IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 28 April 2023, accepted 21 May 2023, date of publication 24 May 2023, date of current version 1 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3279731

APPLIED RESEARCH

An Experimental Evaluation of Control Flow Checking for Automotive Embedded Applications Compliant With ISO 26262

MOHAMMADREZA AMEL SOLOUKI, (Graduate Student Member, IEEE), JACOPO SINI, (Member, IEEE), AND MASSIMO VIOLANTE, (Member, IEEE)
Department of Control and Computer Engineering, Politecnico di Torino, 10129 Turin, Italy
Corresponding author: Jacopo Sini (jacopo.sini@polito.it)

ABSTRACT Random hardware failures (RHFs) may result in data corruption and Control Flow Errors (CFEs). Hardening strategies are employed to mitigate RHFs in embedded systems, either by adding specialized hardware or using Software-Implemented Hardware Fault Tolerance (SIHFT) methods. Numerous SIHFT methods have been presented over the years to improve the reliability of embedded systems. However, evaluating these methods can be challenging in terms of the introduced overhead to the code size and, particularly important for real-time application execution time. Most of them are implemented in the literature using low-level languages such as Assembly. Unfortunately, writing Assembly code is not the preferred development flow for embedded systems applications since functional safety standards require adopting high-level programming languages such as C. Nowadays, there is still a non-negligible portion of code written in the Assembly language where the compiler can automatically insert the SIHFT methods, but these are limited to some high-optimized routines or device drivers. It is possible to compile an application code and then harden the obtained assembly code. But this introduces a greater overhead than just protecting a single statement in the high-level programming language before compiling. Hence, the approach we present in this paper is to apply SIHFT methods against CFEs, known in the literature as Control Flow Checking (CFC), to the application code written in C language, before compiling the application code. To illustrate the proposal, two established software-based control flow error detection techniques implemented in the C programming language were compared, also considering the effects of the optimizations introduced by the compiler. Most SIHFT methods target only soft errors, such as single-event upsets, which typically appear as bit flips. As a result, the diagnostic figures provided in the literature are insufficient to characterize the techniques effectively. To address this gap, in this paper, we consider a scenario from the automotive industry in which the primary concern is permanent random hardware faults, particularly stuck-at faults. Moreover, we propose a classification compliant with ISO26262 to benefit those developers involved in the automotive market, where software-only strategies may be used to balance cost and safety requirements.

INDEX TERMS Automotive applications, fault detection, fault tolerance, software reliability.

I. INTRODUCTION

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

The associate editor coordinating the review of this manuscript and approving it for publication was Shadi Alawneh.

or, at least, to bring the controlled physical process into a safe state. 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 [1].

In this concept, the designers aim to prevent systematic design errors and ensure hardening against Random Hardware Failures (RHFs). 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 [2].

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 [3], checkers [4], or Infrastructure Intellectual Properties (I-IP) [5]. 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) [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18].

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

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 [6]. 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 RHFs detection, defined by the Standard as Detection Coverage (DC).

In a nutshell, our methodology features two novelties: (*i*) the CFC is implemented in the C programming language, and the effects of the compiler optimizations on its effectiveness are considered, and (*ii*) the results are provided in compliance with ISO 26262.

The rest of this paper is structured as follows. We discuss the background and related work in Section II, focusing mainly on software-based hardening techniques and ISO262626-compliant classification. Section III describes

the proposed fault models and the implemented software-based hardening techniques, followed by a theoretical analysis of the effectiveness of the high-level programming language implementation. Section IV reports the experimental setup and the simulation results. Finally, the conclusion is provided in Section V.

II. BACKGROUND

This section shows software-based hardening techniques background mainly focuses on control flow checking techniques. Then ISO26262-compliant classification is explained in detail.

A. SOFTWARE-BASED HARDENING TECHNIQUES

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 RHFs 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). 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 [7].

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 RHFs.

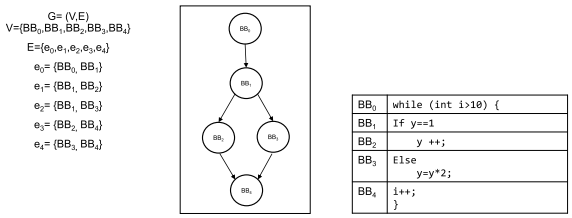

FIGURE 1. Sample code and program CFG example. The execution from basic block BB_1 to BB_2 or from BB_1 to BB_3 are legal, but a jump from BB_1 to BB_4 is illegal and called Control Flow Error (CFE).

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 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 $V = \{v_1, v_2, \ldots, v_n\}$ and the set of edges $E = \{br_{ij} | br_{ij}$ is a branch from v_i 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 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.

Examples of such methods are Enhanced Control Flow Checking using Assertions (ECCA) [8], CFC by Software Signature (CFCSS) [9], Control-flow Error Detection through Assertions (CEDAs) [10] and Assertion for CFC (ACFC) [11]. As explained before, all these approaches are based on comparing the run-time signature computed value with the expected values assigned to each block at the design or compile time. This approach provides misbehaviors detection. We will clarify them below to shed more light on the techniques.

Enhanced Control Flow Checking Using Assertions (ECCA) method was proposed by [8]. 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.

In Control Flow Checking by Software Signatures (CFCSS), which is discussed in [9], 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 [9] 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.

The authors of [10] 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.

Assertions for Control Flow Checking (ACFC), mentioned in [11], 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.

"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 [12]. 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 reassessed before each branch instruction to drop the wrong-successor CFE. The YACCA has fewer undetected errors and higher performance overhead compared to CFCSS.

Software-Based Control Flow Checking (SCFC) was proposed by [13]. 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.

Another approach is Hybrid Error-detection Technique using Assertions (HETA) [14]. 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 [15] 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 subsequentially.

The Relationship Signatures for Control Flow Checking (RSCFC) was proposed in paper [16]. 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.

To summarize, signature monitoring methods like, for instance, YACCA [12], CFCSS [9], CEDA [10], RASM [17], SEDSR [13], and ECCA [8], 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 [16], Software implemented error detection (SIED) [18], and Random Additive Control Flow Error Detection (RACFED) [19] were developed to inspect whether instruction executed in the correct order. Moreover, in [20], a software behavior-based technique is presented to detect CFEs in multi-core architectures. [21] 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. Since other details on control flow errors are outside this paper's scope, if interested, you can refer to [12] for more information.

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 1 reports a comparison between CFC methods. The table reported detection coverage, and overheads are measured by [17] and [19] on implementation done at the assembly level. They used their software-implemented fault injection (SWIFI) tool to validate comparisons between techniques.

B. ISO26262-COMPLIANT CLASSIFICATION

The ISO26262 is a functional safety standard designed for automotive applications. It was released in 2011, and the recent edition, the second edition, was updated in 2018 [1]. This Standard requires detection provision and mitigation systems to respond appropriately to RHFs.

The main part of assessing a design, which is described in part 5 (Product development at the hardware level) of the Standard, is the so-called Failure Mode, Effects, and Diagnostic Analysis (FMEDA). A crucial point of this analysis is to determine whether RHFs are detected as well as to evaluate the effectiveness of the adopted mitigation strategies. For this reason, we are focusing only on the detection mechanisms.

As a piece of information, failure modes for semiconductor components are described in part 11 (Guidelines on the application of ISO 26262 to semiconductors, added in the second update of the Standard).

The core idea of this paper is to compare, in a way compliant with ISO26262, the effectiveness of CFC methods performed in the C programming language on applications developed by means of MBSD. To achieve this result, it is needed to perform fault injections and then classify the results. The model for the injected faults will be described in Section III-A.

The fault injection environment and classifier are explained in [22]. Here, we just provide an overview to make it easier for readers to understand the simulation results without reading the aforementioned paper.

The classification is based on describing the application's behavior by comparing outputs with the fault-free execution (''golden run'') and analyzing the Program Counter (PC) register flow after the fault has been injected. Seven possible outcomes have been defined:

- ''Latent after injection'': fault injected and behavior identical w.r.t. the fault-free run.
- ''Erratic behavior'': behavior different w.r.t. the fault-free run.
- ''Infinite loop'': PC moves in an infinite loop not present in the original program flow but created by the interaction between the source code and the defective PC register.
- ''Stuck at some instruction'': PC remains stuck, pointing to a valid instruction. This happens when the injected

TABLE 1. Compare control flow control techniques.

CFC Method	*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

failure prevents the PC to increase its value (especially when the 3rd bit is involved).
- (Detected) ''by SW hardening'': detected by the CFC.
- (Detected) ''by HW (mechanism)'': PC pointing outside the FLASH/RAM addressing space or any other triggering of hardware traps.
- ''As golden'': detected with an output identical to the golden run. This classification differs from the ''Latent after injection'' since the latter represents a fault that has not been detected, while the ''as golden'' represents a fault that does not have any effect on the output of the application but that has been detected.

Moreover, the other four outcomes are provisioned. ''Latent,'' ''error,'' and ''undefined'' are not related to the application itself but are inserted to monitor the classifier, while the fourth one, ''false positives,'' indicates the presence of defects in the CFC implementation.

The classification required to determine the Diagnostic Coverage (DC), expressed by the ISO26262 in terms of ''detected'' if an embedded mechanism is able to find the presence of the considered RHF, or otherwise ''undetected''.

Considering the ''detected'' class, two other subclasses can be defined: ''safe'' if the RHF cannot have dangerous effects on the user of the item or the surrounding environment, or just ''detected'' if it is not possible to make such an assumption (like in the case of this paper, where the mitigation strategies are not considered).

On the other hand, considering the ''undetected'' class, it is possible to define two subclasses: ''latent'' if the RHF has not any effects on the behavior of the item, or ''residual'' otherwise. A third (not defined by the ISO26262) subclass, called ''false positive,'' has been defined just to describe the probability that the detection mechanism is wrongly triggered. In any case, on an excellent detection mechanism, the frequency of this subclass shall be 0%.

The ISO26262-compliant classifications are computed by the following formulas, considering:
- N the 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 experiment in where the application entered an ''infinite loop'', remained ''stuck at some instruction'', or presenting an ''erratic behavior''.

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

The RHF has been considered detected, following the concept of the Fault Tolerance Time Interval (FTTI) of the Standard, only if the detection happened within a certain amount of machine instructions. For all the simulations of this paper, this value is set to 200 assembly instructions. This has been chosen since it is sufficient to allow the execution of more than one BB.

III. PROPOSED APPROACH

The proposed approach is described in this section, focusing on fault models and the implemented software-based hardening techniques on benchmarks. Then, a theoretical analysis of the effectiveness of the implementation in the high-level programming language is provided.

A. FAULT MODELS

For the sake of this work, 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.

We decided to use the Fault Injection system presented in [22]. The Fault Injection Manager (FIM) described in the paper 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.

B. IMPLEMENTED SOFTWARE-BASED HARDENING TECHNIQUE

The two CFC techniques were implemented manually in the C listing of the two benchmarks, which were produced by means of automatic code generation using Simulink Coder based on the MBSD approach.

The first benchmark is a Finite State Machine (FSM) implementing a timeline scheduler (TS). A timeline scheduler is a periodic task executed based on a timer triggering an interrupt, in charge to run, in a fixed order defined by the system designer, a set of tasks. In our benchmark, we have 15 tasks that shall be executed in a fixed order, granting each of them a 200 ms time slot.

The second benchmark is a software-implemented controller in charge of keeping the liquid level contained in a tank at the desired height with an on-off logic (T). It takes the liquid level inside the tank alongside the current absorbed by the pumps. Based on this data, it decides when to turn the pump on and generates an alarm in case of detection of overcurrents. In this case, it shuts down the pump to avoid damage to its motor.

It is implemented as an FSM and contains the decision logic and an independent monitoring checker to verify that the physical plant properly executes the commands from the decision algorithm.

We chose these two benchmarks since they can represent applications similar to the real automotive one, that cannot be used for intellectual property protection reasons. The first benchmark, referred to in the following as TS, is needed to implement any operating system dealing with periodic tasks. At the same time, the latter, referred to in the following as T, is commonly found in industrial applications and, considering the automotive domain, is similar to algorithms controlling battery charge level in electric and hybrid vehicles.

Figure 2 shows the model-based development flow indicating when the CFC methods were applied in the high-level programming language based on the proposed approach.

We chose YACCA and RACFED since they are based on different philosophies (bit mask vs. random numbers, only inter-block vs. intra-block detection capabilities). Due to these two different philosophies, we chose YACCA due to its extreme simplicity of implementation, while RACFED since it is one of the most recent.

1) YACCA

We adopted the methods described in [12] and [21], where YACCA was proposed as a software-implemented RHF detection mechanism suitable for safety-critical applications.

The program has been divided, as described in section II, 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 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 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.

2) RACFED

Another technique we opted to implement in our benchmark is RACFED [19]. 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 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.

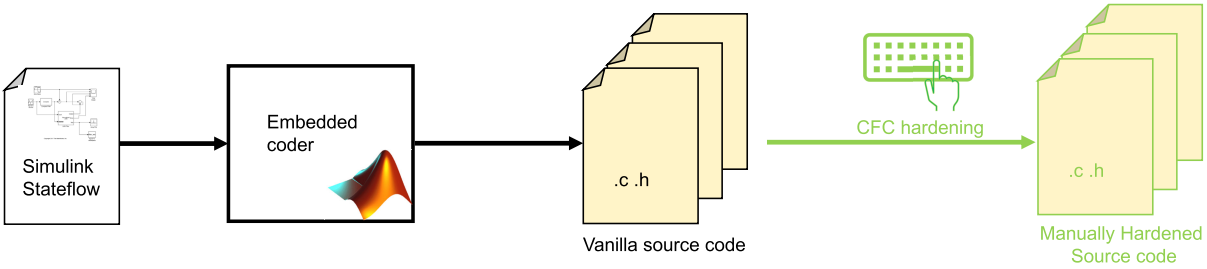

FIGURE 2. Indication, inside the model-based flow, indicating when the CFC is applied using high-level programming languages. The source code is obtained automatically via the Mathworks Embedded Coder from a Simulink semi-formal model, then the obtained source code is manually hardened.

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.

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, 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.

C. IMPLEMENTING CFC METHODS IN HIGH-LEVEL PROGRAMMING LANGUAGES

Usually, CFC techniques are implemented in the assembly language. Nonetheless, it would be too time-consuming and error-prone. Moreover, international standards like the ISO26262 prescribe that the code has to be written, with some minimal exceptions, in a structured way using a high-level programming language. We decided to harden the code by implementing the CFC methods in the C programming language.

For CFC methods, maintaining the instruction order is crucial to allow the correct signature update. Moreover, if the update is based on some arithmetic computations, the compiler can merge all the partial sums to obtain the right numerical results before the correctness verification. These optimizations can affect the detection capabilities of the CFC method. To investigate this possibility, we designed the experiments with each of the 4 optimization levels `O0`, `O1`, `O2`, and `O3`, offered by GCC for RISC-V.

RACFED was developed based on Random Additive Signature Monitoring (RASM) technique [17] 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 [19].

To verify whether the hardened code is correctly translated in Assembly, first, the code of a BB hardened with RACFED should be compiled with no optimizations; then, the obtained assembly code should be investigated. Then, it is possible to verify if the compiler changes the order of the instructions. The result of this experiment is discussed in IV.

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, first, the code of a BB hardened with RACFED should be compiled with optimizations; then, the obtained assembly code should be investigated. The result of these experiments is discussed in IV.

IV. SIMULATION RESULTS

This section shows the obtained simulation results, describes the chosen RISC-V environment, 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 different optimization levels.

FIGURE 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.

A. 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 [23]. 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 [24], providing flexibility to both hardware and software. The benchmark applications considered in this paper were compiled using the GNU RISC-V Toolchain [25].

As the target platform, as described in [22], we choose RISC-V RV32I, simulated at the ISA level thanks to the QEMU (Quick Emulator) [26].

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) [27] 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.

B. HARDENING TECHNIQUE PERFORMANCE ASSESSMENT

As described in section III-B, 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, 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.

C. 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 `O0` 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 (`O1`, `O2`, and `O3`), 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 `O1`, `O2`, and `O3` 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 2. 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 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 2, 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 2 for the experiments without almost all compiler optimizations (O0), and in Table 3 and Table 4 for the three levels of optimization.

D. DIAGNOSTIC COVERAGE

The results shown in Section IV-C were transposed into ISO 26262-compliant classifications, which requires computing the Diagnostic Coverage (DC) of the proposed CFC methods. The obtained results are presented in Table 5 and Table 6. 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 5 is the sum of the last two rows of Table 2 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 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 5, 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.

TABLE 2. 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

TABLE 3. 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. 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

Considering the "Undetected failures" in Table 5, 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 2 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 ≥99%) required by the Standard.

Table 3 and Table 4 show results with different compiler optimizations obtained on the TS benchmark, Table 6 the obtained DCs, and finally Table 7 the corresponding overheads.

Starting from the results in Table 3, the diagnostic coverage (SW only) for YACCA is the best at `O1`, while the DC is similar for `O0` and `O2`, with `O2` being slightly better than `O0`. Overall with all three different compiler optimizations, the DC is improved compared to the case where no compiler optimizations were used (`O0`). 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 4 for RACFED, which also features intra-block detection capability. From `O0` to `O1`, the detections by software increased from 93 to 183. The same pattern, even if less evident, is observed from `O2` (136) to `O3` (156). To explain this phenomenon, we analyzed the generated Assembly code. Between O0 and `O1`, 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 `O2` and `O3`, 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 `O2` and `O3`.

In Table 6 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 O0 to O1, the DC increases by about 24% as the occupied program memory is shrunken by about 20%. The DC for O2 and O3 remains approximately the same as the O1 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 O2 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 O1 optimization, then O2 and O0 are similar, while O3 performs better compared to O2. The ''Residual Undetected'' failures, with O0 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 O2 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 O0 and O1, 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 O0 and about 20% for the optimized versions.

E. 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 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 7 reports the overhead on the program memory represented as ''TSS'' and the overhead on executed instructions represented as ''# exec. instr.'' The 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 benchmarks, 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 overheard 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 O2 and O3, 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.

TABLE 5. 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.
		Safe	*Detected*	*Latent*	*Residual*	
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 6. 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.
		Safe	*Detected*	*Latent*	*Residual*	
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%

TABLE 7. 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%)

For YACCA, with any compiler optimization level, the number of executed instructions is greater than the number of executed instructions in the Vanilla O0. However, the executed instructions overhead decreases drastically from O0 to O1, while this overhead decreases slowly while changing the optimization level from O1 to O2 and from O2 to O3. For RACFED, with O1 compiler optimization level, the number of executed instructions is greater than the number of executed instructions in the Vanilla O0. On the contrary, with O2 and O3 optimization levels, the executed instructions become negative. The executed instructions overhead decreases drastically from O0 to O1, and also from O1 to O2. While this overhead decreases slowly while changing the optimization level from O2 to O3. In conclusion, for both benchmarks, YACCA imposes less TSS overheads while RACFED imposes less executed instructions overheads.

V. CONCLUSION

Numerous Software-Implemented Hardware Fault Tolerance (SIHFT) methods have been presented in the literature to increase the reliability of embedded systems against Random Hardware Failures (RHFs).

The adoption of a specific SIHFT method is challenging because the various methods are available in the literature, lead to the need for an objective comparison methodology.

We proposed a comparison methodology that consists of selecting a set of representative applications, hardening them with selected SIHFT methods, and finally performing the fault injection. Two established CFC methods were selected and applied to two benchmarks to evaluate our approach. Finally, simulation results were reported in compliance with automotive functional safety standard ISO 26262 by evaluating the efficiency of the CFC methods in detecting RHFs. The effects of the compiler optimization on their effectiveness have been investigated by repeating the experiments for each of the 4 optimization levels featured by GCC.

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] *Road Vehicles—Functional Safety*, Standard ISO 26262, 2018.
[2] B. Yue and W. Che, "Data-driven dynamic event-triggered fault-tolerant platooning control," *IEEE Trans. Ind. Informat.*, early access, Oct. 31, 2022, doi: 10.1109/TII.2022.3217470.
[3] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors—A survey," *IEEE Trans. Comput.*, vol. 37, no. 2, pp. 160–174, Feb. 1988.
[4] T. M. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Proc. 32nd Annu. ACM/IEEE Int. Symp. Microarchitecture*, Nov. 1999, pp. 196–207.
[5] C. A. Lisboa, M. I. Erigson, and L. Carro, "System level approaches for mitigation of long duration transient faults in future technologies," in *Proc. 12th IEEE Eur. Test Symp. (ETS)*, May 2007, pp. 165–172.
[6] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Energy efficiency across programming languages: How do energy, time, and memory relate?" in *Proc. 10th ACM SIGPLAN Int. Conf. Softw. Lang. Eng.*, Oct. 2017, pp. 256–267.
[7] *MATLAB Version: 9.13.0 (R2022b)*, The MathWorks Inc., Natick, MA, USA, 2022. [Online]. Available: http://www.mathworks.com
[8] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, pp. 627–641, Jun. 1999.
[9] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 111–122, Mar. 2002.
[10] R. Vemu and J. Abraham, "CEDA: Control-flow error detection using assertions," *IEEE Trans. Comput.*, vol. 60, no. 9, pp. 1233–1245, Sep. 2011.
[11] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," in *Proc. 9th IEEE-Line Test. Symp.*, Jul. 2003, pp. 137–143.
[12] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Improved software-based processor control-flow errors detection technique," in *Proc. Annu. Rel. Maintainability Symp.*, 2005, pp. 583–589.
[13] S. A. Asghari, H. Taheri, H. Pedram, and O. Kaynak, "Software-based control flow checking against transient faults in industrial environments," *IEEE Trans. Ind. Informat.*, vol. 10, no. 1, pp. 481–490, Feb. 2014.
[14] M. Altieri, J. Becker, and F. L. Kastensmidt, "HETA: Hybrid error-detection technique using assertions," *IEEE Trans. Nucl. Sci.*, vol. 60, no. 4, pp. 2805–2812, Aug. 2013.
[15] E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A. Tambara, P. Rech, and H. Quinn, "S-SETA: Selective software-only error-detection technique using assertions," *IEEE Trans. Nucl. Sci.*, vol. 62, no. 6, pp. 3088–3095, Dec. 2015.
[16] A. Li and B. Hong, "Software implemented transient fault detection in space computer," *Aerosp. Sci. Technol.*, vol. 11, nos. 2–3, pp. 245–252, Mar. 2007.
[17] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random additive signature monitoring for control flow error detection," *IEEE Trans. Rel.*, vol. 66, no. 4, pp. 1178–1192, Dec. 2017.
[18] B. Nicolescu, Y. Savaria, and R. Velazco, "SIED: Software implemented error detection," in *Proc. 16th IEEE Symp. Comput. Arithmetic*, Nov. 2003, pp. 589–596.
[19] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random additive control flow error detection," in *Proc. Int. Conf. Comput. Saf., Rel., Secur.* Cham, Switzerland: Springer, 2018, pp. 220–234.
[20] M. Maghsoudloo, H. R. Zarandi, and N. Khoshavi, "An efficient adaptive software-implemented technique to detect control-flow errors in multi-core architectures," *Microelectron. Rel.*, vol. 52, no. 11, pp. 2812–2828, Nov. 2012.
[21] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Proc. 16th IEEE Symp. Comput. Arithmetic*, Nov. 2003, pp. 581–588.
[22] J. Sini, M. Violante, and F. Tronci, "A novel ISO 26262-compliant test bench to assess the diagnostic coverage of software hardening techniques against digital components random hardware failures," *Electronics*, vol. 11, no. 6, p. 901, Mar. 2022.
[23] A. Waterman and K. Asanović, Eds., *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*, document Version 20191214, RISC-V International, Dec. 2019.
[24] S. Di Mascio, A. Menicucci, G. Furano, C. Monteleone, and M. Ottavi, "The case for RISC-V in space," in *Proc. Int. Conf. Appl. Electron. Pervading Ind., Environ. Soc.* Cham, Switzerland: Springer, 2019, pp. 319–325.
[25] (2022). *Gnu RISC-V Toolchain*. [Online]. Available: https://github.com/johnwinans/riscv-toolchain-install-guide
[26] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf.*, 2005, p. 46.
[27] (2022). *The Gnu Debugger*. [Online]. Available: https://www.gnu.org/software/gdb/

MOHAMMADREZA AMEL SOLOUKI (Graduate Student Member, IEEE) received the B.Sc. degree in computer engineering from the Islamic Azad University of Mashhad, Mashhad, Iran, in 2013, and the M.Sc. degree in computer architecture from the Department of Computer Engineering, Qazvin Islamic Azad University, Qazvin, Iran, in 2017. He is currently pursuing the Ph.D. degree with the Department of Control and Computer Engineering (DAUIN), Politecnico di Torino, working under the supervision of Prof. Massimo Violante. He is involved in reliable software for detection, recovery, and resilient behavior. His key topics are validation techniques, functional safety (ISO 26262), model-based development, and embedded software.

JACOPO SINI (Member, IEEE) received the B.Sc. degree in computer engineering, the M.Sc. degree in mechatronic engineering, and the Ph.D. degree in control and computer engineering from Politecnico di Torino, in 2014, 2016, and 2021, respectively. Since 2016, he has been a Research Assistant with Politecnico di Torino. He is involved in several research projects in the area of embedded systems for automotive applications in collaboration with ITT and ELDOR. His research interest includes testing highly dependable embedded items, with a particular focus on autonomous vehicle-enabling technologies.

MASSIMO VIOLANTE (Member, IEEE) received the M.S. and Ph.D. degrees from Politecnico di Torino, Italy. From 2008 to 2009, he was a Research Assistant with the Institute of Physics, Academia Sinica, Tapei, Taiwan. He is currently an Associate Professor with Politecnico di Torino. He is very active in technological transfer activities with industries in the automotive and industrial sectors on topics, such as functional safety, model-based design, and embedded system design and validation. He is scientific responsible of a number of research projects in the area of embedded systems for space, avionic, and automotive applications in collaboration with the European Space Agency, Thales Alenia Space, ITT, CNH Industrial, ELDOR, and Magneti Marelli. He published more than 150 articles in the area of testing and designing reliable embedded systems. He has coauthored two books. His main research interests include the design and validation of embedded system for safety- and mission-critical applications, with emphasis on the use of commercial off-the-shelf components like multi core processors and field programmable gate arrays in automotive, avionic, and space applications, and the development of surface processing and biological/medical treatment techniques using non-thermal atmospheric pressure plasmas, fundamental study of plasma sources, and fabrication of micro- or nanostructured surfaces. He served as the Program Co-Chair and the General Co-Chair for the IEEE Defect and Fault Tolerance in VLSI and Nanotechnology Systems, in 2011 and 2012, and the Program Chair for the IEEE European Test Symposium, in 2012.

Open Access funding provided by 'Politecnico di Torino' within the CRUI CARE Agreement