Improving Software Reliability with Rust: Implementation for Enhanced Control Flow Checking Methods

Improving Software Reliability with Rust: Implementation for Enhanced Control Flow Checking Methods

Jacopo Sini[1], Mohammadreza Amel Solouki[1], Massimo Violante[1], and Giorgio Di Natale[2]
[1]*Department of Control and Computer Engineering Politecnico di Torino,Turin, Italy*
[2]*Univ. Grenoble Alpes, CNRS, Grenoble INP1, TIMA, 38000 Grenoble, France*
{*jacopo.sini, mohammadreza.amelsolouki, massimo.violante*}*@polito.it*, *giorgio.di-natale@univ-grenoble-alpes.fr*

***Abstract*—The C language, traditionally used in developing safety-critical systems, often faces memory management issues, leading to potential vulnerabilities. Rust emerges as a safer and secure alternative, aiming to mitigate these risks with its robust memory protection features, making it suitable for producing reliable code in critical environments, such as the automotive industry. This study proposes employing Rust code hardened by Control Flow Checking (CFC) in real-time embedded systems, which software is traditionally developed by Assembly and C languages. The methods have been implemented at the application level, i.e., in the Rust source code, to make them platform-agnostic. A methodology for leveraging the Rust advantages is presented, such as stronger security guarantees and modern features, to implement these methods more effectively. Highlighting a use case in the automotive sector, our research demonstrates the Rust capacity to enhance system reliability through CFC, especially against Random Hardware Faults. Two CFC algorithms from the literature, YACCA, and RACFED, have been implemented in the Rust language to assess their effectiveness, obtaining 46.5% Diagnostic Coverage for the YACCA method and 50.1% for RACFED. The proposed approach is aligned with functional safety standards, showcasing how Rust can balance safety requirements and cost considerations in industries reliant on software solutions for critical functionalities.**

***Index Terms*—Software Reliability, ISO 26262, Rust Programming Language, Fault Detection**

I. Introduction

In current computing systems, the handling of hardware faults has quickly become a primary design consideration. With process scaling and the adoption of post-CMOS technologies, the occurrence of hardware faults is expected to increase exponentially in the next decade [1], [2]. Hardware faults, including transient ones, are detrimental to building future low-power and high-performance machines. These faults can manifest as Control Flow Errors (CFEs), which are particularly challenging to tolerate as they often result in catastrophic events such as crashes, hangs, or memory corruption. Detection of CFEs is a major challenge in designing reliable and efficient computing systems [3].

The development of safety-critical systems, traditionally dominated by the C language, is prone to memory management issues, leading to undefined behavior and security vulnerabilities. Rust [4] aims to mitigate these risks by robust compile-time memory safety checks, allowing the production of more secure and reliable code. However, its adoption in safety-critical environments, particularly in the automotive industry, remains limited due to its relatively new introduction.

In this research we propose how to harden Rust applications by Control Flow Checking (CFC) methods as an alternative to the traditionally used Assembly and C languages. Our study offers a comprehensive methodology for implementing these methods using Rust, emphasizing its unique features and advantages in real-time embedded systems.

For decades, C has been the predominant choice for low-level system programming due to its zero-cost abstractions and fine control over memory layout [5]. Despite these benefits, C lacks inherent memory safety, leading to numerous vulnerabilities.

Real-time embedded systems must meet stringent requirements, delivering timely results under strict deadlines while operating with limited resources, computing power, and energy. While C/C++ remains common in this domain for their control over system resources, increasing system reliability and security demands alongside power constraints challenge traditional methods for handling hardware faults, particularly those leading to CFEs which, resulting from errors in the sequencing of instructions, can lead to catastrophic system failures such as hangs, crashes, or corrupted data. Traditional approaches, often relying on redundancy to detect or recover from these errors, become less feasible in power-constrained environments. Despite the critical need for reliable computing, methodology for implementing CFC methods in high-level languages are scarce, with most existing proposals utilizing low-level languages like Assembly.

Our research fills this gap by demonstrating the effective use of Rust for CFC implementation. We highlight the advantages of using a high-level language to achieve architecture independence and overcome the mandatory use of certified compilers, which cannot be modified to harden automatically the Assembly code. In comparing Rust to the C programming language, several critical distinctions emerge regarding software reliability [6] and CFC. Rust inherently provides protection against software defects, also known as systematic errors, through robust compile-time checks that enforce memory security and prevent common programming errors [7].

While Rust excels in aiding software defect avoidance, it does not inherently protect against Random Hardware Failures

(RHFs), which pertain to physical faults in the hardware. Therefore, additional measures, such as hardware redundancy and other fault-tolerant design techniques, are required to address RHFs. By leveraging Rust for flow checking in embedded systems, developers can achieve higher software reliability and robustness compared to traditional C implementations. The ability of Rust to detect and prevent common programming errors at both compile-time and runtime, combined with its modern tooling and safety features, makes it a compelling choice since developers can achieve higher software reliability and robustness than traditional C implementation, making it an essential tool for developing reliable embedded software. The purpose of this work is to demonstrate that it is possible to cover both software defects, thanks to the intrinsic capabilities of Rust, and hardware faults, thanks to methods like CFC. This research proposes a classification compliant with ISO 26262:2018, benefiting developers in the automotive market where software-only strategies are used to balance cost and safety requirements.

In summary, in this work we: (i) implemented a Rust application for real-time embedded systems hardened against CFEs by two existing methods, Yet Another Control-Flow Checking using Assertions (YACCA) and Random Additive Control Flow Error Detection (RACFED), described in the following of the paper and highlighted its advantages over traditional Assembly and C languages. (ii) Made a comparison between Rust and C, emphasizing inherent safety features and modern tooling capabilities of Rust. (iii) Demonstrated the effectiveness of Rust in enhancing system reliability on an automotive case study, particularly in handling permanent RHFs. The source code of the application and the detailed results from the experimental campaigns are available at the link: `https://github.com/JacopoSini/Rust4Safety_DATE2025`. (iv) Computed the experimental results in compliance the classification proposed by the ISO26262:2018 part 5 and 11 classification which is the state of the art for automotive industry developers.

The rest of the paper is organized as follows. Section II provides background information on software-based hardening methods and a discussion on the Rust programming language's advantages. In Section III, we present our methodology for implementing CFC techniques in Rust. Section IV covers the experimental results, including diagnostic coverage, overhead, and detection latency. Finally, Section V concludes the paper, summarizing our findings and offering insights into Rust's suitability for safety-critical applications.

II. Background and related work

This section provides a brief overview of software-based hardening methods and discusses the Rust programming language and its advantages. Additionally, functional safety within the automotive industry has been addressed, focusing on the ISO 26262-compliant classification.

A. Software-based hardening techniques

In safety-critical systems, it is crucial to implement techniques capable of detecting and mitigating RHFs, as these failures can result in CFEs that disrupt the correct sequence of instructions. CFC is a Software-Implemented Hardware Fault Tolerance (SIHFT) method designed to identify such erroneous instruction sequences.

The initial step in adding CFC instructions involves creating the Control Flow Graph (CFG) of the program. The CFG is a directed graph representing all the possible execution paths. The source code is divided into Basic Blocks (BBs). Each BB contains only instructions that are different from jump or branch instructions, except for the last one, which may direct execution to another BB. These BBs serve as the nodes of the CFG. Once BBs are identified, static code analysis is performed to determine all potential transitions between the BBs, forming the edges of the CFG.

With the CFG established, CFC is implemented by adding instructions into the source code to verify adherence to the CFG. If a transition not represented in the CFG occurs, a CFE is identified. A common implementation approach for CFC methods is signature monitoring. This involves assigning a unique signature to each BB. Additional instructions compute the signature of the current BB, which is then compared to the expected signature. A mismatch in signatures indicates a CFE. CFC offers several benefits, including low cost and power efficiency, and can be implemented on any Commercial Off-The-Shelf (COTS) device. It can also be used alongside other hardening techniques, such as watchdog timers. However, CFC introduces overheads, such as increased program memory usage and execution time. Despite these drawbacks, CFC remains a valuable tool for detecting CFEs in safety-critical systems.

Examples of CFC methods include Enhanced Control Flow Checking using Assertions (ECCA) [8], CFC by Software Signature (CFCSS) [9], Control-flow Error Detection through Assertions (CEDAs) [10], Assertion for CFC (ACFC) [11], and YACCA [12]. These approaches rely on comparing runtime-computed signatures with their expected values assigned during design or compile-time, thereby detecting incorrect behavior. Relationship Signatures for Control Flow Checking (RSCFC) [13], as well as signature monitoring methods like Yet Another Control Flow Checking using Assertions (YACCA) [12], CFCSS [9], CEDA [10], and ECCA [8], address illegal inter-block jumps by monitoring runtime signatures against compile-time signatures at the BB level. To enhance previous methods by covering illegal intra-block jumps, instruction monitoring techniques such as RSCFC [13], Software Implemented Error Detection (SIED) [14], and Random Additive Control Flow Error Detection (RACFED) [15] have been developed to ensure that instructions are executed within each BB in the correct order and without skipping anyone of them (intra-block detection capability). The primary differences among these methods lie in how signatures are calculated and the nature of the checks performed.

B. Advantages of Rust in Safety-Critical Applications

Rust offers strong memory safety guarantees, making it well-suited for safety-critical systems where avoiding common vulnerabilities like use-after-free, null pointer dereferencing, and data races is crucial [4]. Its *borrow checker* enforces these guarantees at compile time, eliminating many classes of runtime errors that plague C and C++ [16]. This makes Rust highly effective in systems that demand high reliability, such as automotive and aerospace applications.

The language's standard library includes robust concurrency support, with thread APIs and synchronization primitives that ensure memory safety during multi-threaded execution, verified at compile time. Rust's type system further supports the development of reliable and secure code, ensuring that complex safety-critical tasks are handled efficiently.

Compared to C, Rust provides more modern development tools and a safer ecosystem. The built-in tooling for dependency management, automated testing, and detailed compiler diagnostics enhances developer productivity and reduces the likelihood of introducing defects [4]. Rust can produce slightly larger binaries due to its safety abstractions, but this is often offset by performance gains in various workloads. For instance, studies by Wilkens and Perez demonstrated that Rust outperformed C by 26–50% in Dijkstra's algorithm and C++ by 43–78% in the Lovász Local Lemma algorithm, respectively [17], [18].

The use of Rust in embedded systems introduces key benefits: static analysis for data race prevention, flexible memory management, robust concurrency, seamless C/C++ interoperability, and portability enabled by the embedded-Hardware Abstraction Layer (HAL) [19], [20]. These features collectively make Rust a strong candidate for applications where both performance and safety are critical.

C. Functional safety in the automotive industry

The ISO 26262: Road Vehicles - Functional Safety standard, introduced in 2011, addresses the safety aspects of automotive electrical and electronic (E/E) systems, focusing on both random and systematic failures [21]. It is an automotive-specific adaptation of the IEC 61508 standard, which deals with the functional safety of general electronic systems. The standard aims to manage risks arising from malfunctioning behavior of E/E systems, whether due to RHFs or systematic issues. The latest version of the ISO 26262, released in 2018, is divided into eleven parts, encompassing all activities throughout the safety lifecycle of safety-related systems. The standard employs a top-down approach, beginning with a hazard analysis to identify potential hazards and system-level requirements [22]. The parts most relevant to our paper are the third (concept phase), fifth (hardware development), sixth (software development), and eleventh (application to semiconductors) sections. The concept phase involves defining the item and conducting hazard analysis and risk assessment to determine the associated risk level (Automotive Safety Integrity Level, ASIL), safety goals, and the Functional Safety Concept (FSC). The fifth part covers product development at the hardware level, resulting in a list of potential Failure Modes (FMs) that could impact the designed item, particularly its computation unit, which is often a microcontroller. The eleventh part, added in the 2018 update, provides guidance on applying the standard to semiconductor components. The sixth section details the technical safety requirements that must be translated into quality software safety requirements for implementation. This includes self-test and monitoring functions for the operating system, basic software, and application software.

A critical point of every toolchain adopted in the development of safety-critical systems relies on the reliability and determinism of compilers: Ferrous Systems recently qualified their open-source Rust compiler *ferrocene* under the highest integrity levels of both ISO 26262 (ASIL D) and IEC 61508 (SIL 4) [23].

III. Methodology

Two existing CFC methods, YACCA [12] and RACFED [15], have been implemented to assess the effectiveness of the proposed approach. YACCA operates by assigning two compile-time signatures to each basic block: one identifying the BB itself and another representing its valid predecessors. During execution, the runtime signatures are computed and compared against the preassigned ones. This approach ensures that CFEs are detected through bitwise operations between the predecessors mask of the BB in which the flow is entering and the current signature value, with errors leading to an increment of the error signaling variable. While YACCA's simplicity leads to low overhead and efficient implementation in both C and Assembly, its reliance on a signature variable with a bit width equal to the number of BBs becomes complex for programs with more than 64 BBs, requiring careful management of signature updates [12].

RACFED, on the other hand, uses a more complex mechanism to reduce the signature size to a constant length, in this case 64 bits, regardless of the number of BBs. It introduces a two-phase signature update process to detect both inter- and intra-block CFEs. The method assigns random values to each BB's instructions at compile-time, with runtime signatures being updated incrementally after each instruction. This ensures the detection of intra-block errors by monitoring the control flow within larger BBs (those containing more than two instructions). The adjustment values between BBs are computed and updated during the transitions, enhancing RACFED's detection capabilities over YACCA, particularly for errors that occur within BBs [15].

A. Target platform

The target platform chosen for this work is RISC-V RV32I, emulated at the instruction-set level thanks to Quick EMUlator QEMU [24] which is an open-source machine emulator and virtualizer written by Fabrice Bellard. Most of its parts are licensed under GNU General Public License (GPL), others under other GPL-compatible licenses.

B. Fault models

This study focuses on CFC methods, which can detect faults that, directly or indirectly, alter the instruction flow. For sure, those affecting the Program Counter (PC) register have a direct impact on the instruction sequence. They have not been simulated faults that impact data or cause the program to follow an incorrect but legal path present in the CFG, like those that corrupt variables used in conditional assertions (e.g., if-else statements) and lead to incorrect path selection, since the implemented CFCs cannot detect them.

To perform the campaigns it has been adopted the Fault Injection Manager (FIM) system described in [25]. During all the campaigns they have been injected "Permanent" stuck-at faults, meaning that an individual bit of the PC is permanently fixed to 0 or 1 from the time of injection until the end of the simulation. The injection time, the affected bit position, and its state, are selected randomly.

By employing these methodologies and fault models, our study aims to provide a comprehensive evaluation of the Diagnostic Coverage and overheads introduced by CFC methods, offering valuable insights for application developers in safety-critical domains.

IV. Experimental results

Six experimental campaigns have been conducted, each comprising $n = 1000$ injections of "Permanent" faults affecting the Program Counter (PC) of the target.
Two of them have been performed on both RACFED and YACCA hardened applications compiled with no optimization settings of the compiler (referred to in the following as `-O0`), and another two with the first level of optimization (indicated as `-O1`). Since the PC is 32-bit long, injecting a stuck-at fault in its most significant bits leads (i) to no effects or (ii) a considerable jump within the instruction memory, in the latter case raising a hardware trap. For the same reason, since the RV32I supports only 32-bit long instructions, they have not performed injections on the 1st and 2nd bit (since for indicating a valid instruction, both have to be 0s). The considered system features 128 MB of memory. For this reason, in those four campaigns, only faults affecting the bits from the 5th to the 13th position (considering the least significant bit as the 1st) were injected. Injecting from the 14th bit on can only lead to no effects or to triggering an HW trap, since the PC points outside the valid memory address range, while injecting on the 3rd and 4th bits jumps only one or two machine instructions, situations that will lead to an intra-block CFE. For this reason, the remaining two campaigns have been done to assess eventual differences between YACCA and RACFED since the latter provides *intra-block* detection capabilities. They have been performed on the application compiled with no optimization settings but, this time, injecting fault affecting from the 3rd to the 13th bit. These will be indicated as `-O0*`.

A. Classification of Application Behavior under Fault Injection

In this study, the application behavior in faulty conditions has been classified based on their impact on the application from the behavioral point of view. These classifications provide a comprehensive understanding of the faults effects, as summarized below:

- **Latent after injection** (indicated in the following as l): The fault is injected, but the behavior of the application remains identical to the fault-free execution.
- **Erratic behavior** (b_e): The application exhibits deviations from normal execution, indicating the presence of faults possibly affecting safety.
- **Detected by SW hardening** (d_{SW}): The fault is detected by the software-based CFC mechanisms.
- **Safe** (s) if detected and the behavior of the application (excluding the signature) does not change w.r.t. the golden run.
- **Detected by HW mechanism** (d_{HW}): Faults identified by hardware traps.

These categories provide a nuanced view of how injected faults affect the application's control flow and the effectiveness of various detection mechanisms. The detailed experimental results, are presented in Table I. In the campaigns `-O0`, it can be observed that YACCA detected 73 faults, while RACFED 57. In the campaigns `-O0*`, also involving injections into the 3rd and 4th bits, YACCA showed the same performances with 73 detections while RACFED, surprisingly since it features intra-block detection on the contrary to YACCA, reduces its detections from 57 to 46.
Considering the compiler-optimized applications (campaigns `-O1`), as expected, the number of detections decreases significantly since the CFC algorithm is only partially implemented as an effect of the optimizations, in particular considering RACFED, in which the intra-block detection statements are optimized out. The number of detections is aligned to what was obtained in the C language [26], where the same application (compiled without optimizations) they have been recorded 66 detections for YACCA and 52 for RACFED. The hardened application developed in Rust have obtained better results compared to those with C code hardened by automatically generated code resorting to Model-Based Software Design using the Embedded Coder tool of Mathworks Simulink [27] where they recorded 13 detections with YACCA and 35 for RACFED.

B. Diagnostic coverage

The outcomes presented in Section III-B were transposed into ISO 26262-compliant classifications, necessitating the calculation of Diagnostic Coverage (DC) for the evaluation of CFC methods. To be as conservative as possible from the safety point of view, it is important to remark that the signature computed by the CFC algorithm has been considered as an output of the application, so the residual percentages also consider the cases in which the signature does not correspond

TABLE I: Classifier results obtained from the fault injection campaigns assessing the YACCA and RACFED methods implemented manually directly within the Rust code on TS benchmark with different compiler optimizations. No erratic behavior has been detected. The sum for the `o1` campaigns is ≤ 1000 since the number of executed instructions is smaller and, since the injected faults in terms of affected bit and injection time are the same as for the campaigns `o0`, some injections do not happen due to the early end of the application execution.

Classification results	**YACCA**			**RACFED**		
	`o0`	`o0*`	`o1`	`o0`	`o0*`	`o1`
Erratic behavior (b_e)	398	430	157	498	501	108
(Detected) *by HW mechanism* (d_{HW})	392	496	87	444	453	100
Latent after injection (l)	137	1	11	1	0	56
(Detected) *by SW hardening* + Safe ($d_{SW} + s$)	61+12	73+0	16+0	57+0	46+0	30+1
Successful injections $n = b_e + d_{HW} d_{SW} + s + l$	1000	1000	271	1000	1000	295

TABLE II: 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 reported in the table are obtained considering an FTTI of 1000 Assembly instructions. The injection indicated as `o0*` has been performed also targeting the 3rd and 4th bits of the PC.

CFC method	**Compiler Optimization**	**Detected**		**Undetected**		**Stat. error** e
		Safe	***Detected***	***Latent***	***Residual***	
YACCA	`o0`	1.20%	45.30%	13.70%	39.80%	3.10%
YACCA	`o0*`	0.00%	56.90%	0.10%	43.00%	3.10%
YACCA	`o1`	0.34%	44.07%	18.98%	36.61%	9.00%
RACFED	`o0`	0.00%	50.10%	0.10%	49.80%	3.10%
RACFED	`o0*`	0.00%	49.90%	0.00%	50.10%	3.10%
RACFED	`o1`	0.00%	41.69%	1.02%	57.29%	9.00%

to the one (demonstrating that the injection has distrupted the execution flow) of the fault-free execution, but the CFC does not detect the mismatch in the allowed Fault Tolerance Time Interval (FTTI).

In Table II are reported the ISO26262-compliant results obtained from the six campaigns. The values are computed, given that the campaign featured n successful injections, as follows:

- **Safe** = s/n
- **Detected** = $(d_{SW} + d_{HW})/n$
- **Latent** = l/n
- **Residual** = b_e/n.

The symbols used in these formulas are those reported in Table I.

It is possible to observe that the DC, computed as the sum of the Safe and Detected outcomes $DC = (s+d_{SW}+d_{HW})/n$, is about 50%, with few latents and around 45% of residuals. This shows how the proposed approach can aid in increasing the DC at the system level when complementing other detection mechanisms but cannot be considered sufficient by itself.

C. Overheads

Table III provides comprehensive data on the overhead considerations in this study, encompassing two pivotal aspects: (*i*) an increase in the Text Segment Size (TSS), delineating the expanded program memory footprint due to the inclusion of CFC instructions post-compilation. This size increase directly impacts the embedded system's flash memory requirements. (*ii*) Execution time overhead, assessed through ISA-level simulations conducted during the fault injection campaigns, quantifies the additional machine instructions (# exec. instr.) necessary for the execution of the hardened program.

Considering both forms of overhead is crucial for embedded applications, with specific considerations for code size in resource-constrained microcontrollers and the executed instructions, a key determinant of real-time application performance. Comparing the overheads with those obtained in C [26], we observe the following. The overheads for YACCA are quite similar, with a 22% increase in Rust and a 17% increase in C for TSS overhead. In terms of executed machine instructions, Rust shows a 13% increase, while C shows a 5% increase for the case study application. In the case of RACFED, the footprint grows more significantly: a 74% increase in Rust compared to 21% in C for TSS overhead and a 16% increase in Rust versus 3% in C for executed instructions. It's important to note that the overhead in executed instructions is heavily influenced by the application: the more Basic Block (BB) transitions occur within a given time interval, the more signature updates are required, and hence, the more the number of executed instructions increases. The considered application features 38 BBs, and during the execution of the case study happens 68 transitions between the BBs.

D. Detection latency

Since this work deals with embedded software, for the version compiled with `o0` optimization settings, it has also been assessed the Diagnostic Coverage when different Fault Tolerance Time Intervals (FTTIs) were considered, computing it for values equal to 10, 100, 250, 500, 1000, 5000, and 10000 machine instructions executed from the moment of the injection up to the detection, as shown in Figure 1.

TABLE III: Data regarding memory occupation and executed instruction. Vanilla refers to the application that is not hardened from its original form. Are reported the overheads with the different optimization levels.

CFC algorithm	Compiler Optimization	TSS Overhead	# exec. instr. Overhead
Vanilla	`o0` and `o0*`	10318	90252
Vanilla	`o1`	6368	21765
YACCA	`o0` and `o0*`	12556 (+22%)	101594 (+13%)
YACCA	`o1`	9920 (+55%)	23010 (+6%)
RACFED	`o0` and `o0*`	17960 (+74%)	104629 (+16%)
RACFED	`o1`	9244 (+45%)	25440 (+17%)

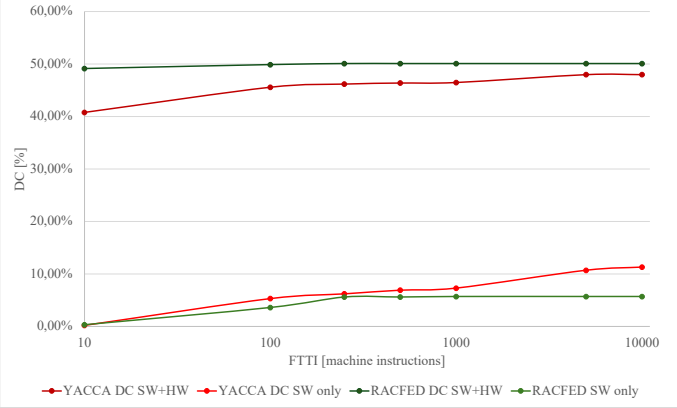

Fig. 1: Experimental results showing the DC of YACCA and RACFED compiled with no optimizations (`o0`) in function to FTTI. DC SW+HW is computed considering the cases where the PC goes outside the legal boundaries of the .text area, under the hypothesis that an independent companion chip can detect this situation.

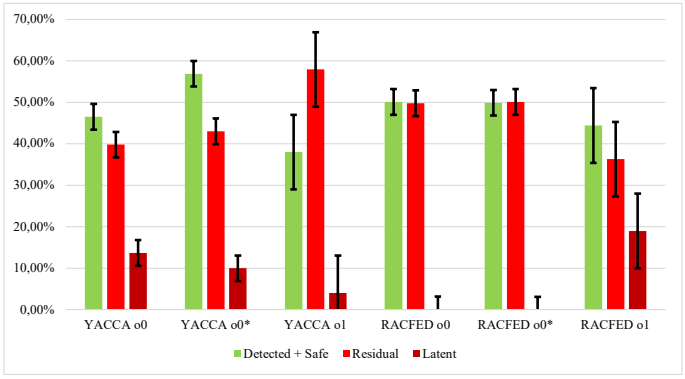

Fig. 2: Experimental results with error bands considering an FTTI of 1000 machine instructions.

E. Statistical error in the reported diagnostic coverage

Considering the statistical error of the obtained ISO26262-compliant classifications, it is possible to assume that the total number of errors to be injected (statistical population) is equal to $N = 2^{16} \cdot 10^4$, since only the addresses between `0x80000000` and `0x8000FFFF` are valid addresses for the PC to point and the considered application is composed of about 10^4 instructions. During the campaigns `o0` and `o0*` they have been injected $n = 1000$ faults, and it is possible to consider the estimate of the true value to be searched (in this case, the number of detections $p = 1/2$ and the cut-off (critical value) $t = 1.96$ (95% of confidence). Given these data, the error e can be computed thanks to the formula proposed by Leveugle in the paper [28]:

$$e = t \cdot \sqrt{\frac{p \cdot (1-p)}{n} \cdot \frac{N-n}{N-1}} = 3.1\%$$

A summary of the results with error bars, considering an FTTI of 1000 machine instructions, is represented in Figure 2. The statistical error for the classification from the campaigns `o1` is larger, around 9%, since only 271 faults (for YACCA) and 295 faults (for RACFED) have been injected before the end of the application run due to the quite smaller number of executed machine instructions w.r.t. the version compiled with `o0` optimizations.

V. Conclusions

Rust can provide significant benefits for critical systems development concerning the avoidance of systematic errors, but by itself, it does not provide an intrinsic protection against RHFs. It represents a unique point in the language design space, bringing the benefits of type and memory safety to systems that cannot afford the cost of garbage collection.

In this paper, it has been shown that Rust although is a viable language choice for those applications for which software-based detection mechanisms, implemented as CFC methods, are required. The results, in terms of DC and overhead, are comparable to those obtained in C with the same application: Rust is indeed suitable for embedded software development, although the choice must not be made without careful consideration of the examined criteria.

Considering the reached diagnostic coverage, it is possible to consider the proposed approach as suitable for low-criticality applications where other mechanisms are not available or as a complement to increase the overall DC at the system level for high-criticality systems when implemented alongside other functional safety mechanisms.

In future work, the authors will propose a back-to-back comparison between the performances of YACCA and RACFED applied to the same application written in both C and Rust programming languages.

REFERENCES

[1] Z. Zhu and J. Callenes-Sloan, "Towards low overhead control flow checking using regular structured control," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 826–829.

[2] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 2002, pp. 389–398.

[3] M. A. Solouki, S. Angizi, and M. Violante, "Dependability in embedded systems: A survey of fault tolerance methods and software-based mitigation techniques," *IEEE Access*, vol. 12, pp. 180 939–180 967, 2024.

[4] S. Klabnik and C. Nichols, *The Rust programming language*. No Starch Press, 2023.

[5] L. Seidel and J. Beier, "Bringing rust to safety-critical systems in space," *arXiv preprint arXiv:2405.18135*, 2024.

[6] A. Pinho, L. Couto, and J. Oliveira, "Towards rust for critical systems," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 19–24.

[7] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 763–779.

[8] 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.

[9] 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.

[10] R. Vemu and J. Abraham, "Ceda: Control-flow error detection using assertions," *IEEE Transactions on Computers*, vol. 60, no. 9, pp. 1233–1245, 2011.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk, "System programming in rust: Beyond safety," in *Proceedings of the 16th workshop on hot topics in operating systems*, 2017, pp. 156–161.

[17] F. Wilkens, "Evaluation of performance and productivity metrics of potential programming languages in the hpc environment," Ph.D. dissertation, 2015.

[18] A. Perez, "Rust and c++ performance on the algorithmic lovasz local lemma," *Project Report. Stanford: Stanford University, Dec*, 2017.

[19] O. K. A. Santoso, C. Kwee, W. Chua, G. Z. Nabiilah *et al.*, "Rust's memory safety model: An evaluation of its effectiveness in preventing common vulnerabilities," *Procedia Computer Science*, vol. 227, pp. 119–127, 2023.

[20] M. Sudwoj, "Rust programming language in the high-performance computing environment," B.S. thesis, ETH Zurich, 2020.

[21] "ISO 26262:2018 Road vehicles – functional safety," 2018.

[22] A. Nardi and A. Armato, "Functional safety methodologies for automotive applications," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 970–975.

[23] F. Systems, "Ferrocene: Open source qualified rust compiler toolchain," https://ferrous-systems.com/ferrocene/, 2023.

[24] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41, no. 46. California, USA, 2005, pp. 10–5555.

[25] 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.

[26] 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.

[27] M. Amel Solouki, J. Sini, and M. Violante, "Implementation of control flow checking—a new perspective adopting model-based software design," *Electronics*, vol. 11, no. 19, p. 3074, 2022.

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