Automatic Data Redundancy in Safety-Critical Applications Using Trait-Based Code Transformation

Automatic Data Redundancy in Safety-Critical Applications Using Trait-Based Code Transformation

Mohammadreza Amel Solouki*, Corrado De Sio*, Maurizio Rebaudengo* and Jacopo Sini*
Department of Control and Computer Engineering, Politecnico di Torino
{name.lastname@polito.it}

***Abstract*—The paper describes a systematic approach for the automatic introduction of data redundancy in a safety-critical application. The transformations aim to make the program capable of detecting potential soft errors caused by transient faults that may alter the program's data. The approach is based on an automated system that leverages the polymorphism features offered by the Rust programming language. Rust is focused on speed, safety, and concurrency, and it also provides powerful high-level abstractions at zero cost. The paper proposes a technique for hardening source code by creating redundant code through an automatic data redundancy system, achieved by defining a trait that enables data duplication and the detection of potential errors. The use of a trait allows the compiler to generate optimized code, without introducing abstraction-related overhead and with minimal programmer effort for boilerplate code generation. Preliminary experimental results are presented, showing the fault coverage achieved by the method, as well as some data related to the performance overhead and the increase in code size.**

***Index Terms*—Software Redundancy, Hardening Source Code, Code Transformation, Transient Faults**

I. Introduction

The rapid proliferation of safety-critical electronic systems, from drive-by-wire automotive controllers to wearable medical devices, has amplified concern over soft and radiation-induced faults in modern semiconductor technology. In embedded processors where cost and power budgets preclude extensive hardware redundancy, Software-Implemented Hardware Fault Tolerance (SIHFT) remains an attractive line of defense [1]. However, classic SIHFT schemes duplicate every instruction, basic block, or entire program and compare their results. Although such full duplication achieves near-ideal fault coverage, it typically inflates the data segment by a factor of two or more, more than doubles execution time, and constrains the coding style through rigid transformation rules, issues that conflict with tight latency and memory envelopes common in embedded workloads [2], [3]. Empirical studies show that not all variables contribute equally to system-level failure semantics: those that govern control-flow decisions or propagate widely through data-flow graphs are far more critical than transient temporaries [4], [5]. This observation motivates a selective strategy that concentrates redundancy on the truly sensitive ones while leaving benigns unguarded. Achieving this balance demands a programming environment capable of precise alias analysis, low-overhead generics, and compile-time enforcement of memory safety, properties that the Rust language provides as first-class features. This paper presents a methodology for selective data hardening exploiting Rust programming language. A static analysis pass ranks program variables according to lifetime, fan-out, and involvement in branch predicates, identifying the subset whose corruption would most likely escalate to silent data corruption. Those variables are transparently wrapped in a lightweight generic wrapper, `Hardened<T>`, that maintains a pair of redundant copies, applies double-write semantics on updates, and performs coherence checks on reads. The proposal leverages Rust's trait system to overload arithmetic, comparison, and indexing operators, ensuring seamless integration with existing code while preserving the language's strict aliasing and ownership guarantees. Fault detection capabilities have been validated by performing fault injection campaigns. This paper makes two concrete contributions. (i) It introduces `Hardened<T>`, a generic Rust wrapper that transparently enforces the duplicate-on-write and comparison-on-read rules; (ii) It reports the first empirical evidence that targeted hardening can detect register-level data corruptions in benchmark application.

The remainder of this paper is organized as follows. Sec. II surveys prior SIHFT; Sec. III explains our trait-based scheme; Sec. IV outlines platform and fault-injection setup; Sec. V reports coverage and overhead; Sec. VI draws conclusions.

II. Background

We first sketch a compact taxonomy of SIHFT techniques, survey key works, and point out the Rust features that make low-cost selective variables duplication possible.

A. Background on Software-Implemented Data Hardening

Transient and permanent faults that alter data values remain a dominant failure mechanism in highly scaled VLSI and nanotechnology systems. When hardware protections (e.g., ECC memories) are insufficient or unavailable, software-implemented data hardening introduces deliberate

*All authors contributed equally to this work.

979-8-3315-1489-1/25/$31.00 ©2025 IEEE

redundancy to detect (and optionally tolerate) corruption at run time [6]. Two design axes characterize most methods: duplication granularity, indicating how many times computation is repeated, and check latency/timing, indicating how soon and often the replicas are compared. Concerning the granularity spectrum, four increasingly coarse options are common [2]: (i) instruction-level duplication, repeating each elementary operation and checking immediately; (ii) duplication of basic or super-blocks, reading out the block's live-out set once per copy; (iii) procedure-level duplication, calling a function twice and comparing the two return values; and (iv) whole program time redundancy, rerunning a diversified variant on the same processor and comparing outputs at checkpoints or at completion. Moving rightward decreases overheads in terms of execution time and energy consumpion but increases error-detection latency. Considering *data hardening instruction-level rules*, a minimal set of three high-level transformation [7] is the following:

1) Replicate every variable x as (x_0, x_1).
2) Mirror each write operation.
3) Assert $x_0 == x_1$ after every read, including branch predicates.

The detection latency is bounded by the interval between a fault occurs, affecting the variable, and the next read operation.

It is possible to selectively apply hardening to safety-critical data with different granularities, as described in the following.

Repeating only selected calls reduces dynamic comparisons, code size, cache traffic, and thus energy dissipation [8]. The worst-case latency equals twice the procedure runtime plus the comparison, suiting workloads whose error rate is moderate and power budget is tight.

Virtual duplex systems [9] execute, serially or in interleaved slices, two independently compiled or otherwise diversified variants; disagreement signals a fault.

Data diversity further widens diagnostic coverage by scaling every constant in one variant by a non-zero factor k and verifying $x_1 == k \cdot x_0$.

Application-specific invariants (range checks, conservation laws, monotonic counters) guard semantic correctness beyond replica agreement, catching errors that manifest as control-flow deviations or silent data corruption [10].

No single technique dominates across all metrics. Fine-grained duplication maximises coverage and minimises latency but may violate timing or energy budgets; coarser schemes invert that trade-off.

B. Related Works

This subsection situates the design axes of II-A within the literature.

Early software techniques such as EDDI [2] and SWIFT [7] duplicate every arithmetic instruction and compare the results. Although coverage is near-ideal, the minimum 2× run-time and code-size inflation (without considering the instructions to check that the two copies are identical) conflict with embedded budgets. Procedure- or program-level replay reduces dynamic checks but increases detection latency [8], [9].

Data diversity and executable assertions catch violations of application invariants [10]; these are orthogonal and could be layered atop our wrapper. Fan-out and lifetime-based ranking has been explored for C binaries [4], [5]. We borrow that idea but enforce redundancy in safe Rust.

Schueller et al. survey Rust safety mechanisms [11]; to our knowledge, this paper is the first to demonstrate trait-based, zero-cost data duplication in Rust.

C. Rust at a Glance

Rust is a statically typed systems language whose defining feature is *ownership*: every value has exactly one owner, and the compiler checks borrows and lifetimes to prevent dangling pointers, data races, and out-of-bounds accesses *at compile time* [12], [13]. Running on LLVM, it routinely matches or outperforms C/C++ on typical embedded workloads (e.g., 26–50 % faster Dijkstra and 43–78 % faster LLL in controlled studies [14], [15]). Based on recent surveys of SIHFT in Rust [11], we highlight five properties that make Rust a natural host for selective data hardening: *Compile-time memory safety* (ownership, lifetimes) removes dangling pointers, races, and out-of-bounds faults [13]. *Zero-cost & polymorphic abstractions.* Traits and generics give C-like binaries while letting us implement redundancy once for `Hardened<T>` and reuse it for *any* T which supports the traits. *Deterministic failure containment.* A well-defined `panic` path, or a custom `no_std` handler, converts any detected incoherence into a controlled task abort. *Concurrency primitives checked at compile time.* Channels, `async/await`, and lock-free atomics carry the same safety guarantees.introducing data races. *Toolchain leverage and migration path.* LLVM preserves inlined redundancy code, and the FFI enables incremental integration with legacy C modules and vendor HALs [16], [17]. These facets minimise the trusted computing base and the energy overhead traditionally associated with SIHFT. Empirical studies already report C-competitive throughput and ongoing efforts toward ISO-26262 certification; those topics are outside this paper's scope but reinforce Rust's suitability for safety-critical deployment.

III. Proposed method

We assume that each program variable v has already been assigned a criticality score $\kappa(v) = w_{\text{fan}} \cdot \text{fanout}(v) + w_{\text{life}} \cdot \text{lifetime}(v) + w_{\text{ctrl}} \cdot \text{cflow}(v)$, which combines fan-out, live-range duration, and control-flow influence. Variables whose $\kappa(v)$ exceeds a designer-chosen threshold form the set $\mathcal{V}_{\text{prot}}$ that we protect with data redundancy.

A. Transformation rules

Our goal is to obtain a method capable of following the classical trio of duplicate-on-write rules. (**R1**) duplicate each variable into two copies (cp_1, cp_2) at initialization. (**R2**) mirror every store on both copies. (**R3**) compare cp_1 and cp_2 on every read and raise an error on mismatch. The rules above are implemented once, generically, via a Rust wrapper that encapsulates the twin copies and overloads the operators used

by the original code. Listing 1 shows the *entire* wrapper; all additional traits (`Sub`, `Index`, ...) follow the same pattern and are available in our repository [18].

```
#[derive(Clone,Copy)]
pub struct Hardened<T>(T,T);                    //
    (R1)
impl<T: Copy + PartialEq +
         core::ops::Add<Output=T>> core::ops::
    Add for Hardened<T>{
    type Output = Self;
    fn add(self, rhs: Self) -> Self {           //
     (R2)+(R3)
        assert!(self.0 == self.1 && rhs.0 ==
    rhs.1);
        Hardened(self.0 + rhs.0, self.1 + rhs
    .1)
    }
}
```

Listing 1: Minimal Hardened wrapper

The `assert!` macro inlines to a single conditional branch, so the wrapper incurs no abstraction penalty; memory overhead is exactly one extra copy per protected variable.

IV. Experimental setup

We assess reliability and overhead through fault-injection experiments, detailing the benchmarks, fault model, and tools.

A. Benchmark Programs

We evaluate the proposed hardening strategy on the compute kernel that exercises the data-access patterns, based on the *Bubble Sort* algorithm, where adjacent comparisons with an early-exit flag, intensify loop-carried data dependencies and conditional swaps. Because redundancy is encapsulated inside the wrapper, the hardened versions differ from the baseline only in a handful of syntactic locations. The present paper reports preliminary figures for *Bubble Sort*, chosen because its data-dependent control flow stresses the proposed redundancy rules.

B. Target platform and compiler settings

All experiments run on a RISC-V RV32IMAC core, instruction-accurately emulated with QEMU [19]. We resorted to the RV32IMAC Instruction Set Architecture (ISA) since the *atomic* (A) extension is needed for implementing the traits of the core library, and the only valid target for the Rust toolchains with the availability of the A extension also features the integer multiplication and division (M) and the compressed instructions (C) ones. The fault-injection campaign was carried out with the Fault Injection Manager (FIM) framework [20]. We first generated the default classifier that FIM derives from the campaing settings and then refined it manually to capture the Rust-specific control-flow patterns [21] of our benchmark. All binaries are compiled for the `riscv32imac-unknown-none-elf` target. We choose `opt-level=0` for guaranteeing that the hardening instructions are fully implemented in the compiled application: at this optimization level, the LLVM mid-end disables dead-store elimination, global value numbering, and loop-invariant code motion, although borrow checking and MIR safety passes still run, so Rust's memory-safety guarantees are preserved. Optimization passes such as `DeadStoreElimination` or `GVN` could remove 'redundant' writes or lift checks outside of loops, thus altering both execution time overhead and the fault detection latency we intend to measure.

C. Fault models

For the fault models, we simulated a failure that impacts a single bit within one of the registers of the affected core. Once the fault is injected, the bit remains fixed at either 0 or 1. Both the injection time and the bit's final state are generated randomly.

D. Injection outcomes classifications

The outcomes of the individual fault injections were categorized into the following classes:

1) **Latent**: The fault was not detected and did not alter the observable behavior of the application.
2) **Detected by traits**: the fault was detected via the trait-based mechanism (i.e., the global variable `ERR_CODE` $\neq 0$) and resulted in a deviation in the behavior.
3) **Safe**: the fault was detected by the trait-based mechanism (`ERR_CODE` $\neq 0$), but it did not cause any observable change in the behavior.
4) **Detected by panic handler**: the fault was detected by the built-in panic handler of Rust.
5) **Residual**: the fault was not detected and led to a change in the behavior of the application. To evaluate the impact of the allotted detection latency, the outcomes were analyzed using five different instruction latency thresholds.

V. Experimental results

We compiled the benchmark and found that a small subset of registers—`sp`, `s0`, and the argument/result registers `a0`–`a3` account for over 70% of dynamic usage; these were therefore chosen as fault-injection targets. We selected specific registers for fault injection: 200 faults were injected into each of the function argument registers `a1` and `a2`. Additionally, we injected 500 faults into the return value register `a0`. Furthermore, although data hardening does not explicitly address control-flow elements, we conducted additional fault injections to assess flow disruption effects by injecting 500 faults into the program counter (`pc`). The results of these injections are presented in Table I. As shown in the table, the hardened implementation is particularly effective in detecting faults affecting the `a0` register, which primarily stores computational results. For this register, with a maximum allowed latency of 1000 machine instructions, all injected (and non-latent) faults were detected, leaving no residual errors. Approximately 75% of these detections are attributable to the hardening strategy, while the remaining 25% are due to Rust's built-in panic handling mechanism. In contrast, faults affecting the program counter (`pc`) result in execution flow disruption, which is more

TABLE I: Injection outcomes; Lat. = latent, Trt. = detected by traits, Pan. = detected by panic; Res. = residual

Latency	pc					a0					a1					a2				
	Lat.	Trt.	Safe	Pan.	Res.	Lat.	Trt.	Safe	Pan.	Res.	Lat.	Trt.	Safe	Pan.	Res.	Lat.	Trt.	Safe	Pan.	Res.
10	495	0	4	1	0	479	0	9	0	12	198	0	0	0	2	197	0	0	0	3
50	493	0	5	2	0	413	4	75	0	8	198	0	0	0	2	197	0	0	0	3
100	491	0	5	4	0	325	5	115	51	4	198	0	0	0	2	197	0	0	0	3
1 k	490	0	5	5	0	260	7	155	78	0	198	0	0	0	2	197	0	0	0	3
10 k	490	0	5	5	0	254	7	159	80	0	198	0	0	0	2	197	0	0	0	3

challenging to detect with the proposed approach. Detection in these cases typically occurs only when the disruption leads to divergent copies and allows the checks to continue executing. Some faults are still caught by Rust's panic handler. Faults injected into the function argument registers `a1` and `a2` are often undetected due to the application's limited use of registers and the compiler's tendency to use these registers for memory addressing in load and store operations: compressed instructions are optimized for a small set of registers including these two, causing these faults to mainly impact read/write operations addresses rather than the actual contents of the hardened variables. Regarding performance overhead, the non-hardened version of the application completes execution—using an array of 10 elements—with 35,197 executed instructions and a memory footprint (sum of `.init`, `.text`, `.rodata`, `.eh_frame`, and `.sbss` sections) of 8,126 bytes. In contrast, the hardened version requires 91,467 instructions to execute (+159%) and occupies 12,454 bytes of memory (+53%). Overall, the hardened binaries detection capabilities is $\approx 50\%$ for faults affecting the return address register `a0`, while the injections in the other registers ended up with latents. This is due to the simplicity of the considered application. For these reasons, more investigations on more complex benchmarks, in terms of register usage footprint, are needed.

VI. Conclusions

In this preliminary paper, we introduced a selective, trait-based data-hardening scheme that leverages Rust's compile-time safety and generics. By wrapping only critical variables in the generic container `Hardened<T>`, the method duplicates data, mirrors writes, and checks reads while preserving algorithmic structure and incurring minimal run-time overhead. A first fault-injection campaign on a RISC-V RV32IMAC target showed complete detection of register-level data corruptions in a *Bubble Sort* kernel, with few residual errors when the allowed detection latency was 1,000 instructions. Memory occupation growth is less than 2×; timing overhead remained less than 3×, considering that all the compiler optimizations are disabled. Future work will (i) extend the benchmark suite to matrix and streaming kernels, (ii) evaluate memory-bit upsets, and (iii) investigate compiler optimizations' effects on the detection coverage and overheads. These steps aim to mature the approach into a deployable component of software-implemented fault tolerance for safety-certifiable systems.

References

[1] M. A. Solouki *et al.*, "Dependability in embedded systems: A survey of fault tolerance methods and software-based mitigation techniques," *IEEE Access*, vol. 12, pp. 180 939–180 967, 2024.

[2] O. Goloubeva *et al.*, *Software-implemented hardware fault tolerance*. Springer Science & Business Media, 2006.

[3] V. B. Thati *et al.*, "An improved data error detection technique for dependable embedded software," in *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2018, pp. 213–220.

[4] T. Wüchner *et al.*, "Robust and effective malware detection through quantitative data flow graph metrics," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings 12*. Springer, 2015, pp. 98–118.

[5] M. Cinque *et al.*, "An empirical analysis of error propagation in critical software systems," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2450–2484, 2020.

[6] B. Kaci *et al.*, "A fault tolerant architecture for data fusion targeting hardware and software faults," in *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing*. IEEE, 2014, pp. 1–10.

[7] L. Parra *et al.*, "A new hybrid nonintrusive error-detection technique using dual control-flow monitoring," *IEEE Transactions on Nuclear Science*, vol. 61, no. 6, pp. 3236–3243, 2014.

[8] V. Sridharan *et al.*, "Reducing data cache susceptibility to soft errors," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 4, pp. 353–364, 2006.

[9] K. Pattabiraman *et al.*, "Symplfied: Symbolic program-level fault injection and error detection framework," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 472–481.

[10] M. Hiller, "Executable assertions for detecting data errors in embedded control systems," in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*. IEEE, 2000, pp. 24–33.

[11] W. Schueller *et al.*, "Evolving collaboration, dependencies, and use in the rust open source software ecosystem," *Scientific Data*, vol. 9, no. 1, p. 703, 2022.

[12] S. Klabnik and C. Nichols, *The Rust programming language*. No Starch Press, 2023.

[13] A. Balasubramanian *et al.*, "System programming in rust: Beyond safety," in *Proceedings of the 16th workshop on hot topics in operating systems*, 2017, pp. 156–161.

[14] F. Wilkens, "Evaluation of performance and productivity metrics of potential programming languages in the hpc environment," Ph.D. dissertation, 2015.

[15] A. Perez, "Rust and c++ performance on the algorithmic lovasz local lemma," *Project Report. Stanford: Stanford University, Dec*, 2017.

[16] O. K. A. Santoso *et al.*, "Rust's memory safety model: An evaluation of its effectiveness in preventing common vulnerabilities," *Procedia Computer Science*, vol. 227, pp. 119–127, 2023.

[17] M. Sudwoj, "Rust programming language in the high-performance computing environment," B.S. thesis, ETH Zurich, 2020.

[18] Rust4Safety_DFTS2025: artifact and source code. (accessed 27 Jul. 2025). [Online]. Available: https://github.com/JacopoSini/Rust4Safety_DFTS2025

[19] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41, no. 46. California, USA, 2005, pp. 10–5555.

[20] J. Sini *et al.*, "A novel iso 26262-compliant test bench to assess the diagnostic coverage of software hardening techniques against digital components random hardware failures," *Electronics*, vol. 11, no. 6, p. 901, 2022.

[21] J. Sini, M. A. Solouki, M. Violante, and G. Di Natale, "Improving software reliability with rust: Implementation for enhanced control flow checking methods," in *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2025, pp. 1–7.