Hardening On-Board Software: a Low-Overhead Compiler-based Approach

Emilio Corigliano, Davide Baroffio, Tomas Antonio Lopez, Federico Reghenzani
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milano, Italy
name.surname@polimi.it

***Abstract*—As challenges related to system complexity, cost, weight, power consumption, and real-time performance are increasingly crucial, mixed-criticality systems become more common in low-cost space avionics. Radiation-induced transient faults further complicate the design, as conventional mitigation strategies exacerbate these issues. This work presents REDDI, a selective Software Implemented Hardware Fault Tolerance (SIHFT) technique that applies fault tolerance at compilation time by recursively protecting developer-annotated resources. We detail this method, introduce the concept of *software spheres*, and address key implementation challenges. The approach is evaluated on a set of benchmarks and a real-world mixed-criticality On-Board Software (OBSW) for experimental sounding rockets, running on an STM32-based On-Board Computer (OBC). Results demonstrate a significant reduction in the overhead typically associated with SIHFT, while maintaining a high fault detection rate in critical code sections, proving the effectiveness of the proposed solution.**

***Index Terms*—SIHFT, fault-detection, compilers, mixed-criticality**

I. Introduction

Radiation-induced transient faults – in particular Single Event Upsets (SEUs) [1] – are a critical design challenge when dealing with low-cost On-Board Computers (OBCs). In this context, radiation-hardened OBCs are not a viable solution for their prohibitive costs, whereas shielding and hardware redundancy – e.g., lockstepping [2] – greatly increase system complexity. The need for protection of critical components in Mixed-Criticality Systems (MCSs) [3] further exacerbates these disadvantages [4], since current hardening techniques do not exploit the separation in criticality levels of the software.

A. Background - SIHFT and ASPIS

Recent solutions employed Software Implemented Hardware Fault Tolerance (SIHFT) techniques, which mitigate – via pure software techniques – the effects of SEUs affecting the data-flow or control-flow of the program. These techniques allow the employment of Commercial Off-The-Shelf (COTS) OBCs for critical systems, reducing costs, weight, and system complexity relative to traditional solutions [5]. SIHFT techniques are traditionally implemented manually by the user, leading to a high development time. To solve this issue, recent research efforts are going towards the design of automatic compiler-based SIHFT techniques. Automatic Software-based Protection and Integrity Suite (ASPIS) [6] falls into this category, implementing data-flow [7] and control-flow [8] [9] hardening at compilation time, using a modern version of the LLVM compiler infrastructure.

In the literature, we can find different compiler-based SIHFT solutions that have been implemented to protect applications from transient faults affecting both data- and control-flows. This work focuses solely on the data-flow protection mechanism implemented in ASPIS, namely Error Detection by Duplicated Instructions (EDDI). Additional protection can be obtained by pairing data-flow hardening with a control-flow graph protection pass, which detects illegal jumps caused by bit-flips in the instruction pointer register or the return address on the stack.

EDDI [7] is a leading state-of-the-art algorithm among data-flow SIHFT techniques. It mitigates bit-flips affecting the data of the program by replicating instructions and comparing their results right before the program stores them in the memory or when the control-flow of the program changes with jump and call instructions. These instructions are called *synchronization points*, wherein the consistency of the program state is verified. ASPIS implements EDDI as a middle-end compilation pass in the LLVM [10] compilation flow, modifying the program at the LLVM Intermediate Representation (LLVM-IR) level. This approach provides many advantages in terms of flexibility, as the transformation can be implemented in a language- and architecture-independent fashion.

Listing 1 presents a snippet of LLVM-IR in which two variables are incremented and then stored into a global variable.

```
0  %b = add i32 %a, 1
   %d = add i32 %c, 1
2  store %b, %critical_var
   store %d, %non_critical_var
4  ...
```

Listing 1. LLVM-IR showing how two variables are incremented and stored into global variables.

Given the input program of Listing 1, EDDI performs the duplication of all the variables and instructions of the program as shown in Listing 2. By default, ASPIS injects consistency checks to verify the integrity of all the operands of store, jump, and call instructions.

Oh and McCluskey [11] enhanced EDDI to support duplicating procedure calls when direct hardening is not feasible. This allows the protection of programs that use external

libraries with already compiled code, which, for this reason, cannot be protected. This principle has been adopted in a previous work on ASPIS [6] to correctly manage duplication among architecture-dependent FreeRTOS components (i.e., part of the *portable* module) that could not be directly instrumented.

```
    %b = add i32 %a, 1
    %b_dup = add i32 %a_dup, 1
    %d = add i32 %c, 1
    %d_dup = add i32 %c_dup, 1
    %b_check = icmp eq i32 %b, %b_dup
    %d_check = icmp eq i32 %d, %d_dup
    %is_consistent = and i1 %b_check, %d_check
    br i1 %is_consistent, label %OkBB, label %ErrBB

OkBB:
    store %b, %critical_var
    store %b_dup, %critical_var_dup
    store %d, %non_critical_var
    store %d_dup, %non_critical_var_dup
    ...
```

Listing 2. EDDI hardened variable example.

B. Related work

Although EDDI seems to provide an easy solution to the problem of fault-tolerance, it introduces a massive overhead for the full duplication of the program to be protected. For this reason, multiple approaches have been tried to reduce the overhead of EDDI. Notable examples are Shoestring by Feng et al. [12], a profiling-based algorithm by Khudia et al. [13], and a genetic algorithm by Arasteh et al. [14]. Finally, some other selective techniques performed critical basic block identification by selecting the ones with the highest number of fan-out nodes (CBD by Abdi et al. [15]) or the ones belonging to the longest path in the Control Flow Graph (CFG) (SDSC by Thati et al. [16]).

A common limitation of the state-of-the-art selective data-flow protection techniques is that they do not exploit knowledge on criticality levels of the resources to harden, implementing uniform protection on the whole code base. The "selectiveness" of these techniques derives from uniformly protecting less of the whole system, selecting automatically which software resources to harden.

C. Contributions

The main focus of this work is the design of the Recursive EDDI (REDDI) algorithm. In particular, our contributions are:

- Formalization of the annotation-driven partitioning of software into *software spheres* and their interactions.
- Implementation of REDDI, a selective compiler-based SIHFT technique for data-flow protection, as a compilation pass in the middle-end of the LLVM framework.
- Improvement of ASPIS for hardening non-trivial C and C++ code.
- Testing of our SIHFT solution on a real-world mixed-criticality On-Board Software (OBSW) and a set of benchmarks for real-time systems.

II. The REDDI algorithm

We present REDDI, a compiler-based approach for selectively hardening mixed-criticality systems which generalizes EDDI. REDDI protects only critical resources selected by the user and all their data dependencies, balancing fault detection capability, computational performance, and memory overhead. This technique has been implemented as a compilation pass in ASPIS, which performs the transformation in the middle-end of the LLVM [10] compilation flow, modifying the program at the LLVM-IR level. This technique shifts the concept of selective data-flow protection, offering maximum protection on critical resources with minimal impact on the overall system performance, as it does not protect non-critical code.

The example in Listing 1 would be hardened as shown in Listing 3, assuming the user annotates `critical_var` as a critical variable. REDDI duplicates the marked variable, all variables required to compute its value, and any variables that depend on it. All the other variables are left untouched.

```
    %b = add i32 %a, 1
    %b_dup = add i32 %a_dup, 1
    %d = add i32 %c, 1
    %is_consistent = icmp eq i32 %b, %b_dup
    br i1 %is_consistent, label %OkBB, label %ErrBB

OkBB:
    store %b, %critical_var
    store %b_dup, %critical_var_dup
    store %d, %non_critical_var
    ...
```

Listing 3. REDDI hardened variable example.

REDDI systematically defines the separation in *software spheres* and their relation with the `to_harden`, `to_duplicate`, and `exclude` source-code annotations used by ASPIS. The following sections will formally define such concepts and describe the REDDI algorithm, along with the additional features necessary to support the hardening of C++ programs.

A. Spheres and Annotations

We divide the software resources into four software spheres: *Sphere of Replication (SoR)*, *Sphere of Duplication (SoD)*, *Sphere of Exclusion (SoE)*, and a *gray area*. By software resources, we refer to global variables (which we call just *variables* for simplicity), instructions, and functions, each of which will be treated differently by REDDI depending on the sphere each resource belongs to. The *SoR* constitutes the trusted area where data and instructions are protected through EDDI duplication. The *SoD* is composed of all the functions whose code is not duplicated, i.e., they are not protected, but that will be called twice when the call instruction has to be hardened (in line with Oh and McCluskey's extension of EDDI [11]). Resources in the SoD comprise functions from already compiled external libraries, where direct function hardening is not feasible due to source code unavailability. The *SoE*, instead, comprises resources that need to be excluded from the duplication: a call to a function in this area will not be hardened, usually to avoid the duplication of side

effects (e.g., interfacing with a hardware peripheral). Finally, the *gray area* represents the untrusted area that serves as a potential expansion zone for the SoR. REDDI moves the needed resources from the gray area to the SoR during the recursive hardening process described in Section II-C.

Three source code annotations enforce the initial division into spheres, which are then expanded during the transformation to maximize the protection of critical resources while leaving the others untouched. The `to_harden` annotation specifies that the marked function or global variable is critical and should be recursively hardened. The `to_duplicate` annotation – admissible only for functions – adds resources to the SoD so that the calls to these functions will be duplicated without the need to harden the callee itself. Finally, the `exclude` annotation is used to define the SoE, so that the resources marked in this way will not be affected by duplication or hardening. The remaining resources initially belong to the gray area. Once the recursive protection is complete, any remaining resources in this area will be left untouched to prevent unnecessary overhead.

B. Basic resource hardening

The process of hardening a variable, a function, or an instruction directly derives from the implementation of EDDI in ASPIS.

Specifically, a variable – global or local – is hardened by creating a duplicate with identical characteristics. REDDI ensures consistency between the original and duplicated variables by recursively hardening all their uses and adding checks only before the synchronization points (i.e., `store`, `branch`, and `call` instructions). The algorithm moves instructions from the gray area to the SoR, since all instructions in the data-flow graph are now replicated. This recursive hardening ensures the duplication of instructions that generate the original value that is being protected, effectively maintaining consistency between original and duplicate resources. Instructions, instead, are hardened by cloning them, with the duplicated version using the corresponding duplicated variables.

Functions are hardened by creating a new function with a modified signature. In particular, the new hardened function accepts two arguments for each original argument, one for the original and one for the duplicated version of the parameter. All the instructions of the hardened function are duplicated, adding consistency checks at every synchronization point. If the function was explicitly marked with the `to_harden` annotation (i.e., it belongs to the SoR), REDDI recursively protects also every global variable used in the function body.

Call instructions, on the other hand, require special treatment. A call instruction to a function in the SoR is hardened by calling the duplicated version of the called function, passing both the original and duplicated versions of each argument. If the function is in the SoD, instead, the call is *duplicated*. REDDI duplicates a call by generating two calls to the unprotected function, one with the original arguments and the other with the duplicated set of arguments.

For example, consider the call in Listing 4, where `my_func` function is called and one argument is accepted (i.e., `my_arg`).

```
0 call void @my_func(ptr @my_arg)
```

Listing 4. example of a function call.

The hardened call will look like Listing 5, where the duplicate version of `my_func` is called (i.e., `my_func_dup`). This hardened function accepts two arguments, one for the duplicate version – `my_arg_dup` – and the other for the original version – `my_arg` – of the parameter passed to the original call.

```
0 call void @my_func_dup(ptr @my_arg_dup, ptr @my_arg)
```

Listing 5. hardened function call.

If the called function has to be duplicated, the call will be cloned and the second version will use the duplicated arguments as shown in Listing 6.

```
0 call void @my_func(ptr @my_arg)
  call void @my_func(ptr @my_arg_dup)
```

Listing 6. duplicated function call.

C. Recursive hardening

REDDI recursively hardens global variables and functions user-annotated with the `to_harden` annotation, hardening all their data-flow and the functions called by the hardened resource. All these resources are inserted in the SoR if they are not already in the SoD or SoE. Particular care is needed to handle the interactions among the three spheres and the gray area. These interactions, in most cases, are represented by calls.

Temporary Argument Duplication (TAD) is a technique introduced in REDDI to address cases where a function call within the gray area or the SoD requires hardening, but some arguments are not in the SoR (i.e., they have not been duplicated). This mechanism ensures consistency by creating a temporary duplicate of non-hardened arguments while balancing performance overhead, recursively hardening only explicitly annotated arguments. Thus, REDDI safeguards critical resources while avoiding unnecessary duplication.

Listing 7 shows an example of a call in the gray area to a SoR function. In this example, `%a` is the critical resource while `%otherVar` and `@function` are gray area resources.

```
0 %a = call ptr @_Znwm(i64 16)
  ...
2 call void @function(ptr %a, ptr %otherVar)
```

Listing 7. An example of a call in the gray area.

Listing 8 shows how TAD applies to this example. The gray area resource `otherVar` is temporarily duplicated right before the function call, allocating the needed space and copying the content of that variable into the space allocated. Then, `function_dup` is called, using the just-created variable as the duplicate version of `otherVar`.

TABLE I
REDDI CALLING CONVENTIONS.

		Callee Function					Argument		
		eSoR	SoR	SoD	SoE	Gray Area	SoR	SoE	Gray Area
Caller Function	SoR	Harden	Harden	Duplicate	-	Harden	Harden	-	Harden
	SoD	**Harden TAD**	-	-	-	-	**Harden TAD**	-	-
	SoE	-	-	-	-	-	-	-	-
	Gray Area	**Harden TAD**	-	-	-	-	**Harden TAD**	-	-

```
%a = call ptr @_Znwm(i64 16)
%a_dup = call ptr @_Znwm(i64 16)
...
%2 = alloca [24 x i8], align 8
call void @llvm.memcpy.p0.p0.i64(ptr align 8 %2, ptr align 8 %otherVar,
      i64 24, i1 false)
call void @function_dup(ptr %a_dup, ptr %2, ptr %a, ptr %otherVar)
```

Listing 8. Temporary argument duplication applied to a call in the gray area.

In the following, we explain how REDDI performs recursive protection in different cases, also covering special instances when balancing protection and duplicated variables coherency is not trivial. In Table I we summarize REDDI's calling conventions in the case of a call from a sphere to a function in another sphere.

1) Recursively protecting functions: REDDI inserts in the SoR – if not already in SoD or SoE – all the functions called by other SoR functions. If the called function is in the SoD, REDDI duplicates the call to the original version of the function. Every call to a function in the SoE is untouched. Function calls made from within the SoD or the gray area are hardened only if the callee is explicitly marked with the `to_harden` annotation (referenced as eSoR). In this case, the callee is moved to the SoR and the call is hardened, invoking the duplicated version of the function and applying TAD to non-hardened arguments. All calls in SoE are kept untouched, calling the original version of the function, to avoid argument duplication and any wrong handling of possible side effects.

2) Recursively protecting variables: Instructions that use variables in the SoR will be duplicated everywhere – SoR, SoD, SoE, and gray area – to ensure consistency, except for calls, which have to be handled separately.

Calls of gray area functions with at least one argument in the SoE are left untouched to avoid side effects; if the arguments are all in the gray area, the call is hardened only if performed in a SoR function. Invocations of gray area functions with at least one argument in the SoR are handled depending on the sphere of the caller and the arguments: If these calls are performed in the SoR, they are hardened; if they are in the SoE, they are kept untouched; finally, if they are performed in the SoD or gray area, they are hardened, applying TAD on the non-hardened arguments.

D. Supporting C/C++ features

Applying the REDDI algorithm on complex programs raises some issues that have to be handled explicitly.

a) Variadic functions: Most modern programming languages allow for *variadic functions*, which are functions that accept a variable number of arguments (e.g., `printf`). We have to handle this kind of function differently, since the way of accessing the variadic arguments is automated through standard macros, which would not work properly if all the arguments were to be duplicated. For this reason, we duplicate only the fixed arguments. Apart from this, both the function body and local variables are still duplicated. Thus, the variadic arguments are locally protected when used inside the function. Listing 9 presents an example of the hardening of a variadic function with one fixed argument.

```
define void @var_func(i32 %0, ...)
define void @var_func_dup(i32 %0, i32 %1, ...)

define void @func() {
  ...
  call void @var_func(i32 %par, ptr %0, ptr %1)
  ...
}

define void @func_dup() {
  ...
  call void @var_func_dup(i32 %par_dup, i32 %par, ptr %0, ptr %1)
  ...
}
```

Listing 9. Hardened variadic function.

b) Indirect calls: Indirect calls are calls in which the address of the callee is determined at runtime, invoking the function through its pointer. In C++, classes can have *virtual methods*, which allow derived classes to provide their own implementations of these methods. The implementation to be used is retrieved through the *virtual table*, effectively performing indirect method calls. The problem with indirect calls is that it is impossible to know whether the called function has a duplicate version. To solve this, REDDI protects the virtual tables of hardened objects so that retrieving a virtual method from them will return a pointer to the hardened version of that method. The REDDI transformation phase, then, blindly duplicates the call parameters of indirect calls, assuming that the call is performed on a duplicate function.

Listing 10 shows a C++ class with a virtual method – `virtMethod` – overridden by `ClassDerived`.

```
class ClassBase {
public:
    virtual void virtMethod() = 0;
};

class ClassDerived : public ClassBase {
public:
    void virtMethod() override {
        ...
    }
};
```

Listing 10. A C++ class with virtual methods.

Listing 11 shows the LLVM-IR code of the virtual table and class constructor of the `ClassDerived` class. The virtual table `_ZTV12ClassDerived` is an array of pointers to the virtual methods (i.e., `_ZN12ClassDerived10virtMethodEv`, which is the mangled name for the `ClassDerived` implementation of `virtMethod`). In `_ZN12ClassDerivedC2Ev` – the constructor of `ClassDerived` – the address of the virtual table `_ZTV12ClassDerived` is saved as the first field of the created object.

```
@_ZTV12ClassDerived = constant { [3 x ptr] } {
[3 x ptr] [ptr null, ptr @_ZTI12ClassDerived, ptr
      @_ZN12ClassDerived10virtMethodEv]
}

define void @_ZN12ClassDerivedC2Ev(ptr %0) {
  call void @_ZN9ClassBaseC2Ev(ptr %0)
  store ptr getelementptr inbounds ({ [3 x ptr] }, ptr
      @_ZTV12ClassDerived, i32 0, inrange i32 0, i32 2), ptr %0, align 8
  ret void
}
```

Listing 11. LLVM-IR for a C++ constructor of a class with virtual methods.

REDDI hardens this example as in Listing 12 by creating `_ZTV12ClassDerived_dup`, the duplicate version of the virtual table, and `_ZN12ClassDerivedC2Ev_dup`, the duplicate version of the constructor. In the duplicate version of the constructor, the store of the virtual table to the first element of the object is duplicated, resulting in the redundancy of the virtual table address.

```
@_ZTV12ClassDerived_dup = constant { [3 x ptr] } {
[3 x ptr] [ptr null, ptr @_ZTI12ClassDerived, ptr
      @_ZN12ClassDerived10virtMethodEv_dup]
}

define void @_ZN12ClassDerivedC2Ev_dup(ptr %0, ptr %1) {
  call void @_ZN9ClassBaseC2Ev_dup(ptr %0, ptr %1)
  store ptr getelementptr inbounds ({ [3 x ptr] }, ptr
      @_ZTV12ClassDerived_dup, i32 0, i32 0, i32 2), ptr %1, align 8
  store ptr getelementptr inbounds ({ [3 x ptr] }, ptr
      @_ZTV12ClassDerived_dup, i32 0, i32 0, i32 2), ptr %0, align 8
  ret void
}
```

Listing 12. LLVM-IR for a hardened C++ constructor of a class with virtual methods

c) Global constructors: In C++, all global objects are constructed before the main starts. This is implemented in LLVM with a function for each compilation unit, where all the global objects are created. These functions are inserted in an intrinsic LLVM structure called `llvm.global_ctors`, structured as shown in Listing 13.

```
@llvm.global_ctors = appending global [2 x {i32, ptr, ptr}] [
  {i32, ptr, ptr} {i32 65535, ptr @_GLOBAL__sub_I_file1.cpp, ptr null},
  {i32, ptr, ptr} {i32 65535, ptr @_GLOBAL__sub_I_file2.cpp, ptr null}
]
```

Listing 13. Global constructors structure.

However, if a global object has to be hardened, the global constructor to be called is the duplicate version of the constructor. In the case that a whole compilation unit is hardened (e.g., the `file1.cpp`), also the function calling the global constructors of the compilation unit is hardened and, for this reason, the `llvm.global_ctors` has to be updated like in Listing 14.

```
@llvm.global_ctors = appending global [2 x {i32, ptr, ptr}] [
  {i32, ptr, ptr} {i32 65535, ptr @_GLOBAL__sub_I_file1.cpp_dup, ptr
      null},
  {i32, ptr, ptr} {i32 65535, ptr @_GLOBAL__sub_I_file2.cpp, ptr null}
]
```

Listing 14. Hardened global constructors structure.

d) Volatile variables: In C and C++, the `volatile` keyword indicates that a variable's value may be changed externally, preventing compiler optimizations. It is commonly used for memory-mapped hardware interactions, where reads and writes cause side effects. To avoid unintended behaviors, REDDI includes all `volatile` operations in the SoE, ensuring correct hardware behavior.

III. Experiments

REDDI is deployed as an LLVM middle-end pass within the ASPIS pipeline[1], operating at LLVM-IR and controlled by three annotations (`to_harden`, `to_duplicate`, `exclude`). A simple transformation pass called `AddAnnotation` serves as a helper to annotate all the resources of an LLVM-IR module passed in input. All the annotated modules are then linked via `llvm-link` and fed to REDDI. Optionally, we could also protect the control-flow graph of the program with a Control-Flow Checking (CFC) algorithm after a pre-processing of the Module, which simplifies the control-flow graph and eliminates eventual dead code. However, the experiments presented do not include CFC hardening. This way, we highlight the detection capabilities of REDDI alone. Finally, the modified LLVM-IR module passes through the LLVM backend and is linked with eventual external libraries, yielding the protected object file to be executed. ASPIS compilation flow is presented in Fig. 1.

We tested REDDI by hardening a set of benchmarks and the mixed-criticality OBSW of an experimental sounding rocket. These are two real-time applications built over the Miosix real-time operating system [17] [18] and running on an STM32-based OBC. The goal of these experiments is to compare REDDI and EDDI for overhead and fault detection capabilities.

A. Testing campaign on benchmarks

We split the set of benchmarks of the ABSURD[2] suite into critical (`ACS`, `ANN`, and `JPG`) and non-critical (`CAN`, `SCL`, and `LAT`). The fault detection campaign consists of 50,000 experiments in which we induced bit-flips on registers, stack, `.data`, and `.bss` sections with uniform distribution relative to their size. The heap is excluded since it is not used by any benchmark. The faults were injected via a debugger.

The fault injections were performed on the program compiled in four configurations: non-protected (`No prot`), protecting the critical global variables with REDDI (`REDDI GV`),

[1]For these experiments, we used ASPIS compiled with version 18.1.8 of the LLVM framework.

[2]Available at `https://github.com/HEAPLab/ABSURD`

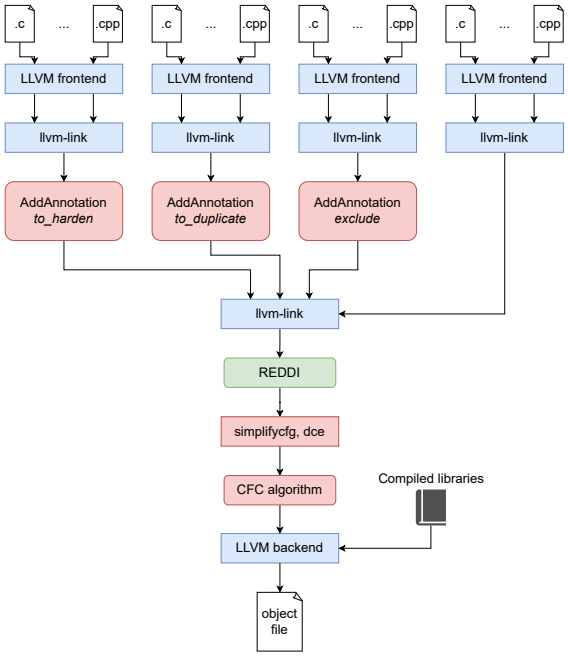

Fig. 1. ASPIS compilation flow

protecting the critical functions with REDDI (`REDDI Fn`), and protecting the whole program with EDDI (`EDDI`).

An injection test can produce no effect (`NE`) or produce an effect on the test (`ERR`). These effects can be further categorized into: the program got stuck in an infinite loop (`Loop`), the board rebooted for a hardware fault (`HF`), REDDI detected a data mismatch (`DCD`), or the data corruption has not been detected (`SDC`). Silent Data Corruptions (SDCs) can be divided into Critical Data Corruptions (CDCs) (`CDC`), where at least one critical benchmark produced an SDC, and non-CDCs (`non-CDC`), where only non-critical benchmarks had SDCs. The results are shown in Table II.

We observe that the number of erroneous runs (ERR) increases when injecting faults into hardened software configurations. This phenomenon can be attributed to the increased memory usage associated with more aggressive protection mechanisms, i.e., when more resources are hardened. Protection techniques inherently expand the memory footprint due to the creation of duplicate variables, thereby increasing the likelihood that a fault injection affects a critical memory location. However, despite the rise in effective SEUs, the rate of SDC and Loop errors consistently decreases with more aggressive protections. This demonstrates that this technique can offer a tunable way of balancing the overhead and the protection offered to the system.

To study the impact on execution time, we measured the run time of each benchmark, for each protection configuration, over 5,000 isolated runs, without injecting any fault. The results showed a general improvement in overhead with respect to EDDI, as reported in Table III. We can notice how, in some cases, EDDI slightly outperformed REDDI. We attribute

TABLE II
INJECTION OUTCOMES PER SOFTWARE CONFIGURATION OF BENCHMARKS.

	NE	ERR	Loop	HF	DCD	SDC	
						CDC	non-CDC
No prot	47482	2518	38	202	0	1965	313
%	-	100	1.51	8.02	0	78.04	12.43
REDDI GV	46467	3533	22	153	1253	1809	296
%	-	100	0.62	4.33	35.47	51.20	8.38
REDDI Fn	44527	5473	16	149	4979	62	267
%	-	100	0.29	2.72	90.97	1.13	4.88
EDDI	42486	7514	11	114	7301	21	67
%	-	100	0.15	1.52	97.17	0.28	0.89

TABLE III
THE MEASURED OVERHEAD IMPARTED BY PROTECTING EACH INDIVIDUAL BENCHMARK WITH DIFFERENT CONFIGURATIONS.

	ACS	ANN	JPG	CAN	SCL	LAT
No prot	1x	1x	1x	1x	1x	1x
REDDI GV	1.54x	2.71x	2.92x	1.46x	3.03x	2.24x
REDDI Fn	1.90x	2.73x	3.01x	3.14x	2.94x	2.21x
EDDI	1.92x	2.72x	2.93x	3.14x	2.94x	2.23x

this increase in overhead to the TAD mechanism introduced by REDDI. Under certain circumstances, this technique could generate heavy `memcpy` instructions that would outweigh the benefits of not duplicating the entire data-flow.

Another testing campaign was aimed at evaluating the impact that fault protection has on the hyperperiod of the whole mixed-criticality system. This campaign comprised 5,000 tests, with the results shown in Fig. 2, where we plot the time overhead of the different techniques compared to the non-protected program. An important consideration about the hyperperiod is that this measurement depends greatly on the heaviest benchmark. In our case, the benchmark `JPG` was one order of magnitude longer to run than the other benchmarks, and hardening this resource led to a substantial increase in the overhead of the whole application.

In the following section, we explore the protection of a lightweight critical resource in a complex system, underlining the true potential of this technique.

B. Testing campaign on a real OBSW

The second test subject is a mixed-criticality OBSW[3] based on Miosix, deployed on the main OBC of the experimental sounding rocket developed by Skyward Experimental Rocketry in 2024 for their flight in European Rocketry Challenge (EuRoC). The task of the system is to reach as precisely as possible 3,000 meters of altitude and then deploy the

[3]Available at:
`https://git.skywarder.eu/avn/swd/obsw/-/tags/lyra-euroc24`

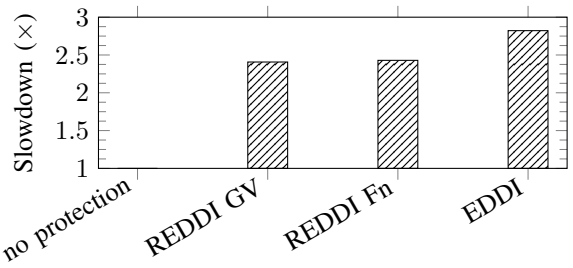

Fig. 2. Comparing the hyperperiod of the mixed-criticality system under different protection configurations.

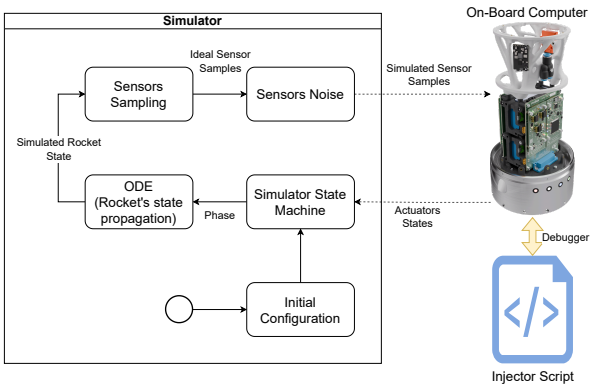

Fig. 3. Scheme of the Hardware-In-the-Loop simulation

recovery system (i.e., parachutes) for a safe descent. The OBSW deploys the parachutes through an expulsion charge when the Apogee Detection Algorithm (ADA) – a Kalman Filter (KF) which uses the barometers to estimate altitude and vertical velocity – detects that the vertical velocity of the rocket approaches zero.

We performed 1,500 tests on an unprotected OBSW and another 1,500 with the ADA recursively protected with REDDI, which represents the critical resource in this mixed-criticality system. The injection is performed on the small but highly critical memory region associated with this algorithm. Each of these tests was performed by running a complete Hardware-In-the-Loop (HIL) test, which reproduces the real flight conditions through a simulator.

Fig. 3 shows the high-level scheme of the HIL simulation. At each simulation step, the simulator propagates the rocket's state, accounting for the current state and the position of the actuators. The propagated rocket's state is then sampled by the sensors modelled on the simulator, which mimic the sampling of the sensors with real frequencies and noise models. These sensor samples are then sent to the OBSW, which will seamlessly use them as the real sensor samples thanks to the OBSW HIL framework. The rest of the OBSW is tested as in a real flight, testing its functional and performance properties. The actuators will be moved as in flight, and their states are sent back to the simulator to close the loop, using this data for the next simulation step.

In Fig. 4 we show that a bit-flip in the critical memory re-

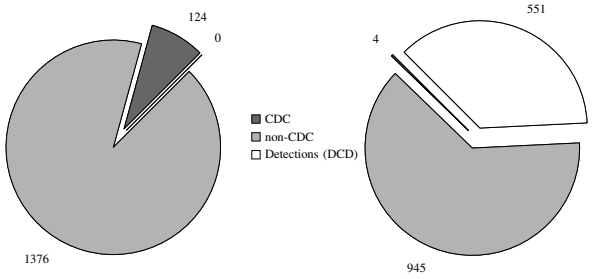

Fig. 4. Results of the fault injection campaign conducted on an unprotected OBSW (left) and on one protected with REDDI (right).

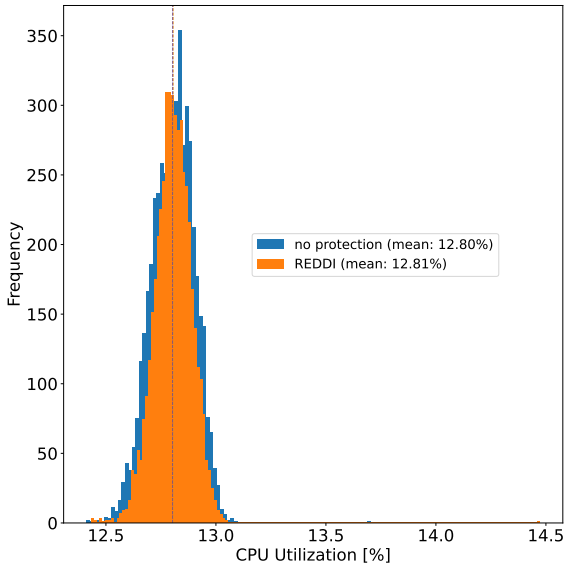

Fig. 5. Comparison of CPU utilization between no protection and REDDI on OBSW.

gion – i.e., the region associated with the ADA – can lead to a failure of the recovery software, which could potentially cause a catastrophic failure of the whole system. In our tests, this case repeated for 8.27% in the non-protected OBSW version, while it was lowered to 0.27% in the protected version, with a Data Corruption Detected (DCD) rate of 36.7%.

To quantify the computational overhead of the protection, we ran a separate campaign with no fault injections. CPU utilization was measured by executing a calibrated busy-wait in the lowest-priority thread. Preemptions by higher-priority tasks increase the observed delay.

We compute the utilization as $U = 100\% \cdot \left(1 - \frac{D_{exp}}{D_{obs}}\right)$. D_{exp} is the delay with no preemption and D_{obs} is the delay measured during execution. During each HIL run, the CPU utilization is measured when the full system – including ADA – is active. We then compare the distributions of the mean utilization across 5,000 runs for the non-protected and REDDI-protected OBSWs. Fig. 5 shows that REDDI's protection negligibly impacts the computational load of the system.

Moreover, the object file size overhead between the non-protected and protected OBSW is negligible, increasing from 1.170 MB for the non-protected case to 1.174 MB when the ADA is recursively hardened. This represents a size overhead

of only $1.0036\times$, negligible compared to most state-of-the-art techniques, which typically incur a minimum overhead of at least $2\times$ [19]. Such a high overhead would render system protection unfeasible in this scenario, as the microcontroller's flash memory is limited to 2 MB.

IV. Limitations and future work

This work can serve as the basis for many enhancements and future studies. The consistency of the output program could be better assured through testing of particular features of the supported programming languages to find cases where the hardened program is inconsistent. Further optimizations of the techniques could also aim at improving the fault-detection/overhead ratio. REDDI can also be the base for future selective recovery techniques for MCS. Moreover, the current version of ASPIS has some known limitations that could be overcome in future versions of the suite. Indirect calls of variadic functions are not well protected since indirect calls are blindly hardened without knowing if the function is variadic or not. Pointer comparison in synchronization points is a difficult task since, from the LLVM version 17, the framework has completely passed to opaque pointers and, for this reason, if a variable is a pointer, it is not explicit which type it points to. For this reason, when it is not trivial to resolve the pointer type, comparisons between pointers are avoided.

V. Conclusions

Grounded in the division of the software into *spheres* (SoR, SoD, SoE, and gray area), and the formalization of their interactions with calling conventions, REDDI provides a solid theoretical basis for a selective and recursive hardening of user-annotated functions or global variables. Beyond this foundation, we enhance C/C++ support for hardening of variadic functions, indirect calls, and global constructors.

By enabling recursive protection of critical resources, REDDI outperforms EDDI in terms of runtime overhead and overall hyperperiod inflation, while maintaining low CDC rates. Across benchmarks, REDDI generally reduces execution-time overhead compared to EDDI, with a few counter-examples explained by TAD costs. On a real OBSW, fault-injection campaigns on the apogee-detection algorithm show a drop in catastrophic failures from 8.27% to 0.27%, with a fault-detection rate (DCD) of 36.7%, demonstrating that selective protection of the critical resource is effective. This is achieved with negligible runtime and memory overhead.

In conclusion, REDDI enhances the applicability of compiler-based fault detection in real-world MCS, offering a tunable trade-off between overhead and fault detection to meet the specific constraints of each application.

Acknowledgment

This work has been supported by the National Resilience and Recovery Plan (PNRR) through the National Center for HPC, Big Data, and Quantum Computing.

References

[1] R. F. Hodson, J. A. Pellish, R. A. Austin, M. J. Campola, R. L. Ladbury, K. A. LaBel, G. R. Allen, R. Gaza, and E. M. Willis, "Avionics radiation hardness assurance (rha) guidelines," National Aeronautics and Space Administration, Tech. Rep., 2021.

[2] S. Reinhardt and S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, June 2000, pp. 25–36.

[3] F. Reghenzani, Z. Guo, L. Santinelli, and W. Fornaciari, "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)*, 2022, pp. 27–39.

[4] A. Burns and R. I. Davis, "Mixed criticality systems-a review:(february 2022)," *Department of Computer Science, University of York, Tech. Rep (2013)*, 2022.

[5] F. Reghenzani and W. Fornaciari, "Towards certifiable software-implemented hardware fault tolerance," in *2024 IEEE 14th International Symposium on Industrial Embedded Systems (SIES)*, 2024, pp. 10–16.

[6] D. Baroffio, F. Reghenzani, and W. Fornaciari, "Enhanced compiler technology for software-based hardware fault detection," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 29, no. 5, Sep. 2024. [Online]. Available: https://doi.org/10.1145/3660524

[7] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[8] ——, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, March 2002.

[9] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random additive signature monitoring for control flow error detection," *IEEE Transactions on Reliability*, vol. 66, no. 4, pp. 1178–1192, Dec 2017.

[10] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[11] N. Oh and E. McCluskey, "Error detection by selective procedure call duplication for low energy consumption," *IEEE Transactions on Reliability*, vol. 51, no. 4, pp. 392–402, 2002.

[12] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," ser. ASPLOS XV. New York, NY, USA: Association for Computing Machinery, 2010, p. 385–396. [Online]. Available: https://doi.org/10.1145/1736020.1736063

[13] D. S. Khudia, G. Wright, and S. Mahlke, "Efficient soft error protection for commodity embedded microprocessors using profile information," in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, ser. LCTES '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 99–108. [Online]. Available: https://doi.org/10.1145/2248418.2248433

[14] B. Arasteh, A. Bouyer, and S. Pirahesh, "An efficient vulnerability-driven method for hardening a program against soft-error using genetic algorithm," *Computers & Electrical Engineering*, vol. 48, pp. 25–43, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045790615003419

[15] A. Abdi, S. A. Asghari, S. Pourmozaffari, H. Taheri, and H. Pedram, "An optimum instruction level method for soft error detection," *International Review on Computers and Software*, vol. 7, no. 2, pp. 637–641, 2012.

[16] V. B. Thati, J. Vankeirsbilck, J. Boydens, and D. Pissort, "Selective duplication and selective comparison for data flow error detection," in *2019 4th International Conference on System Reliability and Safety (ICSRS)*, Nov 2019, pp. 10–15.

[17] M. Maggio, F. Terraneo, and A. Leva, "Task scheduling: A control-theoretical viewpoint for a general and flexible solution," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4, pp. 1–22, 2014.

[18] G. Audrito, F. Terraneo, and W. Fornaciari, "Fcpp+miosix: Scaling aggregate programming to embedded systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 869–880, March 2023.

[19] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "Swift: software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, 2005, pp. 243–254.