Compiler-Injected SIHFT for Embedded Operating Systems

Davide Baroffio
davide.baroffio@mail.polimi.it
Politecnico di Milano, Milano, Italy

Federico Reghenzani
federico.reghenzani@polimi.it
Politecnico di Milano, Milano, Italy
European Space Agency, Noordwijk, Netherlands

ABSTRACT

Random hardware faults are a major concern for critical systems, especially when they are employed in high-radiation environments such as aerospace applications. While specialised hardware already exists for implementing fault tolerance, software solutions, named Software-Implemented Hardware Fault Tolerance (SIHFT), offer higher flexibility at a lower cost. This work describes a compiler-based approach for inserting instruction-level fault detection mechanisms in both the application code and the operating system. An experimental evaluation on a STM32 board running FreeRTOS shows the effectiveness of the proposed approach in detecting faults.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; **Software fault tolerance**; • **Computer systems organization** → *Dependable and fault-tolerant systems and networks.*

KEYWORDS

SIHFT, Compilers, Fault Detection, Embedded Systems, Safety

ACM Reference Format:
Davide Baroffio and Federico Reghenzani. 2023. Compiler-Injected SIHFT for Embedded Operating Systems. In *20th ACM International Conference on Computing Frontiers (CF '23), May 9–11, 2023, Bologna, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3587135.3589944

1 INTRODUCTION

Safety- and Mission-critical systems often require a certain degree of resilience to random hardware faults. Such techniques are usually implemented via specialised hardware, for instance, via redundancy of identical processors or entire devices. While hardware implementations are effective from a reliability standpoint, they are very expensive, not only in terms of design and production costs but also power, energy, thermal, and other non-functional metrics [6]. *Software-Implemented Hardware Fault Tolerance (SIHFT)* has been proposed to reduce these costs by moving the hardware fault tolerance to the software-level. In addition, to reduce the cost of the implementation, SIHFT is an enabler for the use of *Commercial Off-The-Shelf (COTS)* hardware components in critical systems, which are usually not designed to be resilient to faults. The most common hardware fault is the *Single Event Upset (SEU)*, which is a transient error occurring randomly during the system execution and usually consists of a bit flip in a memory component of the system. SEUs are a major concern in critical systems, especially, but not only, in aerospace applications. SIHFT solutions mitigate this problem by detecting faults via program state checking and the determination of whether a fault occurred or not. SIHFT techniques able to perform fault recovery also exists. For instance, an application-level redundancy employs a hypervisor that runs multiple replicas of an application and compare the output. With two replicas, the SIHFT method is able to only detect faults, while with three instances it is possible to use a voting mechanism to even correct the output. Other fault recovery techniques include recovery blocks, task re-execution, error-correcting codes (ECC), and others [3, 10]. While application-level techniques are good to detect faults or recover from them when the fault happens in the memory region of the application, they cannot solve the problem of faults occurring during the OS routines execution. This is a barrier, especially for *Real-Time Operating Systems (RTOS)*, that must guarantee both functional correctness and timing correctness [9].

1.1 State of the Art

Instruction duplication is a common SIHFT technique already used in safety-critical systems [12]. In literature, the seminal paper by Oh et al. [8] describes the *Error Detection by Duplicated Instructions (EDDI)* approach. EDDI is the first automatic compiler transformation that introduces instruction duplication and consistency checks in the assembly code to implement fault detection. A full description of the EDDI approach is provided later in Section 2.1. Instruction duplication is, however, insufficient to protect against faults occurring in registers that affect the execution flow (for instance, a SEU in the Program Counter). Several approaches to solve this problem have been developed [7, 13, 14] and usually consist of a run-time verification on the Control-Flow Graph (CFG). They detect unexpected execution paths which deviate from the normal execution flow. One of these approaches, CFCSS [7], is described later in Section 2.2. Reis et al. [11] proposed SWIFT, a fault detection technique that employs both data and execution flow protection with the underlying assumption that the memory features ECC. NZDC [2] improved SWIFT by adding more instructions redundancy and consistency checks, and, at the same time, exploiting advanced features of modern architectures to lower the overhead. Finally, Bohman et al. [1] proposed COAST, a platform-independent automatic compiler instrumentation implementing SIHFT. In contrast to previous works that focus more on high-performance devices, super-scalar processors, or architectures providing specific features, the authors focus more on embedded systems, where such micro-architectural solutions might not be available. Although COAST also provides execution flow protection, their work focuses mainly on instruction

Original program	**Duplicated program**
`LDR r0, [r2]`	`LDR r0,  [r2]`
	`LDR r10, [r12]`
`ADD r0, r1, #1`	`ADD r0,  r1, 1`
	`ADD r10, r11, 1`
	`CMP r0,  r10`
	`BNE error_handler`
`STR r0, [r2]`	`STR r0,  [r2]`
	`STR r10, [r12]`

Figure 1: An example of EDDI transformation, expressed using ARM assembly syntax.

duplication and triplication mechanisms. Our approach differs, and improves, the COAST framework for the following main reasons:

- We provide protection also in the case of multi-level pointers, as subsequently explained in Section 3.2;
- The protection across different compilation units is supported;
- The passes compile and run with the modern versions of LLVM (15.0.0 at the time of writing).

1.2 Contributions

In this paper, we focus on the problem of detecting transient hardware faults, particularly SEUs, by exploiting compiler-injected SIHFT techniques in both application and oeprating system (OS) code. In particular, we focus on instruction-level fault detection, which exploits two mechanisms: *data protection* and *execution flow protection*. The proposed fault detection scheme is general and agnostic with respect to the specific mechanism used for fault recovery.

The novel contribution of this article is twofold:

(1) A set of *passes* implemented in the LLVM software stack to automatically introduce fault detection techniques in both tasks and OS, in a transparent way to the programmer. Given the fact that SIHFT techniques are implemented as LLVM passes, our approach is independent from both the source language and the processor architecture.
(2) An experimental evaluation of the proposed SIHFT fault detection techniques applied to a FreeRTOS system running on a real board, with the goal to quantify the improvement of the reliability metrics. Specific microbenchmarks have been developed to test the resilience of the OS itself.

2 BACKGROUND ON SIHFT

This section provides further details on EDDI [8], a data protection technique, and CFCSS [7], an execution flow protection technique.

2.1 Memory protection

The EDDI instruction-level parallelism is the traditional SIHFT approach for memory protection. This technique duplicates the program assembly instructions and employs different memory areas to store the original and duplicated content. The same operation is performed for data contained in registers. EDDI then adds comparison instructions to perform consistency checks before the so-called *synchronisation points*. The *synchronisation points* are composed of all the branch and store instructions. Figure 1 illustrates the assembly transformation. The `LDR` instruction loading the memory pointed by the address contained in `r2` is duplicated by loading from the address (contained in `r12`) of the duplicated memory area. The arithmetic operation `ADD` is duplicated as well, with each copy working on a different set of registers. After that, because `STR` is a synchronization point, EDDI adds a comparison between the original (`r0`) and the copy (`r10`). If they do not match, the `BNE` instruction jumps to a user-defined `error_handler` that will implement the fault recovery mechanism. Finally the `STR` instructions are duplicated as well, saving the values in two different portions of memory.

The original EDDI mechanism does not consider function calls by design and limits the discussion to simple branches. In our work, as later described, we also considered call and return instructions because they are suitable synchronisation points as well.

2.2 Control Flow Graph protection

For CFG verification, we adopted an approach based on the *Control Flow Checking via Software Signatures* (CFCSS) [7] technique. CFCSS defines a transformation at compile-time divided into two phases: first, it assigns a unique signature s_i to each basic block B_i of the program CFG. Then, CFCSS adds instructions for computing the run-time signature, which is stored in a special register `G` initialised at 0. This register is then compared at the beginning of each basic block with the static signature computed earlier. More specifically, let s_i and s_j be the static signatures of, respectively, the basic blocks B_i and B_j, such that B_j is a successor of B_i. Then, the computation of the run-time signature in register `G` is inserted at the beginning of B_j as follows:

$$\mathtt{G} = \mathtt{G} \oplus d_j$$

with $\oplus$ defined as the bit-wise XOR operation and $d_j = s_i \oplus s_j$ is a value statically computed at compile-time. After the update, `G` and s_j are compared and, in case of a mismatch, a user-defined error handling function is executed.

There are cases that are not supported by this approach as-is. For instance, if a basic block B_j has multiple predecessors, the run-time signature would have different values depending on the basic block the program came from. In order to tackle this, the mechanism defines the so-called *Run-Time Adjusting Signature*, which is a value stored in a dedicated register `D`. An instruction for computing `D` is inserted in each predecessor B_i of B_j right after the instruction updating `G`. We say that a basic block B_k is the *neighbour basic block* of a basic block B_i if they have a successor basic block B_j in common. Let s_i, s_j and s_k be the signatures of the basic blocks B_i, B_j and B_k such that B_i and B_k are neighbours sharing B_j as successor.

Thus, in the basic block B_k, `D` is set to 0. Whereas in B_i, we compute `D` as follows:

$$\mathtt{D} = s_i \oplus s_k$$

In the successor B_j, on the other hand, the instruction computing the run-time signature `G` becomes:

$$\mathtt{G} = \mathtt{G} \oplus \mathtt{D} \oplus d_j$$

with $d_j = s_k \oplus s_j$. This case can be easily extended to the scenario in which B_j has $n > 2$ predecessors by using the same node B_k

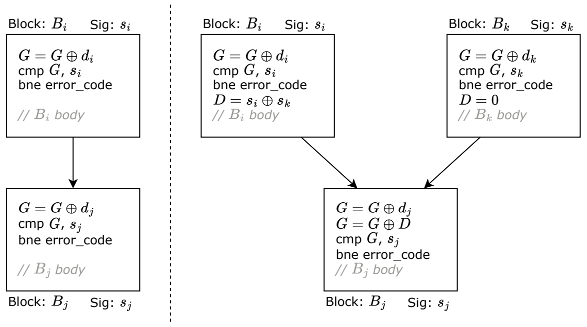


Figure 2: Two examples of the CFCSS transformation: in the left case, the block B_j has only one predecessor, while in the right case, B_j has two predecessors making the addition of the Run-Time Adjusting Signature necessary.

for computing the run-time adjusting signature D in every predecessor B_i of B_j. Figure 2 provides an example of two small CFGs transformed employing CFCSS in the cases in which B_j has 1 and 2 predecessors.

3 AUTOMATED COMPILER-INJECTED SIHFT

To implement the compiler-based approach presented in this article, we followed an approach similar to the COAST framework by Bohman et al. [1]. The implementation as a set of LLVM passes makes the approach independent from the source code and from the architecture. Indeed, LLVM passes are compiler routines that perform transformations on the *Intermediate Representation (IR)*, a code internally used by LLVM to represent the computer program being compiled.

Our framework implements the following LLVM passes:

(1) Transformation of function return values to arguments (`FuncRetToRef`)
(2) Duplication of instructions (`DuplicateInstructions`)
(3) Injection of CFG protection (`CFGVerification`)
(4) Replication of global variables (`DuplicateGlobals`)

The compilation flow is depicted in Figure 3: the LLVM front-end compiles the source code of each compilation unit into IR code, and the multiple IR files are then merged into a single IR file via `llvm-link`. Our pass `FuncRetToRef` transforms the functions to `void` routines and `DuplicateInstructions` adds instruction duplication. At this stage, the IR may contain several empty basic blocks generating complex CFG structures. Therefore, the IR is optimised by using the `simplifycfg` pass provided by LLVM. CFG protection is then injected by the `CFGVerification` pass. When it is not possible to run the passes on some source files (for instance, drivers written in C but containing assembly instructions), their IR is then merged at this stage by invoking `llvm-link` again. Finally, the global variables are protected by the `DuplicateGlobals` pass, and the object file can be generated by the LLVM back-end. The next paragraphs detail the function of each pass.

3.1 Transformation of function return values to arguments

The first pass applied to the IR is `FuncRetToRef`. This pass prepares the IR by transforming functions that have a return value into

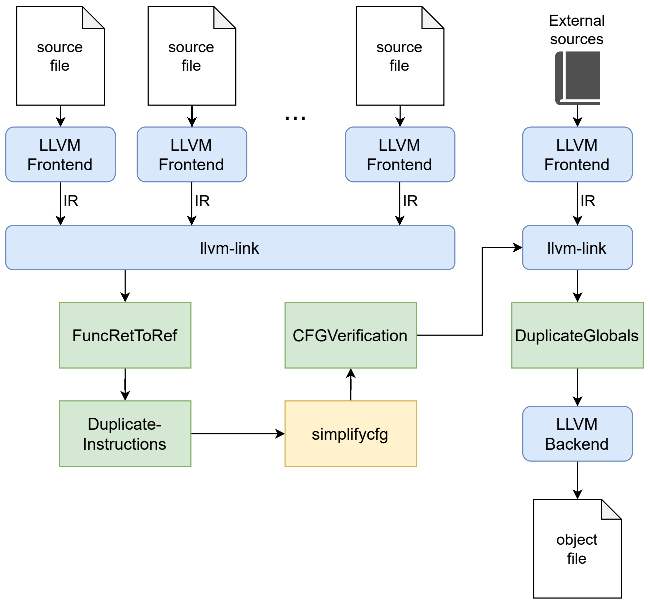


Figure 3: High-level scheme of the compilation flow.

`void` functions for which the return value is passed by reference as a function argument. This transformation is necessary for the subsequent pass in order to perform the duplication on the return value, which becomes a function argument. For example, let `sum` be a function performing the arithmetic sum between two integers and having the following prototype in LLVM IR:

```
define dso_local i32 @sum(i32 noundef %0,
                          i32 noundef %1)
```

The pass transforms the function to a corresponding routine with `void` as the new return type and an additional pointer argument in the prototype:

```
define dso_local void @sum(i32 noundef %0,
                           i32 noundef %1,
                           ptr noundef nonnull %ret_ptr)
```

The attribute `nonnull` is an attribute to state that the pointer is always valid (similar to the C++ pass-by-reference): this attribute allows better optimisations. Let us call `orig_f` the original function and `void_f` the transformed function. After the function prototype transformation, for each return instruction `ret <ty> %ret_val`, the pass:

- adds a `store` instruction, in order to save the returned value into the new pointer argument:
  `store <ty> %ret_val, ptr %ret_ptr`
- replaces the original return with `ret void`.

The IR of the callers of the function must be modified as well. The return value can be used by the caller in two ways that need to be handled differently by our pass.

In the first case, a `store` operation of the function return value immediately follows the `call` instruction. This implies that an `alloca` instruction is already present at the beginning of the caller function, i.e. the necessary memory has already been allocated for saving the function output. Thus, the pass removes this `store` instruction since a `store` is already performed in the callee (see previous paragraph). Finally, the original `call` instruction is replaced with a `call void` to the corresponding `void_f` function, using the value of the `alloca` as the pointer to the new argument.

The second case occurs when the function return value is used directly in a virtual register without storing it in the memory. Consequently, there is no `alloca` for the return pointer in the caller, which is needed to call the new `void_f` function. Hence, the pass creates an `alloca` instruction at the beginning of the caller function and replaces the `call` instruction with the `call void`, similarly to the previous case. In addition, it creates a `load` instruction that replaces all the previous *uses* of the original return value. For clarity, we report this snippet composed of two IR instructions as an example:

```
%ret_val = call i32 some_func()
%1 = add i32 %ret_val, 1
```

Following the transformation, it becomes:

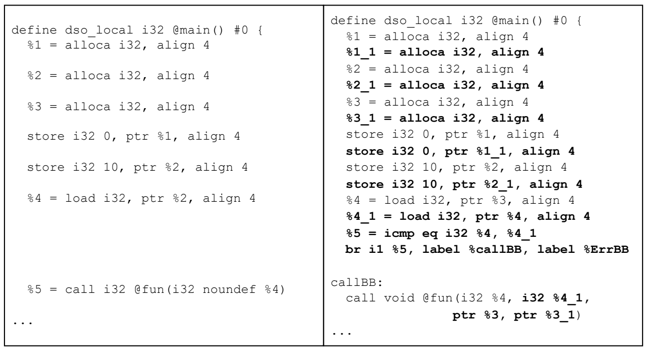
```
%ret_val = alloca i32
...
call void some_func_ret(ptr %ret_val)
%1 = load i32 %ret_val
%2 = add i32 %1, 1
```

3.2 Injection of duplicated instructions

`DuplicateInstructions` is the main pass providing memory protection. It is the IR implementation of a modified version of the EDDI mechanism. Several modifications to the original approach are necessary due to the level of abstraction provided by the LLVM IR. Indeed, EDDI considers only simple branches while we need to take into account also function calls by duplicating both function arguments and return values. Global variables also require a special handling. In particular, our pass cannot protect complex types (such as C `structs`) and pointers when used in the global scope, because they may be unexpectedly updated by functions external to the current compilation unit. Section 3.4 explains in detail this issue and the taken countermeasures.

The first operation performed by the pass consists in duplicating each non-constant global variable of the compilation unit. Then, the pass duplicates the function arguments. Let us call `void_f` the function output of the `FuncRetToRef` pass and `dup_f` the transformed function output of `DuplicateInstructions` pass. Each `dup_f` has a different prototype than the original because it has all the parameters duplicated. The `dup_f` also contains the duplication of the return value parameters added by the `FuncRetToRef` pass, effectively providing return value duplication. The final part iterates over all the instructions of the compilation unit performing a transformation depending on the LLVM IR instruction class:

(1) `AllocaInst`. The instruction is just duplicated and inserted after the original one. We will refer to this transformation simply as *duplication* from now on.
(2) `BinaryOperator`, `UnaryOperator`, `GetElementPtrInst`, `LoadInst`, `CmpInst`, `PHINode`, `SelectInst`. The instruction is duplicated, then the pass verifies that all of its operands have been duplicated in the context of previous instructions. For each non-duplicated operand, the pass recursively tries, if possible, to duplicate it with the same criteria explained in this list.



```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4

  %2 = alloca i32, align 4

  %3 = alloca i32, align 4

  store i32 0, ptr %1, align 4

  store i32 10, ptr %2, align 4

  %4 = load i32, ptr %2, align 4




  %5 = call i32 @fun(i32 noundef %4)

...
```

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %1_1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %2_1 = alloca i32, align 4
  %3 = alloca i32, align 4
  %3_1 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 0, ptr %1_1, align 4
  store i32 10, ptr %2, align 4
  store i32 10, ptr %2_1, align 4
  %4 = load i32, ptr %3, align 4
  %4_1 = load i32, ptr %4, align 4
  %5 = icmp eq i32 %4, %4_1
  br i1 %5, label %callBB, label %ErrBB

callBB:
  call void @fun(i32 %4, i32 %4_1,
                 ptr %3, ptr %3_1)
...
```

Figure 4: Example of LLVM IR code before (left) and after (right) applying the `DuplicateInstructions` pass.

(3) `StoreInst`, `AtomicRWInst`, `AtomicCmpXchgInst`. Both the instruction and its operands are duplicated as described in point (2). In addition, since these instructions perform a store operation in memory, we consider them as suitable *synchronisation points* and the pass adds consistency checks for each pair of operands: the pass inserts a `cmp` instruction to compare their values and adds `and` instructions to compute the logical conjunction of all the `cmp` instructions. Subsequently, the pass adds a conditional branch on the result, jumping to a user-provided error handler routine if any of the operands mismatches. In case one of the operands is of `ptr` type, a special handling is required since the original pointer and its copy point to different areas of memory by design, thus comparing the addresses would always lead to a mismatch and the incorrect execution of the error handling routine. Our pass solves this problem by finding the original memory location by following the chain of `store` instructions and pointers. If it is not possible to find the original memory location, our pass does not compare the operands, reducing the detection capability but preventing false positive detections.
(4) `BranchInst`, `SwitchInst`, `ReturnInst`, `IndirectBrInst`. The approach is similar to the one used for the point (3) with the exception that the instruction is not duplicated because it alters the control flow. The protection will be provided by the next pass `CFGVerification`.
(5) `CallBase`. The modifications are the same as the ones described in the previous point (4), but this time considering the function arguments as the instruction operands. In addition to that, we check whether a `dup_f` version of the called function exists and, in the positive case, the pass substitutes the function call with the `dup_f` using the interleaved duplicated arguments as parameters. Otherwise, due to the fact that the caller is duplicated and the callee is not, the pointers passed as arguments might have been modified by the callee without modifying the duplicate in the caller. To cope with this inconsistency, the pass adds instructions to re-align the duplicated value after the `CallBase` instruction.

3.3 Injection of CFG protection

The `CFGVerification` pass implements a modified version of the CFCSS algorithm described in Section 2.2. The original algorithm requires G and D to be stored in unique registers, however, two problems arose during the development of the LLVM pass:

- Since the IR is platform-independent, no information on the actual registers available in the target architecture is available, and enforcing the allocation of G and D into dedicated registers would be against the IR philosophy.
- The original algorithm considers only basic blocks and not functions. Instead, each function requires dedicated G and D memory locations, that could not, consequently, be stored in registers. The reason is that computing G and D requires the knowledge of a basic block's predecessors and successors: this requirement is not met when the functions called are not predictable at compilation time (*e.g.* callback functions invoked via relative jumps). In particular, this causes both a problem for the callee because it will likely find an invalid run-time signature into `G`, and for the caller, because the values of the registers `G` and `D` are overwritten by the callee.

In order to deal with these issues, two virtual registers containing the local G and D values are created for each function, instead of having single instances in the global scope. Each function manages its own G and D values separately and in local scope of the function. The virtual registers are then transformed in real hardware registers or moved to memory by the register allocator of the LLVM backend. Other than this difference, the our algorithm is equivalent to CFCSS.

3.4 Replication of global variables

In some cases, not all the functions can be compiled with our passes. Typical examples are driver functions containing inline assembly. In this case, clearly, no IR representation is available and the passes cannot work and it can happen that some global variables are updated inside those functions excluded from the transformation.

The `DuplicateGlobals` pass has the objective of solving this issue. The pass consists of a duplication of the `store` instructions that have as destination the global variables duplicated during `DuplicateInstructions`. For instance, if `some_global` is a global `int` variable and has `some_global_dup` as its duplicate, the single instruction `store i32 10, ptr @some_global` belonging to one of the excluded functions is transformed to:

```
store i32 10, ptr @some_global
store i32 10, ptr @some_global_dup
```

The `DuplicateGlobals` is necessary to enable the compatibility between functions instrumented by the pass and functions that cannot be instrumented by the pass.

3.5 Limitations

Our framework has some limitations regarding the coverage of all the possible input code features. In particular: (1) global variables of complex types are not duplicated (but correctly managed); (2) two subsets of LLVM IR instructions – i.e. exception handling instructions and vector instructions – are not currently supported[1]; (3) The local storage of `G` and `D` prevents the detection of illegal branches that jumps at the very beginning of the first basic block of a function. We plan to solve all of these limitations and improve the detection capabilities in a future version of the framework.

[1]In any case, this limitation is irrelevant in the experiments of Section 4 because they are not supported by the architecture/OS used.

4 EVALUATION OF A REAL SYSTEM BASED ON FREERTOS

We tested our set of LLVM passes on a real board running the FreeRTOS. The objective of this experimental campaign is to test the ability of the pass to detect faults, especially in the OS routines that are usually the most challenging to protect. A similar previous work, which also analysed FreeRTOS, exists in literature [5]. However, the authors run experiments emulating the processor and by considering faults occurring only in registers and cache, excluding the main memory. In our experimental evaluation, instead, we injected faults on the real hardware and tested the memory locations, stressing the FreeRTOS functions via specifically developed microbenchmarks. The code is open-source and publicly available[2].

4.1 Hardware Setup and Methodology

We compiled FreeRTOS with the LLVM and its passes described in Section 3 for a NUCLEO-L152RE board equipped with a microcontroller STM32L152RET6. While applying SIHFT techniques to application tasks is quite straightforward, we decided to investigate the application of our framework to the OS. For this reason, the workload used in this experimental evaluation is composed of a set of microbenchmarks to test the functionalities of FreeRTOS. They are described in the next Section 4.2. Then, in order to simulate SEUs in our system, we exploited the debugging interface available in the NUCLEO board via the ST-link GDB server. A set of scripts orchestrates the whole testing process in a transparent way with respect to the code running on the target. A host PC controls the experiments by providing commands to the GDB server that halts the execution of the target code, injects the fault at a random memory location, and resumes the execution, by verifying the effect of the injected fault. In particular, the possible results are: *Fault Detected (memory protection)*, *Fault Detected (CFG protection)*, *Fault Detected (hardware)*[3], *Silent Data Corruption*[4], *Loop*[5], and *No Effect*[6].

The experiments have been performed on the original version of FreeRTOS, in order to establish a baseline, and then on the version compiled with our framework. In both cases, faults were injected in two different scenarios: faults occurring in registers and faults occurring in memory locations. We injected approximately 12 000 faults in a 10-hours campaign for each scenario.

4.2 Microbenchmarks

We developed a set of microbenchmarks to test the main components of FreeRTOS: tasks, queues, stream buffers, and timers. Because the microbenchmarks are implemented as tasks in the system, the schedule generated by the scheduler has been also implicitly tested. The 8 tasks are implemented as traditional real-time periodic tasks and scheduled with the default round-robin policy. To

[2]https://doi.org/10.5281/zenodo.7788290
[3]Some faults are detected by the hardware due to, for instance, the access to illegal memory addresses or the execution of incorrect instructions.
[4]The software produced an incorrect result, but no fault was detected.
[5]The target is stuck in an infinite loop.
[6]The fault produced no effect because, for instance, occurred in an unused memory region.

Table 1: Results of the fault injection experiments without and with our framework. Percentages are expressed as the ratio with respect to the total number of errors in the experiment.

	Registers				**Memory**			
	w/o SIHFT	%	**w/ SIHFT**	%	**w/o SIHFT**	%	**w/ SIHFT**	%
Total injections:	11988		13026		11933		12915	
No Effect:	10775		11090		11553		12629	
Errors:	1213	100	1936	100	380	100	286	100
Loop:	290	23.9	141	7.2	39	10.3	21	7.3
Silent Data Corruption:	117	9.6	9	0.5	46	12.1	10	3.5
Fault Detected (hardware):	806	66.5	525	27.1	295	77.6	166	58.1
Fault Detected (data):	-		595	30.7	-		51	17.8
Fault Detected (execution flow):	-		666	34.5	-		38	13.3

increase the probability of observing faults, the system utilisation ratio was near 1. The microbenchmark suite is composed of:

- `vTaskTaskTest`: it tests the task-related functions, such as task creation, deletion, suspension, resume, priority assignment.
- `vTaskQueueTest`X: they are four tasks ($1 \leq X \leq 4$) testing different scenarios of queue insertion, reading, resetting, synchronisation, creation, and deletion.
- `vTaskBufferTestSend` and `vTaskBufferTestReceive`: they test message buffers in a producer–consumer fashion.
- `vTaskTimerTest`: it tests the timer-related function: creation, reading, modification, and reloading.

The majority of the FreeRTOS functions have been compiled with our framework, with some exceptions. Static queues creation, sleep functions, the timer handling function `xTimerGenericCommand`, and the context switch routine are currently not possible to be protected by our passes due to the presence of highly architecture-dependent features. For instance, the context switch routine is directly implemented in assembly, bypassing the LLVM IR.

4.3 Results

Table 1 provides an overview of the experimental results. The amount of total injections is the sum of the number of injected faults that had no effect on the execution and the number of faults that caused errors, both detected and undetected. Each column of the table represents a separate test, detailing the injection outcome for each combination of target (Memory/Registers) and SIHFT protection (enabled/disabled). The different amount of total injections for each test is due to the complex injection setup needed to interact with the debugging interface. As expected, the number of faults that produce an error – i.e. a deviation from a correct result or execution – are approximately 10% for registers and 3% for memory.

Thanks to our framework, the amount of *Loop* errors – i.e. when the system is unresponsive because stuck in infinite loops – in registers is reduced from 23.0% to 7.2% (ratio of the total errors). *Loop* errors in the memory dropped from 10.3% to 7.3%. Even if not applied in this experimental evaluation, the *Loop* errors can be also detected by watchdog timers. *Silent Data Corruption* errors, which are the most critical because undetectable, have been almost zeroed in the registers case (from 9.6% to 0.5%) and reduced from 12.1% to 3.5% in the memory. The number of faults detected by the hardware dropped from 66.5% to 27.1% in the register case, and from 77.6% to 58.1% in the memory case. This reduction improves the real-time capabilities of the system since in most cases recovery from hard faults requires a system reboot, while for SIHFT-detected faults it usually suffices to run a predefined recovery routine.

In conclusion, our compiler-injected SIHFT is able to detect 65.2% of the errors in the registers (for a total detection of 92.1% including hardware detection, and 99.5% if we include watchdogs) and 31.1% of the errors in the memory (for a total detection of 89.2% including hardware detection, and 96.5% including watchdogs). Overall, compared to the total number of injected faults, we can state that our system compiled with our pass is able to tolerate 99.3% of the SEUs – which increases to 99.92% if we consider the expiration of the watchdog as valid detection.

4.4 Preliminary overhead characterisation

In order to evaluate the introduced overhead, we run *DES* and *Matrix Multiply* benchmarks from the Mälardalen suite [4]. Our SIHFT mechanism increases the average execution time by about 4.12x for *DES* and 4.38x for *Matrix Multiply* with respect to the original code with no protections. These overheads are in line with the aforementioned previous works. We also computed the overhead in terms of binary size, resulting in 2.95x increase in the program code section of the FreeRTOS binary.

5 CONCLUSIONS

In this article, we described a set of LLVM passes that automatically introduces SIHFT techniques for fault detection in the code, independently from the source code or the underlying architecture. We then showed how the passes are effective by running them on FreeRTOS and a real board through a set of specifically developed micro-benchmarks, obtaining an overall fault tolerance to SEU over 99%. The number of Silent Data Corruption errors is < 1% for the registers and < 4% for the memory.

ACKNOWLEDGMENTS

This work has received funding by the European Space Agency (OSIP no. 4000133770 / 21 / NL / MH / hm), the ICSC National Research Center in High-Performance Computing, and EU H2020 Marie Sklodowska-Curie project APROPOS (Grant Agreement no. 956090).

REFERENCES

[1] 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 (2019), 223–232. https://doi.org/10.1109/TNS.2018.2886094

[2] Moslem Didehban and Aviral Shrivastava. 2016. NZDC: A Compiler Technique for near Zero Silent Data Corruption. In *Proceedings of the 53rd Annual Design Automation Conference* (Austin, Texas) *(DAC '16)*. Association for Computing Machinery, New York, NY, USA, Article 48, 6 pages. https://doi.org/10.1145/2897937.2898054

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

[4] 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), Vol. 15)*, Björn Lisper (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 136–146. https://doi.org/10.4230/OASIcs.WCET.2010.136 The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.

[5] 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)*. IEEE, Grenoble, France, 1452–1455. https://doi.org/10.23919/DATE51398.2021.9474205

[6] Nahmsuk Oh. 2001. *Software Implemented Hardware Fault Tolerance.* Ph. D. Dissertation. Stanford University.

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

[8] 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 (2002), 63–75. https://doi.org/10.1109/24.994913

[9] Federico Reghenzani, Zhishan Guo, and William Fornaciari. 2023. Software Fault Tolerance in Real-Time Systems: Identifying the Future Research Questions. *ACM Comput. Surv.* (mar 2023), 30 pages. https://doi.org/10.1145/3589950 In Publishing.

[10] 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. https://doi.org/10.1109/RTAS54340.2022.00011

[11] 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. https://doi.org/10.1109/CGO.2005.34

[12] 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.

[13] 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 (2017), 1178–1192. https://doi.org/10.1109/TR.2017.2754548

[14] Ramtilak Vemu and Jacob Abraham. 2011. CEDA: Control-Flow Error Detection Using Assertions. *IEEE Trans. Comput.* 60, 9 (2011), 1233–1245. https://doi.org/10.1109/TC.2011.101