Enhancing Automotive Embedded Applications: A Comprehensive Evaluation of Control Flow Checking Methods

Enhancing Automotive Embedded Applications: A Comprehensive Evaluation of Control Flow Checking Methods

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

***Abstract*—Mitigating the risks posed by Random Hardware Failures (RHFs) is crucial to prevent data corruption and Control Flow Errors (CFEs) in embedded systems. This paper addresses these concerns through the application of Software-Implemented Hardware Fault Tolerance (SIHFT) methods, emphasizing compatibility with high-level programming languages such as C. Current SIHFT methods, often implemented in low-level Assembly, present challenges in terms of overhead to code size and real-time execution. Our proposed approach focuses on pre-compilation application of SIHFT methods, specifically Control Flow Checking (CFC), to identify CFEs within C-language-based code. We conducted a comparative analysis of two established software-based CFE detection methods in C, seamlessly integrating CFC methods into the application behavioral model. Our methodology ensures ISO26262 compliance, crucial for the automotive sector, offering a software-only strategy that aligns with safety and cost considerations.**

***Index Terms*—control flow checking, functional safety, automotive applications, fault tolerance**

I. Introduction

Embedded systems play an increasingly pivotal role across military, medical, and commercial domains. Ensuring these systems exhibit high reliability and correct functionality, while mitigating reasonable risks, is paramount. The operational robustness of these systems, particularly their functional safety, hinges on the careful selection of hardware and software components.

Functional safety is a critical facet of the product safety process, emphasizing the mitigation of unreasonable risks. FuSa standards offer reference life cycles for the implementation of embedded systems, necessitating that systems perform tasks correctly within defined timeframes or, at a minimum, safely bring controlled physical processes to a halt. Predominantly rooted in IEC 61508, standards like ISO 26262 specifically target functional safety in automotive industry applications, addressing safety-critical tasks. Initially released in 2011 and subsequently updated in 2018, ISO 26262 guides the development of automotive systems, ensuring adherence to stringent safety requirements [1].

Designers operating within this paradigm strive to prevent systematic design errors and fortify systems against the impact of Random Hardware Failures (RHFs). RHFs, affecting physical components like central processing unit registers or memory locations, are unavoidable due to the inherent nature of electronic components. While systematic errors can be mitigated through meticulous life cycle implementation, RHFs demand specialized attention.

Various techniques are employed to reduce hazards and bolster safety, with a specific focus on enhancing fault tolerance levels to improve system reliability and integrity. Introducing hardening techniques, as detailed in existing literature, often involves adding redundancy through extra hardware components or software instructions in the application. However, this method may incur higher costs as it necessitates replicated hardware modules or custom hardware with error detection mechanisms. In contrast, software redundancy techniques prove more flexible and cost-effective for error detection [2]. These techniques introduce additional code without hardware modifications, facilitating the monitoring of the application's correct execution. Software-implemented detection methods, such as Control Flow Checking (CFC), have been proposed to enhance the reliability of embedded systems [3]–[9].

In the automotive sector, Model-Based Software Design (MBSD) has emerged as a well-established approach for developing applications [10]. Employing a semiformal model using tools like Mathwork Simulink, MBSD simplifies developer activities and circumvents the use of low-level languages mandated by safety standards like ISO26262 (as requested by part 6 of ISO 26262 Standard [1]). This approach particularly targets automotive applications developed to manage the safety of safety-critical systems throughout their lifecycles.

However, the implementation of CFC techniques for diverse case studies poses challenges, necessitating a rigorous assessment of their effectiveness. Existing literature often details CFC techniques using Assembly language, providing low-level implementation examples that lack portability. This paper addresses these challenges, proposing an approach that implements CFC countermeasures in high-level programming languages without compromising error detection capabilities. The experimental results align with ISO 26262 automotive functional safety standards, demonstrating the effectiveness of

the proposed CFC methods in detecting RHFs.

Our approach involves the implementation of two established CFC detection methods in C-level languages and on Model-Based Software Design (MBSD), complemented by benchmarks relevant to the automotive industry. This contribution aims to enrich the understanding of software-implemented control flow checking and provide practical insights for developers and researchers, particularly in the automotive industry.

This work is organized as follows: Section II provides an overview of hardening techniques, Section III explains our case study, while Section IV presents the experimentally measured effectiveness. Finally, Section V provides conclusions.

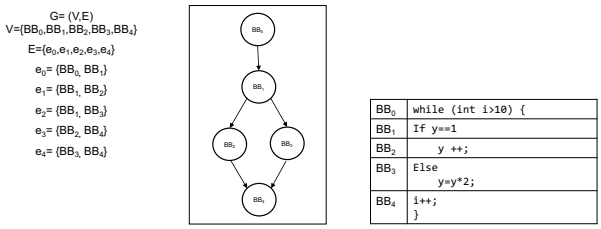

Fig. 1: Sample code and program CFG example. The execution from basic block BB_1 to BB_2 or from BB_1 to BB_3 are legal, but a jump from BB_1 to BB_4 is illegal and called Control Flow Error (CFE) [12].

II. Background

This section briefly overviews hardware-based fault tolerance techniques and software-based hardening methods. Additionally, we delve into functional safety within the automotive industry, focusing on ISO26262-compliant classification. This scheme is pivotal for assessing fault tolerance methods' effectiveness while ensuring compliance with industry-specific safety standards and regulations.

A. Hardware-based fault tolerance techniques

Hardware-based techniques have two main groups *i)* redundancy-based and *ii)* hardware monitors.The first technique relies on hardware or time redundancy, adding extra hardware components to detect and eventually reduce faults.

Hardware redundancy can be applied through *i)* passive, *ii)* active and *iii)* hybrid methods. In contrast, the second technique 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. These include, for example, watchdog processor checkers; Infrastructure Intellectual Properties (I-IP); runtime signatures using either reference signatures or verification of the integrity of signatures with error correction codes.

Hardware-based techniques have a high cost, verification, testing time, and overhead, leading to higher power consumption [11].

B. Software-based hardening techniques

The corruption of the execution order of instructions is known as a control flow error (CFE). A CFE is a violation of the control flow graph (CFG) of the program. The CFG is an oriented graph representing the program flow. Basic Blocks (BBs) are the vertices while the legal transitions between them are the edges. A BB is a sequence of consecutive instructions with exactly one entry and one exit point, meaning there is no branch or jump instructions except the last one. Figure 1 shows the sample code alongside an example of the CFG for the corresponding program.

CFEs are typically partitioned into two categories: inter-block CFEs and intra-block CFEs. An inter-block CFE is an invalid jump through the program between two different basic blocks, while an intra-block CFE is an invalid jump within the same basic block. Both types of CFE can cause the program or system to halt, crash, or provide erroneous output, potentially leading to hazardous situations.

Among the various error protection techniques available in the literature, Control Flow Checking (CFC) is considered one of the most effective for those faults affecting the control flow of embedded software. Examples of such methods are 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]. Relationship Signatures for Control Flow Checking (RSCFC) [8], signature monitoring methods like, for instance, YACCA [7], CFCSS [4] and CEDA [5], address illegal inter-block jumps during application execution by monitoring run-time signatures with compile-time signatures at the BB level. 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 time or compile-time. It allows the detection of misbehaviors. The fundamental difference among these techniques is how signatures are computed and checks are performed. To improve the previous methods and allow them to cover unallowable intra-block jumps, instruction monitoring techniques, such as the previously described RSCFC [8], Software Implemented Error Detection (SIED) [3], and Random Additive Control Flow Error Detection (RACFED) [9] are used to examine the correct order of instruction execution.

C. Functional safety in the automotive industry

The ISO 26262: Road Vehicles - Functional Safety standard, published in 2011 [1], addresses the safety aspects of automotive E/E architectures, considering both random and systematic system failures. It is an automotive-specific adaptation of the IEC 61508 standard, the functional safety focusing on general electronic systems. This means managing risks emerging from malfunctioning behavior (due to random hardware failures or systematic failures) of E/E systems The Standard is divided into eleven parts, covering all activities during the safety life cycle of safety-related systems. The process prescribed in ISO 26262 uses a top-down approach in which, hazard analysis is first 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 needed to define the risk level associated with its functionality (Automotive Safety Integrated Level, ASIL), the safety goals (SGs) to be achieved, and its functional safety concept (FSC).

The fifth phase 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. Since the computation unit is a microcontroller, part 11 introduced in the 2018 version is useful since describes the application of the Standard to semiconductor components. The sixth phase is that technical safety requirements must be detailed down to quality software safety requirements to be implemented in the software. These are self-test and monitoring functions for the operating system, basic software, and application software [1].

III. Case Study

To enhance the robustness of the software, two distinct approaches were employed. as shown in Figure 2:

1) Source Code Generation and Manual Hardening: The source code was initially generated directly from the Simulink StateFlow chart through the Embedded Coder. Subsequently, the source code underwent manual hardening. It is imperative to highlight that adhering to functional safety standards dissuades the implementation of embedded software in assembly code due to associated testing complexities.
2) Integration of CFC Techniques into Simulink Model: CFC techniques were directly integrated into the Simulink Model, reflecting the application behavior. This aligns with the MBSD commonly employed in the automotive industry. MathWorks [13], a widely adopted tool for MBSD, was selected for its compatibility and Simulink's popularity in describing behavior models. The Stateflow package within Simulink facilitated the development of Finite State Machines (FSM).

To evaluate the efficacy of our approach, we opted for two established CFC methods, namely YACCA [7] and RACFED [9]. The selection of MathWorks [13] was driven by its ubiquity and acceptance within the MBSD paradigm.

YACCA [7] is favored for its simplicity in implementation and is available in both assembly and C language. This simplicity typically results in minimal overheads in terms of instructions and code size. However, it necessitates a signature variable with a bit width equal to the number of Basic Blocks (BBs), making implementation challenging when BBs exceed 64.

RACFED [9], while more intricate, utilizes random numbers, enabling the use of a smaller signature (64 bits suffice in every scenario). It incorporates a two-phase signature update and has the capability to detect intra-block Control Flow Errors (CFEs). Theoretically, this positions RACFED to offer superior detection capabilities compared to YACCA.

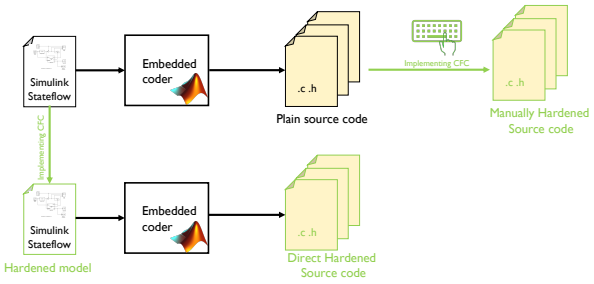

Fig. 2: Code Hardening Approaches

This case study, through the integration of these CFC methods into an MBSD framework, provides a comprehensive examination of their performance in enhancing software robustness.

A. Target platform

To assess the robustness of the proposed methodologies, we conducted benchmarking on a target platform based on RISC-V (RV32I). The emulation of this platform was achieved at the instruction-set level through the utilization of QEMU (Quick Emulator) [14], an open-source machine emulator and virtualizer.

The choice of RISC-V as the underlying architecture for our platform provides notable advantages. Specifically, it allows developers to tailor the architecture to specific applications. Furthermore, it fosters an environment open to diverse hardware vendors without incurring royalty-related constraints. This characteristic aligns with the open nature of RISC-V, empowering developers with customization capabilities.

We employed the GNU RISC-V Toolchain to compile the benchmark applications considered in this paper. This toolchain ensures compatibility and optimal performance within the RISC-V architecture.

The GNU Debugger (GDB) [15] was seamlessly integrated into our setup, serving as the primary interface to interact with the QEMU emulator. As an open-source debugger, GDB provides a robust and flexible environment for debugging and analysis. The utilization of QEMU and the GNU RISC-V Toolchain [16] in our experimental setup guarantees a standardized and replicable environment for benchmarking. This consistency is crucial for accurately evaluating the performance of the proposed methodologies.

B. Fault models

This subsection of the work focuses on a specific subset of faults, specifically those impacting the Program Counter (PC) register. The rationale behind this choice lies in the inherent capabilities of CFC methods, which are primarily designed to detect faults influencing the program flow. Consequently, faults affecting data or leading the program along an incorrect but valid path within the CFG are beyond the scope of CFC detection.

For clarity, faults such as those causing a deviation from the correct path in conditional assertions (e.g., if-else statements)

due to corruption in the associated variable are not within the purview of CFC. These faults, although potentially impactful, fall outside the detection capabilities of the chosen CFC approach. As a result, their detection does not necessitate experimental validation.

To conduct fault injection experiments, we adopted the Fault Injection system outlined in [17]. This system, presumably detailed in the referenced source, provides a robust and reliable framework for introducing faults into the system, allowing for the systematic evaluation of the CFC method's effectiveness in detecting faults affecting the Program Counter register. The specific faults injected, coupled with the chosen Fault Injection system, contribute to the rigor and reproducibility of our experimental setup.

C. ISO26262-compliant classification

This section serves as a valuable methodology for software developers, offering them a means to comprehensively assess the efficacy of their software in detecting and, when possible, mitigating failures impacting the onboard computation unit. The paper facilitates this assessment by presenting classifications derived from simulation results in a format compliant with ISO26262 standards. In the context of real-time safety-critical systems, our focus centers on two key aspects: *i)*Diagnostic Coverage (DC) and *ii)* Overhead. Diagnostic Coverage, a key metric, quantifies the effectiveness of the detection mechanism. Simultaneously, we prioritize assessing the overhead regarding executed instructions, as preserving real-time performance is imperative. Formally, the N results, where N represents the number of injected faults, are categorized as follows:

- N the number of injections;
- L the number of "latent after injection" outcomes;
- D_{HW} the number of simulations where a hardware mechanism has detected the RHF;
- D_{SW} the number of simulations where the RHF has been detected by the CFC;
- U the experiment in where the application entered an "infinite loop", remained "stuck at some instruction", or presenting an "erratic behavior".

$$\text{Safe} = \frac{\text{As golden}}{N}$$

$$\text{Detected} = \frac{D_{HW} + D_{SW}}{N}$$

$$\text{Latent} = \frac{L}{N}$$

$$\text{Residual} = \frac{U}{N}$$

$$\text{False positive} = \frac{\text{false positive}}{N}$$

The Diagnostic Coverage (DC) is calculated using the formula:

$$\text{DC} = \frac{DSW \text{ (Number of simulations where the RHF is detected by Control Flow Checking)}}{N \text{ (Total number of fault injections conducted)}}$$

This equation provides a quantitative measure of the effectiveness of the CFC methods in detecting RHFs during fault injection simulations. This metric serves as a crucial indicator of the diagnostic capabilities of the implemented fault detection mechanism.

IV. Experimental results

In this study, we opted for two benchmarks: *i)* timeline scheduler (TS) and *ii)* tank level (T).

The first benchmark (timeline scheduler (TS)) was 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 of running a set of tasks in a fixed order defined by the system designer. In our benchmark, we had 15 tasks that to be executed in a fixed order, granting each of them a 200 ms time slot.

The second benchmark (tank level (T)) was 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, shutting down the pump to avoid damage to its motor.

These benchmark applications have 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.

A. Fault injection results

Seven distinct outcomes were considered in classifying the application's behavior, each providing insights into the impact of injected faults on the Program Counter (PC) register flow. These outcomes include "Latent after injection," where the fault is injected, but the behavior remains identical to the fault-free run, and "Erratic behavior," indicating a deviation from the normal execution. Additionally, classifications like "Infinite loop" and "Stuck at some instruction" capture specific behaviors related to the PC register, revealing insights into potential issues caused by the injected faults.

Furthermore, the classification extends to detection mechanisms, with distinctions like "(Detected) by SW hardening," signifying detection by the Control Flow Checking (CFC), and "(Detected) by HW (mechanism)," indicating detection triggered by hardware traps. Notably, a category labeled "As golden" differentiates from "Latent after injection," representing detected faults that do not impact the application's output.

We conducted multiple injection campaigns alongside these classifications, each introducing 1000 *Permanent* faults affecting the target's PC. *Permanent* faults imply a scenario where a bit inside the affected register remains permanently stuck at 0 or 1 from the moment of injection until the end of the simulation. Injection characteristics were randomly chosen, including injection time, affected bit position, and state.

It is important to note that the sum in Table I may be less than 1000 when some injections fail due to the random injection time occurring after the algorithm's end.

TABLE I: Classifier results obtained from the fault injection campaign assessing [18] [12].

Approach	Classification result	T Benchmark		TS Benchmark	
		MBSD	C-Level	MBSD	C-Level
YACCA	Latent after injection	791	883	110	166
	Erratic behavior	0	29	0	0
	Infinite loop	0	20	261	142
	(Detected)bySW +Safe	0+13	26+40	112+0	141+0
	(Detected)byHW +Safe	2	2	512	551
RACFED	Latent after injection	771	945	133	83
	Erratic behavior	0	0	0	0
	Infinite loop	0	0	167	314
	(Detected)bySW +Safe	1+34	0+52	305+0	79+0
	(Detected)byHW +Safe	0	3	395	524

TABLE II: ISO 26262-compliant classification of the results obtained from the fault injection campaign [18] [12].

Approach	CFC Methods	Benchmarks	Detected		Undetected		False Pos.
			Safe	Detected	Latent	Residual	
MBSD	YACCA	T	1.61%	0.25%	98.14%	0.00%	0.00%
	RACFED	T	4.22%	0.12%	95.66%	0.00%	0.00%
	YACCA	TS	0.00%	51.80%	9.10%	39.10%	0.00%
	RACFED	TS	0.00%	70.00%	13.30%	16.70%	0.00%
C-level	YACCA	T	4.00%	2.80%	88.30%	4.90%	0.00%
	RACFED	T	5.20%	0.3%	94.50%	0.00%	0.00%
	YACCA	TS	0.00%	69.20%	16.60%	14.20%	0.00%
	RACFED	TS	0.00%	60.30%	8.30%	31.40%	0.00%

TABLE III: 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. [18] [12].

Approach	CFC Methods	Benchmarks	TSS Overhead	# exec. instr. Overhead
MBSD	Vanilla	T	9012	33460
	YACCA	T	10432 (+15.7%)	33498 (+0.1%)
	RACFED	T	12804 (+42.0%)	33534 (+0.2%)
	Vanilla	TS	1736	3991
	YACCA	TS	6056 (+249%)	10771 (+170%)
	RACFED	TS	7320 (+322%)	7492 (+87.7%)
C-level	Vanilla	T	9012	42593
	YACCA	T	10512(+16.6%)	44668(+4.9%)
	RACFED	T	10966(+21.7%)	43864(+3.0%)
	Vanilla	TS	1736	3991
	YACCA	TS	2496 (+ 43.8%)	4182 (+ 4.9%)
	RACFED	TS	6271 (+ 261%)	5770 (+ 44.6%)

This comprehensive classification system, coupled with injection campaign results, offers a thorough understanding of the application's response to permanent faults and the effectiveness of the detection mechanisms.

B. Diagnostic coverage

The outcomes presented in Section IV-A were transposed into ISO 26262-compliant classifications, necessitating the calculation of DC for the evaluation of CFC methods. Table II reveals that the RACFED method outperformed the YACCA method, aligning with expectations due to its additional features, such as intra-block detection and a two-phase signature update, which were not utilized in our benchmark. It is noteworthy that no "safe" detected failures were observed for the TS benchmark, while "safe" detected failures predominated in the T benchmark for both MBSD and manual hardening approaches. This distinction underscores the methodological variations in the benchmark scenarios, contributing to nuanced detection outcomes for the two benchmarks under different hardening techniques.

C. Overheads

Table III provides comprehensive data on the overhead considerations in this study, encompassing two pivotal aspects: (*i*) the augmentation in Text Segment Size (TSS), delineating the expanded program memory footprint attributable 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.

Delving into 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. The correlation between CFE detection capabilities and the introduction of hardening instructions is evident. The analysis reveals that the lower DC observed in the T benchmarks aligns with the overhead in terms of the number of executed instructions. Notably, variations in text segment sizes between benchmarks arise from implementing hardening techniques using Embedded Coder's inline option.

V. Conclusions

In the realm of software-implemented Control Flow Error (CFE) detection techniques, the literature lacks comprehensive guidelines aiding developers and researchers in method selection and implementation within models or high-level programming languages. To address this gap, we conducted an experimental study implementing two established CFE detection techniques using a Model-Based Software Design approach and the C language. Our investigation focused on evaluating these techniques across three critical criteria: *i)* diagnostic coverage, *ii)* code size overhead, and *iii)* the number of executed instructions, directly impacting the worst-case execution time of the application.

This study serves as a valuable resource for software developers and researchers, particularly those engaged in the automotive industry, offering insights into the most suitable CFE detection techniques for mitigating random hardware failures. Furthermore, the portability ensured by high-level programming language or model implementations extends the applicability of this approach, making it feasible for adoption in AUTOSAR-compliant applications [19], [20]. Our proposed methodology aligns with technical requirements in scenarios where code independence from the platform is paramount.

Our methodology exhibits potential for broader application across diverse industrial domains, such as unmanned aerial vehicles, owing to its comprehensive nature surpassing the confines of the automotive sector. In this context, we have opted for automotive industry benchmarks to cater to the requisites of automotive functional safety standards concerning the utilization of high-level programming languages.

References

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

[2] M. A. Solouki, S. Angizi, and M. Violante, "Dependability in embedded systems: A survey of fault tolerance methods and software-based mitigation techniques," *arXiv preprint arXiv:2404.10509*, 2024.

[3] 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.

[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] 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.

[10] K. Vinoth Kannan, "Model-based automotive software development," in *Automotive Embedded Systems: Key Technologies, Innovations, and Applications*. Springer, 2021, pp. 71–87.

[11] S. Safari, M. Ansari, H. Khdr, P. Gohari-Nazari, S. Yari-Karin, A. Yeganeh-Khaksar, S. Hessabi, A. Ejlali, and J. Henkel, "A survey of fault-tolerance techniques for embedded systems from the perspective of power, energy, and thermal issues," *IEEE Access*, vol. 10, pp. 12 229–12 251, 2022.

[12] 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.

[13] T. M. Inc., "Matlab version: 9.13.0 (r2022b)," Natick, Massachusetts, United States, 2022. [Online]. Available: https://www.mathworks.com

[14] 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.

[15] "The gnu debugger [online] available," https://www.gnu.org/software/gdb/, 2022.

[16] "Gnu risc-v toolchain [online] available," https://github.com/johnwinans/riscv-toolchain-install-guide, 2022.

[17] 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.

[18] 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.

[19] "AUTOSAR Main Requirements," https://www.autosar.org/fileadmin/standards/R22-11/FO/AUTOSAR_RS_Main.pdf, accessed: 2024.

[20] S. Fürst and M. Bechter, "Autosar for connected and autonomous vehicles: The autosar adaptive platform," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, 2016, pp. 215–217.