Enhanced Compiler Technology for Software-based Hardware Fault Detection

DAVIDE BAROFFIO, Politecnico di Milano, Italy
FEDERICO REGHENZANI, Politecnico di Milano, Italy
WILLIAM FORNACIARI, Politecnico di Milano, Italy

Software-Implemented Hardware Fault Tolerance (SIHFT) is a modern approach for tackling random hardware faults of dependable systems employing solely software solutions. This work extends an automatic compiler-based SIHFT hardening tool called ASPIS, enhancing it with novel protection mechanisms and overhead-reduction techniques, also providing an extensive analysis of its compliance with the non-trivial workload of the open-source Real-Time Operating System FreeRTOS. A thorough experimental fault-injection campaign on an STM32 board shows how the system achieves remarkably high tolerance to single-event upsets and a comparison between the SIHFT mechanisms implemented summarises the trade-off between the overhead introduced and the detection capabilities of the various solutions.

CCS Concepts: • **Computer systems organization** → **Reliability**; *Real-time operating systems*; *Embedded systems*; • **Software and its engineering** → **Compilers**.

Additional Key Words and Phrases: Fault Detection, Embedded Systems, Compilers, SIHFT, Real-Time Operating Systems

ACM Reference Format:
Davide Baroffio, Federico Reghenzani, and William Fornaciari. 2024. Enhanced Compiler Technology for Software-based Hardware Fault Detection. *ACM Trans. Des. Autom. Electron. Syst.* 1, 1, Article 1 (January 2024), 23 pages. https://doi.org/10.1145/3660524

1 INTRODUCTION

Recent trends in Integrated Circuitry (IC) manufacturing, such as the decrease in voltages and the shrinkage of the silicon surface, raise new challenges each year, not only regarding the hardware but also for developers of modern critical systems, since these trends facilitate the occurrence of soft errors [5] and hardware failures [7]. Indeed, the family of critical systems includes the set of applications for which the failure of a component – either hardware or software – can have severe consequences on the environment and individuals, as well as business operations and equipment [18]. Achieving high resilience is crucial in critical domains also because critical systems are often exposed to physical phenomena like electromagnetic interference and radiation. Such phenomena affect hardware components and can manifest at the software level as a so-called Single Event Upset (SEU), typically modelled as a transient fault causing a single bit-flip in a memory component of the system.

The article is an extension of the conference paper "Compiler-Injected SIHFT for Embedded Operating Systems" (DOI: 10.1145/3587135.3589944).

Authors' addresses: Davide Baroffio, davide.baroffio@polimi.it, Department of Electronics, Information and Bioengineering, Politecnico di Milano, Milano, Italy, 20133; Federico Reghenzani, federico.reghenzani@polimi.it, Department of Electronics, Information and Bioengineering, Politecnico di Milano, Milano, Italy, 20133; William Fornaciari, william.fornaciari@polimi.it, Department of Electronics, Information and Bioengineering, Politecnico di Milano, Milano, Italy, 20133.

Manuscript submitted to ACM

Traditionally, SEUs are mitigated by adopting hardware-based solutions such as the physical shielding of the IC components or hardware replication mechanisms for error detection and correction. These solutions, however, exacerbate some problems typical of the design of critical systems, especially related to costs and non-functional properties. For instance, adopting Heavy-Ion Tolerant (HIT) hardened memory cells [34] – which is common in the aerospace safety-critical domain – introduces around 30% penalty for energy consumption, whilst also occupying twice the area and weight than equivalent Commercial-Off-The-Shelf (COTS) components [29]. These additional energy, area, and weight requirements may impact the design of the system at large, like in the case of spacecraft design, having problematic consequences on the overall project [24].

Software-Implemented Hardware Fault Tolerance (SIHFT) represents a relatively novel set of solutions that aim at providing, at the software level, resilience against hardware failures. SIHFT has many advantages with respect to hardware solutions since it allows the adoption of COTS components, significantly lowering production and maintenance costs, and satisfying high-performance and low-energy requirements [13]. In this work, we will focus on the SIHFT techniques able to detect the occurrence of a transient fault. Fault recovery strategies and the detection of permanent faults are outside the scope of this paper and left as future work.

1.1 Previous work, structure, and contributions

In our previous conference paper [4], we proposed an architecture- and language-independent approach based on the LLVM compiler framework for the detection of hardware faults by implementing two relevant state-of-the-art SIHFT solutions for data protection and Control-Flow Checking (CFC) at the level of the LLVM Intermediate Representation (IR). The techniques have been combined and tested – both in terms of detection rate and overhead – for the compilation of the entire open-source Real-Time Operating System FreeRTOS running on a real-world COTS embedded device. Remarkably, the resulting system achieved an overall tolerance to SEUs over 99% by using software-only techniques. The tool, named ASPIS[1] (Automatic Software-based Protection and Integrity Suite), has been further improved both in terms of detection rate and overhead by implementing novel and state-of-the-art overhead reduction strategies and CFC techniques. The flexibility of the toolchain increased as well, reducing the user effort required for the compilation with our framework. This article extends the conference paper by describing how we further improved ASPIS by tackling the challenges that arise from the adoption of SIHFT solutions in an RTOS, in terms of both the overhead they introduce and their detection capabilities.

Compared to the conference paper, this paper provides, in addition to a comprehensive description of the approach, the following novel contributions:

- The implementation of a further state-of-the-art algorithm for Control-Flow Checking (CFC), namely Random Additive Signature Monitoring (RASM) [33];
- The improvement of RASM achieving inter-function CFC, also by modifying the underlying open-source Real-Time Operating System FreeRTOS;
- Two overhead reduction techniques;
- The analysis of the compliance of FreeRTOS kernel components with our automated SIHFT injection tool;
- An extended experimental evaluation of the proposed techniques, comparing the overhead and detection capabilities of various combinations of the implemented SIHFT mechanisms.

[1]The source code is available at https://github.com/HEAPLab/ASPIS.

This paper is structured as follows: Section 2 provides some algorithmic background about the SIHFT techniques we implemented, Section 3 provides implementation details of the ASPIS passes and the optimisations we introduced, and Section 4 describes how we tackled the challenges that arose whilst compiling FreeRTOS with ASPIS. The experimental setup and the results are described in Section 5. Section 6 provides a literature review and positions our work with respect to previous research and Section 7 concludes this work, outlining the future possible research paths enabled by this paper.

2 BACKGROUND

Data and control-flow protections are a set of SIHFT techniques for detecting SEUs altering the memory content and the execution flow of a program. This section describes in detail the state-of-the-art techniques that represent the foundations and the starting point of our solution and provides the necessary background also on the functioning of FreeRTOS, the case-study OS we protected with ASPIS. As already mentioned, a detailed literature review of similar approaches is, instead, reported in Section 6.

2.1 Data Protection

SIHFT techniques for data protection usually implement software redundancy in the data or the computation. Since the most relevant, diffused, and flexible approach to software-based redundancy is Error Detection by Duplicated Instructions (EDDI) by Oh et al. [22], we adopted it as the baseline for our data protection mechanism. EDDI is an assembly-level instruction redundancy mechanism that duplicates all the program code, interleaving the original instructions and their copies. EDDI relies on the assumption that the hardware provides separate memory regions and sets of registers for the main instructions and their copies. The algorithm inserts assembly instructions to perform consistency checks before specific instructions that are called *synchronisation points*. Before each synchronization point instruction, the algorithm adds the comparison on its operands, jumping to a user-defined error procedure in case of a mismatch. The most suitable candidate instructions for synchronization points are typically branch- and store-like instructions since they can be used to verify the correctness of conditional branches and maintain the consistency of the memory content.

Oh and McCluskey [23] extended EDDI by allowing the developers to manage the function call instructions in two different ways: 1) functions whose bodies have to be duplicated by standard EDDI, and 2) functions whose bodies are not duplicated but are invoked twice by the caller, eventually comparing their results at the end. We leveraged this approach to extend our data protection mechanism for managing library functions for which the source code is not available and, in general, code that is not possible to transform due to implementation issues. Section 4 will provide some practical examples of this differentiation applied to the FreeRTOS source code.

2.2 Control-Flow Protection

One of the main drawbacks of instruction duplication is the focus limited to data protection, ignoring SEUs that alter the program execution flow. For example, a bit-flip in the program counter could cause the program to unexpectedly jump to a different location of the code (a so-called illegal branch), affecting the correctness of the output. Control-Flow Checking (CFC) is the family of SIHFT techniques that focuses on protecting the Control-Flow Graph (CFG) of the program against illegal branches. Most CFC approaches in the literature rely on one or multiple run-time signatures. These signatures are updated at run-time following the execution path and compared to the ones predicted at compile-time. Usually, they are checked one time per basic block. The following paragraphs report the two signature-based CFC

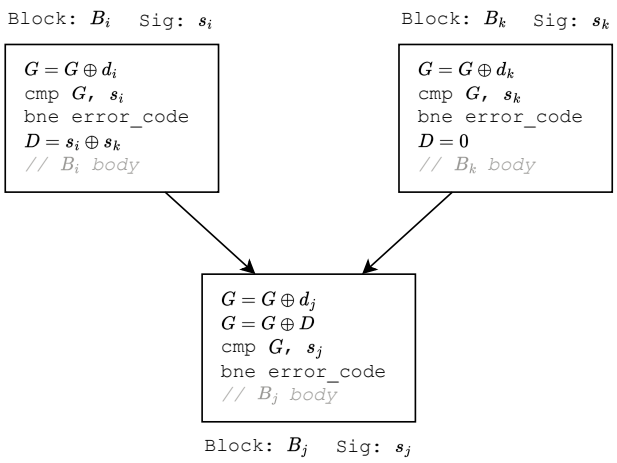

Fig. 1. Example of CFCSS signature updates with a basic block having multiple predecessors.

solutions from the literature: CFC via Software Signatures (CFCSS) [21] and Random Additive Signature Monitoring (RASM) [33].

2.2.1 Control-Flow Checking via Software Signatures (CFCSS). CFCSS is one of the most widely known signature-based CFC techniques. Similarly to the majority of signature-based methods, CFCSS assigns a unique signature to each basic block of the program and performs periodic checks – typically after each branch instruction – between the static compile-time signature and a run-time signature stored in a register G. The algorithm inserts instructions to update the run-time signature as G = G $\oplus$ d_j at the beginning of each basic block B_j. The symbol $\oplus$ represents the bit-wise XOR operation and $d_j = s_i \oplus s_j$, where s_j and s_i are the signatures of B_j and of its predecessor B_i. After the update, G and s_j are compared and, in case of a mismatch, the program jumps to a user-defined procedure for fault recovery.

In case a basic block B_j has multiple predecessors, the algorithm requires an additional signature called *adjusting run-time signature* that is stored in another dedicated register D. The idea is that the predecessors of B_j have to store in D a value that adjusts the run-time signature depending on the signature of the predecessor chosen by the successor for computing d_j. If B_j uses the signature of its predecessor B_k for computing $d_j = s_j \oplus s_k$, CFCSS inserts the instruction D = $s_h \oplus s_k$ into each predecessor B_h of B_j, right after the comparison of G with the static signature. Finally, the adjusting run-time signature D is XOR-ed with the run-time signature G in B_j so that no mismatch occurs: G = G $\oplus$ D. Figure 1 provides a simple example of how CFCSS uses D for blocks having multiple predecessors.

2.2.2 Random Additive Signature Monitoring (RASM). CFCSS suffers from the drawbacks of all the CFC mechanisms that adopt a single signature update per basic block. More specifically, they are vulnerable to illegal branches from the middle of the body of a basic block to the beginning of another and from the end of a basic block to the middle of the body of another. Solutions adopting a double update of the signature, like RASM, solve these issues. The comparison is depicted in Fig. 2. Other than the double signature update, RASM differs from CFCSS because it requires only one run-time signature for standard CFC – which increases to two run-time signatures if we enable inter-function checking – and performs the run-time signature update on conditional branches using the same condition of the conditional branch itself. For nomenclature consistency with CFCSS, we call the run-time signature G, and we call D the additional run-time signature for inter-function CFC.

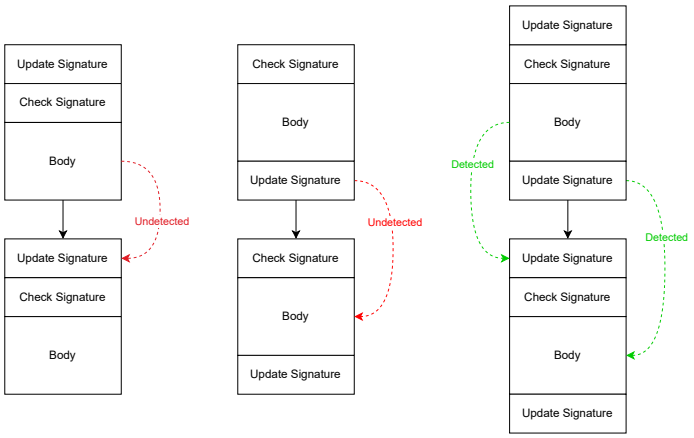

Fig. 2. One signature update approaches (left and centre) fail to detect the two depicted illegal branches, while two signature update approaches (right) are able to detect them.

Similarly to CFCSS, unique signatures s_i are assigned to each basic block B_i in the CFG at compile time. Specifically, RASM requires two random numbers whose sum is the unique signature s_i assigned earlier, called n_i and r_i[2], i.e., $s_i = n_i + r_i$. The static signature s_i and its addenda are used for both determining the validity of a branch at the beginning of a basic block and for computing each update to the run-time signature. According to the original RASM specification, the values of the signatures and their addenda are chosen randomly, with no specific precautions necessary in the randomization process apart from ensuring that there is no shared state between the signatures, i.e. $s_i \neq s_j \wedge s_i \neq n_k$ for all distinct i, j, k. The algorithm inserts the first update instruction $\mathtt{G} = \mathtt{G} - r_i$ at the beginning of each basic block B_i. Then, the signature verification is added to check whether the condition $\mathtt{G} == n_i$ holds. In the negative case, it means that an illegal branch has been detected and the program jumps to a user-defined error routine. Afterwards, the algorithm inserts the second run-time signature update at the end of each basic block. The objective of this update is to ensure that all outward edges of the basic block can be taken using an adjustment value a_i. The update instructions differ depending on whether the terminator of the basic block, i.e. the last instruction of the block, is a return statement or not. If the terminator is a return statement, we allocate a value called R_i and a prime number p_i. We then compute the adjustment value $a_i = (p_i + r_i) - R_i$, we add an instruction updating $\mathtt{G} = \mathtt{G} - a_i$, and a comparison between G and R_i that is used to jump to a user-defined fault handling routine in case of a mismatch. If the last instruction is not a return statement, the algorithm updates the run-time signature depending on the successors: for each successor B_j of B_i, we compute $a_i = n_i - (n_j + r_j)$ and, finally, we update the run-time signature $\mathtt{G} = \mathtt{G} - a_i$. This case trivially extends to function calls by considering the first block of the callee as the successor when updating the run-time signature.

The original algorithm also provides a solution to indirect branches and calls, with the limitation that the candidate successor basic blocks must known at compile-time.

2.3 FreeRTOS Execution Flow

Before diving into the description of the ASPIS toolchain, we provide some background information on FreeRTOS and its functioning to better understand the impact of the ASPIS transformations on the kernel components of the OS.

[2]Variable names differ from the one presented in the original RASM publication [33] for the sake of clarity.

FreeRTOS implements a real-time operating system which is suitable for embedded systems thanks to its small memory footprint and timing predictability. The OS features a small amount of components, providing APIs for task creation, inter-task communication, and, most importantly, task scheduling. All the kernel components that are platform-independent are contained in three source files: `tasks.c`, `queue.c`, and `list.c`, with the first managing task lists and scheduling, the second managing inter-task communication, and the third implementing a list data structure which is also used by the previous two files. Other than that, FreeRTOS requires some platform-dependent libraries, mainly composed of assembly code, and macros that are needed for dynamic memory management, interrupt handling, and context switching, which are designed for the specific processor architecture.

The scheduler is invoked upon system ticks and performs context switches according to the active set of tasks, which is represented by a collection of Task Control Blocks (TCBs). FreeRTOS by default creates an *Idle* task, which is used for garbage collection and to run the scheduler (when in non-preemptive mode), and a *Timer* task for timed events management in case the timers are enabled. All these software components leverage a set of data structures that represent the state of FreeRTOS at execution time. Previous research [20] categorized these components and analyzed their vulnerability to SEUs, defining a baseline for estimating the elements that can and cannot be protected by ASPIS. We analyzed the degree of protection that ASPIS can provide to FreeRTOS by studying the software targets categorized in their work, the results of our analysis are described in Section 5.2.4.

3 AUTOMATED COMPILER-INJECTED SIHFT

ASPIS is a set of LLVM *passes* that implement the SIHFT algorithms described in Sections 2.1 and 2.2 at the LLVM IR level, meaning that the transformations are platform- and language-independent. We designed the passes `FuncRetToRef`, `EDDI` and `DuplicateGlobals` to implement the EDDI data protection mechanism, while `CFCSS` and `RASM` are implementations of the original CFCSS and RASM algorithms in LLVM[3]. Figure 3 depicts the ASPIS compilation flow. As a preliminary step, ASPIS takes a set of source files and transforms them via the LLVM front-end, producing the respective set of LLVM IR files that will be linked together by the `llvm-link` tool in order to ease the transformations across multiple compilation units. After the application of `FuncRetToRef`, `EDDI`, and either CFC pass, ASPIS utilizes `llvm-link` once again to incorporate external sources into the compilation unit. The output of the linking then is transformed by `DuplicateGlobals` for maintaining the consistency of the global variables that are modified by the external functions.

The next paragraphs outline the major features of the passes for each protection mechanism – data protection and CFC – describing the implemented algorithms, the challenges encountered during development and how they have been solved, in addition to the novel concepts introduced by this work.

3.1 Data Protection

The data protection mechanism in place is enforced by three passes: `FuncRetToRef`, `EDDI` and `DuplicateGlobals`. Their subsequent application implements IR-level redundancy by duplicating most of the program IR instructions and performing other transformations to guarantee data integrity, including the duplication of return values and function arguments, the duplication of global variables, and keeping consistent the global variables across the external modules. This data protection mechanism is further detailed in the next paragraphs: Section 3.1.1 describes the behaviour of the passes `FuncRetToRef`, `EDDI`, and `DuplicateGlobals`, Section 3.1.2 explains the motivations behind the need and the

[3]The `EDDI` and `CFCSS` passes were respectively called `DuplicateInstructions` and `CFGVerification` in the original conference paper. They have been renamed to make clear which are the original algorithms.

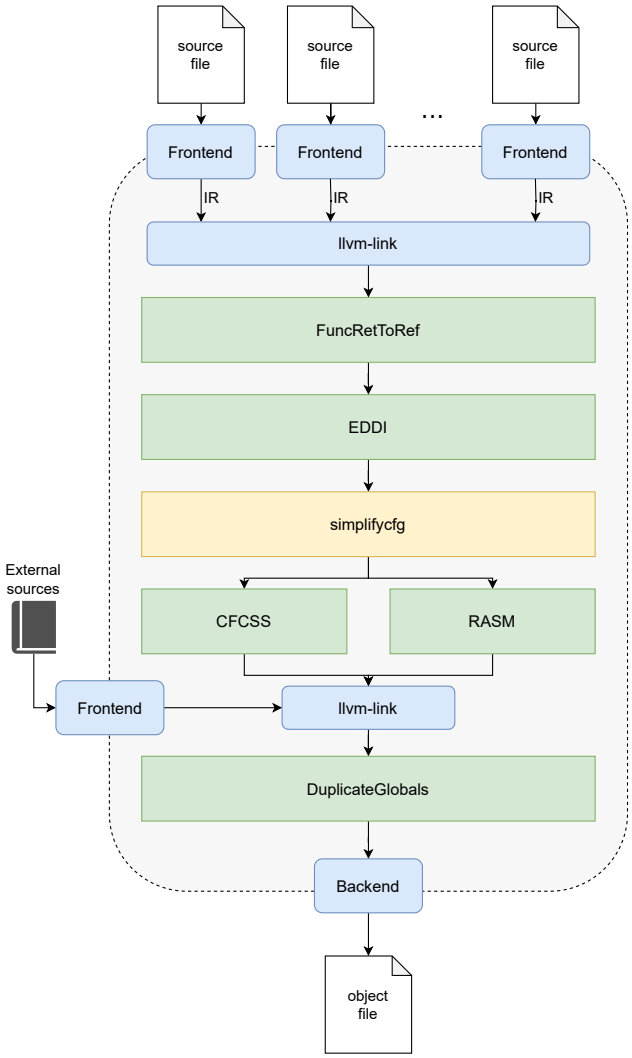

Fig. 3. High-level perspective of the ASPIS compilation sequence as the subsequent application of custom (in green) and pre-defined (in yellow) LLVM passes.

effects of these transformations, and Section 3.1.3 describes the overhead-reduction techniques that we implemented in ASPIS.

3.1.1 Data Protection Passes. `FuncRetToRef` is a preliminary pass that transforms non-void functions into void functions moving the return value to the parameter list. Indeed, this transformation requires a modification of the function body, transforming the original IR `return` instruction into a `return void` instruction, as well as a modification of the body of each caller to correctly pass the return parameter to the new function signature and handle the return pointer after the procedure call.

`EDDI` is the core pass implementing instruction duplication, which is an almost straightforward implementation of the original EDDI algorithm. The procedure can be divided into three major steps: duplicate all the non-constant

global variables, duplicate the arguments in the function signatures, and iterate over the instructions of the program performing duplication and synchronisation points insertion.

Finally, `DuplicateGlobals` finalises the data protection transformation by iterating over all the excluded functions to maintain the integrity of the global variables between internal (duplicated) functions and external (non-duplicated) functions.

3.1.2 Motivations behind these transformations. The implementation of the three passes for data protection enhances the original EDDI algorithm, solving the challenges that arose from the translation of the original approach to standard the LLVM IR abstraction level.

The first issue was related to the impossibility of assigning different sets of registers and memory regions as required by the original EDDI since LLVM IR is architecture-independent and, therefore, has no knowledge of the available hardware registers. The separation of the memory components is required to prevent interference between the data of the two copies of the program.

Proposition 3.1. *The application of* `EDDI` *and* `DuplicateGlobals` *transformations is equivalent to the adoption of different sets of registers and memory regions for each copy of the program used in the original EDDI.*

Since the data of a program is represented by the set of variables in the source code, irrespective of their physical location in memory or registers, ASPIS only cares about duplicating these variables, without the need for two separate regions for storing data. The `EDDI` pass straightforwardly protects variables within the function scope by duplicating the `alloca` LLVM instructions. However, challenges arise when dealing with global variables used by functions external to the code compiled with ASPIS (for instance, libraries). In fact, it may happen at run-time that only one copy of a global variable is updated by some non-duplicated code, which causes a mismatch when the program returns to the duplicated code. These external functions are represented by the "External Sources" block in the ASPIS compilation flow (Figure 3). To solve this problem, we need to split the global variable types into non-complex (e.g. integers, floats) and complex (e.g. pointers, data structures, and arrays) types. On the former, ASPIS applies the `DuplicateGlobals` pass: after integrating the external libraries in the IR (via the `llvm-link` tool as showed in Figure 3), `DuplicateGlobals` duplicates each `store` on non-complex global variables of these external functions so that data consistency is guaranteed. Regarding the variables of the complex type, instead, `EDDI` directly duplicates them assuming that such variables are not modified by functions external to the compilation unit. In practice, we relax this assumption by allowing developers to mark globals to not duplicate thanks to a custom LLVM annotation so that they can be used safely in external functions. This limitation on complex types is caused by the challenges posed by the need to explore all the uses of the members of the complex variables, which is an intricate, long and, in some cases, not even feasible process. ASPIS also provides selective function call duplication, in line with the work of Oh and McCluskey [23], for guaranteeing consistency on variables that are passed as parameters to library functions. Some examples will be provided in Section 4. We conclude that the combination of `EDDI` and `DuplicateGlobals` guarantees data integrity by duplicating all the variables of the program.

Another problem is that the original EDDI algorithm does not explicitly manage `call` and `return` instructions because it just considers them as branch instructions surrounded by a set of `load`/`store` instructions that perform parameter and return value passing. Therefore, the protection of the data flow between procedure calling is guaranteed by the duplication of these instructions done by EDDI. To apply this scheme, LLVM should know architecture-specific

details (e.g., the calling convention) that are, instead, not available at the LLVM IR level[4], which instead uses the high-level `call` and `return` primitives to abstract all the implementation details.

Proposition 3.2. *The combination of* `FuncRetToRef` *and* `EDDI` *provides data flow protection across function calls and returns.*

Our pass `EDDI` performs function argument duplication by design, and the instructions using the duplicated arguments are duplicated as well, guaranteeing a full-fledged parameter duplication upon function invocation. Additionally, running `FuncRetToRef` before the `EDDI` pass appends a new parameter as a return pointer at the end of the list of function arguments so that `EDDI` duplicates it as well. Regarding the original `return` instruction, `FuncRetToRef` substitutes it with a `store` followed by a `return void`, with the former being duplicated by `EDDI`, effectively enabling the return value duplication. Therefore, `FuncRetToRef` and `EDDI` provide inter-function data flow protection since they allow the duplication of function arguments and return values.

The original EDDI specification does not discuss how to manage complex data types at synchronisation points. For example, let us consider a data variable and a pointer to this variable. The original EDDI algorithm duplicates both of them obtaining two copies of the data variable and two copies of the pointer allocated in different memory regions, each pointer pointing to one of the variables. Hence, with the original EDDI algorithm, any comparison of pointers to these variables always leads to a mismatch. The problem also affects arrays and data structures.

Proposition 3.3. *ASPIS's pass* `EDDI` *improves the original EDDI algorithm by offering the protection of pointers, simple arrays, and data structures.*

When feasible, `EDDI` determines the original content of the pointer, possibly following the chain of `load-store` instructions in the case of a multi-level pointer. When the pointer content is found, the pointed value and its copy are compared. If one of the two pointers suffered an SEU, the pointed data will likely mismatch, hence the mechanism achieves pointer protection. Finally, ASPIS does not compare data structures by design on each synchronisation point that uses the data structure itself. Instead, it compares the members of a data structure when they are eventually retrieved and used as operands of a synchronisation point, effectively protecting them. Regarding arrays, ASPIS compares all their items at synchronization points. Nonetheless, since comparing all the elements may introduce unacceptable overhead, we reduced the number of checks by avoiding the comparison on nested arrays or arrays of complex data types. In fact, similarly to data structures, we observed that the data contained in the array is likely to be used in a synchronisation point, hence we perform the comparison on the single array element uses instead of the whole array.

3.1.3 Overhead Reduction. One of the main drawbacks of SIHFT solutions is the considerable timing overhead they introduce. In the specific case of instruction duplication, we expect an increase in the execution time of a factor of two due to the need to execute twice the amount of code of the original program. On top of that, there is the non-negligible overhead introduced by the consistency checks, which are performed upon `store`, `branch`, and `call` instructions. In principle, reducing the number of consistency checks should not affect the execution correctness, but only postpone the detection to the next synchronisation point. Based on this observation, we provide some tweaking parameters in ASPIS that can be enabled or disabled to reduce the number of consistency checks injected in the code. One of the parameters enables or disables Full Duplication with Selective Checking (FDSC), which is a technique that inserts consistency checks in critical blocks identified by using a heuristic from the state of the art based on the number of their fan-outs [2].

[4]This issue looks like a limitation but it is actually a design feature of LLVM to remain architecture-independent at IR-level.

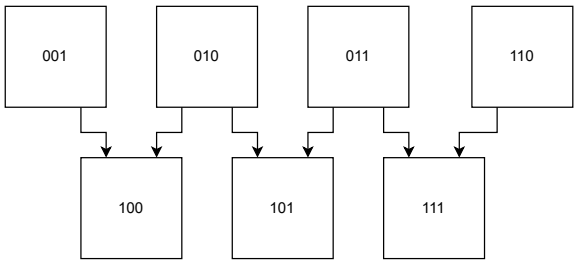

Fig. 4. Corner case not considered by the original CFCSS algorithm. The boxes represent the basic blocks and the number is the binary representation of the compile-time signature of each basic block.

We also provide a set of parameters for selecting what instructions the pass considers as synchronisation points between `store`, `call`, and `branch` instruction families. These simple tweaks have been used for the implementation of a novel selective checking technique, we call it selective-EDDI (sEDDI), which is a revised version of EDDI that does not consider `store` instructions as synchronisation points.

3.2 Control-Flow Graph Monitoring

Concerning CFC, ASPIS includes two alternative solutions: `CFCSS` and `RASM`. They are applied after the `EDDI` pass to prevent it from creating duplicate instructions also on the code that performs the integrity checks on the CFG signatures. In fact, protecting the code for CFC is not required under the assumption that the faults manifest as SEUs.

3.2.1 The `CFCSS` *Pass.* The `CFCSS` pass is an almost straightforward implementation of the original CFCSS algorithm. However, due to the abstraction level of the LLVM IR, it is not possible to allocate registers for the pair of run-time signatures. Instead of adopting a set of global variables, we opted for a per-function CFC, similar to what we did for memory allocation in `EDDI`, meaning that the `CFCSS` pass performs control-flow graph monitoring only between the basic blocks within the same function. Adopting a pair of global run-time signatures would cause problems due to the unpredictability of some program features such as indirect function calls and interrupts. Therefore, we employed a pair of local variables for each function, such that the callees do not alter the contents of the run-time signatures of the callers.

Some corner cases are not considered by the original CFCSS specification when determining the predecessors of a successor – which can be seen as a neighbour – upon computing the adjusting signature `D`, leading to ambiguities and undefined behaviours. As a simple example, consider the CFG in Figure 4. The original CFCSS algorithm defined d_i as depending on the signature of one of its predecessors, for instance, we can set $d_{100} = s_{100} \oplus s_{001}$, meaning that B_{100} has chosen B_{001} as the predecessor for computing d_i. Therefore, both its predecessors B_{001} and B_{010} should use s_{001} for computing `D`. But B_{010} has another successor B_{101} with multiple predecessors, hence setting $\texttt{D} = s_{001} \oplus s_{010}$ would require that B_{101} uses the signature of its predecessor B_{001} for computing d_{001}. B_{100} and B_{101} have B_{010} as a common predecessor, so they can use s_{010} as the signature of the predecessor for computing $d_{100} = s_{100} \oplus s_{010}$ and $d_{100} = s_{101} \oplus s_{010}$. However, the reasoning does not extend to the block B_{111}, since it shares no predecessor with the other two blocks. The problem is that, according to the original algorithm, it is unclear which predecessor signature should be used by the blocks B_{100}, B_{101}, and B_{111} for computing d_i. Although scenarios like this – in which blocks share no predecessors – are not uncommon in real-world programs, the CFCSS specification does not tackle the problem. Therefore, we extend the CFCSS original approach by defining the concept of neighbourhood as follows:

Definition 3.4 (Neighbourhood). A basic block B_i is in the *neighbourhood* of a basic block B_j – and in such a case we write $B_i \in neigh(B_j)$ – if either of these conditions hold:

- $succ(B_i) \cap succ(B_j) \neq \emptyset$, i.e., B_i and B_j have at least a common successor, or
- $B_i \in neigh(B_k)$ and $B_j \in neigh(B_k)$, i.e., B_i and B_j are both in the neighbourhood of the same basic block B_k.

With our extension, all basic blocks of the same neighbourhood use the same basic block B_i as the neighbour block. We apply the same reasoning to all the successors of the blocks in the neighbourhood, i.e., all the basic blocks B_j having a predecessor in a neighbourhood deterministically use the signature of the same basic block B_i for computing $d_j = s_j \oplus s_i$, effectively tackling the issue described earlier.

3.2.2 The `RASM` *Pass.* Contrarily to `CFCSS`, `RASM` has multiple advantages such as the double signature update, conditional signature update, indirect branch protection, and return block protection. This last feature has been leveraged for the implementation of the inter-function CFC version of RASM (inter-RASM or iRASM for short), which will be described in Section 3.2.3. Focusing on the standard RASM, in practice, its implementation as a pass is very similar to the one of CFCSS. Similarly to `CFCSS`, the pass inserts the two local variables for storing the run-time signatures. In principle, only one local variable should suffice, i.e. the one storing the current run-time signature `G`, but the extra variable is necessary later for the inter-function CFC version of `RASM` described in Section 3.2.3.

Additionally, the main difference in our approach with respect to the original RASM algorithm concerns the assignment of the static signatures s_i. Specifically, the signature s_i is divided into two parts, as required by the specification: n_i and r_i. However, the original algorithm does not provide precise indications on how to pick the two random numbers, so we opted for assigning sequential numbers divisors of 2 to n_i and 1 to r_i, such that the following relation holds:

$$s_i = n_i + r_i = n_i + 1$$

and $n_i = 2 + n_{i-1}$, $\forall i$. This is one of the simplest ways of enforcing no shared state between the signatures, i.e., $s_i \neq s_j \wedge s_i \neq n_k$ for all distinct i, j, k. We demonstrate the equivalence – in terms of safety – with the original RASM by observing that a control-flow error can only be caused by an SEU targeting either a memory component able to produce an alteration in the control-flow graph (such as the program counter) or the memory region containing the signature. In the first case, the SEU would cause a jump to an instruction in another part of the code that is assumed to be valid (i.e. part of the ISA). But since no two blocks share the possible signature states at the beginning and in the body of the basic blocks, respectively s_i and n_i, the signature would mismatch as soon as a check is encountered. Considering, instead, the case of an SEU on the signature leading to any value different from the valid signature expected by the successor block, the next update will indeed invalidate the signature, leading to a mismatch. Our approach guarantees SEUs resiliency since only one signature is admissible at the beginning of a basic block and only one signature is admissible during the execution of a basic block. We suspect that the original randomicity requirement of the RASM specification was needed to minimise the probability that two basic blocks in different compilation units share the same signature. However, we tackle this issue by linking with `llvm-link` all the compilation units we plan on protecting, as shown in Fig. 3. In this way, we ensure that a unique signature is assigned statically to each basic block of the protected code.

After the signature assignment with the criteria just mentioned, the `RASM` pass hardens the code implementing the same algorithm described in the original RASM specifications. We also apply the `lowerswitch` LLVM pass (not depicted in the compilation flow figure for simplicity), which, as the name suggests, "lowers" `switch` instructions into

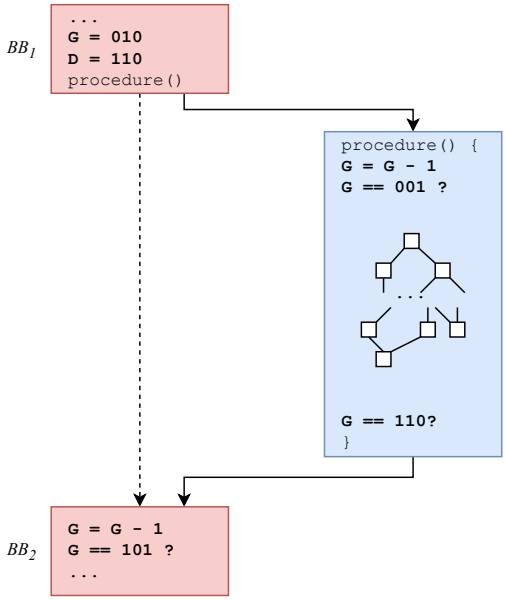

Fig. 5. Example of CFG with `inter-RASM` enabled.

a sequence of consecutive `branch` instructions before the application of our compilation pipeline for simplifying the insertion of the conditional signature updates required by RASM.

3.2.3 Achieving Inter-Function CFC. The two approaches described thus far do not protect against control-flow errors that jump at the very first instruction of a function. In fact, they use local variables as per-procedure run-time signatures which are reset at the beginning of each function call, hence an illegal branch landing at the first instruction of a function will reset the signature, which makes the fault undetectable. In order to protect against such errors, the code needs to trace at run-time the sequence of functions called in order to set, upon function call/return, the signature for the basic block following the `call`/`return` instruction. Indeed, a unique pair of signatures is required globally, which makes it straightforward to implement them as two global variables `G` and `D`. In ASPIS, we implemented a novel inter-function CFC version of the `RASM` pass, which we call `Inter-RASM`, or `iRASM` for short. `Inter-RASM` is based on the intuition that the caller sets the signature of the first basic block of the callee as the run-time signature before performing the call and signals the callee that the signature it expects at the end of the procedure is the one of the basic block following to the call. The implementation leverages the R_i adjusting signature, which is now stored dynamically in `D`. Since the run-time signature `G` is checked against `D` upon return (in compliance with the original RASM specification), the basic block reached after the return has a run-time signature `G` that matches the one it expects. This raises the problem of multiple nested function calls, which are obviously extremely common in programming. In order to prevent the callee from overwriting `D` when it, in turn, calls another function, it backs up `D` before the call and then resets it after the nested call has terminated. This slight modification to the original algorithm is based on the features provided by RASM itself. Its implementation, however, requires the addition of some checking instructions at the beginning of each function of `G` against a fixed well-known value, which we call the `INIT_SIGNATURE`, to check whether the function is the entry point of the program or not (e.g. the `main()` of a C/C++ application).

Figure 5 provides an example of CFG compiled with `iRASM`. Blocks of the same colour are part of the same function. The CFG of `procedure()` is only outlined for the sake of simplicity, and the instructions performing the checks on `INIT_SIGNATURE` at the beginning of each function are omitted as well. The dashed line represents the predecessor-successor relationship modelled by LLVM, yet the actual block executed after BB_1 is the entry block of `procedure`. Hence, before calling `procedure()`, BB_1 sets `G` and `D` according to the values expected at the beginning of `procedure()` and of BB_2, respectively.

4 ENABLING ASPIS IN FREERTOS

Most of the SIHFT literature we reviewed does not provide insight into the applicability of their techniques to specific workloads. The only exception is the paper by James and Goeders [16], which evaluated the proposed COAST framework on FreeRTOS. Likewise, we evaluate the applicability of our framework to FreeRTOS by providing an estimation of the amount of programming effort required by the user to make its code compliant with ASPIS. In fact, although ASPIS can be used to protect the entire FreeRTOS kernel codebase without any major programming effort, the interaction with external components and drivers (i.e., port-dependant components) must be carefully adapted in the context of code duplication and control-flow checking. In particular, most of the concerns derive from the use of external functions and their use of FreeRTOS global variables, together with the problem of context switches and interrupt routines in the case of `inter-RASM`. The following will describe the only modifications required to the FreeRTOS kernel to be compliant with ASPIS.

4.1 Handling FreeRTOS Global Data Structures

FreeRTOS employs several global variables in order to ease the interaction between the different kernel components. The management of global variables of non-complex data types is performed directly within ASPIS thanks to `EDDI` and `DuplicateGlobals` passes. However, the assumption of ASPIS by which complex variables are not modified in external functions does not hold, in general, in the case of a FreeRTOS port. For instance, the port-dependant assembly function `xPortPendSVHandler` saves the top of the stack into `pxCurrentTCB`, which is a global variable that, as the name suggests, contains the TCB of the currently running task. Since this function is written in assembly, we had to manually adjust the code in order to mirror changes also on the duplicate `pxCurrentTCB_dup`, which will be created at compilation time.

Other than the aforementioned modification, we needed to exclude all the global variables that serve as stacks of statically allocated tasks using a source-level annotation. These are the only variables of the FreeRTOS kernel that are excluded from the duplication. Indeed, since the data is already duplicated within each stack, there is no need to duplicate the stacks as well.

4.2 Architecture-Dependent Functions Management

Since all the functions belonging to the `Drivers` and `Portable` modules are port-dependent, including the already mentioned `xPortPendSVHandler`, they have been excluded from the SIHFT compilation. Other than exploiting in-line assembly, these functions deal with other architecture-dependent features, for instance, they perform read/write operations in specific memory regions that are reserved for GPIO. Memory management functions, such as `malloc` and `free`, also belong to the set of excluded functions of the `Portable` FreeRTOS module. Library functions like `memcpy` and `memset` are excluded as well. However, we give developers the option to set some functions as “to duplicate” via Clang annotations. ASPIS, in such a case, duplicates the calls to the annotated function, providing selective function

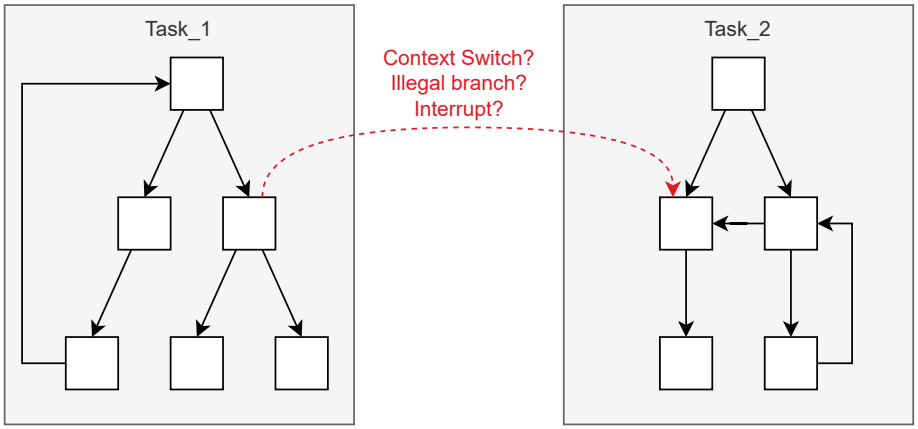

Fig. 6. Outlined CFG of two tasks, demonstrating the indistinguishability of an illegal branch w.r.t. context switches and interrupts.

call duplication. Therefore, we created a set of wrappers for duplicating calls to external functions as well as a set of wrappers for performing calls to original non-duplicated functions from external sources. In particular:

- `pvPortMalloc` has been duplicated in case of dynamic TCB and timer creation.
- `pvPortFree` has been duplicated in case of TCB and timer deletion, matching what was done for `pvPortMalloc`.
- `memcpy` has been duplicated in `prvCopyDataToQueue()` and `prvCopyDatafromQueue()`, two primitives of the `queue.c` kernel component for saving and reading data from a queue, respectively. In fact, only one of the two queue replicas would be updated when calling the first function, whereas the read would be performed only from one replica when calling the second function.

Not performing duplication would lead to some serious correctness problems such as overflows in task stacks or ghost updates on variables (i.e. the code would update only one copy of the variable), which in turn may lead to an incorrect error detection.

One notable example linked to the problem of global variables duplication is the global `xTimerQueue`, which requires the duplication of all the three functions described above in order to work correctly. In particular, no `memcpy` duplication would prevent the update of `xTimerQueue` clone. Moreover, also assuming `memcpy` is duplicated, not duplicating `pvPortMalloc` would prevent the creation of the double timers, causing a command sent on `xTimerQueue` to be executed twice on the same timer, while not duplicating `pvPortFree` would lead to an overflow in the case `pvPortMalloc` duplication is enabled.

4.3 The `inter-RASM` algorithm with Context Switch and Interrupts

Interrupts and context switches are surely the most concerning aspects of the compilation of an entire operating system with CFC enabled. The main issue is that interrupts and context switches are indistinguishable from illegal branches due to their unpredictability in altering the CFG of the program. Figure 6 provides a clear example of the indistinguishability between an illegal branch, a context switch, and an interrupt: the execution can be moved to another task because of a context switch, or to another routine for interrupt handling, but it can also happen that the control flow has been altered due to a CFE. Therefore, applying `inter-RASM` on the unmodified FreeRTOS kernel will lead to a mismatch in all the above cases.

Our solution employs two global variables as the run-time signatures and assigns a pair of run-time signatures to each task, backing the signatures up and restoring them at each context switch. This required the addition of two integer fields to the FreeRTOS TCB data structure, acting as per-task run-time and adjusting run-time signatures. The context switch procedure was modified as well to perform the switching between the signature pairs. Finally, we also added some code intercepting interrupt routines for backing up and restoring the signatures before and after the routine body, respectively.

5 EXPERIMENTAL EVALUATION

In order to evaluate the effectiveness of ASPIS, we ran an experimental campaign aimed at assessing its SEU detection capabilities and the introduced overhead in terms of timing and size.

5.1 Experimental Setup

We hardened the OS employing multiple combinations of protection passes by employing `EDDI` and its `FDSC` and `sEDDI` optimisations for instruction duplication in conjunction with the three passes for CFC: `CFCSS`, `RASM`, and `inter-RASM`. We combined them to test a total of 10 configurations, including the configuration without any SIHFT mechanism, which serves as our baseline performance. The resulting code ran on a NUCLEO-L152RE board equipped with the STM32L151RET6 microcontroller and was used to measure the ability of ASPIS to improve SEU detection and the introduced overhead. In our experimental setup, the NUCLEO board was connected to a host machine governing the experiments via gdb and the ST-Link gdb server.

5.2 Fault detection

The objective of this part of the experimental campaign was to investigate the detection capabilities of ASPIS on the OS components. We used ASPIS to compile a version of FreeRTOS running specifically developed microbenchmark tasks designed to test the majority of FreeRTOS core features: Tasks, Queues, Message Buffers, and Timers, in parallel with two tasks executing the MatMult and CRC benchmarks from the Mälardalen [14] and MiBench [15] benchmark suites, respectively. The experiment is orchestrated by the host machine that performs the fault injection at random time instants, targeting the RAM address space and the processor registers with a uniform distribution.

5.2.1 Interpretation of the possible outcomes. The host machine observes the execution after the injection to discover data corruptions or timing faults that were not detected by the protection mechanisms in place. Specifically, data corruptions are discovered by checking the output of the program against a pre-computed golden run, while the timing integrity is checked using a so-called watchdog timer. A watchdog requires the analysed task to send periodic heartbeats in order to detect whether timing errors occurred.

The orchestrator determines the output of each injection depending on the following possible outcomes:

- *No Effect.* The fault did not cause any modification to the execution, meaning that the output matches the one of a pre-computed golden run.
- *Loop.* This is a "stuck-at" fault, i.e., the program does not continue the expected execution path but gets stuck in an infinite loop.
- *Silent Data Corruption (SDC).* The execution correctly finishes but the actual output of the program does not match the one of the golden run.

- *HW-Detected.* The fault injected caused an error that is captured by the hardware (e.g. illegal address/opcode). The hardware triggers the execution of an OS-level routine to manage the error.
- *EDDI-Detected.* The ASPIS data protection mechanism in place detected a mismatch between the two copies of the data.
- *CFC-Detected.* The ASPIS CFC mechanism in place detected an illegal branch.

Watchdogs are very common in the context of real-time systems, therefore we can consider loop faults to be easily detectable. However, waiting for a watchdog expiration requires idle cycles in which the program could already be performing a recovery, worsening the real-time capabilities of the system. Therefore, even if we consider loop faults to be detectable, they are still relevant when evaluating the detection capabilities. HW-detected faults, on the other hand, are immediately detected since the processor would raise an exception due to an illegal instruction or an illegal memory access. In this case, even if we did not consider recovery techniques in our discussion, we can assume that recovery from a hard fault is not trivial. Therefore we consider mechanisms lowering the number of HW-detectable faults as more effective than others. Finally, the most important outcome, that we actually used as a reliability metric, concerns the amount of SDCs that the system suffers. The relevance of this category of errors comes from the fact that in practice they are indistinguishable from correct program executions, meaning that ASPIS was not able to detect in any way that an error occurred and an incorrect output was produced.

5.2.2 Injection campaign. We injected 60000 bit-flips between registers and memory assuming a uniform distribution on each memory word by considering the size of the memory and the registers space: $80KB$ of RAM ($81920B$) against 16 32-bit registers ($64B$). Therefore our injector had the probability of injecting a fault within the register space of $P_r = \frac{64B}{80KB+64B} = \frac{64B}{81920B+64B} = 1/1281 \approx 0.0008$, and the probability of injecting a fault within the memory space of $P_m = 1 - P_r = 1280/1281 \approx 0.9992$. Given our statistical fault injection model, we use the following widely used formula for computing the margin of error rate e [19]:

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

where p is the a priori estimate of the percentage of faults causing a failure, which we conservatively set to $p = 0.5$ to maximize the sample size as suggested by the original article [19]. Then, $t = 2.5758$ is the cut-off point corresponding to a 99% confidence level computed with respect to the normal distribution, N is the sample space size[5], and $n = 60\,000$ is our sample size. Under the conservative assumption that N is infinite, we estimate the error rate of our injection campaign by computing the limit: $\lim_{N\to\infty} e = 0.00526 = 0.526\%$.

5.2.3 Results. The fault injection results for each configuration of ASPIS are provided in Table 1. The table reports the percentage for each outcome with an error $e = 0.526\% < 1\%$ and a confidence of 99% together with the percentages by considering only effective faults, which represents the effectiveness of ASPIS provided that the SEU is not masked by the program. Expectedly, `EDDI` is one of the best data protection mechanisms, achieving the highest results in terms of detection rate. However, when considering the number of SDCs, the three mechanisms for data protection provide comparable protection. This is probably due to the overhead introduced by the checks of `EDDI`, which increase the execution time and the liveliness period of the variables.

Concerning CFC, although `CFCSS` manifests the highest detection rate, the most efficient mechanism in terms of SDCs is `inter-RASM`, while `CFCSS` and `RASM` provide slightly worse but still comparable protection. With the former

[5] N = (memory size × time instants)

Configuration	No Effect	Loop		SDC		HW-Detected		EDDI-Detected		CFC-Detected	
	$\%_{tot}$	$\%_{tot}$	$\%_{effective}$	$\%_{tot}$	$\%_{effective}$	$\%_{tot}$	$\%_{effective}$	$\%_{tot}$	$\%_{effective}$	$\%_{tot}$	$\%_{effective}$
No SIHFT	97.133	0.925	32.267	0.625	21.802	1.317	45.930	-	-	-	-
EDDI + CFCSS	92.265	0.532	6.874	0.017	0.215	1.178	15.234	5.453	70.502	0.555	7.175
EDDI + RASM	92.225	0.462	5.938	0.022	0.279	1.232	15.841	5.703	73.355	0.357	4.587
EDDI + inter-RASM	91.173	0.447	5.060	0.015	0.170	1.797	20.355	6.173	69.940	0.395	4.475
FDSC + CFCSS	92.865	0.513	7.195	0.022	0.304	1.298	18.197	4.768	66.830	0.533	7.475
FDSC + RASM	92.737	0.498	6.861	0.012	0.161	1.242	17.095	5.143	70.812	0.368	5.071
FDSC + inter-RASM	91.518	0.507	5.974	0.012	0.138	1.828	21.556	5.742	67.695	0.393	4.637
sEDDI + CFCSS	92.925	0.512	7.232	0.023	0.330	1.362	19.246	4.675	66.078	0.503	7.114
sEDDI + RASM	92.615	0.502	6.793	0.015	0.203	1.335	18.077	5.170	70.007	0.363	4.920
sEDDI + inter-RASM	91.567	0.495	5.870	0.013	0.158	1.855	21.996	5.603	66.443	0.467	5.534

Table 1. Fault injection results showing the percentages of outcomes with respect to the total amount of injections ($\%_{tot}$) and only the effective faults ($\%_{effective}$).

achieving a lower SDC rate with `EDDI` and the latter achieving a lower SDC with `sEDDI` and `FDSC`. Regarding loop faults, the three mechanisms for CFC provide almost the same degree of protection, but, once again, `inter-RASM` manifests the lowest average amount of loop faults when considering only effective faults.

5.2.4 Protection of FreeRTOS. Mamone et al. [20] studied the resiliency of the FreeRTOS against SEUs by performing statistical fault injection on the kernel's most important data structures grouping them depending on their usage. By keeping track of the injection targets during our fault injection campaign, we performed a similar analysis by gathering the variables that suffered from SDCs, filtering them between application-specific and FreeRTOS-specific locations, and grouping them extending the classification nomenclature we found in the literature [20] as follows:

- GKVARS: The set of global kernel variables of FreeRTOS.
- TCBVARS: The set of TCB structures in the set of FreeRTOS variables.
- DLDLST: The lists containing information about the delayed tasks.
- RDYLST: The lists containing information about the ready tasks.
- MTXQVARS: The set of data structures used by FreeRTOS for inter-task communication, including queues and message buffers.
- TMRVARS: The set of data structures related to FreeRTOS timer handling.
- APPVARS: Application-specific locations and other variables that do not belong to the other categories.

The SDCs and loop faults for each of the categories are represented in Table 2 and Table 3, respectively showing the total percentage of injections that caused SDCs and loop faults. The data highlights how all protection mechanisms in place were able to significantly reduce the number of SDCs, achieving zero faults in all targets except the data structures for inter-task communication, while loop faults are completely zeroed out, meaning that ASPIS is able to greatly increase the real-time capabilities of the system.

5.3 Timing and Size Overhead

In this case, we implemented two tasks running the DES encryption and Matrix Multiplication (MM) benchmarks from the Mälardalen benchmark suite [14], one task running the Lift benchmark from the TACLeBench benchmark suite [9], and two tasks running CRC and SHA from the MiBench suite [15]. The overhead experiments have two goals: measure the execution time overhead and the binary size overhead. In particular:

Target	No SIHFT	EDDI + CFCSS	EDDI + RASM	EDDI + iRASM	FDSC + CFCSS	FDSC + RASM	FDSC + iRASM	sEDDI + CFCSS	sEDDI + RASM	sEDDI + iRASM
GKVARS	3.703%	0%	0%	0%	0%	0%	0%	0%	0%	0%
TCBVARS	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
DLDLST	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
RDYLST	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
MTXQVARS	1.680%	0.068%	0.069%	0.137%	0.134%	0.069%	0%	0.069%	0.068%	0%
TMRVARS	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
APPVARS	0.659%	0.018%	0.024%	0.014%	0.022%	0.012%	0.014%	0.026%	0.016%	0.016%

Table 2. SDC rate per FreeRTOS target.

Target	No SIHFT	EDDI + CFCSS	EDDI + RASM	EDDI + iRASM	FDSC + CFCSS	FDSC + RASM	FDSC + iRASM	sEDDI + CFCSS	sEDDI + RASM	sEDDI + iRASM
GKVARS	18.519%	0%	0%	0%	0%	0%	0%	0%	0%	0%
TCBVARS	5.348%	0%	0%	0%	0%	0%	0%	0%	0%	0%
DLDLST	7.778%	0%	0%	0%	0%	0%	0%	0%	0%	0%
RDYLST	0.042%	0%	0%	0%	0%	0%	0%	0%	0%	0%
MTXQVARS	1.200%	0%	0%	0%	0%	0%	0%	0%	0%	0%
TMRVARS	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
APPVARS	0.753%	0.637%	0.554%	0.537%	0.617%	0.598%	0.609%	0.615%	0.602%	0.593%

Table 3. Loop rate for FreeRTOS target.

- To test the timing overhead, we compiled multiple versions of FreeRTOS, each one running with one of the benchmarks, and we measured the time required for the benchmark execution. Results were collected via the debugging interface of the board by placing a break-point on a dedicated function from which it is possible to extract the correct execution time of the task.
- Regarding the binary size overhead, we simply observed the size of the binary files produced at the end of each compilation. This value provides us insights into the size of the program code, rather than the number of actually executed instructions.

Configuration	DES	MM	Lift	CRC	SHA	*Avg.*
EDDI + CFCSS	5.472x	4.276x	4.724x	3.889x	5.083x	4.689x
EDDI + RASM	6.291x	5.053x	5.801x	4.638x	6.255x	5.6081x
EDDI + inter-RASM	10.749x	7.041x	8.052x	6.572x	8.626x	8.208x
FDSC + CFCSS	2.672x	3.086x	2.985x	2.676x	2.718x	2.827x
FDSC + RASM	3.090x	3.488x	3.385x	3.063x	2.969x	3.199x
FDSC + inter-RASM	5.392x	4.889x	4.812x	3.959x	4.175x	4.645x
sEDDI + CFCSS	3.387x	3.027x	2.993x	2.674x	2.728x	2.962x
sEDDI + RASM	3.700x	3.637x	3.486x	3.059x	3.110x	3.399x
sEDDI + inter-RASM	6.190x	4.877x	4.631	4.184x	4.191x	4.814x

Table 4. Increase in benchmark execution time with respect to No SIHFT configuration.

The timing and size overheads are summarised in Table 4 and in Table 5, respectively. The data highlights that `CFCSS` is the most lightweight CFC mechanism, performing slightly better than `RASM`. `Inter-RASM`, on the other hand,

Configuration	*Size (B)*	*Increase*
No SIHFT	29 160	1.00x
EDDI + CFCSS	130 176	4.46x
EDDI + RASM	151 968	5.21x
EDDI + inter-RASM	280 352	9.61x
FDSC + CFCSS	113 152	3.88x
FDSC + RASM	131 232	4.50x
FDSC + inter-RASM	248 544	8.52x
sEDDI + CFCSS	113 152	3.88x
sEDDI + RASM	130 992	4.49x
sEDDI + inter-RASM	248 272	8.51x

Table 5. Size of the program code (`.text` sections) of FreeRTOS having microbenchmarks as tasks.

introduces a massive overhead, with up to 9x penalty on the execution time. Concerning instruction duplication, the overheads are comparable for `FDSC` and sEDDI, while, unsurprisingly, `EDDI` introduces the greatest overhead due to its checks at every `store`, `branch`, and `call` instruction.

5.4 Overhead-Detection Trade-off

The experimental evaluation we conducted highlighted that there is no "best" solution for achieving resilience at a low cost. This is one of the major concerns of SIHFT solutions as it represents an engineering challenge for developers of critical systems who have to select the solution that fits their needs depending on the specific domain of application.

Figure 7 illustrates the overhead-detection trade-off of ASPIS. The X-axis represents the increase factor in terms of both size and average timing overhead, whereas the Y-axis represents on a logarithmic scale the percentage of SDCs suffered by the resulting system on 60,000 injections. Each combination of data protection and CFC solutions is represented by a point in the two plots, as described in the legend on the right-hand side of the figure. Clearly, the most efficient solutions are the ones closer to the bottom-left corner of the two plots. It is reasonable to assume that critical systems engineers would prioritise execution time over binary size, therefore we focus on the timing overhead plot (on the right). We can observe that most solutions achieve a remarkably low SDC rate. `FDSC` obtains the lowest rates with the smallest overhead, yet both are comparable to the ones of `sEDDI`, while `EDDI` provides a low SDC rate with a much higher timing penalty.

6 RELATION WITH PREVIOUS WORKS

There are multiple SIHFT techniques in the literature that enforce consistency of the execution state at diverse granularity levels. For example, application-level redundancy consists of running multiple replicas of an application and comparing their output. The orchestration of the interleaving between the two replicas and the consistency checks is typically left to a hypervisor. Indeed two replicas enable only fault detection, while three instances provide Triple Modular Redundancy (TMR), which leverages a voting mechanism for recovery purposes. Other than TMR, fault recovery can also be performed by SIHFT mechanisms such as recovery blocks, task re-execution, and Error-Correcting Codes [25, 13]. Other than application-level duplication, we already mentioned instruction-level duplication (EDDI), and its extension to the procedure level [23].

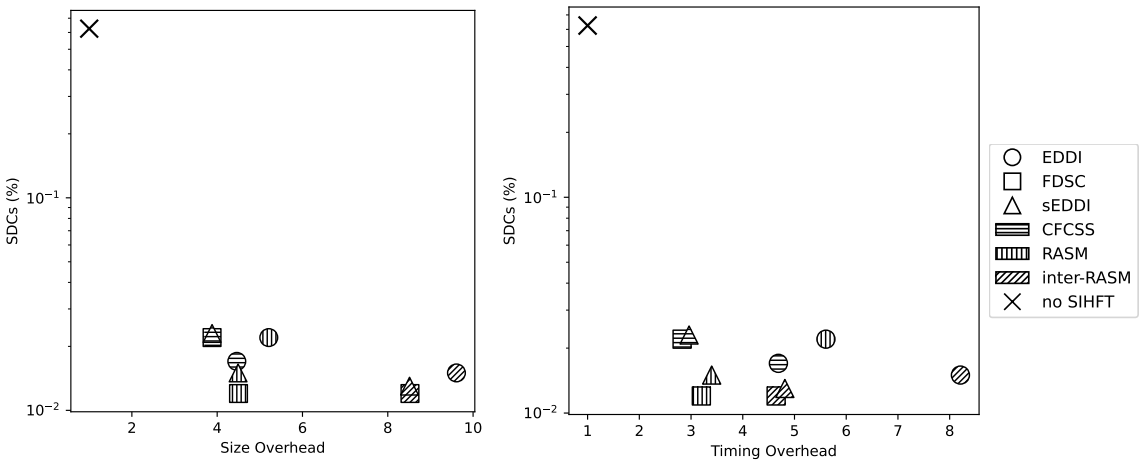

Fig. 7. Size and timing overhead w.r.t. the percentage of SDC for each SIHFT mechanism.

Other works focused on reducing the overhead of SW-based redundancy by reducing the duplicated code while keeping the protection level reasonably high, exploiting the fact that a significant amount of SEUs causes architecture-level faults, such as segmentation faults, that are inherently masked by the processor [28]. This family of techniques is known as "selective duplication". Examples of selective duplication are: Shoestring by Feng et al. [10] and other selective duplication techniques exploiting code analysis mechanisms to determine the vulnerable instructions like profiling [17], genetic algorithms [3] and critical basic block identification [2, 31]. Remarkably, critical block identification has been used for our `FDSC` optimization, which implements the heuristics of Critical-Block Duplication (CBD) [2] for checkpoint insertion.

Regarding CFC techniques, other than the already mentioned CFCSS and RASM algorithms that are implemented in ASPIS, some notable mentions are Control-flow Error Detection using Assertions (CEDA) [35] and Path Sensitive Signatures (PaSS) [36]. None of these mechanisms implements intra-block protection, preventing instruction skip faults from being detected. This kind of fault can be mitigated by different approaches known in the literature as Instruction Monitoring techniques, like RACFED [32], which, however, introduces a much higher overhead. Even if some researchers explained concerns related to the effectiveness of CFC techniques [30, 27], these techniques provide protection to the execution flow with relatively low overhead, and they are considered a valuable tool in the security domain [1] as a stand-alone mechanism against targeted faults tampering with the control flow.

Research shows the advantages of combining computational redundancy and CFC techniques, like ASPIS. Reis et al. [26] presented SWIFT, which employs CFCSS together with EDDI under the assumption that the underlying memory provides ECCs. Didehban et al. [8] presented near-Zero Silent Data Corruption (nZDC), an improved version of SWIFT adding more instruction redundancy and consistency checks, while at the same time exploiting micro-architectural features of modern devices to lower the overhead. Bohman et al. implemented COAST [6], a platform-independent approach for inserting both DMR and TMR as a set of passes for the LLVM compiler framework, without making assumptions on the underlying hardware in contrast to SWIFT and nZDC. COAST aligns with the approach we described in this paper and has also been tested on FreeRTOS via hardware emulation [16]. They tested the detection capabilities of their instruction triplication (TMR) solution on the FreeRTOS kernel achieving a 3-4x overhead and a total of 0.52%

SDCs with respect to the total number of effective faults in registers, cache and dcache. Overall, the combinations provided by ASPIS have comparable overhead and the majority of them manifest a lower SDC rate. Other than that, ASPIS has also greater usability since the module can be compiled out-of-tree for a more recent version of LLVM and provides a command-line interface to the automatised pipeline. Finally, Sharif et al. described COMPAS [30], an LLVM-based compiler framework implementing techniques such as CFCSS, CEDA, nZDC, SWIFT, and others, for the RISC-V architecture, while also comparing their protection degree carrying out a Monte Carlo analysis. The techniques implemented in COMPAS have been extended to the security domain as well in COMPASeC [12], demonstrating how they can be adopted for ensuring protection against attacks relying on hardware fault injection for instruction skipping.

7 CONCLUSIONS

This paper described an improvement with respect to our previous work [4] and other state-of-the-art works by providing more tools for platform-independent and compiler-injected SIHFT. The novel implementation of ASPIS provides further protection as well as overhead-reduction techniques. The overhead reduction techniques allow us to find some efficient trade-off combinations of SIHFT that make our system at least as effective as other state-of-the-art solutions with comparable features, achieving resiliency to up to 99.842% of effective faults, which is the largest rate in the literature we reviewed. Our experiments concluded that the best option by considering the trade-off between timing overhead and detection is represented by `FDSC` + `RASM` – which combines a novel overhead-reduction technique with a state-of-the-art CFC solution – and suffers from only 0.012% of the total faults injected with an overhead of less than 3.5x.

The future research directions from this paper include the study of the effects of introducing other compiler optimisations, the implementation of recovery techniques, the expansion of the framework to the domains of security and distributed computing, and the compliance of ASPIS with the strict requirements and standards of safety-critical domains.

ACKNOWLEDGMENTS

This work has received funding from National Resilience and Recovery Plan (PNRR) through the National Center for HPC, Big Data and Quantum Computing [11].

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (CCS '05). Association for Computing Machinery, Alexandria, VA, USA, 340–353. ISBN: 1595932267. DOI: 10.1145/1102120.1102165.

[2] Athena Abdi, Seyyed Amir Asghari, Saadat Mozaffari, Hassan Taheri, and Hossein Pedram. 2012. An optimum instruction level method for soft error detection. *International Review on Computers and Software*, 7, (Jan. 2012), 637–641.

[3] Bahman Arasteh, Asgarali Bouyer, and Sajjad Pirahesh. 2015. An efficient vulnerability-driven method for hardening a program against soft-error using genetic algorithm. *Computers & Electrical Engineering*, 48, 25–43. DOI: https://doi.org/10.1016/j.compeleceng.2015.09.020.

[4] Davide Baroffio and Federico Reghenzani. 2023. Compiler-injected sihft for embedded operating systems. In (CF '23). Association for Computing Machinery, Bologna, Italy, 337–343. DOI: 10.1145/3587135.3589944.

[5] R. Baumann. 2005. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22, 3, 258–266. DOI: 10.1109/MDT.2005.69.

[6] Matthew Bohman, Benjamin James, Michael J. Wirthlin, Heather Quinn, and Jeffrey Goeders. 2019. Microcontroller compiler-assisted software fault tolerance. *IEEE Transactions on Nuclear Science*, 66, 1, 223–232. DOI: 10.1109/TNS.2018.2886094.

[7] S. Borkar. 2005. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25, 6, 10–16. DOI: 10.1109/MM.2005.110.

[8] Moslem Didehban and Aviral Shrivastava. 2016. Nzdc: a compiler technique for near zero silent data corruption. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 1–6. DOI: 10.1145/2897937.2898054.

[9] Heiko Falk et al. 2016. TACLeBench: a benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)* (OpenAccess Series in Informatics (OASIcs)). Martin Schoeberl, (Ed.) Vol. 55. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:10.

[10] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: probabilistic soft error reliability on the cheap. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS XV). Association for Computing Machinery, Pittsburgh, Pennsylvania, USA, 385–396. ISBN: 9781605588391. DOI: 10.1145/1736020.1736063.

[11] William Fornaciari et al. 2023. Risc-v-based platforms for hpc: analyzing non-functional properties for future hpc and big-data clusters. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Cristina Silvano, Christian Pilato, and Marc Reichenbach, (Eds.) Springer Nature Switzerland, Cham, 395–410. ISBN: 978-3-031-46077-7. DOI: 10.1007/978-3-031-46077-7_26.

[12] Johannes Geier, Lukas Auer, Daniel Mueller-Gritschneder, Uzair Sharif, and Ulf Schlichtmann. 2023. Compasec: a compiler-assisted security countermeasure to address instruction skip fault attacks on risc-v. In *2023 28th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 1–7.

[13] O. Goloubeva, M. Rebaudengo, M.S. Reorda, and M. Violante. 2006. *Software-Implemented Hardware Fault Tolerance*. Springer US, New York, NY. ISBN: 9780387329376.

[14] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET Benchmarks: Past, Present And Future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)* (OpenAccess Series in Informatics (OASIcs)). Björn Lisper, (Ed.) Vol. 15. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 136–146. ISBN: 978-3-939897-21-7. DOI: 10.4230/OASIcs.WCET.2010.136.

[15] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. 2001. Mibench: a free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 3–14. DOI: 10.1109/WWC.2001.990739.

[16] Benjamin James and Jeffrey Goeders. 2021. Automated software compiler techniques to provide fault tolerance for real-time operating systems. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 1452–1455. DOI: 10.23919/DATE51398.2021.9474205.

[17] Daya Shanker Khudia, Griffin Wright, and Scott Mahlke. 2012. Efficient soft error protection for commodity embedded microprocessors using profile information. *SIGPLAN Not.*, 47, 5, (June 2012), 99–108. DOI: 10.1145/2345141.2248433.

[18] John C. Knight. 2002. Safety critical systems: challenges and directions. In (ICSE '02). Association for Computing Machinery, Orlando, Florida, 547–550. ISBN: 158113472X. DOI: 10.1145/581339.581406.

[19] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. 2009. Statistical fault injection: quantified error and confidence. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, 502–506. DOI: 10.1109/DATE.2009.5090716.

[20] Dario Mamone, Alberto Bosio, Alessandro Savino, Said Hamdioui, and Maurizio Rebaudengo. 2020. On the analysis of real-time operating system reliability in embedded systems. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 1–6. DOI: 10.1109/DFT50435.2020.9250861.

[21] N. Oh, P.P. Shirvani, and E.J. McCluskey. 2002. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51, 1, 111–122. DOI: 10.1109/24.994926.

[22] N. Oh, P.P. Shirvani, and E.J. McCluskey. 2002. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51, 1, 63–75. DOI: 10.1109/24.994913.

[23] Nahmsuk Oh and Edward J McCluskey. 2001. Low energy error detection technique using procedure call duplication. In *Proceedings of the 2001 International Symposium on Dependable Systems and Networks*.

[24] Federico Reghenzani. 2023. Enabling software technologies for critical cots-based spacecraft systems. In *Proceedings of the 20th ACM International Conference on Computing Frontiers* (CF '23). Association for Computing Machinery, Bologna, Italy, 236–242. DOI: 10.1145/3587135.3592765.

[25] Federico Reghenzani, Zhishan Guo, Luca Santinelli, and William Fornaciari. 2022. A mixed-criticality approach to fault tolerance: integrating schedulability and failure requirements. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, Milano, Italy, 27–39. DOI: 10.1109/RTAS54340.2022.00011.

[26] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. 2005. Swift: software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*. IEEE, San Jose, CA, USA, 243–254. DOI: 10.1109/CGO.2005.34.

[27] Abhishek Rhisheekesan, Reiley Jeyapaul, and Aviral Shrivastava. 2019. Control flow checking or not? (for soft errors). *ACM Trans. Embed. Comput. Syst.*, 18, 1, Article 11, (Feb. 2019), 25 pages. DOI: 10.1145/3301311.

[28] Alessandro Savino, Stefano Di Carlo, Gianfranco Politano, Alfredo Benso, Alberto Bosio, and Giorgio Di Natale. 2012. Statistical reliability estimation of microprocessor-based systems. *IEEE Transactions on Computers*, 61, 11, 1521–1534. DOI: 10.1109/TC.2011.188.

[29] ECSS Secretariat. 2016. *Space product assurance - Techniques for radiation effects mitigation in ASICs and FPGAs handbook*. (ECSS-Q-HB-60-02A ed.). European Space Agency. Noordwijk, The Netherlands.

[30] Uzair Sharif, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. 2022. Compas: compiler-assisted software-implemented hardware fault tolerance for risc-v. In *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, 1–4. DOI: 10.1109/MECO55406.2022.9797144.

[31] Venu Babu Thati, Jens Vankeirsbilck, Jeroen Boydens, and Davy Pissort. 2019. Selective duplication and selective comparison for data flow error detection. In *2019 4th International Conference on System Reliability and Safety (ICSRS)*, 10–15. DOI: 10.1109/ICSRS48664.2019.8987731.

[32] Jens Vankeirsbilck, Niels Penneman, Hans Hallez, and Jeroen Boydens. 2018. Random additive control flow error detection. In *Computer Safety, Reliability, and Security*. Barbara Gallina, Amund Skavhaug, and Friedemann Bitsch, (Eds.) Springer International Publishing, Cham, 220–234.

[33] Jens Vankeirsbilck, Niels Penneman, Hans Hallez, and Jeroen Boydens. 2017. Random additive signature monitoring for control flow error detection. *IEEE Transactions on Reliability*, 66, 4, 1178–1192. DOI: 10.1109/TR.2017.2754548.

[34] R. Velazco, D. Bessot, S. Duzellier, R. Ecoffet, and R. Koga. 1994. Two cmos memory cells suitable for the design of seu-tolerant vlsi circuits. *IEEE Transactions on Nuclear Science*, 41, 6, 2229–2234. DOI: 10.1109/23.340567.

[35] Ramtilak Vemu and Jacob Abraham. 2011. Ceda: control-flow error detection using assertions. *IEEE Transactions on Computers*, 60, 9, 1233–1245. DOI: 10.1109/TC.2011.101.

[36] Ze Zhang, Sunghyun Park, and Scott Mahlke. 2020. Path sensitive signatures for control flow error detection. In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (LCTES '20). Association for Computing Machinery, London, United Kingdom, 62–73. ISBN: 9781450370943. DOI: 10.1145/3372799.3394360.

Received 30 November 2023; revised 21 February 2024; accepted 13 April 2024