

Guidelines for Implementing Control Flow Checking into Automotive Embedded Applications Developed with C Language

Original

Guidelines for Implementing Control Flow Checking into Automotive Embedded Applications Developed with C Language / Sini, Jacopo; AMEL SOLOUKI, Mohammadreza; Violante, Massimo. - (2023), pp. 1-6. (Intervento presentato al convegno IEEE Nordic Circuits and Systems Conference (NorCAS) tenutosi a Aalborg, Denmark nel 31 October 2023 - 01 November 2023) [10.1109/norcas58970.2023.10305466].

Availability:

This version is available at: 11583/2983636 since: 2023-11-08T09:53:39Z

Publisher:

IEEE

Published

DOI:10.1109/norcas58970.2023.10305466

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

IEEE postprint/Author's Accepted Manuscript

©2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

Guidelines for Implementing Control Flow Checking into Automotive Embedded Applications Developed with C Language

Jacopo Sini, Mohammadreza Amel Solouki, and Massimo Violante
Department of Control and Computer Engineering
Politecnico di Torino, Turin, Italy
{jacopo.sini, mohammadreza.amelsolouki, massimo.violante}@polito.it

Abstract—Embedded systems, such as automotive applications, are increasingly used in safety-critical systems. The correct and reliable implementation of such systems depends on many factors, including the design of the system hardware, software, fault-tolerance mechanisms, and the choice of programming language, followed by the test, verification, and validation techniques employed. Even well-designed systems are not exempt from having defects that stem from their physical properties, and these imperfections can cause unforeseen and dangerous actions in safety critical systems. This paper focuses on isolating or mitigating the effects of Random Hardware Failures (RHF). Hardening strategies are employed to mitigate RHF in embedded systems, either by adding specialized hardware or using Software-Implemented Hardware Fault Tolerance (SIHFT) methods. SIHFT methods are applied to various applications to harden them against Control Flow Errors (CFEs). This paper presents a guideline for applying a subset of SIHFT methods called Control Flow Checking (CFC) methods to application code written in C language. The motivation is that in the literature few guidelines can be found that provide insight on implementing CFC methods with high-level programming languages. Most proposals implement CFC methods in low-level languages such as assembly. The rationale behind developing high-level language implementations lies in the pursuit of architecture independence as well as the inadequacy of a certified compiler for the target platform that can conveniently incorporate Certified Functionally Correct into the compiled assembly/machine language code.

Index Terms—control flow checking, automotive applications, software reliability

I. INTRODUCTION

Embedded systems are becoming increasingly common in a variety of industries. These systems must be reliable and safe. These characteristics of the system depend on both system hardware and software. Several techniques can be applied to improve reliability and safety, including hardware and software redundancy to harden systems. 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. Hardware redundancy requires replicated hardware modules or custom hardware with fault detection mechanisms. Software redundancy, on the other hand, executes additional instructions without modifying any

hardware components and allows tracking the proper execution of the application. In addition, software redundancy techniques are much more adaptable and economical solutions for error detection.

Software-Implemented Hardware Fault Tolerance (SIHFT) methods are more flexible and cost-effective as they do not require adding any special hardware components, because they exploit what is already available in common Commercial Off The Shelf (COTS) products. They add extra code to the application to monitor its correct execution.

Among various SIHFT methods, this paper focuses on Control Flow Checking (CFC) methods [3-12]. CFC methods check the application's control flow to ensure it is executing correctly.

Implementing CFC methods can be challenging, and their effectiveness needs to be assessed. The literature on CFC methods provides most examples written in Assembly language. This is usually considered the most suitable strategy since compilers can automatically add the hardening method instructions to the application. However, for some embedded applications, using Assembly language is not the preferred development flow, since functional safety standards (part 6 of ISO 26262 Standard [1]) mandate the use of high-level programming languages such as C whenever possible as well as, use of certified compilers. Moreover, the automotive industry production chains are typically complex because different companies develop software components that are then integrated into the system, especially within the AUTOSAR framework. This leads to the need to guarantee a minimum acceptable hardening level regardless of the platform chosen to run the integrated products.

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

This work is organized as follows: Section II discusses the significance of C in the automotive industry as well as provides

an overview of Software-Implemented Hardware Fault Tolerance. In Section III, the article expounds on implementation best practices, while Section IV delves into the efficacy of the technique as measured through experiments. Lastly, Section V provides conclusions.

II. BACKGROUND

A. Using the C language in automotive industry applications

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

However, programming errors are relatively easy to make, and the language lacks proper support for error detection, posing a potential danger to safety-critical systems. Consequently, several constraints, such as the MISRA C guidelines [2], have been developed, limiting the use of problematic language features. In addition, various tools and techniques like static analysis tools, code reviews, and unit testings are available to enhance the security of C codes. Nonetheless, C language's weaknesses, like incomplete type checking, lack of exception handling mechanisms, and limited run-time error checking, increase the need to incorporate SIHFT to the code written in C to guarantee safety even after eliminating programming defects.

B. Software-Implemented Hardware Fault Tolerance

In safety-critical systems, it is important to use techniques that can detect and mitigate Random Hardware Failures (RHF). RHF can lead to control flow errors (CFEs) that can cause wrong execution order of instructions. CFC is a SIHFT method that can detect wrong execution order of instructions. The first step of adding CFC instructions, is to create the Control Flow Graph (CFG) of the program. The CFG is a directed graph that shows the possible execution paths through the program: the source code is divided into Basic Blocks (BBs). Each BB is comprised of lines of code in which no jump or branch instructions exist, except the last instruction of the BB which can be a jump or branch instruction to jump to another BB. BBs are the nodes of the CFG. After the BBs are determined, the code is statically analyzed to find all the possible transitions between the BBs. These transitions are the edges of the CFG. Once the CFG is complete, CFC is implemented by inserting extra instructions into the source code to check if CFG is respected. In case a transition which is not present in the CFG occurs, a Control Flow Error (CFE) has happened. A common way to implement the extra instructions for CFC methods is signature monitoring. Signature monitoring works by assigning a unique signature to each BB in the program. Extra instructions inserted into the program, can compute the signature of the current BB. The computed signature is then compared to the expected signature of the BB. If the signatures do not match, then a CFE is detected. CFC has several advantages. It is a low-cost, low-power technique that can be implemented on any COTS device. It can also be complemented by other

hardening techniques, such as watchdogs. However, CFC also has some drawbacks. For example, it adds some overheads to the program, such as an increase in the size of the occupied program memory due to the CFC instructions and execution time overhead. Despite its disadvantages, CFC is a valuable technique for detecting CFEs in safety-critical systems.

Examples of CFC methods include Enhanced Control Flow Checking using Assertions (ECCA) [3], CFC by Software Signature (CFCSS) [4], Control-flow Error Detection through Assertions (CEDAs) [5], Assertion for CFC (ACFC) [6], and Yet Another Control-Flow Checking using Assertions (YACCA) [7]. All these approaches are based on comparisons of the value of the signatures computed at run-time with their expected values assigned to each BB at the design or compile-time. It allows the detection of incorrect behavior. Relationship Signatures for Control Flow Checking (RSCFC) [8], signature monitoring methods, for instance, YACCA [7], CFCSS [4], CEDA [5], and ECCA [3]; address illegal inter-block jumps during application execution by monitoring run-time signatures with compile-time signatures at the BB level. To improve previous methods by covering illegal intra-block jumps, instruction monitoring methods, such as RSCFC [8], Software Implemented Error Detection (SIED) [9], and Random Additive Control Flow Error Detection (RACFED) [10] have been developed to check that instructions are executed in the correct order. The essential differences among these methods are in the way signatures are computed and checks are performed.

III. IMPLEMENTATION GUIDELINES

Implementation guidelines are a set of rules or suggestions that can be used for the implementation of CFCs by taking into account the specific features of C programming language. We adopted the algorithms described in [11] and [10], commonly known as YACCA, and RACFED. We chose these methods because they are based on different philosophies (bit mask vs. random numbers) and have different detection capabilities (inter-block vs. intra-block). The YACCA [11] method gives each BB entry and exit point a distinct signature. The advantage of this method is in the capability of detecting CFEs that happened when the program flow jumped from inside of one 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 because the signature is re-evaluated prior to each branch instruction to eliminate the CFE for the incorrect successor. To identify both inter-block and intra-block CFEs, RACFED [10] was created based on Random Additive Signature Monitoring (RASM) method [12]. Two gradual signature updates and one signature verification for each BB are used in RASM which is a signature monitoring method. 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







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

Fig. 1: Instructions on how to read the Control Flow Graphs represented in this paper.

application. However, RACFED extends this functionality by inserting gradual signature updates after each instruction inside the Run Time Signature (RTS) variable. To understand this guideline, Figure 1 explains how to read Control Flow Graphs. The control graph is a visual representation of the control flow of a program. The nodes in the graph represent the statements in the program, and the edges represent the control flow between the statements.

A. Functions or macros needed in C language

The YACCA and RACFED algorithms use different methods for performing the TEST and SET operations, with algorithm 1 used for TEST in YACCA and algorithm 2 used for RACFED, and algorithm 3 used for SET in YACCA and algorithm 4 used for RACFED. In RACFED, the Run Time Signature (RTS) is updated after each basic block statement by summing a random number, and the total of these random numbers is subtracted before signature checking to allow for intra-block detection capabilities. The SET operation is conducted in two phases within the actual algorithm. To optimize YACCA, the predecessor's mask is retrieved from the last TEST call in the implementation, as TEST always occurs before SET. For more details on CFC methods, see [13], which provides implementation examples.

Algorithm 1 TEST operation (YACCA)

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

B. Switch-case construct

For a switch-case construct, Figure 2 shows the positions of the TEST and SET statements for the entry BB which is indicated with the `switch(...)` statement. Meanwhile, Figure 3 shows the positions of the TEST and SET statements for the exit BB which is indicated with a `}` character.

Algorithm 2 TEST operation (RACFED). bb represents the ID of the BB which is calling TEST(). RTS is an array containing the compile-time signature of every BB.

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

Algorithm 3 Set operation (YACCA)

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

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

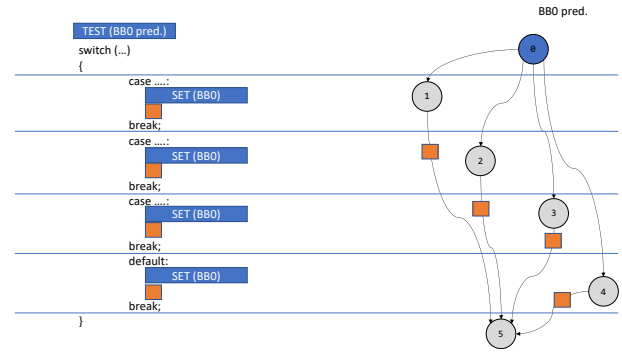


Fig. 2: Positions of the TEST and SET operations for inter-block CFE detection inside the `switch-case` constructs for the entry block (indicated as 0, in blue.)

C. If-else construct

For the `if-else` construct, Figure 4 illustrates the positions of the TEST and SET statements for the entry BB, which is indicated with the `if(...)` statement. While Figure 5

Algorithm 4 Set operation (RACFED). bb represents the current BB, while bb_{+1} its expected successor. `subRanPrevVal` and `CTS` are arrays (with lengths equal to the number of BBs) containing respectively the random number sums and the compile-time signature for every BB.

- 1: **SET**(bb, bb_{+1})
- 2: $RTS = RTS - subRanPrevVal[bb]$
- 3: $adjVal = (CTS[bb] + subRanPrevVal[bb]) + (CTS[bb_{+1}] + subRanPrevVal[bb_{+1}])$
- 4: $RTS = RTS + adjVal$

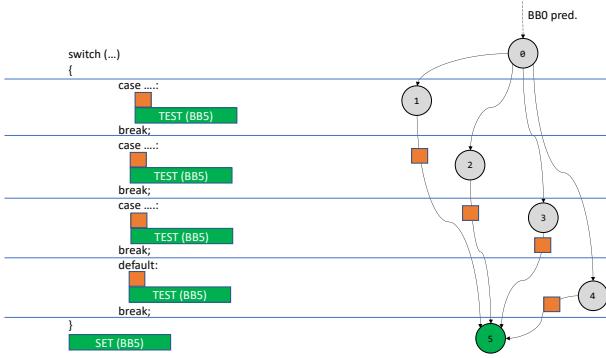


Fig. 3: Positions of the TEST and SET operations for inter-block CFE detection inside the `switch-case` constructs for the exit block (indicated as 5, in green.)

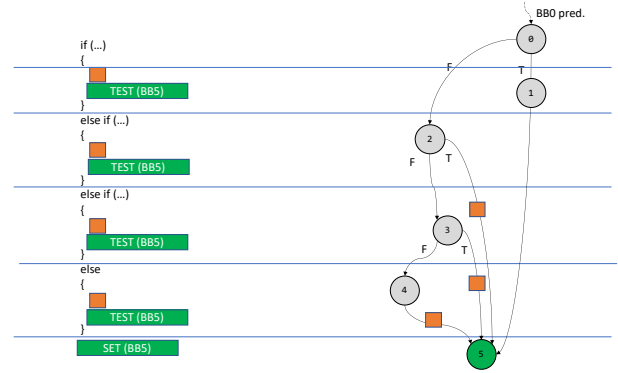


Fig. 5: Positions of the TEST and SET operations for inter-block CFE detection inside the `if-else` constructs for the exit block (indicated as 5, in green.)

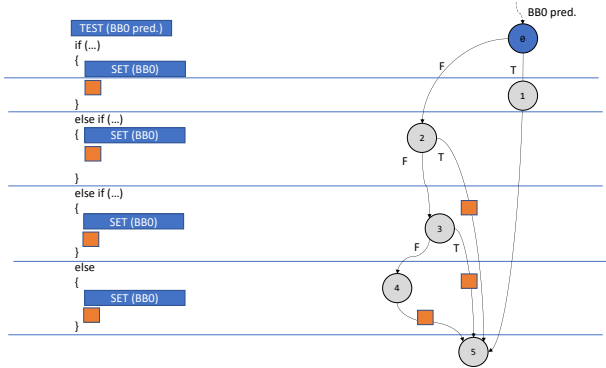


Fig. 4: Positions of the TEST and SET operations for inter-block CFE detection inside the `if-else` constructs for the entry block (indicated as 0, in blue.)

displays the positions of the TEST and SET statements for the exit block which is indicated with a `}` character.

If there is no `else` statement, an `else` statement should be included. In this situation, it is strongly recommended to add a comment clarifying that the `else` statement was inserted to the initial algorithm to facilitate the implementation of the CFC.

D. Function calls

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

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

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

The following implementation steps are taken:

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

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

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

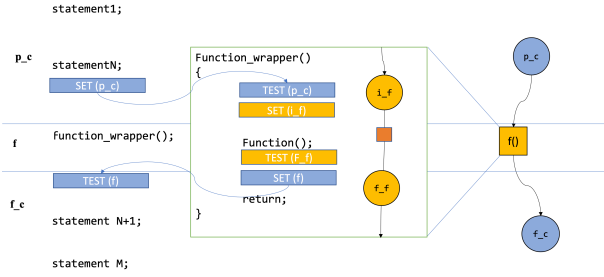


Fig. 6: Positions of the TEST and SET operations for inter-block CFE detection for a function call.

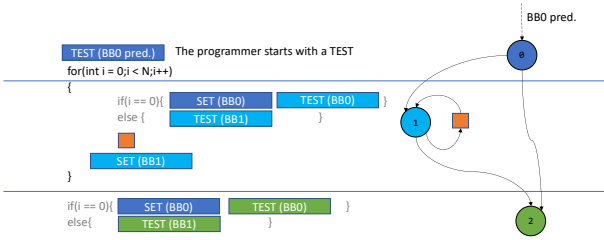


Fig. 7: Positions of TEST and SET operations to detect inter-block CFEs for a for loop.

as static variables to avoid conflicts. Local variables are used in this case.

E. For loops

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

IV. TESTING AND VALIDATION APPROACH

This section compares implementations of two established CFC methods in C language, YACCA and RACFED. We have implemented both methods in C and run them on two benchmarks: (i) Timeline Scheduler (TS) and (ii) Tank Level

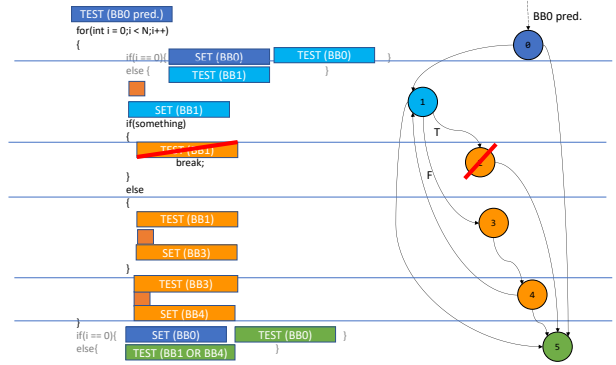


Fig. 8: Positions of TEST and SET operations to detect inter-block CFEs for a for loop containing a break instruction inside it.

Controller (T). The benchmarks were compiled using the GNU RISC-V toolchain and simulated at the instruction set level using QEMU. We used the fault injection system presented in [15] to inject permanent faults affecting the program counter. We chose this register because it directly affects the instruction flow. Two classes of detected faults were considered: (i) safe and (ii) just detected. Safe faults cannot lead to safety violations, while just detected faults are those that are detected but could potentially lead to a safety violation. The experimental results comply with ISO 26262 automotive functional safety standards [14]. Table I shows the diagnostic coverage (DC) of each CFC method for the two benchmarks. DC is defined as the ratio of the total number of faults detected to the total number of faults injected. The results reveal that RACFED outperforms YACCA in terms of DC for both benchmarks. This is attributed to the fact that RACFED can detect intra-block faults, while YACCA can only detect inter-block faults. Additionally, Table II presents the overhead of each method. In this work, two types of overheads are taken into account: (i) the increasing Text Segment Size (TSS), a measure of the amount of program memory that has been occupied as a result of the CFC instructions that were added to the program's instructions after the hardened program was compiled. As a result, the embedded system's flash memory needs to be larger. (ii) Execution time overhead, calculated as the additional number of machine instructions (# exec. instr.) required to run the hardened program given the ISA-level simulation used to run the campaigns. The overhead has been calculated, comparing the parameters measured for the hardened versions to those measured for the non-hardened versions. Although the RACFED overheads are slightly higher than YACCA's, both methods show low overhead levels suitable for real-time applications. In summary, our results show that RACFED is a more effective CFC method than YACCA. RACFED has higher DC and lower overheads, making it a better choice for safety-critical systems.

TABLE I: ISO 26262-compliant classification of the cumulative results obtained from the fault injection campaigns on the benchmarks. [14].

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 II: 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. [14].

CFC method	Benchmark	Compiler Optimization	TSS Overhead	# exec. instr. Overhead
Vanilla	T	00	9012	42593
YACCA	T	00	10512 (+16.6%)	44668 (+4.9%)
RACFED	T	00	10966 (+21.7%)	43864 (+3.0%)
Vanilla	TS	00	1736	3991
YACCA	TS	00	2496 (+43.8%)	16689 (+318.17%)
RACFED	TS	00	6271 (+261.23%)	5770 (+44.58%)

V. CONCLUSIONS

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

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

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

REFERENCES

- [1] "Iso 26262:2018 road vehicles – functional safety," 2018.
- [2] The MISRA Consortium Limited. (2023) Misra c:2023 guidelines for the use of the c language in critical systems. [Online]. Available: <https://www.misra.org.uk/>
- [3] Z. Alkhalifa, V. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 627–641, 1999.
- [4] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.
- [5] R. Vemu and J. Abraham, "Ceda: Control-flow error detection using assertions," *IEEE Transactions on Computers*, vol. 60, no. 9, pp. 1233–1245, 2011.
- [6] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," in *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003*. IEEE, 2003, pp. 137–143.
- [7] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Improved software-based processor control-flow errors detection technique," in *Annual Reliability and Maintainability Symposium, 2005. Proceedings*. IEEE, 2005, pp. 583–589.
- [8] A. Li and B. Hong, "Software implemented transient fault detection in space computer," *Aerospace science and technology*, vol. 11, no. 2-3, pp. 245–252, 2007.
- [9] B. Nicolescu, Y. Savaria, and R. Velazco, "Sied: Software implemented error detection," in *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*. IEEE, 2003, pp. 589–596.
- [10] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random additive control flow error detection," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2018, pp. 220–234.
- [11] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*. IEEE, 2003, pp. 581–588.
- [12] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random additive signature monitoring for control flow error detection," *IEEE transactions on Reliability*, vol. 66, no. 4, pp. 1178–1192, 2017.
- [13] "CFC guidelines example in C language," <https://github.com/JacopoSini/CFCGuidelinesExamples>, 2023.
- [14] M. A. Solouki, J. Sini, and M. Violante, "An experimental evaluation of control flow checking for automotive embedded applications compliant with iso 26262," *IEEE Access*, vol. 11, pp. 51 185–51 198, 2023.
- [15] 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, 2022.