Novel Control Flow Checking Implementations for Automotive Software

# Novel Control Flow Checking Implementations for Automotive Software

Francesco Cosimi
*Huawei Research Center and University of Pisa*
Pisa, Italy
francesco.cosimi@phd.unipi.it

Jacopo Sini
*Control and Computer Eng. Politecnico di Torino*
Turin, Italy
jacopo.sini@polito.it

Antonio Arena
*Functional Safety Team Huawei Research Center*
Pisa, Italy
antonio.arena1@huawei.com

Prof. Massimo Violante
*Control and Computer Eng. Politecnico di Torino*
Turin, Italy
massimo.violante@polito.it

***Abstract*—Safety-critical applications shall be implemented on highly dependable systems, and a part of their reliability is based on checking if the software is executed correctly. Various techniques are available for this purpose, like Control Flow Checking (CFC). Many CFC algorithms can be found in the literature, but their detection performances are assessed in theoretical scenarios, when implemented in Assembly language. The international standard on functional safety for automotive applications is ISO26262. It mandates to develop using high-level programming languages and the computation of the Diagnostic Coverage (DC). The DC measures the effectiveness of the chosen hardening method, in order to detect various Failure Modes (FMs). This paper discusses two alternative solutions, one software-only, and the other involving customized hardware, for these concerns: (i) address the FMs affecting the computation units described by Table 30 of part 11 of the ISO26262 (ii) guarantee the Freedom From Interference between the hardening method and the monitored entity.**

***Index Terms*—Functional Safety, Control Flow Check, Hardware Acceleration, Interference, ISO26262**

## I. Introduction

Every modern vehicle is equipped with many electronic systems, and due to the complexity of the application, it is necessary to analyze the potential risk of malfunctioning. Since 2018, a new version of ISO 26262 has regulated safety of onboard electronic devices [1]. This Standard classifies Functional Safety (FuSa) risks through an Automotive Safety Integrity Level (ASIL).

The execution of a safety-related task on the processing unit of an Electronic Control Unit (ECU) can be supported by peripherals that cover certain FuSa aspects. For example, a Performance Monitoring Unit (PMU) and Error Management Unit (EMU) [2], may be useful to monitor the health of the hardware (HW) and tasks recovery action when/where needed. On the other hand, Execution Tracing Unit [3] might help to reconstruct the flow of the task and to perform timing validation even for multi-core applications. Moreover, if a semiconductor component is developed as a part of an item compliant to the ISO 26262, it shall be implemented on the base of hardware safety requirements, derived from the top-level safety goals of the item, through the technical safety concept [4].

In particular, if the safety-related functionality requires to implement ASIL-D software (SW) safety requirements, usually the corresponding hardware, where the code is executed, shall fulfill requirements with the same ASIL against systematic hardware faults. By the way, the absence of systematic defects is not the only target of FuSa, even the possibility of Random Hardware Failures (RHFs), due to the physical failure of hardware elements, has to be considered. The FuSa standard requires avoiding the presence of systematic defects by application of a formal development process, called "safety life-cycle", and the provision of mechanisms to detect and mitigate RHFs. However, traditional automotive multi-core microcontrollers do not guarantee the same ASIL level; thus, the integrator shall find some countermeasures to run ASIL-D software requirements on cores with lower ASIL.

This paper proposes a novel strategy based on ASIL-Decomposition of Technical Safety Requirements (TSRs) aimed at controlling semiconductor failure modes through software safety mechanisms. In particular, the ISO26262 Part-11 table 30 has been taken as reference, where failure modes affecting digital components are listed and categorized by the part/subpart they involve [4].

Among the various RHFs detection techniques available in the literature, for those faults affecting the control flow of embedded automotive software, Control Flow Checking (CFC) is considered one of the most effective ones. CFC is aware of which are the correct transitions between regions of instructions called Basic Block (BB) and can verify if possible hazards are caused by failures occurring in safety-related systems.

This paper contains two novelties:

- A software-only solution implemented in the C language. For this case, ISO26262-compliant results on its effectiveness have been obtained thanks to the testbench presented in [12] in a realistic automotive scenario.
- A hardware accelerated solution (with an external custom peripheral) to improve performance and guarantee Freedom From Interference (FFI).

Both the solutions are implementation of Yet Another Control-Flow Checking using Assertions (YACCA) [5] algorithm, based on comparing the value of the signatures computed at run-time with their expected values assigned to each Basic Block (BB) at the design or compile-time.

The paper is organized in six sections. First section (this) gives an introduction about Functional Safety developing methods. Section II introduces CFC techniques. Section III shows how CFC can be applied, through ASIL-Decomposition. Then, Section IV presents a SW-only implementation for a Parking Pawl and its results. Section V is about a hardware acceleration of YACCA algorithm with a dedicated peripheral. Section VI concludes the paper and introduces future works.

## II. Background

### A. Software-based hardening techniques

The main idea beyond the CFC logic is to verify if the program performs tasks (and/or instructions) in the correct order. It is based on signature monitoring, and it does not require special hardware or operating system. The idea is to insert some redundant instructions into the source code; this makes it low cost and adaptable to any Commercial Off-The-Shelf (COTS) device. These inserted instructions increase the execution time, and their effects on real-time application shall be assessed.

If we analyze the theoretical aspects, CFC relies on the concept of Control Flow Graph (CFG). CFG is an oriented graph, composed of a set of vertices, each one representing a BB, and a set of edges, representing the legal transitions between the BBs. BBs are defined as regions of the code without any branch or jump instructions except for the last one. If, at run-time, a transition not present in the transitions-set, happens, we are in presence of a Control Flow Error (CFE), detected by CFC. To harden a piece of code with CFC, we need three steps:

- CFG is generated.
- At compile-time (a-priori) values of signatures are assigned to each BB.
- Real-time signature computation and comparison methods are added to the original application.

During hardened software component execution, the signature values computed at run-time are compared with the compile-time ones. In the case a CFE is detected, an error variable is set, allowing to trigger possible mitigation strategies.

### B. Common Control Flow Check Algorithms

Common CFC algorithms are Enhanced Control Flow Checking using Assertions (ECCA) [6], CFC by Software Signature (CFCSS) [7] and Yet Another Control-Flow Checking using Assertions (YACCA) [5]. The essential difference among these techniques is how signatures are computed, and how checks are performed. The methods mentioned above update signature only before performing a BB transition, limiting them to detect only CFEs causing illegal transitions between them (this is called inter-block detection).

In order to obtain more coverage for these methods, detecting if some of the BB instructions are skipped or repeated (intra-block detection), new algorithms can be introduced. Relationship Signatures for Control Flow Checking (RSCFC) [8], Software Implemented Error Detection (SIED) [9], and Random Additive Control Flow Error Detection (RACFED) [10] update their signatures after each statement, allowing to check if they are executed in the correct order.

## III. Road to CFC and Some Solutions

In the scenario of a multi-core microcontroller, in which each core can have a different ASIL, decomposition might be applied, by splitting the Technical Safety Requirements (TSRs) with highest ASILs in two: the computing-intensive part on lower integrity cores, and the checking of the latter proper functionality on higher-integrity ones.

This strategy is aimed to cover the failure modes of digital components with lower ASIL as specified in ISO 26262-11 Table 3, by introducing software safety mechanisms. The idea is to decompose the TSR in hardware and software requirements, assuming that all the validation and testing activities are applied by the integrator.

The considered application to be hardened, as explained in the subsections III-B and III-C, is in charge to control the steering lock of a car. Since an engagement of the steering lock during the driving can provoke an accident, the steering lock functionality is safety related and classified as ASIL-D.

### A. Technical Safety Concept (TSC) example

This section provides an example of TSC, assuming that an ASIL-D TSR is allocated to the system under analysis. The generic TSR can be specified as follows:

**TSR**: In case of failure, the system shall apply the safety related function within 100ms, ASIL-D.

By following the usual requirements deployment process [11], the corresponding HW and SW Safety Requirements (HWSR and SWSR) inherit the same ASIL of parent TSR. The objective is to be able to satisfy the TSR, and the corresponding hardware and software requirements, also in those scenarios where, because of task mapping constraints, the safety related function cannot be completely executed on cores with the highest safety integrity (e.g. no lockstep cores).

### B. Dedicated CPU Approach (Multi-core)

This scenario assumes that multiple safety related functionalities are already allocated to the core(s) with highest integrity, and that other safety related requirements cannot be allocated on those. The strategy is to apply ASIL decomposition on TSR mentioned in Section III-A and obtain:

- **HWSR1**: The MicroController Unit (MCU) shall detect failures of the hardware elements involved in the execution of the task, ASIL-X(D).
- **HWSR2**: The MCU shall detect failures of the hardware elements involved in the Outbound External Communication MCU Function, ASIL-X(D).
- **SWSR1**: The application SW shall detect unintended output and shall react within 100ms, ASIL-D.
- **SWSR2**: The Basic Software (BSW) shall detect MCU failure modes, listed in ISO26262-11 Table 30, ASIL-Y(D).
- **TSR2**: Sufficient independence shall be demonstrated between the elements implementing SWSR2, HWSR1 and HWSR2, ASIL-D.

where ASIL-X + ASIL-Y = ASIL-D.

Thus, the proposal is to allocate the SWSR1 to the core with lower integrity (ASIL-X). On the other hand, SWSR2 shall be allocated on cores with higher integrity in order to fulfill the independence constraints. At the same time, the two HWSR are fulfilled by the core with lower integrity. This solution is feasible provided that the effectiveness of the SW safety mechanisms, allocated on the core with the highest integrity, complies with the initial safety requirements.

### C. Use a Dedicated Hardware Peripheral

In this solution we assume that the ASIL decomposition applied in the previous chapters is implemented by a dedicated HW peripheral. This scenario may happen in those cases where a customized hardware can be developed or modified during the design of the system. The increasing trend of RISC-V architectures for automotive applications opens the door

to modifying the host hardware accordingly to needs to (i) achieve the requested safety integrity level and (ii) minimize the performance overhead. The safety requirements HWSR1, HWSR2 and SWSR1 are still the same as in subsection III-B. Instead, HWSR3 replaces SWSR2 as follows:

- **HWSR3**: The MCU shall detect MCU failure modes as described in ISO26262-11 Table 30, ASIL-Y(D).
- **TSR2**: Sufficient independence shall be demonstrated between the elements implementing HWSR1, HWSR2 and HWSR3, ASIL-D.

where ASIL-X + ASIL-Y = ASIL-D.

To maximize the performance and obtain a fully-integrated device, the HWSR3 can be allocated to the CFC peripheral and can be sided by the implementation of custom instructions.

## IV. YACCA for FFI Approach

This section describes the results obtained by the implementation of the TSC discussed in the subsection III-B, on an emulation of a RISC-V RV32I microcontroller. The used testbench has been described in [12]. It comprises a Fault Injection Manager in charge of injecting the faults in the memory and register file. Moreover, it features a classifier that, by comparing each fault-affected run with the fault-free, labels each run to fill in the simulation results as in Table I.

TABLE I
ISO26262-11 simulation results.

| Failure mode | Injected faults | Detected | Detection Coverage |
|---|---|---|---|
| CPU_FM2: unintended instruction(s) flow executed | 1000 | 230 | 23% |
| CPU_FM3: incorrect instruction flow timing (too early/late) | 964 | 219 | 22.7% |
| CPU_FM4: incorrect instruction flow result | 36 | 11 | 30.6% |

These data came out by performing 1000 injections, on a random bit from the 16th to the 4th of the Program Counter (PC), to be sure that the fault will modify the program's flow. Repeating the same experiment is needed to obtain a figure of the Diagnostic Coverage (DC) YACCA algorithm can achieve in a realistic automotive scenario.

### A. Criticisms on FFI

As shown in the experimental results, the DCs are low, and this is particularly critical considering that the nominal functionality (the steering lock) is safety related. Another point of concern regards the necessity to guarantee real-time performances. This aspect is twofold: (i) the CFC does not have to interfere with the real-time performances of the application/system and (ii) the CFC is generally not capable by itself to detect failures that increase the execution time. To address these concerns, in Section V, we propose the use of the HW peripherals.

## V. YACCA Dedicated Hardware Peripheral

This section is about the implementation of the TSC described in III-C, by realizing a custom HW peripheral based on YACCA algorithm. Here we highlight the improvements brought by hardware acceleration and the correlated weaknesses. Following list tries to evaluate which are the main advantages of HW-acceleration:

- *Test* and *Set* functions in RISC-V can be atomic, reducing timing overhead, because the bits in the masks can be set without protection.
- Ease time measurement between consecutive *Test* and *Set*.

On the other hand, implementing new transistors and connections brings disadvantages:

- Extend area/power overhead, and higher RHF probability.
- Limit the maximum number of monitorable tasks and nodes, according to the size of the ID/bitmask (i.e. YACCA) or the amount of monitoring blocks.

A way to mitigate these issues is showed in the next section.

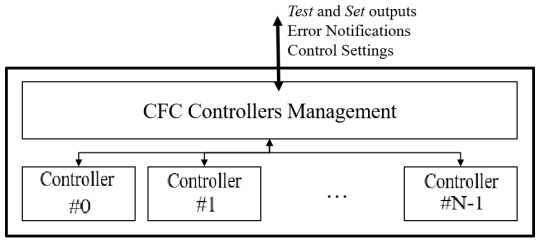


Fig. 1. Block diagram of a CFC peripherals with N controllers and a Controller's Management block.

### A. YACCA Architecture

The aspects of hardware custom implementation we are more interested in are the capability of covering CFE, the reduction of time overhead and guarantee independence between monitor and monitored items. Figure 1 shows the top-level architecture using a block diagram. In the picture we can see N blocks performing YACCA algorithm and a Management supports supervision, control and management of settings of sub-blocks. Each controller contains the sub-blocks shown in Fig. 3. YACCA equation (1) is implemented in combinatory logic.

$$error = [(ID\&(\sim predecessors\_mask)) > 0]?1 : 0 \quad (1)$$

In a software implementation of CFC the mask and the ID may assume any size, according to monitored application's number of BBs. The hardware in this case has harder constraints, but it can be adapted adding some logic. As shown in Fig. 2 we can add more replicas of the same logic (X1, X2, X3...) and allow the selection of the right ones, corresponding to the maximum number of Basic Blocks in the widest task. In Fig. 2 we can allow a maximum of "n-bit X Cn" BBs for a "X Cn" selection, where Cn is the number of replicas, and n-bit is the number of bits contained in each of them. For example, a typical automotive implementation has about 200 BBs, which means we have to implement a minimum of Cn = 25 blocks of combinatory YACCA 8-bit logic (8-bit X 25).

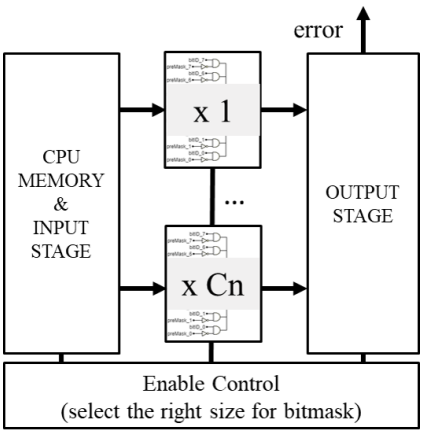


Fig. 2. Block diagram of the logic needed to change the maximum number of BB allowed in the monitored application.

In Fig. 3 the device detects the *Set* and *Test* functions' outputs. Every time one of the functions is executed and detected the counters are increased by +1. Control Logic is enabled and start checking only if the values of the two instructions' counters match. The timer has nearly the same function of a Window Watchdog, if there is a mismatch between the the counters for too long (timeout) *Error* is set.

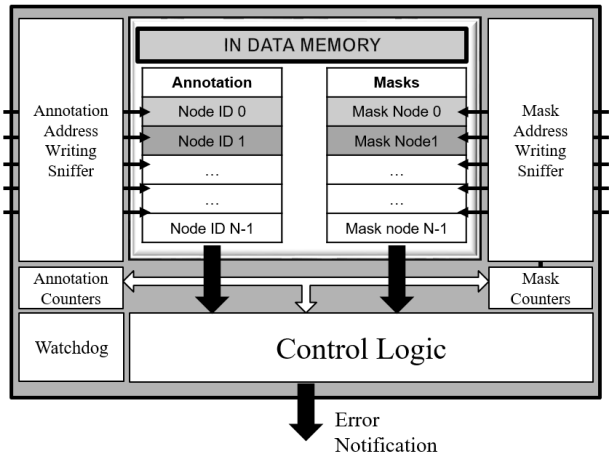


Fig. 3. This picture shows how the function detection mechanism, the WhatchDog Timer and the Control Logic are linked inside a controller.

### *B. A Solution to the Number of Monitorable Tasks Constraint*

To overcome problems due to peripheral size, and allow the monitoring of more tasks than the number of controller in Fig. 1, we can add dynamism to the way the Controllers are managed. In fact the N controllers in the device can be divided in two categories: fixed ones and dynamic ones. The fixed controllers will always check the same tasks, never changing. The dynamic ones' control registers will be updated at every context change, cooperating with the scheduler.

### *C. Analysis of Failure Coverage for YACCA peripheral*

The identified Failure Modes No Annotation/Mask transferred as requested, Annotation/Mask transferred when not requested, Annotation/Mask transferred too early/late are covered by timer/watchdog detecting a non-negligible delay (timeout) between Annotation/Mask updates. Moreover, Failure Mode Annotation/Mask transferred with incorrect value is mitigated by the Control Logic, which detects the mismatch between the Mask and the wrong Annotation [4].

## VI. Conclusions and Future Works

This paper presents two alternative Technical Safety Concepts, one software-only, the other with custom hardware, aimed to detect the Failure Modes of computation units described in Table 30 of ISO26262 part 11. For the first proposal are also reported data about its Diagnostic Coverage. Two main reasons lead to develop hardware implementations, when customization is an option: (i) implement the Freedom From Interference by design, to perform ASIL- decomposition, and (ii) reduce the time overhead, particular concern in real-time applications.

In future works, the following aspects will be considered:

- Measure of the DC of YACCA in AUTOSAR scenarios by performing fault injection on COTS microcontrollers.
- Comparison between the DCs obtained with HW-based and SW-only implementations.
- Complement CFC with data hardening techniques.

## References

[1] A. Ismail, W. Jung, "Research Trends in Automotive Functional Safety", 2013 International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (QR2MSE), Chengdu, China
[2] F. Cosimi, F. Tronci, S. Saponara and P. Gai, "Analysis, Hardware Specification and Design of a Programmable Performance Monitoring Unit (PPMU) for RISC-V ECUs," 2022 IEEE International Conference on Smart Computing (SMARTCOMP), 2022, pp. 213-218, doi: 10.1109/SMARTCOMP55677.2022.00056.
[3] F. Cosimi, F. Tronci, S. Saponara and P. Gai, "Analysis, Design and Synthesis of an Execution Tracing Unit (ETU) based on AUTOSAR Run-Time Interface (ARTI)", International Conference on Applications in Electronics Pervading Industry, Environment and Society, 2022, Springer, Cham
[4] ISO26262, Part-11
[5] O. Goloubeva et al. "Improved software-based processor control-flow errors detection technique," Annual Reliability and Maintainability Symposium, 2005. Proceedings., 2005, pp. 583-589, doi: 10.1109/RAMS.2005.1408426.
[6] Z. Alkhalifa et al. "Design and evaluation of system-level checks for online control flow error detection," IEEE Transactions on Parallel and Distributed Systems, vol. 10, Issue. 6, pp. 627-641, 1999.
[7] N. Oh et al. "Control-flow checking by software signatures," in IEEE Transactions on Reliability, vol. 51, no. 1, pp. 111-122, March 2002, doi: 10.1109/24.994926.
[8] A.Li et al. Software implemented transient fault detection in space computers. Aerospace Science and Technology 11, 2–3 (2007), 245–252
[9] B.Nicolescu et al.."SIED:Software implemented error detection,"In Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems. 589–596.
[10] J. Vankeirsbilck et al. "Random Additive Control Flow Error Detection," in Computer Safety, Reliability and Security Procedings, Cham: Springer International Publishing, 2018, pp. 220–234.
[11] ISO26262, Part-4
[12] J. Sini et al."A Novel ISO 26262-Compliant Test Bench to Assess the Diagnostic Coverage of Software Hardening Techniques Against Digital Components Random Hardware Failures" MDPI Electronics 11, no. 6: 901.https://doi.org/10.3390/electronics11060901