in the CFG.

In this work, two different CFC are used for their characteristic: "Yet Another Control-Flow Checking using Assertion" (YACCA) for its simplicity and the possibility to be applied in C and also in Assembly, using a bit field power of 2 signature, so for the BB 15, bit 15 is set to 1. This method raises the criticalities when the CFG contains more than 64 BBs.

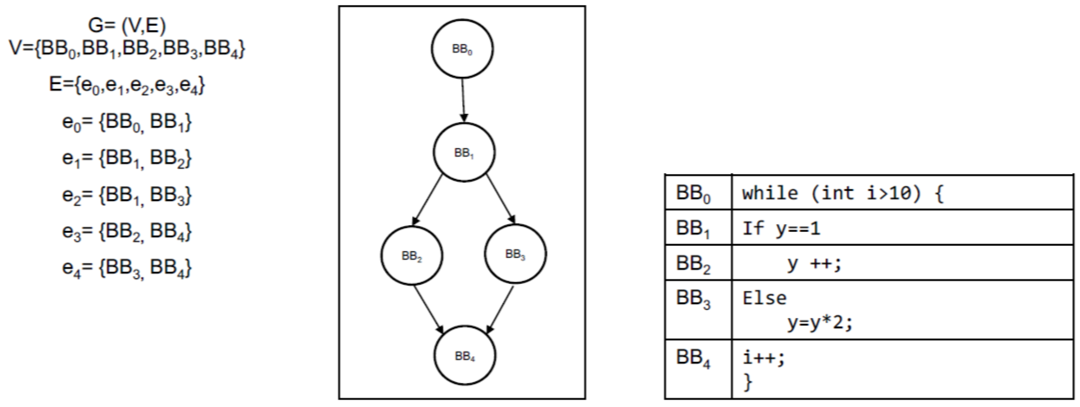
The other CFC hardening method is the "Random Additive Control Flow Error Detection" (RACFED). It uses random numbers as signatures, overcoming the constraint on the number of possible BB in YACCA. Moreover, RACFED can detect intra-block CFE.

3.2.1 YACCA

Yet Another Control-Flow Checking using Assertion, (YACCA) is a method to check the correct flow of execution in the application using assertion over signatures.

Each BB entry has a unique signature; before executing the instructions inside the BB the program checks if the predecessor mask belongs to a legal one. Then, after executing the code, before exiting the BB the CFC assigns the new signature. If the program jumps from an BB that is not a legal predecessor, the CFE is detected.

YACCA uses two variables to perform the CFC: *ERR_CODE* and ID_s. Before starting the program, *ERR_CODE* is set to 0 and ID corresponds to the signature of the first BB executed.


$G = (V,E)$
$V = \{BB_0, BB_1, BB_2, BB_3, BB_4\}$
$E = \{e_0, e_1, e_2, e_3, e_4\}$
$e_0 = \{BB_0, BB_1\}$
$e_1 = \{BB_1, BB_2\}$
$e_2 = \{BB_1, BB_3\}$
$e_3 = \{BB_2, BB_4\}$
$e_4 = \{BB_3, BB_4\}$

BB_0	`while (int i>10) {`
BB_1	`If y==1`
BB_2	`    y ++;`

BB_3	`Else`
	`    y=y*2;`

BB_4	`i++;`
	`}`

Figure 3.4. Example of a CFG in C [5]

Entering into the new BB the system performs the TEST operation, written in Algorithm 1 3.1. The program checks if the variable ID_s corresponds to ID. If yes, the execution of the program proceed normally;, otherwise, the CFE is detected and the ERR_CODE is increased of 1.

The SET operation, written in 3.2, is executed before exiting the BB, 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 flow of the program is correct, the ID_s will be all set to 0 by the AND operation and the signature can be set to the ID of it's legal successor by OR and the bit-wise operation.

This method cannot detect intra-block illegal jumps and criticalities can rise when the Control Flow Graph has more than 64 BBs.

```
Algorithm 1: TEST operation (YACCA)
1: TEST($RTS$, predecessor_mask)
2: if $RTS \wedge (\neg\ predecessor\_mask)$ then
       CFE detected
       end
3: Continue normal execution
```

Table 3.1. Algorithm to do TEST operation in YACCA [8].

```
Algorithm 2: SET operation (YACCA)
1: SET($RTS$, $predecessor\_mask$, $BB\_ID$)
2: $RTS = RTS \wedge \neg predecessor\_mask$
3: $RTS = RTS \vee (1 << BB\_ID)$
```

Table 3.2. Algorithm to do SET operation in YACCA [8].

In the listings 3.1 and 3.2 are represented in order the function in C to test and set the bitwise mask of each basic block. The part used in the debugging was also included in the code reported below. It was used to check whether false positive errors were declared after the implementation of the hardening method.

In the listing 3.4 and 3.3 is presented the implementation in Rust. It is possible to notice how the two codes in C and in Rust are very similar because both are high-level languages, while the main difference is in the use of unsafe regions in Rust for handling the change of property of the variable.

```c
void YACCA_TEST(uint64_T predecessors_mask) {

    last_predecessor_mask = predecessors_mask;
    ERR_CODE = ERR_CODE + ((ID & (~predecessors_mask)) >0 ? 1:0);

    if (ERR_CODE!=0) {
        ERR_CODE=0;
        #if (DEBUG_PRINTS == DEBUG_ENABLE)
          printf ("ERR_CODE=               ");
```

```
            printf("%llu\n",predecessors_mask);
        #endif
    }
}
```

Listing 3.1. YACCA TEST operation in C

```
void YACCA_SET(uint8_T BB_ID) {
  if(BB_ID < 63) {
    ID=(ID& (~ last_predecessor_mask)) ^ ((uint64_T) 1<<BB_ID);
  } else {
    ERR_CODE = 255;
    #if (DEBUG_PRINTS == DEBUG_ENABLE)
      printf("BB with an ID > 63!!!\n");
    #endif
  }
}
```

Listing 3.2. YACCA SET operation in C

```
fn YACCA_TEST(predecessors_mask : u64){
    unsafe{
        last_predecessor_mask = predecessors_mask;
        ERR_CODE = ERR_CODE + if ID & (! predecessors_mask ) > 0 {1} else
    {0};
    }
    let ERR_CODE_local = unsafe{ERR_CODE};
    let ID_local = unsafe{ID};
    if ERR_CODE_local > 0 {
        println!("Failed YACCA TEST for mask {predecessors_mask} when ID
    is {ID_local}");
    }
    if ERR_CODE != 0{
        ERR_CODE = 0;
    } */
}
```

Listing 3.3. YACCA TEST operation in RUST

```
fn YACCA_SET(BB_ID : u8){
    if BB_ID < 63{
        unsafe{
            ID = (ID & (! last_predecessor_mask)) ^ (1<<BB_ID);
        }
    } else {
        unsafe{
            ERR_CODE = 255;
        }
    }
}
```

Listing 3.4. YACCA SET operation in RUST

3.2.2 RACFED

Random Additive Control Flow Error Detection (RACFED), uses random numbers to assign a signature to each BB and it is possible to detect intra-block Control Flow Error

(CFE) and also inter-block CFEs. Checking the signature with RACFED is composed by four steps:

a) At compiler time, to each BB is given two random values, a Compile Time Signature (cts) and a SubRanPrevVal, the sum of these two values must be unique. The sum of these two values returns the first signature to be manipulated.

b) The instructions monitoring done by RACFED is given by adding a random number to the compile time signature (cts) in each BB with three or more instructions. This monitoring aims to detect intra-block CFEs, so skipping some instructions inside the same BB.

c) To check whether the BB is the correct successor, from the compile-time signature the SubRanPrevVal value is subtracted and checked with the value stored in memory, as shown in 3.3. Then, after execution of each line of code inside the BB, a random value is added to the compile time signature.

d) At the end of the basic block, an adjust value is computed subtracting to the successor signature, given by the sum of the compile-time signature and the SubRanPrevVal, the sum of the predecessor compile-time signature, and the sum of all the random values added in the step before. The adjust value is then summed to the actual signature to obtain the new signature of the next block of instructions.

This method increases the overhead of the code, but is very effective in detecting illegal jumps inside the same block, when it contains more than three instructions.

Algorithm 3: TEST operation (RACFED). *bb* represent 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
```

Table 3.3. Algorithm to do TEST operation in RACFED [8].

Algorithm 4: SET operation (RACFED). *bb* represents the current BB, while bb_{+1} its expected successor. subRanPrevVal and CTS are arrays (with length 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
```

Table 3.4. Algorithm to do SET operation in RACFED [8].

In the listings 3.6 and 3.5 are represented in order the function in C to test and set the bitwise mask of each basic block. The part used in the debugging was also included in the code reported below. It was used to check whether false positive errors were declared after the implementation of the hardening method.

In the listing 3.8 and 3.7 is presented the implementation in Rust. It is possible to notice how the two codes in C and in Rust are very similar because both are high-level languages, while the main difference is in the use of unsafe regions in Rust for handling the change of property of the variable.

```c
void CheckSig(int32_T current_mask){
          if  (signature!=cts[current_mask]){
                ERR_CODE=current_mask;
                #if (DEBUG_PRINTS == DEBUG_ENABLE)
                printf ("ERR_CODE=");
                printf("%d\n",current_mask);
                #endif
        }
}
```

Listing 3.5. RACFED TEST operation in C

```c
// initialization of the SET operations
void UpdSigBegin (int32_T current_mask){
    signature =signature - subRanPrevVal[current_mask];
}

// adding random number when Basic Blocks with more than three
   instructions
void AddRand(int32_T rand){
    signature=signature + rand;
}

// final part of setting the signature in RACFED
void UpdSigEnd(int32_T current_mask,int32_T successor_mask){

    adjustValue= - (cts[current_mask]+sumBB[current_mask] ) + (cts[
   successor_mask]+subRanPrevVal[successor_mask]);
    signature=signature+adjustValue;

    #if (DEBUG_PRINTS == DEBUG_ENABLE)
        printf ("CurrentBB=%d\n",current_mask);
        printf ("signature=%d\n",signature);
        printf("BB=%d Zero=%d\n",current_mask,signature- (cts[current_mask
   ]+sumBB[current_mask] ));
    #endif
}
```

Listing 3.6. RACFED SET operation in C

```rust
fn CheckSig(current_mask: usize){
    unsafe{
        let current_signature = SIGNATURE;
        if (current_mask < CTS.len() && current_signature != CTS[
   current_mask] as i32){
            ERR_CODE = current_mask as i32;
            println!("Failed YACCA TEST for mask {current_mask}");
        }
    }
}
```

Listing 3.7. RACFED TEST algorithm in Rust

```
fn UpdSigBegin(current_mask: usize){
    if(current_mask < CTS.len()){
        let prev_val = SUB_RAN_PREV_VAL[current_mask];
        unsafe{
            let current_signature = SIGNATURE;
            SIGNATURE = current_signature - prev_val;
        }
    } else {
        unsafe{
            ERR_CODE = 1000;
        }
    }
}

fn AddRand(rand: i32){
    unsafe{
        let current_signature = SIGNATURE;
        SIGNATURE = current_signature + rand;
    }
}

fn UpdSigEnd(current_mask: usize, successor_mask: usize){
    if(current_mask < CTS.len() && successor_mask < CTS.len()){
        let adjust_value = -(CTS[current_mask] + SUM_BB[current_mask] as
    i32) + (CTS[successor_mask] + SUB_RAN_PREV_VAL[successor_mask]);
        unsafe{
            let current_signature = SIGNATURE;
            SIGNATURE = current_signature + adjust_value;
        }
    } else {
        unsafe {
            ERR_CODE = 1000;
        }
    }
}
```

Listing 3.8. RACFED SET algorithm in Rust

3.3 Experimental Setup

Safety-critical system must be compliant with international functional safety standards, the system must be developed to be reliable.

Fault models are described by *what* inject, the type of fault, *when* inject, so determine the Fault Injection Time, and *where* inject the fault, choosing a bit position that is relevant to the scope of the research. For example, the CFC can detect only errors in the flow of instructions and cannot detect a fault that corrupts the value of a variable.

Fault models, following the description proposed in paper [6] are composed by a *controller*, that generate the commands that are feed to the *load generator*, that produces the stimuli that act as input for the target input. The controller is also given to the *monitor* and to the *injector*. The latter reads a *faultload* library and generates the fault description. The last component of a fault model is the *monitor*, that logs the readouts of the campaigns.

For this work of thesis, Quick EMUlator (QEMU) [27], an open source machine emulator and virtualizer used for CPU emulation, was used. This approach makes the test bench agnostic with respect to the specific instruction test because it can simulate different Instruction Set Architecture (ISA).

QEMU was used together with Gnu Debugger (GDB), allowing the Fault Injection Manager (FIM) to interact with the simulation environment.

In the following sections, a better description of the Fault Injection Manager, the type of fault injected and the overall functioning of the test bench is provided.

3.3.1 Fault injection Strategy and Tools

Fault Injection (FI) campaigns are a fundamental step in verifying the behavior of the system when a random hardware failure affects the program. FI campaigns can be performed by hardware and physical components, i.e. taking the target for which the application is developed and performing fault injection, with an external power supply, a neutron beam, or modifying the voltage or current of the item. These methods are costly and require a large number of specimens to perform the fault injection campaign.

However, these types of campaigns performed directly on the hardware are needed to determine the probability distribution of faults on the hardware.

In this work of thesis, a test bench, described in detail in [6], compliant with ISO 26262 standard will be used to perform fault injection into the Program Counter (PC) of the target application.

3.3.2 Types of Faults Injected

The faults injected into the program are only faults that CFC can detect, so the one which affects the standard flow of the application. CFC can detect only errors in the flow of execution of the program, not if a value assumes another value erroneously or if the program takes a conceptually wrong path but present in the CFG, for example, a wrong branch in the "if-else" decision.

To perform fault injection, two types of fault are injected into the Program Counter (PC): *Permanent* and *PermanentStuckAt*.

Both failures are bit-flip where the bit remains stuck at 0 or 1 from the time the fault is injected to the end of the execution. Permanent fault attacks only a bit in the target register, while PermanentStuckAt affects the whole register.

For this work only Permanent faults are injected into the system, moreover it is possible to choose only a subset of bit where to apply the fault. With this precaution, it is possible to avoid the fault exiting the memory space and being detected by HW traps.

3.3.3 Description of the test bench

The description of test bench [6] used in this work thesis is described below and a graphical representation of it is given in figure 3.5:

1. GCC, GNU Compiler Collection, was selected to compile the file from the source code for the target platform. In this case a RISC-V RV32I microcontroller, since it is an open-source ISA and it is raising interest in various fields, automotive and aerospace. The cross-compilation produces an *.ELF* file, which contains the machine-executable code and the memory map.
2. A setting file is prepared, containing information on the campaign to be executed.
3. The setting file is fed to the Fault Injection Manager (FIM), which reads it and generates all the scripts into the CMD file needed to perform the fault injection campaign.
4. The scripts are launched and perform the golden run, the run without any faults, and then all the runs with the fault injected. For each run, a log file containing the output of the program is saved.
5. The classifier is compiled and run, taking the golden run and each log of the faulty run as input.
6. The results of the classifier are converted into ISO 26262 compliant classification.

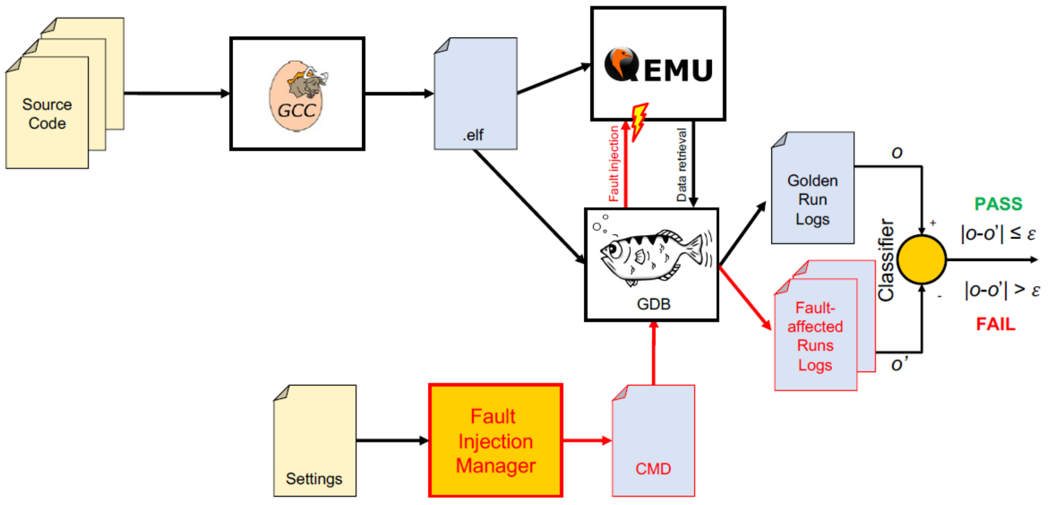


Figure 3.5. The fault injection methodology for application developed in C, Figure from [6]

The fault injection for the Rust implemented application, depicted in figure 3.6, uses almost the same items used in the test bench represented in figure 3.5 while the main difference is the use of `Cargo`, that is the compiler developed specifically for Rust.

The main component of a fault injection campaign is the Fault Injection Manager (FIM). Its structure is hierarchic, describing the campaign, formed by the watches, so the variables that must be monitored in the campaign, and the faults to be injected

A *watch* is described by:

- Symbol: the name of the variable to be watched

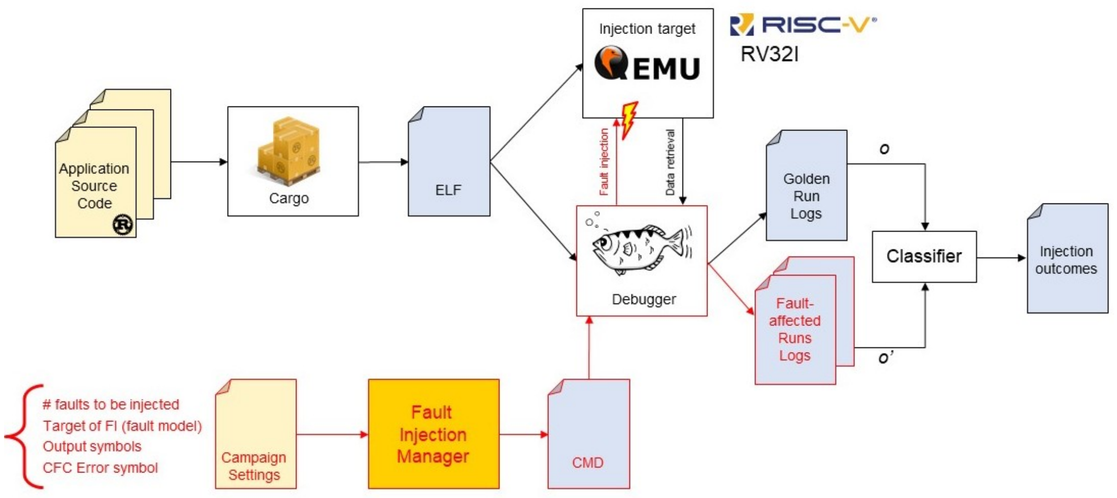


Figure 3.6. The fault injection methodology for application developed in Rust, Figure from [7]

- Address: the address of the memory to be watched.
- Description: a description of the watch, needed for the report
- DetectionWatch and DetectedCondition: if DetectionWatch is true, it means that the variable contains the result of a RHF detection mechanism. The DetectedCondition contains the condition that represents whether the detection occurred in relation to the DetectionWatch variable.

The *fault* is described in the setting file by:

- Type: the type of fault injected in the campaign, in this case Permanent
- Target: the name of the target register
- `bitPositionMask`: a 64-bit mask, used to perform bitwise-level configuration of the fault.
- Fault Tolerance Time Interval (FTTI): maximum time allowed to the system from fault injection until fault detection occurs.
- Minimum injection time: minimum time before the fault injection happens.
- Maximum injection time: maximum time to inject the fault in the system.
- Permanence time: the number of instruction executed after the fault injection if the software under testing does not set the termination condition.
- Number of injections: the number of faults to be randomly generated and injected.

The other main component of the test bench is the classifier, a file that takes as input the logs of the golden run, so the one without any faults, and the log files of the fault campaigns. It compares the outputs logs obtained during the executions, and determines if the computation result has changed after the injected faults.

3.4 Implementation and Hardening

The fault injection campaign was executed on the hardened code obtained from the embedded coder plug-in supplied by Mathworks. As described in paragraph 3.1.2, the program was created using the MATLAB Simulink graphical language environment. The C code was generated directly employing the embedded coder and then manually hardened to integrate the CFC mechanism as shown in figure 3.7.

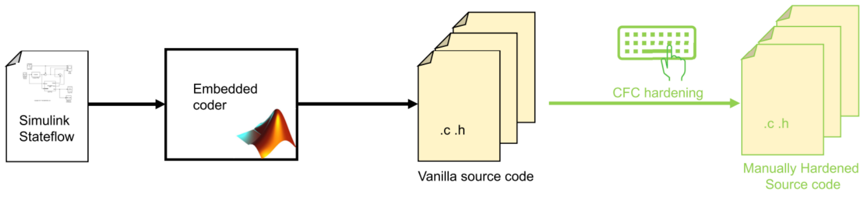


Figure 3.7. The sequence of action to generate the code following the MBSD approach

3.4.1 Integrating CFC Mechanism into Generated Code

Control Flow Checking methods are usually integrated in Assembly programming language, but this can result in error-prone and time consuming. Moreover the international standard ISO 26262 defines mandatory the utilization of high level programming language. For this, the CFC hardening was integrated directly to the code.

To integrate the CFC into the generated code, the first step is to create the Control Flow Graph (CFG). It was generated manually, identifying beforehand all the basic blocks and the possible transition between them.

After this step, it was assigned at each BBs a signature at compile-time. For the YACCA method, the signatures follow the sequential numbering of the BBs, because the signature mask is a bit shift equal to it.

For RACFED, it was randomly generated two arrays with length nineteen, of numbers between one and 255. These two arrays are the cts, compile time signature, and `subRanPrevVal`, that constitute the signature mechanism for the RACFED method. It was manually checked that all signatures were unique between them and that none of the adjVal was a negative number.

The same values for the signature were also used in the Rust code. For integrating the algorithm of SET and TEST of both YACCA and RACFED, it was used the one provided in the paper [7] and found also in the repository `https://github.com/JacopoSini/Rust4Safety_DATE2025`.

It is important for the logs of the FIM to keep the same names of the variables even after the compiler. For this reason, in the Rust code is applied the `[#no_mangle]` attribute. This attribute prevents the mangling of the name variable, avoiding problems in the generations of the log files when computing the fault campaign with the Rust programming language program.

3.4.2 Rust variable handling

The main characteristic of Rust, that is also the main reason for which it was selected for this comparison, it is the ability to handle memory in a safer way with respect to C.

For this reason every variable must be initialized before using and for variables initialized in the for loop it behaves strangely. The delay implemented in the subsystem to detect any variation of the pedals, described in section 3.1.1, is translated by the Embedded Coder in a for loop.

For loop requires special attention when implementing Control Flow Checking hardening methods, when checking the signatures of the Basic Blocks. As shown in Figure 3.8, it needs an "if-else" conditional statement to check if it is the first transition in the for-loop. If yes, so the index of the for-loop is still at 0, it checks if the signature correspond to the previous Basic Block, otherwise it checks if the previous Basic Block executed was the one inside the for-loop. The same reasoning is done at the end of the segment of code.

While in C the value of the index is retained during the computation so the final if to check which value of the index `i` does not returns any problem, when implementing it on Rust it is necessary an auxiliary variable to keep in memory the last value of the index `i` assumed, as shown in listing 3.9.

Moreover the variable `idx_delay`, defined inside the for loop is in `<usize>` type, so it needs an explicit casting as `uint32` to be then compared in the second part of the implementation of Control Flow Checking.

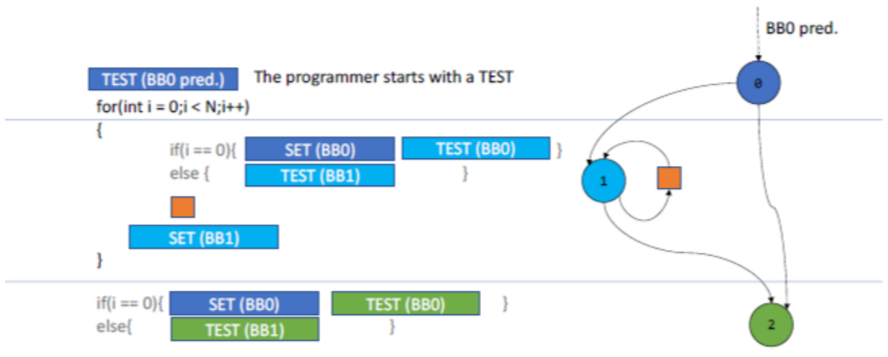


Figure 3.8. The algorithm of SET and TEST in the for loop implemented in paper [8]

```
let mut idx_ext: u32 = 0;

```

```
    [...]

for idx_delay in 0..9 {
    idx_ext = (idx_delay + 0) as u32;

    if idx_delay == 0 {
        YACCA_SET(16);

        YACCA_TEST(1<<16);
    } else {
        YACCA_TEST(1<<17);
    }
    self.dw.Delay1_DSTATE[idx_delay] = self.dw.Delay1_DSTATE[idx_delay +
   1];
    self.dw.Delay_DSTATE[idx_delay] = self.dw.Delay_DSTATE[idx_delay + 1];
    YACCA_SET(17);
}

    if idx_ext == 0 {
        YACCA_SET(16);

        YACCA_TEST(1<<16);
    } else {
        YACCA_TEST(1<<17);
    }
```

Listing 3.9. For loop implementation with YACCA CFC in Rust

Chapter 4

Fault Injection Campaign Planning

In this chapter will be listed all the metrics to perform the Fault Injection Campaign and to evaluate the results in compliance with the standard ISO 26262.

In the first section are presented the seven classes where the behavior of the faulty application can fall in comparison with the result of the "golden run".

The kind of overhead that the application can address and the statistical error to evaluate the fault injection campaign is also presented.

Lastly, there is a first result of a fault injection campaign using the YACCA Control Flow Checking method implemented in the C programming language.

4.1 Fault Detection Coverage

To evaluate the performance of the Control Flow Checking method, it is necessary first to observe how many faults are detected from the software implementation and determine how the faulty application behaves with respect to the fault-free one.

Following the classes described in the paper [14], it is possible to collect the behavior of the faulty one in seven classes, listed below:

- *"Latent after injection"*, where the behavior of the application with the fault injected and the fault-free are the same.
- *"Erratic behavior"*, where the behavior is different from the fault-free application.
- *"Infinite loop"*, the faulty application enters in an infinite loop not present in the original program flow, but caused by an erroneous path of the program.
- *"Stuck at some instruction"*, the faulty application is stuck in some correct instruction of the program but it prevents the program to increase its value.
- *"Detected by SW"*, the error is detected by CFC

- *"Detected by HW"*, the program points to some invalid memory address or falls into some hardware traps.
- *"As Golden"*, the fault is detected but the outcome is equal to the fault-free application.

In figure 4.1 the classifier is represented with the possible transitions between them, dividing the graph into two macro zones, undetected and detected.

In the "Detected" zone there are the false positive, the detected by the hardware or software, and the "as-golden" when detected and the outcome is equal to the fault-free one.

In "Undetected" there are the latent, latent after injection and all the transitions between these states and an anomalous behavior of the application as an erratic one or an infinite loop or the transition to the detected one, when the fault is detected by the application.

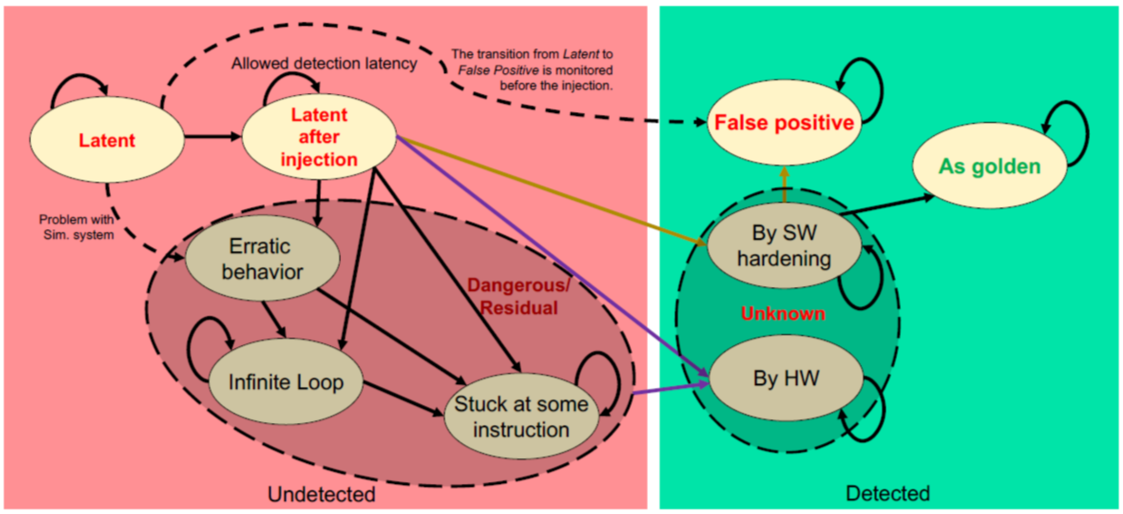


Figure 4.1. The classifier Finite State Machine (FSM), Figure from [6]

The ISO 26262-compliant classification is calculated by the following formulas taking into account:

- N, the number of injections.
- L, the number of "latent after injection" outcomes.
- D_{HW}, the number of RHF detected by hardware.
- D_{SW}, the number of RHF detected by software, using the CFC.
- U, how many times the application entered in an "infinite loop","stuck at some instruction" or presenting "erratic behavior".

The formulas used are as follows:

- Safe $= \frac{\text{As golden}}{N}$
- Detected $= \frac{D_{HW}+D_{SW}}{N}$
- Latent $= \frac{L}{N}$
- Residual $= \frac{U}{N}$
- False Positive $= \frac{\text{False positive}}{N}$

4.2 Performance Overhead Analysis

In this work of thesis, the two overheads analyzed are the increase of Text Segment Size (TSS) and the execution time overheads.

The TSS represents the increase in the size of the occupied program memory after the hardening process, increasing the memory required by the program.

The execution overhead highlights the extra number of instruction needed by the program to execute.

Both of these overheads are essential in the analysis of the system, whereas the embedded system is running on a hardware with stringent requirements over the execution time and the available memory.

4.3 Statistical Error

Another metric to consider is the statistical error associated with the fault injection campaign. Presented in the paper [7], applying the formula proposed by Leveugle in [28]:

$$e = t * \sqrt{\frac{p(1-p)}{n} * \frac{N-n}{N-1}}$$

Where p is the estimate of the true value being searched, unknown a priori but in the interval between 0 and 1. It is usually posed at p = 0.5.

N is the initial population size and t is the confidence level.

In our case, n represent how many faults are injected into the system in observation.

4.4 First example of fault injection campaign

A first example of 718 faults was injected in the application, in this moment only for the YACCA implementation in the C code, as reported in the two tables below 4.1 and 4.2.

In this campaign the detection rate was very low, but no false positive detection was triggered, this means that the implementation works correctly.

The high value of "Latent after Injection" outcomes in the fault injection campaign is due to the fact the system under observation is of the finite-state type, so it is possible that the fault lies in some state not evaluated after the fault injection. Moreover, YACCA does not detect intra-blocks Control Flow Errors, and this also can explain the high value in the classification.

Classification Result	**YACCA**		**RACFED**	
	C	RUST	C	RUST
Latent after injection	704	//	//	//
Erratic Behavior	0	//	//	//
Infinite Loop or Stuck at some instruction	0	//	//	//
(Detected) by SW hardening + Safe	14	//	//	//
(Detected) by HW	0	//	//	//

Table 4.1. Classifier of the result after the faults injection campaigns with detection latency equal to 100.

Classification Result	**YACCA**		**RACFED**	
	C	RUST	C	RUST
Latent after injection	700	//	//	//
Erratic Behavior	0	//	//	//
Infinite Loop or Stuck at some instruction	0	//	//	//
(Detected) by SW hardening + Safe	18	//	//	//
(Detected) by HW	0	//	//	//

Table 4.2. Classifier of the result after the faults injection campaigns with detection latency equal to 1000.

Chapter 5

Conclusion and future works

5.1 Summary

In this thesis work, an application was developed that simulate an emergency post-collision brake system following the Model-Based Software Design (MBSD) approach. It was developed using Simulink, a graphical language environment, provided by Mathworks.

Then, the code was generated from this application using Mathworks tool such as the Embedded Coder and Simulink coder. They allow the user to generate a code from the system developed in the graphical environment, selecting both the preferred programming language and the target platform.

In this case, the programming language selected was C, due to its high usage in the automotive field, and the target platform was RISC-V, an open source Instruction Set Architecture (ISA).

The code was then hardened by hands, implementing the two Control Flow Checking methods chosen, in this case Yet Another Control-flow Checking using Assertion (YACCA) and Random Additive Control Flow Error Detection (RACFED). They were selected because they are the most complete among all the CFC methods available to us so far. YACCA is the most complete among the CFC that uses bitwise signatures and RACFED permits to detect intra-blocks Control Flow Errors.

The focus of this thesis is also to present the comparison between the same application code developed in two different programming languages. In this case, the other programming language selected was Rust, for its capability to manage memory issues that can arise in C programming language applications.

The campaign was then set, defining the evaluating metrics and an example of results was presented only for the YACCA implementation in the C programming language.

The natural continuation of this thesis work is to obtain the experimental data of the fault injection campaigns. These results will be categorized and classified in the metrics explained in the previous chapter and see the real efficacy of the two Control Flow Checking methods with respect also to the two different programming languages in which they were developed.

The results and the back-to-back comparison of the two programming languages will be published in a paper later this year, providing a parallel evaluation of YACCA and RACFED not only between them but also between the two different programming languages. The results, hopefully, will also highlight how the implementation of Rust with the memory management makes the system more reliable with respect to the one developed in C.

Lastly the code to implement the Control Flow Checking in both the programming languages, Rust and C, will be published as open-source on GitHub.

5.2 Future works

The results of the fault injection campaign are not the last point of this study because it is open to further investigation and studies of the main points and characteristics.

In this work, only the implementation of a Control Flow Checking method was considered, without exploiting all the intrinsic Rust safeguards such as boundary checks on arrays or the borrowing rules to manage concurrency access to the variables.

For completion, would be advised to design a bottom-up study that pinpoints all the Rust intrinsic mechanisms with a fault model that targets the memory causing forced out-of-bounds memory access. Comparing the results with a C code without these implementation would highlight whether Rust's native features reduce the need for external hardening.

Although the work presented includes both Rust and C implementations, future campaigns could adopt exactly the same algorithm in both languages, without additional hardening, to measure differences in failure modes, coverage, and overhead. By isolating each language's inherent strengths or vulnerabilities, this controlled comparison would highlight whether Rust's built-in checks meaningfully outperform C in realistic fault scenarios.

Future experiments should test a range of additional fault models, such as injections into array indices, general-purpose registers, or peripheral I/O, to observe memory corruption effects and subtle concurrency issues. These broader fault scenarios are vital to assessing whether Rust's safe memory model can either prevent or swiftly detect memory-related bugs with less code overhead than comparable C solutions.

Lastly, the interplay between Rust's intrinsic protections and external control flow checking (CFC) methods deserve deeper exploration. Studies could examine how different combinations of built-in safety features and SIHFT techniques (like YACCA and RACFED) affect coverage and runtime overhead. Understanding whether Rust's memory safety can reduce reliance on software-implemented hardening thus, lowering additional overhead, would guide decisions about adopting Rust for safety-critical systems.

Collectively, these directions would provide a more complete understanding of how Rust's design fundamentally impacts safety, reliability, and overall performance in embedded environments, offering actionable insights for teams evaluating Rust as a replacement or supplement to C in future automotive and other safety-critical applications.

Bibliography

[1] Ondrej Burkacky, Johannes Deichmann, and Jan Paul Stein. Automotive software and electronics 2030, 2019.

[2] Jens Lienig and Hans Bruemmer. *Fundamentals of electronic systems design*. Springer, 2017.

[3] ISO 26262...the tale of Transient and Permanent Faults - Verification Horizons — blogs.sw.siemens.com. https://blogs.sw.siemens.com/verificationhorizons/2022/10/27/similar-but-different-the-tale-of-transient-and-permanent-faults-in-iso-26262/. [Accessed 03-2025].

[4] Mohammadreza Amel Solouki, Shaahin Angizi, and Massimo Violante. Dependability in embedded systems: a survey of fault tolerance methods and software-based mitigation techniques. *IEEE Access*, 2024.

[5] Mohammadreza AMEL SOLOUKI. *Simulation Techniques For Rapid Software Development and Validation*. PhD thesis, Politecnico di Torino, 2024.

[6] 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), 2022.

[7] Jacopo Sini, Mohammadreza Amel Solouki, Massimo Violante, and Giorgio Di Natale. Improving software reliability with rust: Implementation for enhanced control flow checking methods. In *2025 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2025.

[8] Jacopo Sini, Mohammadreza Amel Solouki, and Massimo Violante. Guidelines for implementing control flow checking into automotive embedded applications developed with c language. In *2023 IEEE Nordic Circuits and Systems Conference (NorCAS)*, pages 1–6. IEEE, 2023.

[9] João P. Trovao. New concepts in automotive electronics [automotive electronics]. *IEEE Vehicular Technology Magazine*, 16(2):113–123, 2021.

[10] João P. Trovao. Trends in automotive electronics [automotive electronics]. *IEEE Vehicular Technology Magazine*, 14(4):100–109, 2019.

[11] Vijaykrishnan Narayanan and Yuan Xie. Reliability concerns in embedded system designs. *Computer*, 39(1):118–120, 2006.

[12] Philip Payman Shirvani. *Fault-tolerant computing for radiation environments*. Stanford University, 2001.

[13] Victor P. Nelson. Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7):19–25, 1990.

[14] Mohammadreza Amel Solouki, Jacopo Sini, and Massimo Violante. An experimental evaluation of control flow checking for automotive embedded applications compliant with iso 26262. *IEEE Access*, 11:51185–51198, 2023.

[15] A. Avizienis. Toward systematic design of fault-tolerant systems. *Computer*, 30(4):51–58, 1997.

[16] Jelena N Nedeljković, Sandra M Đošić, and Goran S Nikolić. A survey of hardware fault tolerance techniques. In *2023 58th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*, pages 223–226. IEEE, 2023.

[17] Les Hatton. Safer language subsets: an overview and a case history, misra c. *Information and Software Technology*, 46(7):465–472, 2004.

[18] Ayushi Sharma, Shashank Sharma, Sai Ritvik Tanksalkar, Santiago Torres-Arias, and Aravind Machiry. Rust for embedded systems: Current state and open problems. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 2296–2310, 2024.

[19] The Rust Programming Language - The Rust Programming Language — doc.rust-lang.org. https://doc.rust-lang.org/book/. [Accessed 18-03-2025].

[20] ISO 26262-1:2018 — iso.org. https://www.iso.org/standard/68383.html. [Accessed 2025].

[21] EX30 Post-impact braking | Volvo Support UK — volvocars.com. https://www.volvocars.com/uk/support/car/ex30/article/47d2c97fd33effd3c0a8cc3718c999b7-52d4fb14596cb66ac0a8b09723ed9727-8664b2fa77a7e089c0a [Accessed 21-03-2025].

[22] Automatic Post-Collision Braking System - Car Terms | SEAT — seat.com. https://www.seat.com/car-terms/a/automatic-post-collision-braking-system. [Accessed 21-03-2025].

[23] kia.com. https://www.kia.com/content/dam/kia2/in/en/content/ev6-manual/topics/chapter6_7_4.html. [Accessed 21-03-2025].

[24] Jessica Shea Choksey. What is a Post-Collision Braking System and How Does It Work? https://www.jdpower.com/cars/shopping-guides/what-is-a-post-collision-braking-system-and-how-does-it-work, Sep 06, 2021. [Accessed 21-03-2025].

[25] Introduction - C2Rust Manual — c2rust.com. https://c2rust.com/manual/. [Accessed 03-2025].

[26] Jamey Sharp. GitHub - jameysharp/corrode: C to Rust translator — github.com. https://github.com/jameysharp/corrode. [Accessed 03-2025].

[27] Welcome to QEMU 2019's documentation! QEMU documentation — qemu.org. https://www.qemu.org/docs/master/. [03-2025].

[28] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 502–506, 2009.