 doi:10.20944/preprints202208.0160.v1

Article

Implementation of Control Flow Checking - A New Perspective Adopting Model-Based Software Design

Mohammadreza Amel Solouki [1], Jacopo Sini[1] and Massimo Violante[1]

[1] Department of Control and Computer Engineering, Politecnico di Torino, Turin, Italy; {mohammadreza.amelsolouki, jacopo.sini, massimo.violante}@polito.it
* Correspondence: jacopo.sini@polito.it

Abstract: A common requirement of embedded software in charge of safety tasks is to guarantee the identification of those Random Hardware Failures (RHFs) that can affect digital components. RHFs are unavoidable. For this reason, functional safety standards, like the ISO 26262 devoted to automotive applications, require embedded software designs able to detect and eventually mitigate them. For this purpose, various software-based error detection techniques have been proposed over the years, focusing mainly on detecting Control Flow Errors. Many Control Flow Checking (CFC) algorithms have been proposed to accomplish this task. However, applying these approaches can be difficult because their respective literature gives little guidance on the their practical implementation in high-level programming languages, and they have to be implemented in low-level code, e.g., assembly. Moreover, the current trend in the automotive industry is to adopt the so-called Model-Based Software Design approach, where an executable algorithm model is automatically translated into C or C++ source code. This paper presents two novelties: firstly, the compliance of the experimental data on the capabilities of Control Flow Checking (CFC) algorithms with the ISO 26262 automotive functional safety standard; Secondly, by the implementation of the CFC algorithm in the application behavioral model is automatically translated. There is no need to modify the code generator. The assessment was performed using a novel fault injection environment targeting a RISC-V (RV32I) microcontroller.

Keywords: fault injection; functional safety; automotive applications; fault tolerance;

1. Introduction

Embedded systems are being increasingly used in safety-critical and general-purpose applications, where failure can have serious consequences. The design of these systems is a complex process because it requires design methods to be integrated in the hardware and software to meet their functional and non-functional requirements. These applications range from small applications, like mobile phones, to complex and crucial applications, such as medical, aerospace, and automotive systems.

The operation of systems, particularly their functional safety aspects, relies on the hardware and software adopted. Different techniques are generally used to reduce hazards and improve safety to avoid the presence of unreasonable risks. It is key to assess whether systems are working correctly and *Fault Tolerance Techniques* (FTT) take into account redundancy in the software (extra instruction codes) and hardware (external hardware components). From the hardware perspective, redundancy and diversity are the most common techniques. Instead, in terms of the software, the techniques include design diversity, hazard prevention, and hazard detection,which are needed to provide consequential and timely management when hazards occur. The core idea is to avoid the presence of systematic errors in the design (defects), but unfortunately, the hardware components can be affected by inevitable failures due to their physical nature, called *random* (since they can happen at any moment which is not possible to forecast) and *hardware failures* (HRFs). Furthermore, providing an appropriate and unified design methodology is necessary to

 doi:10.20944/preprints202208.0160.v1

guarantee the safe and correct operations of target applications. Thus, many safety standards for example, MIL-STD-882D, a military standard, IEC 61508, on functional safety for electrical, electronic, and programmable electronic safety-related systems, and ISO 26262 targeting automotive applications have been developed to cover the safety management of safety-critical systems throughout their lifecycles.

FTT requires structural modules containing two basic properties, *i)* self-protection and *ii)* self-checking [1]. The first property means that a component must be able to protect itself from external contamination by detecting errors in the information passed to it by other interacting components. Moreover, self-checking means that a component must be able to detect internal errors and take appropriate actions to prevent the propagation of those errors to other components. The coverage of error detection mechanisms used in a design is determined by the cost of the additional redundancy for the hardware by adding external components, or the software by adding the external instruction at run-time or compile-time. In some cases, both approaches can be use together based on the integrity level determined thanks to Hazard Analysis and Risk Assessment (HARA). Hardening the hardware components is more expensive than software methods because adding extra complementing error detection methods is costly.

A reasonable solution is software implemented to increase the reliability of embedded systems. Many software-implemented control flow error (CFE) detection techniques have been proposed to protect embedded systems. These techniques add extra control variables and update instructions to the target programs. At specific points in the target program, run-time and compile-time values of the control variable are compared to one another. Any mismatch indicates a CFE has occurred and has been detected. A CFE can be an inter-block CFE or an intra- block CFE. An inter-block CFE is an invalid jump between two different basic blocks (BBs), while an intra-block CFE is an invalid jump within the same BB. Both types of CFE can lead to hazardous situations by causing the affected program or system to halt, crash, or provide erroneous output.

In this paper, we present two novelties with respect to the state of the art.
The first one is hardening two case study applications based on two established CFC techniques through Model-Based Software Design (MBSD). Assessing the effectiveness of CFC techniques can be difficult. In contrast with the literature, where these are implemented at the assembly level, we apply them directly in the Mathworks Simulink Model to compare the reported error detection ratios with our experimental results. The advantages of implementing at the model level are simplifying developer activities and avoiding the use of low-level languages as required by functional safety standards. In addition, based on our proposal, it could be more user-friendly to implement CFC methods on each source code written in high-level languages.
The second novelty is that the experimental results are classified in compliance with ISO 26262 automotive functional safety standards.

The remainder of this paper is structured as follows. We discuss background in section 2 with related work and give a short overview of hardware fault tolerance 2.1, CFC techniques 2.2 and overview of functional safety in the automotive industry 2.3. In section 3 the proposed approach is discussed, explains fault models 3.1, and 3.2 shows the implementation method. In section 4 shows the results with detail of the target platform 4.1, assessment 4.2, fault injection results 4.3. overhead results 4.5, and diagnostic coverage 4.4. In the final section, we conclude our proposal in section 5

2. Background

2.1. Hardware-based fault tolerance techniques

Hardware-based techniques have two main groups *i)* redundancy-based and *ii)* hardware monitors. The first group relies on hardware or time redundancy. In contrast, the second group adds special hardware modules to the system's architecture to monitor the control flow of the programs inside the processors and memory accesses performed by them, such as watchdog processors [2], checkers [3], or Infrastructure Intellectual Properties

 doi:10.20944/preprints202208.0160.v1

(I-IP) [4]. Hardware-based techniques have a high cost, verification and testing time, and area overhead, which also leads to higher power consumption.

2.1.1. Redundancy

Hardware redundancy is the most common technique, which is the addition of extra hardware components for detecting or tolerating faults [5], [6]. For example, instead of using a single core/processor, more cores/processors can be exploited so that each application is executed on each core/processor, then the fault can be detected or even corrected. Hardware redundancy can be applied through *i)* passive, *ii)* active and *iii)* hybrid methods.

i) Passive Redundancy

Examples of this redundancy are *N* modular redundancy (NMR), such as Triple Modular Redundancy (TMR), and using voting techniques. These techniques are referred to as *M*-of-*N* systems, which means that the system consists of N components, and the correct operation of this system is achieved when at least M components correctly work. The TMR system is a 2-of-3 system with M = 2 and N = 3, which is realized by three components performing the same action, and the result is voted [5], [6].

*ii)*Active Redundancy

This type of active hardware redundancy includes duplication with comparison (DWC), Standby-sparing (SS), Pair-and-a-spare technique, and watchdog timers. In DWC, two identical hardware components perform the exact computation in parallel, and their output is compared. Therefore, the DWC technique can only detect faults but cannot tolerate them because the faulty component cannot be determined [5], [6]. In standby-sparing, one module is operational, and one or more modules are standby or spares. If the fault is detected in the main component, it will be omitted from the operation, and the spare component will continue the execution[5], [6]. Meanwhile, pair-and-a-spare is a combination of DWC and SS techniques. For instance, two modules are executed in parallel, and their results will be compared to detect the fault [5], [6].

*iii)*Hybrid Redundancy

The basic concept of this method is integrating the main features of both active and passive hardware redundancies. *N* modular redundancy with spare, sift-out modular redundancy, self-purging redundancy, and triple duplex architecture are examples of hybrid hardware redundancy [5], [6]. The basic concept of self-purging is based on NMR with spare techniques. All modules are active and participate in the function of the system. In sift-out modular redundancy, there are N identical modules. However, they are configured in the system through special circuits (comparators, detectors, and collectors). The triple duplex architecture combines the DWC technique with TMR, which helps to detect the faulty module and remove it from the system.

2.1.2. CFC techniques

In general, deploy dedicated monitoring hardware like a watchdog processor or extra hardware within the processor to monitor the control flow by comparing the instruction addresses with stored expected addresses or runtime signatures with reference signatures or by verifying the integrity of signatures with error correction codes [7], [8], [9], [10], and[11]. For example, control flow checking by execution tracing (CFCET) [11] uses execution tracing to transmit the runtime branch instruction address and the branch target address to an external watchdog processor. The watchdog processor compares these addresses with the reference ad- dresses stored in its associative memory to detect a deviation in control flow. WDP [12], and Online signature learning and checking (OSLC) [8] use the watchdog

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 August 2022 doi:10.20944/preprints202208.0160.v1

processor to compare with the reference control flow. ASIS [7] uses a hardware signature generator and watchdog monitor to check the control flow of several processors.

2.2. Software-based hardening techniques

CFE detection techniques have been proposed in the literature over the years. In order to detect CFEs, such techniques insert extra control variables, called signatures, managed by set and test instructions into the target program. The set instructions usually modify the signature incrementally (by applying changes with logical or arithmetical operations that input the current value to compute the next one to be assigned). In terms of test, compare the signature with the expected one determined at compile-time. By adding instructions, recalculate these control variables at run time and compare them to the expected compile-time value.

A basic block (BB) is a sequence of consecutive instructions with precisely one entry and one exit point. A program can be described by its CFG, which is an oriented graph, with the BBs as the vertices and the intentional paths between BBs as the edges. A transition between BBs are not represented in set edges is recognized as an error. This error is a violation of the CFG of the program.

In this section, some of the CFC techniques are discussed. Methods are based on comparing the value of the signatures computed at run-time with their expected values assigned to each block at the design or compile-time.

Enhanced Control Flow Checking Using Assertion (ECCA) technique has been proposed in [13]. Its idea is to assign three compile-time variables to each BB for comparison and update. : `BID`, `NEXT1`, and `NEXT2`. `BID` is a unique prime number larger than 2 that serves as the compile-time signature. `NEXT1` and `NEXT2` hold the signatures of the possible successor blocks of the current BB. The redundancy instructions are added for two proposed. Firstly, the *test* function is executed at the beginning of the BB to check whether the predecessor BB is permissible or not. The second function is a *set* that is executed at the end of BB and updated signature for possible successor BB. Because the multiplication and division operations used in assertions come at the expense of increased performance overhead.

When the processor executes a new BB, particular assertions check the control flow using the involved BB identifiers. However, the authors of ECCA claim that this can be resolved by grouping the different cases into one BB.

Control Flow Checking by Software Signatures (CFCSS), discussed in [14], is a technique that assigns two compile-time variables to each BB based on a predecessor's relationship. The first variable is the compile-time signature s_i, a unique randomized bit sequence. The second variable is d_i, defining the first predecessor's proper branch to the current BB. The global signature variable G is used to track the execution at run-time; in a fault-free run, it always corresponds to the signature of the basic block currently being executed. When control is transferred from one BB to another, CFCSS calculates the destination BB's signature from the current BB's signature by using the XOR function to figure out the difference between the signatures of the current and destination BBs. In the case of an error-free run, the run-time signature should hold the same value as the compile-time signature of the current BB. When a BB has multiple incoming edges, the update phase uses an extra variable,`D`. Pre- predecessor BBs are guaranteed to update this variable then the run-time signature can be updated to the proper value regardless of which predecessor has executed.

YACCA (Yet Another Control-Flow Checking using Assertions) technique was explained in [15]. It assigns a unique signature to each BB entry and exit point. The advantage of using this method is the possibility of detecting CFEs happening when the program flow jumps from the inside of a BB to one of its legal successors even if the successive BB gives back the control to the BB affected by the wrong jump. This is possible since the signature is re-assessed before each branch instruction to drop the wrong-successor CFE.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 August 2022 doi:10.20944/preprints202208.0160.v1

Relationship signatures for control flow checking (RSCFC) [16] is based on extracting the relationship between the blocks then assigning signatures to every block. It has encoded the control flow relations between different BBs into the unique formatted signatures and then inserted control flow checking instructions into the head and the end of every BB separately. The method has a higher fault detecting rate than CFCSS, but this algorithm needs more performance overhead, and the express ability of the signature is confined by the machine word length.ANDing run-time signatures detect the faults in the flow control program with the information at the beginning and end of the blocks. This technique allows the detection of inter-block CFEs thanks to three variables: a compile-time signature s_i, the CFG locator L_i, and the cumulative signature m_i.

The technique that proposes the detection of soft errors using software redundancy is Soft Error Detection using Software Redundancy (SEDSR), an inter-block CFE detection technique proposed by [17].

Based on this technique, critical blocks have to be identified in the CFG. Critical blocks are duplicated at compile-time and have to be identified in the CFG.

SEDSR assigns only one compile-time variable to each BBs. This variable shows the valid successor BBs of the current BBs. The bit position on the variable shows the position of the successor of the current BB. If there are n BBs in the CFG of the program, then s_i has to be n bits wide, and the bits of the successor BBs are set to *1*. The run-time signature S is verified at the beginning of each BB. The verification checks whether the current BB is the previous BB's valid successor. In case of an error-free run, the bit on the position associated with the current BB should be set. If that bit is *0*, an error has occurred. The run-time signature S is updated in the middle of each BB and assigns the compile-time signature s_i of the current BB to S.

To further increase the error detection ratio, the same authors of the latter also proposed the Software-based Control Flow Checking (SCFC) technique to extend their SEDSR technique [18]. Two run-time variables have used. The first is a variable containing the run-time identifiers of the BBs, `ID`, and the second contains the run-time signature S. SCFC uses the same compile-time signature s_i as in SEDSR.

First, run-time verification is performed at the beginning of each BB. This verification checks whether or not `ID` holds the compile-time identification number (`ID`) of the current BB. A mismatch indicates a CFE has occurred. The second run-time verification, i.e., the verification of the run-time signature `S`, is performed in the middle of each BB. This verification is the same as SEDSR and checks if the current BB is a valid successor of the previous one.

To achieve a higher CFEs detection `ID` and `S` are updated at different places in the BB. `S` is updated in the middle of the BB after being verified. `ID` updated the compile-time id of the successor block at the end of the BB.

Random Additive Control Flow Error Detection (RACFED) [19] is the CFE detection technique that uses run-time signature updates to detect inter and intra-block CFEs. It detects CFEs by inserting a control variable and updating instructions in the target code.

A control variable update instruction is inserted at the beginning of each BB. This update instruction is followed by a verification instruction for the run-time value of the control variable compared to the compile-time. Control is transferred to the error if any mismatch between the two values is detected. Table 1 reports a comparison between each CFC method as discussed above. The reported detection coverage and overheads are measured by [20] and [19] on implementation done at the assembly level. They used their software-implemented fault injection (SWIFI) tool to validate comparisons between techniques. It used the CFG of the program to find possibilities to flip a single bit in the program counter (PC) register so that an interblock CFE is caused. Since this paper considered CFC methods, this approach resembles industry-relevant architectures such as AUTOSAR (AUTomotive Open System ARchitecture), in which a graph representation of a function is constructed by dividing the function into basic blocks at each (conditional) branch. The objective of AUTOSAR is to establish an open industry standard for the automotive

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 August 2022 doi:10.20944/preprints202208.0160.v1

software architecture between suppliers and manufacturers [21]. The standard comprises a set of specifications describing software architecture components and defining their interfaces [22].
Many companies now use AUTOSAR specifications for building automotive products and bringing them on the market. Using the AUTOSAR trademark implies conformance to AUTOSAR specifications, which is essential for those products' interoperability, reuse, portability, and scalability. Therefore they have to demonstrate their conformance to the AUTOSAR standard. Functional safety is one of the main objectives [23] as AUTOSAR will support safety-related applications and thereby has to consider the upcoming ISO 26262 standard.

The AUTOSAR platform is emerging in the automotive domain as an open industry standard for developing in-vehicular systems. To cope with the growing complexity of modern vehicles, it provides a modular software architecture with standardized interfaces and a run-time environment (RTE) that separates application-level software components from the underlying basic software modules and the actual hardware.
The third mechanism that the WdM provides is logical monitoring, which focuses on detecting control flow errors, which occur in one or more program instructions that are processed either in an incorrect sequence or are not processed at all.
the WdM provides logical monitoring, which focuses on detecting control flow errors, which occur when one or more program instructions are processed either in an incorrect sequence or are not processed.

Table 1. Compare CFC Methods. UV indicates Used Variables, S indicates Signatures, DP indicates detection performance [%], CO indicates Code size overhead [%] and EO indicates Execution time overhead[%]

Algorithm	*UV*	*S*	*intra-block*	*DP*	*CO*	*EO*
ECCA	4	prime-numbers	χ	73.5	36.0	244.8
CFCSS	2	randomized-bit	χ	75.8	15.2	76.6
YACCA	2	bit-field	χ	82.8	30.0	203.2
RSCFC	2	bit-field	✓	49.4	17.5	86.8
SEDSR	3	bit-field	✓	46.8	12.3	67.1
SCFC	3	bit-field	✓	60.4	22.9	115.7
SIED	2	random numbers	✓	52.4	14	115.7
RACFED	3	random numbers	✓	N.A.	N.A.	81.5

2.3. Functional safety in the automotive industry

In safety-critical embedded systems, ensuring adequate safety levels represents the primary factor in the success of these systems. ISO 26262 is an international standard for the functional safety of electrical and electronic systems titled "Road vehicles - Functional safety." It was released in 2011, and the current edition is the second one, released in 2018 [24]. ISO 26262 was derived from the generic functional safety standard IEC 61508 to address the specific needs of electrical and electronic systems and focuses on malfunctioning behaviors. The Standard is divided into eleven parts, covering all activities during the safety life cycle of safety-related systems, including electrical, electronic, and software elements that provide safety-related functions. The process prescribed in ISO 26262 uses a top-down approach in which, first, hazard analysis is conducted to identify potential hazards and system-level requirements. The most important parts related to our paper are the third (concept phase), fifth (development at the hardware level) and sixth (development at the software level).

The third is the "concept phase", when the item is defined. From the definition, it is possible to perform the hazard analysis and risk assessment [25] [26] needed to define the risk level associated with its functionality (Automotive Safety Integrated Level, ASIL), the

 doi:10.20944/preprints202208.0160.v1

safety goals (SGs) to be achieved, and its functional safety concept (FSC).
Based on the obtained SGs and their ASILs, actions to prevent the presence of systematic failures or to mitigate random hardware failures (RHFs) have to be taken in phases five and six. The fifth one is about product development at the hardware level. An essential result of this phase is the list of the possible Failure Modes (FMs) that can affect the designed item and, in particular, its computation unit. A detailed description of the application of the safety life cycle to semiconductors is given in part 11 [24].
The design group develops a embedded software in parallel to the hardware design. It shall be developed by following part six of the standard to avoid the presence of defects (a.k.a. bugs) in the code (prevention against systematic errors). The possibility of unavoidable RHFs shall be taken into account. Hence the need to implement hardening techniques like CFC.

3. 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 the most recent.

To harden the code, we instrumented by hand CFC techniques into the Simulink Model representing the application behavior. Because in the automotive industry, the software is usually developed by the Model-Based Software Design Approach. Moreover, based on functional safety requirements, it is not recommended to implement by hand on the assembly code. Regarding our approach, the hardened C code has been obtained from the code generator Embedded Coder. As a result, modification of the code generator is not needed. Because it simply generates the source code and CFC. The worse impacts granularity because statements are not added only by the compiler as hardening by hands the C code, but also by Embedded Coder. Based on the observation, measuring the detection rate experimentally is needed.
To assess our proposal, we implemented YACCA[15], and RACFED [19] to compare two different CFC algorithms. The first is based on bitfields, assigning a single bit signature to each BB, and lacking intra-block detection capability. On the other hand, the latter uses signatures based on random numbers and features intra-block detection.
The advantages of YACCA are its implementation simplicity usually (in assembly and C language), leading to small overheads in both instructions and code terms, but it requires a signature variable with a bit width equal to the number of BBs. It can be challenging to implement them when there are more than 64 BBs.
RACFED is a more complex method, but due to the usage of random numbers, it can use a smaller signature (64 bits are sufficient in every case). Moreover, for the reasons mentioned above, it shall provide better detection capabilities from a theoretical point of view.
The metrics measured on our benchmark application are available in Table 7 Moreover, these methods have not been implemented through a traditional high-level programming language (like C, C++, or ADA) but resort to a graphical representation of its functionality in the form of a physical/control model or a Finite State Machine (FSM).
The common adopted tools to develop these models are the ones developed by the company MathWorks [27].
Their popular tool for describing behavior models is Simulink®, while its package StateFlow is used to develop FSMs. Since these are the most commonly implemented software units, CFC is perfect for hardening them against RHFs.

3.1. Fault models

For the sake of this work, we considered only faults affecting the Program Counter (PC) register, directly or indirectly(for example, the control flow jumps to a location outside the program area or inside the program area but to a wrong address). We chose this register since it directly affects the instruction flow.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 August 2022 doi:10.20944/preprints202208.0160.v1

We used the tool presented in [28] to perform the fault injection campaigns. The Fault Injection Manager (FIM) described in the paper mentioned above features two fault models, first, *Permanent* and second *PermanentStuckAt*.
The *permanent* affects only one bit of the register that remains from the injection on fixed to 0 or 1, while the *PermanentStuckAt* affects the entire register globally, making it stuck to a fixed value. Here, we used only the *permanent*, choosing the injection time, the affected bit, and its state randomly.
It is possible to choose a subset of bits that can be affected to a bitmask to increase injection efficiency. Avoiding injection on higher bits causes the PC immediately to move outside the .text area, triggering hardware failure detection mechanisms.
More in detail, since we know that the RV32I has 32bit-long instructions and that the addressing is done addressing blocks of 8 bits, we can consider that injecting on the first two bits has no meaning since such an injection raises a hardware trap for misaligned memory access during the fetch of the instructions. So, it is convenient needed to start from the 3rd bit, considering that the number of "skipped" instructions $I_{skipped}$ when the fault-affected bit at a position N assumes the wrong value is equal to $I_{skipped} = 2^{N-2}$. For example, if the 6th bit is affected, it provokes to skip $2^{6-2} = 16$ instructions.

3.2. Implemented software-based hardening technique

Simulink allows code generation starting from models by using Embedded Coder. With this process, C, C++, optimized MEX functions and HDL code can be easily generated. The Target Language Compiler Tool (TLC), is included in Simulink, allowing the developer to customize the generated code accordingly to the requested platform starting from any model. It's used to convert the model into C code.Our proposal is shown in Figure 1.
We chose these techniques (YACCA and RACFED) for our empirical study.

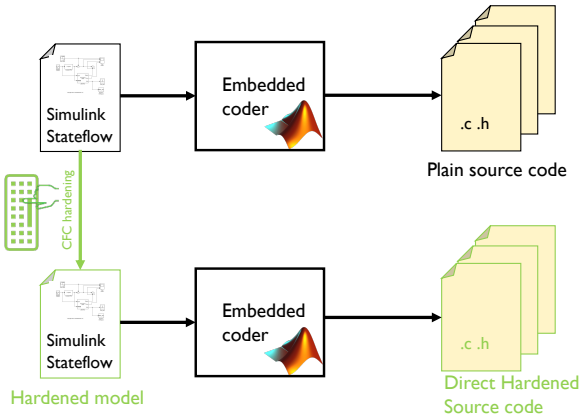

Figure 1. Model Based approach for implementing the CFC methods. The benchmarks are obtained by hardened them in the model, than generating the source code automatically thanks to Embedded Coder.

3.3. YACCA

Yet Another Control-Flow Checking using Assertions (YACCA) technique is an inter-block CFE, has been proposed in [15]. It assigns two unique compile-time variables. The first variable is a unique compile-time signature ID_i, and the second variable is `ERR_CODE`. Initialization, 1 bit is assigned for each BB as a unique ID, `ERR_CODE` is set to 0, and the ID_i is equal to the first BB's ID. Then the program reaches the BB; the check instruction checks

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 August 2022 doi:10.20944/preprints202208.0160.v1

that if ID_i is not equal to ID, the `ERR_CODE` becomes 1. When the program exits BB, the ID_i has to update to the legal successor by operation between itself and the corresponding mask. The advantage of using this method is the possibility of detecting CFEs happening when the program flow jumps from the inside of a BB to one of its legal successors even if the successive BB gives back the control to the BB affected by the wrong jump.

3.4. RACFED

There are four steps in the compile-time process to implement RACFED:

- First, all needed compile-time variables `compile-time signature (CTS)` and (subRanPrevVal), those are random numbers, are assigned to all BBs.
- Then, the instruction monitoring is implemented with random value that called `run-time signature (RTS)`.These update instructions help to detect intra-block CFEs.
- Next, the first run time signature update by subtracting `RTS` and `subRanPrevVal` of the current BB then the signature verification instructions are inserted in each BB.
- Then, if there are more than 2 payload instaruction, `RTS` is updated by adding a random number
- Finally, the last signature update is inserted in each BB. The `RTS` is updated with an `adjustment value`. `adjustment value` will be reach based on
  sum of the (`compile-time signature (CTS)` and random numbers that assigned for payload instructions of current BB subtracting with (`compile-time signature (CTS)` and `subRanPrevVal`) of the successors BBs.

4. Experimental results

In order to assess the diagnostic coverage obtained with YACCA (an example is shown in Figure 2) and RACFED implemented by the MBSD, we hardened the benchmark application with CFC techniques and validated them with the performance assessment system described in [28]. In this study, we opt two benchmarks: *i)* timeline scheduler (TS) and *ii)* tank level (T). The first benchmark is a Finite State Machine (FSM) implementing a timeline scheduler. A timeline scheduler is a periodic task, executed thanks to a timer triggering an interrupt, in charge to run a set of tasks, in a fixed order defined by the system designer. In our benchmark, we have 15 tasks that shall be executed in a fixed order, granting each of them 200 ms time slot.
Since the scheduler is a needed core component, we chose this application as a case study. In particular, the timeline is the simplest one that it is possible to implement to allow more than one task to run into an embedded system, allowing to share the computation unit between different applications. The second one 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. 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 over-currents, shutting down the pump to avoid damage to its motor.
As this application is widespread in the automotive industry, developers can extend its idea to other applications, such as estimated battery level, which is a crucial feature in modern hybrid cars.
We prepared implementations on the source code that has been generated directly from the Simulink Stateflow chart via the Embedded Coder.
Such a benchmark application has been chosen since this kind of algorithm is expected in the automotive environment, for example, battery management during regenerative braking or in other functional safety environments to keep the right level in fire extinguisher plants.

4.1. Target platform

As the target platform, as described in [28], is RISC-V RV32I, simulated at the instruction-set level thanks to the QEMU (Quick Emulator)[29]. It is an open-source machine emulator and virtualizer written by Fabrice Bellard. Most of its parts are licensed under GNU General

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 August 2022 doi:10.20944/preprints202208.0160.v1

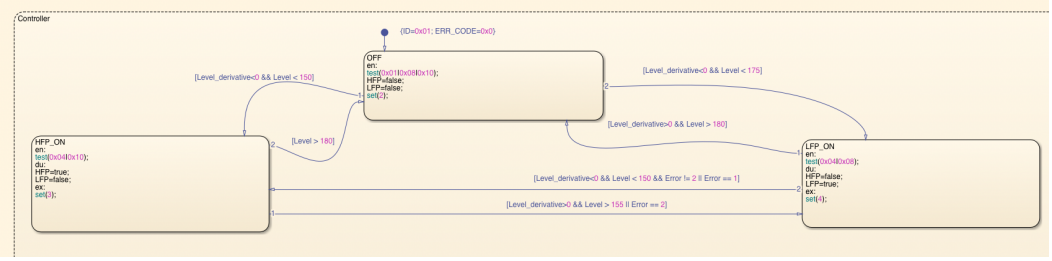

Figure 2. The YACCA-hardened tank level controller. It is possible to see the test and set functions.

Public License (GPL), others under other GPL-compatible licenses. The main reason QEMU has been used in our proposal is to make the test bench agnostic concerning the specific instruction set since it can emulate many different ISAs.
The GDB 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.

4.2. Hardening technique performance assessment

Two different versions of each benchmark application have been prepared, then two CFC methods(YACCA and RACFED) were used to harden each application *i)* timeline scheduler (TS) and *ii)* tank level (T).
The first benchmark is a Finite State Machine (FSM) implementing a timeline scheduler. A timeline scheduler is a periodic task, executed thanks to 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 one 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. 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. We prepared implementations on the source code that has been generated directly from the Simulink Stateflow chart via the Embedded Coder.

Table 2. Classifier results obtained from the fault injection campaign assessing the YACCA algorithm implemented on the timeline scheduler.

Classification result	**# of occurrences**
Latent	0
Latent after injection	110
Erratic behavior	0
Infinite loop or *Stuck at some instruction*	261
(Detected) *by SW hardening* + Safe	112 + 0
(Detected) *by HW mechanism*	512
As golden	0
False positive	0
Undefined	0
Error	0

4.3. Fault injection results

The main idea of this classification is to describe the behavior of the application after the fault has been injected[28].
To do it, seven possible outcomes have been defined:

- *Latent after injection*: fault injected and behavior identical w.r.t. the fault-free run.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 August 2022 doi:10.20944/preprints202208.0160.v1

Table 3. Classifier results obtained from the fault injection campaign assessing the YACCA algorithm implemented on Tank Level.

Classification result	# of occurrences
Latent	0
Latent after injection	791
Erratic behavior	0
Infinite loop or *Stuck at some instruction*	0
(Detected) *by SW hardening* + Safe	0 + 13
(Detected) *by HW mechanism*	2
As golden	0
False positive	0
Undefined	0
Error	0

- *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.
- (Detected) *by SW hardening*: detected by the software hardening mechanism.
- (Detected) *by HW* (mechanism): PC pointing outside the FLASH/RAM addressing space.
- *As golden*: detected and with an output identical to the golden run.

Moreover, other four outcomes *latent*, *error*, and *undefined*, not related to the application by itself but inserted to monitor the classifier, and *false positives* of the detection algorithm, are provisioned. We conducted 4 campaigns, each one of 1000 injections of a *Permanent* fault affecting the Program Counter (PC) of the target.
All the faults have been injected into the PC since the CFC can detect only those FMs directly or indirectly affecting the program flow. Considering this known limitation, we know without any need for experimental results that those 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.
The results have been obtained from the classifier for YACCA implemented on the timeline scheduler is available in Table 2 and for tank level in Table 3. Moreover, the results obtained for RACFED for the same benchmarks is reported in Table 4 and Table 5 .

Table 4. Classifier results obtained from the fault injection campaign assessing the RACFED algorithm implemented on the timeline scheduler.

Classification result	# of occurrences
Latent	0
Latent after injection	133
Erratic behavior	0
Infinite loop or *Stuck at some instruction*	167
(Detected) *by SW hardening* + Safe	305 + 0
(Detected) *by HW mechanism*	395
As golden	0
False positive	0
Undefined	0
Error	0

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 August 2022 doi:10.20944/preprints202208.0160.v1

Table 5. Classifier results obtained from the fault injection campaign assessing the RACFED algorithm implementedon Tank Level.

Classification result	# of occurrences
Latent	0
Latent after injection	771
Erratic behavior	0
Infinite loop or *Stuck at some instruction*	0
(Detected) *by SW hardening* + Safe	1 + 34
(Detected) *by HW mechanism*	0
As golden	0
False positive	0
Undefined	0
Error	0

4.4. Diagnostic coverage

The classification required to determine the DC should be expressed 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 ISO 26262) 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 results shown in Section 4.3 have been transposed into ISO 26262-compliant classifications, obtaining the results presented in Table 6.

To compute the Diagnostic Coverage (DC), we considered as detected all the faults ended up with a detection by the software, a timeout after the last SET instruction (miming the behavior of a windowed watchdog) and the one detected by the hardware (PC pointing outside the limits of the instruction memory.

For YACCA, we have a DC, respectively, of 51.8 % for the timeline scheduler(TS) and 1.86% for the tank level (T). For RACFED, we obtained a DC of 70.0% for the TS and 4.34% for T. Regarding the CFC algorithms that we opted for, we can say that RACFED is more effective than YACCA. It is an expected result since it also provides, alongside intra-block detection not exploited in our benchmark, also a two-phases signature update.

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

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

4.5. Overheads

Table 7 presents the data about the overhead of the hardening techniques considering two different aspects: the increase of the program size (evaluated in terms of increase in its text segment) and the number of actually executed machine-code instructions (a reliable metrics to estimate its effect on its execution time).

It is possible to see that the explanation on the lower DC for the T benchmarks is confirmed considering the overhead in terms of executed instructions number: for both the hardening techniques, we have a stronger impact. This result is expected: the more

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 August 2022 doi:10.20944/preprints202208.0160.v1

Table 6. ISO 26262-compliant classification of the results obtained from the fault injection campaign.

Algorithm	Detected		Undetected		False Positive
	Safe	*Detected*	*Latent*	*Residual*	
YACCA TS	0.00%	51.80%	9.10%	39.10%	0.00%
RACFED TS	0.00%	70.00%	13.30%	16.70%	0.00%
YACCA T	1.61%	0.25%	98.14%	0.00%	0.00%
RACFED T	4.22%	0.12%	95.66%	0.00%	0.00%

the hardening instructions are executed, the more they have the possibility to detect CFEs. Considering the T benchmark, the impact of the CFCs in terms of executed instructions is almost negligible (a little number of BBs transactions happen).
The huge difference in terms of text segment sizes between the two benchmarks can be explained in terms of the number of BB presents: the functions related to the hardening techniques are implemented with the *inline* option of Embedded Coder, so a greater number of BBs requires in a proportional way more C language statements.

Table 7. Data regarding memory occupation and executed instruction. T indicates the results for the algorithm applied on the tank level controller and TS indicates for timeline scheduler

Benchmatk	Text segment size [bytes]	# of executed instructions
Vanila TS	1736	3991
YACCA TS	6056 (+249%)	10771 (+170%)
RACFED TS	7320 (+322%)	7492 (+87.7%)
Vanila T	9012	33460
YACCA T	10432 (+15.7%)	33498 (+0.1%)
RACFED T	12804 (+42.0%)	33534 (0.2%)

5. Conclusions

This paper has presented the diagnostic performances of two case studies.
Following a Model-Based Software Design approach, the hardening against RHFs has been implemented at a high level of abstraction directly into the behavioral models, then automatically translated to a high-level language (in this case, C). This allowed the implementation of the CFC directly within the Simulink Stateflow environment.

For this study, we considered YACCA and RACFED as the CFC algorithms. These allow the program behavior to be observed with effective and cost-efficient error detection.

Table 6 demonstrates that the Diagnostic Coverage (Safe + Detected) achieved depends on the nature of the hardened algorithm. If the number of BB transitions increases, the CFC methods can more efficiently detect RHFs. The TS performed transitions continuously, reaching a DC of 51.8 % for YACCA and 70.0 % for RACFED.
On the contrary, for the T benchmark, where only external events trigger the transitions, the DC dropped to 1.86 % and 4.40 %, respectively.
In both cases, RACFED performed better than YACCA.
Another evident phenomenon related to the TS benchmark shows that the execution overheads for the CFC methods (both YACCA and RACFED) were similar to the assembly level implementation (c with Table 1). For YACCA we observed + 170% in the MBSD implementation and + 202.3 % in the assembly experimental, while for RACFED +87.7% and + 108.4%, respectively.

This empirical study of CFC could be beneficial for software developers and researchers, especially those involved in the automotive industry, in order to identify the most appropriate techniques for RHF detection.

 doi:10.20944/preprints202208.0160.v1

Author Contributions: Conceptualization, M.A.S, J.S. and M.V.; methodology, M.A.S, J.S. and M.V; software, J.S. and M.A.S; validation, M.V.; resources, M.V.; data curation, J.S.; writing—original draft preparation, M.A.S and J.S; writing—review and editing, M.A.S, J.S. and M.V.; visualization, J.S.; supervision, M.V.; project administration, M.V.; funding acquisition, M.V. All authors have read and agreed to the published version of the manuscript.

Conflicts of Interest: Declare conflicts of interest or state "The authors declare no conflict of interest." Authors must identify and declare any personal circumstances or interest that may be perceived as inappropriately influencing the representation or interpretation of reported research results. Any role of the funders in the design of the study; in the collection, analyses or interpretation of data; in the writing of the manuscript, or in the decision to publish the results must be declared in this section. If there is no role, please state "The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results".

 doi:10.20944/preprints202208.0160.v1

The following abbreviations are used in this manuscript:

ACFC	Assertions for Control Flow Checking
BB	Basic Block
CEDA	Control-flow Error Detection using Assertions
CFC	Control Flow Checking
CFCET	Control Flow Checking by Execution Tracing
CFCSS	Control Flow Checking by Software Signatures
CFE	Control Flow Error
CFG	Control Flow Graph
COTS	Commercial Off-The-Shelf
DC	Diagnostic Coverage
DWC	Duplication With Comparison
ECC	Error Correcting Code
ECCA	Enhanced Control Flow Checking using Assertions
FI	Fault Injection
FIM	Fault Injection Manager
FM	Failure Mode
FMEDA	Failure Mode, Effects, and Diagnostic Analysis
FSC	Functional Safety Concept
FSM	Finite State Machine
FTA	Fault Tree Analysis
FTTI	Fault Tolerance Time Interval
HSI	Hardware/Software Interfaces
I-IP	Infrastructure Intellectual Properties
IP	Intellectual Property
ISA	Instruction Set Architecture
ISO	International Standard Organization
MBSD	Model-Based Software Design
NMR	N modular redundancy
OSLC	Online Signature learning and Checking
PC	Program Counter
PLD	Programmable Logic Device
RACFED	Random Additive Control Flow Error Detection
RHF	Random Hardware Failure
RISC	Reduced Instruction Set Computing
RSCFC	Relationship Signatures for Control Flow Checking
SEooC	Safety Element out of Context
SETA	Software-only Error-detection Technique using Assertions
SG	Safety Goal
SIED	Software implemented error detection
SIHFT	Software Implemented Hardware Fault Tolerance
SS	Standby-sparing
TLC	Target Language Compiler Tool
TMR	Triple Modular Redundancy
TSC	Technical Safety Concept
TSR	Technical Safety Requirements
WD	Watchdog
YACCA	Yet Another Control-Flow Checking using Assertions

References

1. Russell J. Abbott, Resourceful Systems for Fault Tolerance, Reliability, and Safety, ACM Computing Surveys, Vol. 22, No. 1, March 1990, pp. 35–68.
2. A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors—A survey," IEEE Trans. Comput., vol. C-37, no. 2, pp. 160–174, Feb. 1988.
3. T. Austin and T. Diva, "DIVA: A reliable substrate for deep submicron microarchitecture design," in Proc. ACM/IEEE Int. Syp. Microarchi- tecture. Washington, DC, USA: IEEE Computer Society, Nov. 1999, pp. 196–207.
4. 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), Los Alamitos, CA, USA, May 2007, pp. 165–170.
5. E. Dubrova, Fault-Tolerant Design. New York, NY, USA: Springer-Verlag, 2013.
6. I. Koren and C. M. Krishna, Fault-Tolerant Systems. San Mateo, CA, USA: Morgan Kaufmann, 2007.
7. J. B. Eifert and J. P. Shen. 1995. Processor monitoring using asynchronous signature instruction streams. In Proceedings of the 25th International Symposium on Fault-Tolerant Computing, 1995, "Highlights from Twenty-Five Years'." 106. DOI:https://doi.org/10.1109/FTCSH.1995.5326
8. H. Madeira and J. G. Silva. 1991. On-line signature learning and checking: Experimental evaluation. In CompEuro'91. Proceedings of the 5th Annual European Computer Conference on Advanced Computer Technology, Reliable Systems and Applications.642–646. DOI:https://doi.org/10.1109/CMPEUR.1991.257464
9. T. Li, M. Shafique, J. A. Ambrose, S. Rehman, J. Henkel, and S. Parameswaran, "RASTER: Runtime adaptive spatial/temporal error resiliency for embedded processors," in Proc. 50th ACM/EDAC/IEEE Design Autom. Conf. (DAC), 2013, pp. 1–7. [25] G. Miremadi, J. Ohlsson, M. Rimen, and J. Karlsson. 1998. Use of time, location, and instruction signatures for control flow checking. In Proceedings of the DCCA-5 International Conference.
10. L. Parra, A. Lindoso, M. Portela, L. Entrena, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Marínez-Álvarez. 2013. Efficient mitigation of data and control flow errors in microprocessors. In Proceedings of the 2013 14th European Conference on Radiation and Its Effects on Components and Systems (RADECS'13). 1–4. DOI:https://doi.org/10.1109/RADECS.2013. 6937381
11. A. Rajabzadeh and S. G. Miremadi. 2006. CFCET: A hardware-based control flows checking technique in COTS processors using execution tracing. Microelectron. Reliab. 46, 5 (2006), 959–972.
12. T. Michel, R. Leveugle, and G. Saucier. 1991. A new approach to control flow checking without program modification. In Proceedings of the 21st International Symposium on Fault-Tolerant Computing, 1991.
13. Z. Alkhalifa, V. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Design and evaluation of system-level checks for online control flow error detection," IEEE Transactions on Parallel and Distributed Systems, vol. 10, Issue. 6, pp. 627-641, 1999.
14. N. Oh, P. P. Shirvani and E. J. McCluskey, "Control-flow checking by software signatures," in IEEE Transactions on Reliability, vol. 51, no. 1, pp. 111-122, March 2002, doi: 10.1109/24.994926.
15. O. Goloubeva, M. Rebaudengo, M. S. Reorda and M. Violante, "Improved software-based processor control-flow errors detection technique," Annual Reliability and Maintainability Symposium, 2005. Proceedings., 2005, pp. 583-589, doi: 10.1109/RAMS.2005.1408426.
16. Aiguo Li and Bingrong Hong. 2007. Software implemented transient fault detection in space computers. Aerospace Science and Technology 11, 2–3 (2007), 245–252
17. SeyyedAmir Asghari, Atena Abdi, Hassan Taheri, Hossein Pedram, and Saadat Pourmozaffari. 2012. SEDSR: soft error detection using software redundancy. Journal of Software Engineering and Applications 5, 09 (2012), 664.
18. 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.
19. J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random Additive Control Flow Error Detection," in Com- puter Safety, Reliability and Security Procedings, Cham: Springer International Publishing, 2018, pp. 220–234.
20. J. Vankeirsbilck, N. Penneman, H. Hallez and J. Boydens, "Random Additive Signature Monitoring for Control Flow Error Detection," in IEEE Transactions on Reliability, vol. 66, no. 4,

pp. 1178-1192, Dec. 2017, doi: 10.1109/TR.2017.2754548.

21. Heinecke, Harald et al.: AUTOSAR – Current results and preparations for exploitation, Eurofo-rum conference May 3rd, 2006
22. Fennel, Helmut et al.: Achievements and Exploitation of the AUTOSAR Development Partner-ship, SAE Convergence Congress, Detroit, 2006
23. AUTOSAR Main Requirements:AUTOSAR_MainRequirements.pdf
24. ISO 26262:2018 Road Vehicles –Functional Safety (2018)
25. J.Sini and M.Violante "A simulation-based methodology for aiding advanced driver assistance systems hazard analysis and risk assessment", Microelectronics Reliability, Volume 109, 2020, 113661, ISSN 0026-2714.
26. J. Sini, M. Violante, V. Dodde, R. Gnaniah, L. Pecorella "A Novel Simulation-Based Ap-proach for ISO 26262 Hazard Analysis and Risk Assessment," 2019 IEEE 25th Interna-tional Symposium on On-Line Testing and Robust System Design (IOLTS), pp.253-254,doi: 10.1109/IOLTS.2019.8854385.
27. Mathworks. Mathworks.http://www.mathworks.com, April 2022.
28. J. Sini, M.Violante and F.Tronci "A Novel ISO 26262-Compliant Test Bench to Assess the Diag-nostic Coverage of Software Hardening Techniques Against Digital Components Random Hard-ware Failures" MDPI Electronics 11, no. 6: 901.https://doi.org/10.3390/electronics11060901
29. Bellard,F: QEMU, a Fast and Portable Dynamic Translator. In ATEC '05 Proceedings of the Annual Conference on USENIX Annual Technical Conference, Anaheim Ca, USA,2005