Divergent Multi-Version Execution (DME): Canonical Instruction-Trace Based Fault Detection via Structural Address-Space Decorrelation

Petro Baran
Independent Researcher
zoshytlogic@gmail.com
Uzhgorod, Februare 2026

Abstract

Redundancy-based fault tolerance traditionally executes identical binaries with identical memory layouts. Although effective against many transient faults, such uniformity leaves systems vulnerable to correlated control-flow errors: a single perturbation of the program counter may redirect all replicas along the same incorrect path, producing silent data corruption.

This paper introduces *Divergent Multi-Version Execution (DME)*, a software-only execution model that enforces structural address-space diversity while preserving identical instruction semantics. Instead of comparing physical addresses or full architectural state, DME compares *canonical instruction traces* that incorporate opcodes, register operands, immediate values, *and instruction results* (loaded values, computation outputs, and stored data). Per-instruction hashing of these comprehensive layout-independent traces converts any logical or data divergence into an immediately detectable mismatch with latency bounded by one instruction.

We formalize the canonical trace model, derive probabilistic bounds on hash-collision failure and correlated control-flow aliasing under decorrelated layouts, and demonstrate an implementation on commodity 32-bit microcontrollers. Results show that deterministic, low-latency fault detection is achievable in software with minimal memory overhead, making DME practical for resource-constrained safety-critical systems.

Fundamental principle: *under fault-free execution, all replicas produce identical canonical traces; under any fault, structural decorrelation guarantees maximally divergent outcomes.*

1 Introduction

Safety-critical and industrial control systems rely heavily on redundancy to detect or mask faults. Hardware lockstep, Triple Modular Redundancy (TMR), and N-version programming execute multiple replicas of a program and compare their results. These techniques assume that identical execution contexts imply identical correctness.

However, most redundancy mechanisms preserve identical code and data layouts. Deterministic control-flow faults—for example, corruption of the program counter or branch target—may therefore redirect all replicas to the same incorrect location. Because execution remains mutually consistent, such faults can escape detection and produce silent data corruption.

This observation motivates treating *address-space structure* as an independent dimension of diversity. Rather than replicating identical layouts, replicas may preserve logical semantics while deliberately differing in their physical placement of code and data.

We introduce *Divergent Multi-Version Execution (DME)*, a software-only execution model based on three principles:

1. independent compilation to decorrelate address layouts,

2. deterministic lockstep execution across replicas, and

3. comparison of canonical instruction traces rather than raw state.

The key insight is that program correctness is defined by the *logical sequence of executed instructions*. Physical addresses are merely implementation artifacts. By canonicalizing instructions to remove layout-dependent information, replicas can be compared at the semantic level.

DME therefore occupies a design point between hardware lockstep and N-version programming: replicas share identical semantics but differ structurally, achieving diversity without multiple independent software implementations.

2 System Model

Algorithm 1 Per-instruction synchronized DME execution

```
for each logical step t do
    for i = 1 to N do
        Fetch_i
        Execute_i
        Canonicalize_i
        Hash_i
    end for
    Compare(H_1, H_2, ..., H_N)
end for
```

We consider $N \geq 2$ replicas of the same source program executing on a shared processor in deterministic interleaved lockstep.

Each replica:

- is compiled independently,
- has private code and data regions,
- preserves identical opcode-level semantics, and
- has an isomorphic control-flow graph.

Thus logical behavior is identical under fault-free execution, while physical addresses differ.

Assumption 2.1 (Replica independence)**.** *Replica-local architectural states experience independent faults.*

Assumption 2.2 (Private address spaces)**.** *Code and data of different replicas do not alias.*

Assumption 2.3 (Common-mode faults)**.** *Clock, power, or global hardware failures may affect all replicas identically and are out of scope.*

2.1 Address-Space Decorrelation

For CFG $G = (V, E)$ and replica n, mapping

$$\phi_n : V \to \mathbb{N}$$

assigns distinct addresses such that

$$\forall n \neq m,\ \phi_n(v) \neq \phi_m(v).$$

2.2 Signed and Distance-Decoupled Control-Flow Placement

To further suppress correlated control-flow aliasing, the compiler decorrelates both the direction and the magnitude of control-transfer displacements. For a transfer instruction located at address a, the logical target block v is mapped as

$$\phi_n(v) = a + \sigma_n \cdot \Delta_n(v), \quad \sigma_n \in \{+1, -1\},$$

where σ_n is a replica-specific polarity and $\Delta_n(v)$ is a replica-specific displacement.

Consequently, logically identical edges are encoded with different signed offsets across replicas. For example, one replica may realize a transfer as $PC+ = 64$, while another realizes the same logical transfer as $PC- = 96$. Both the sign and the magnitude therefore vary between replicas.

This transformation eliminates shared relative-address structure and significantly reduces the probability that identical numerical program-counter perturbations resolve to the same logical block in multiple replicas.

3 Canonical Instruction Trace Model

3.1 Canonicalization

Physical addresses differ between replicas and must not influence comparison. We therefore define a

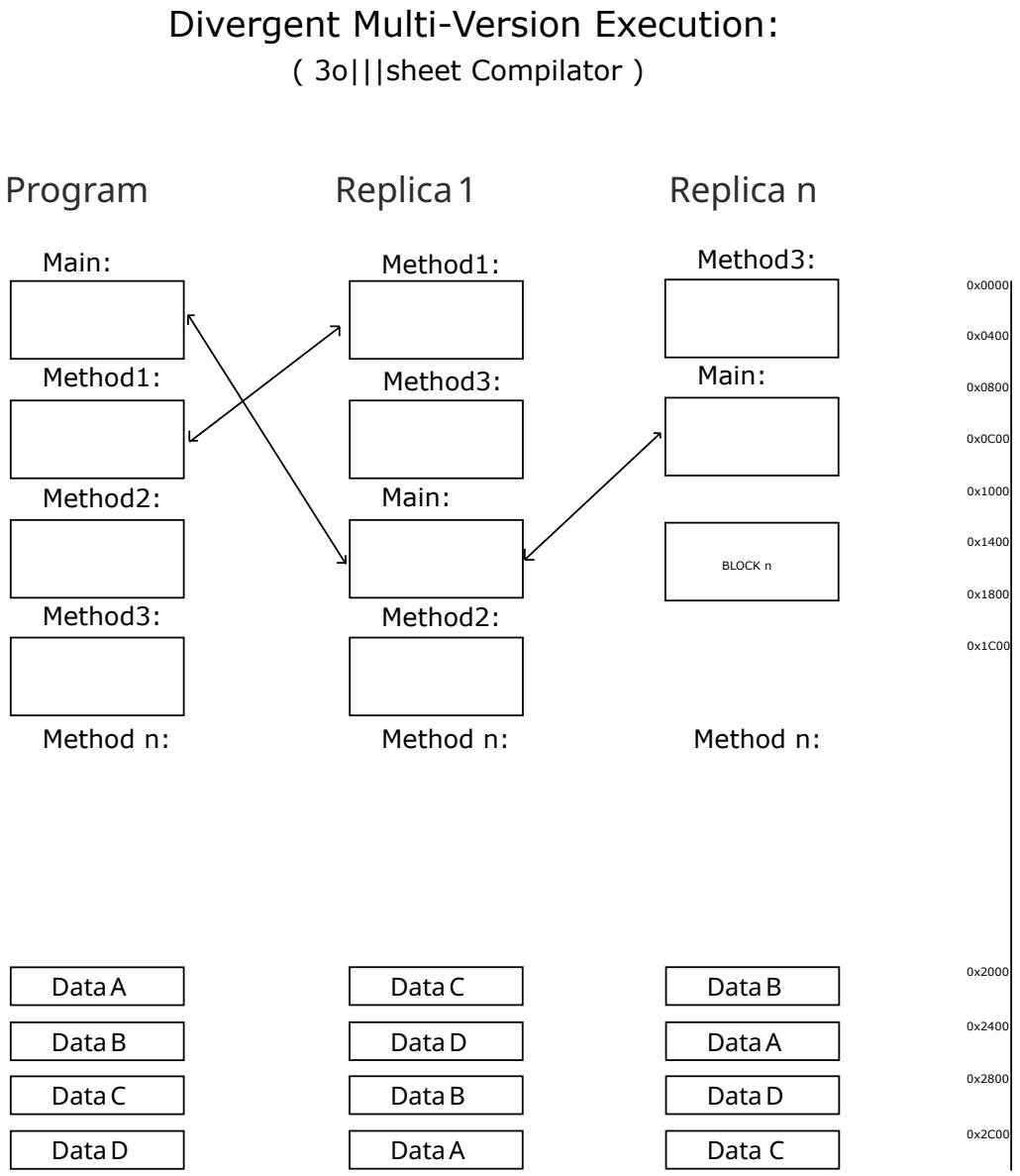

Figure 1: Example memory layout of methods, logical blocks, objects, and variables produced by the 3o|||sheet compiler

layout-independent canonical form that captures the full semantic effect of each instruction.

Definition 3.1 (Canonical instruction)**.** *A func-*

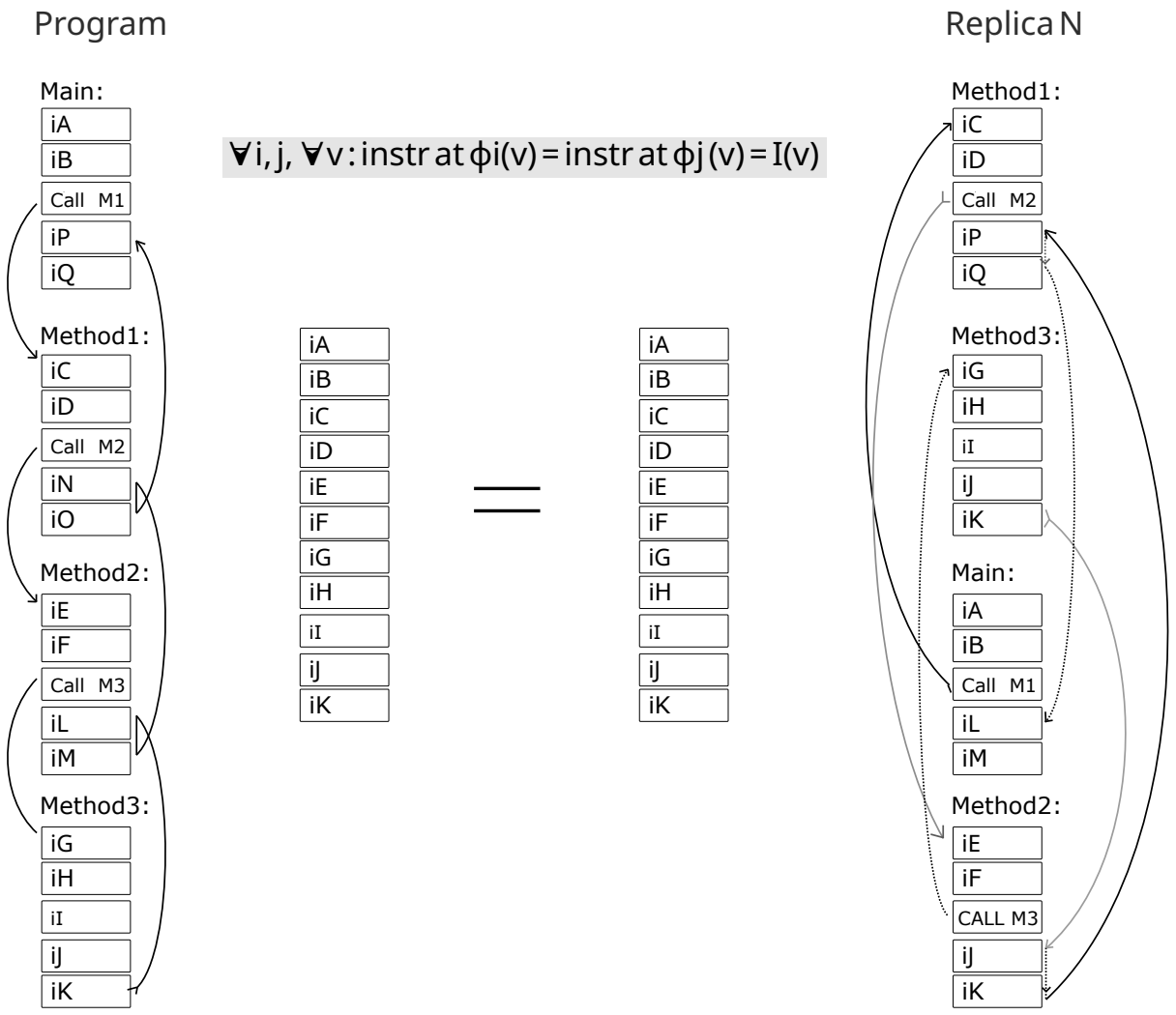

Figure 2: Normal (fault-free) execution. Each replica's methods, objects, and variables are relocated in memory, with recompilation applying new addresses for instruction use during runtime.

tion

$$C(I, s)$$

maps an instruction and the resulting architectural state to a representation containing:

- *opcode and condition codes,*
- *source and destination register identifiers,*
- *immediate values,*
- loaded memory values *(for load instructions),*
- computed results *(for ALU operations),*
- stored values *(for store instructions).*

Absolute or relative addresses are explicitly excluded.

The canonical trace becomes

$$\hat{T}_n = (C(I_n^0, s_n^0), C(I_n^1, s_n^1), C(I_n^2, s_n^2), \dots),$$

where s_n^t denotes the architectural state of replica n after executing instruction I_n^t.

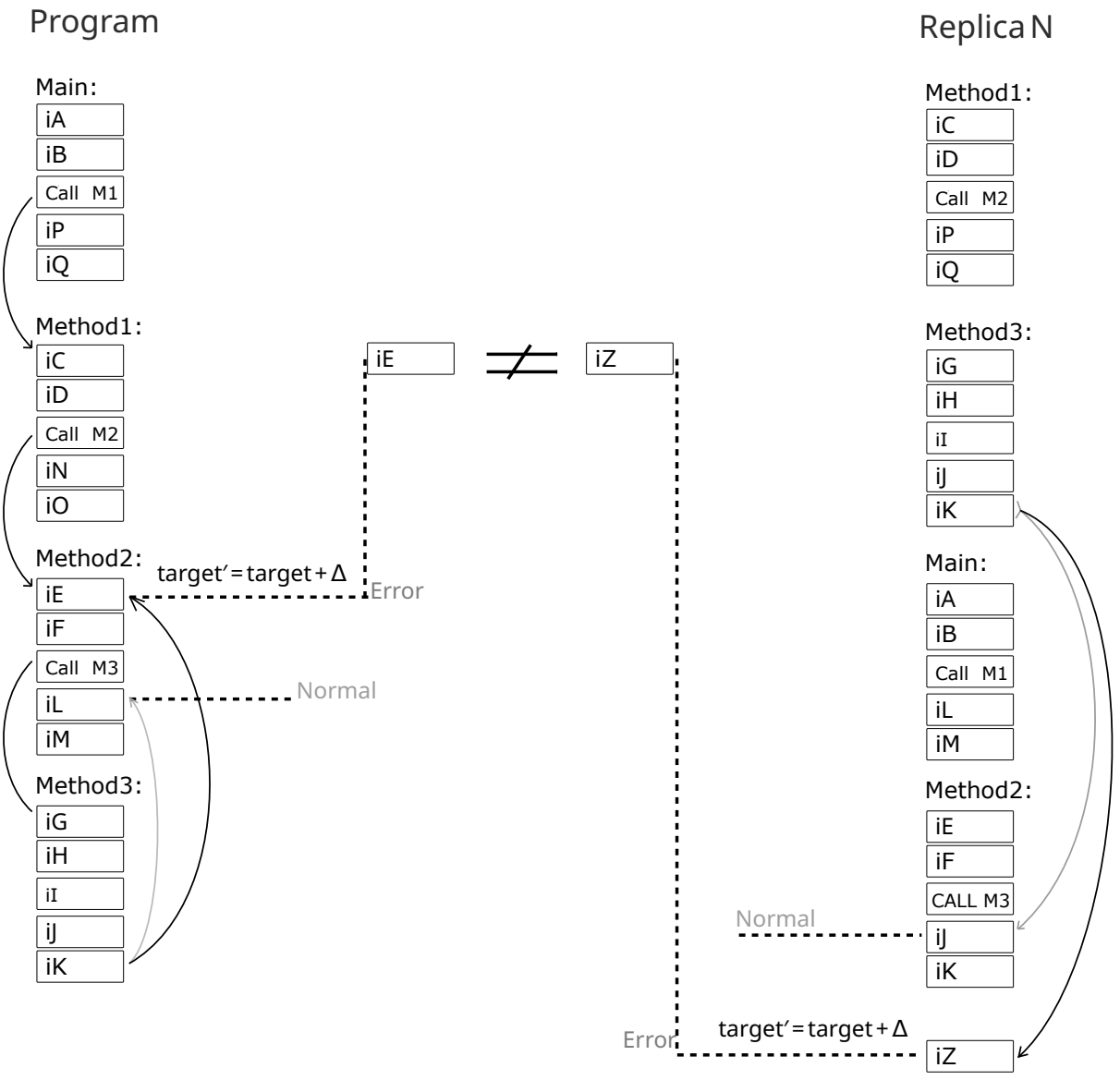

Figure 3: Faulty execution trajectory example. Under normal conditions, logical operations proceed identically and sequentially. An error in the program counter, thanks to address-space decorrelation, causes replicas to diverge to different code regions even with identical numerical perturbations. This divergence is instantly captured via mismatched canonical trace hashes.

Under fault-free execution,

$$\hat{T}_i = \hat{T}_j \quad \forall i, j.$$

4 Trace Hashing and Comparison

Each replica maintains a running hash over its canonical trace, incorporating both the instruction

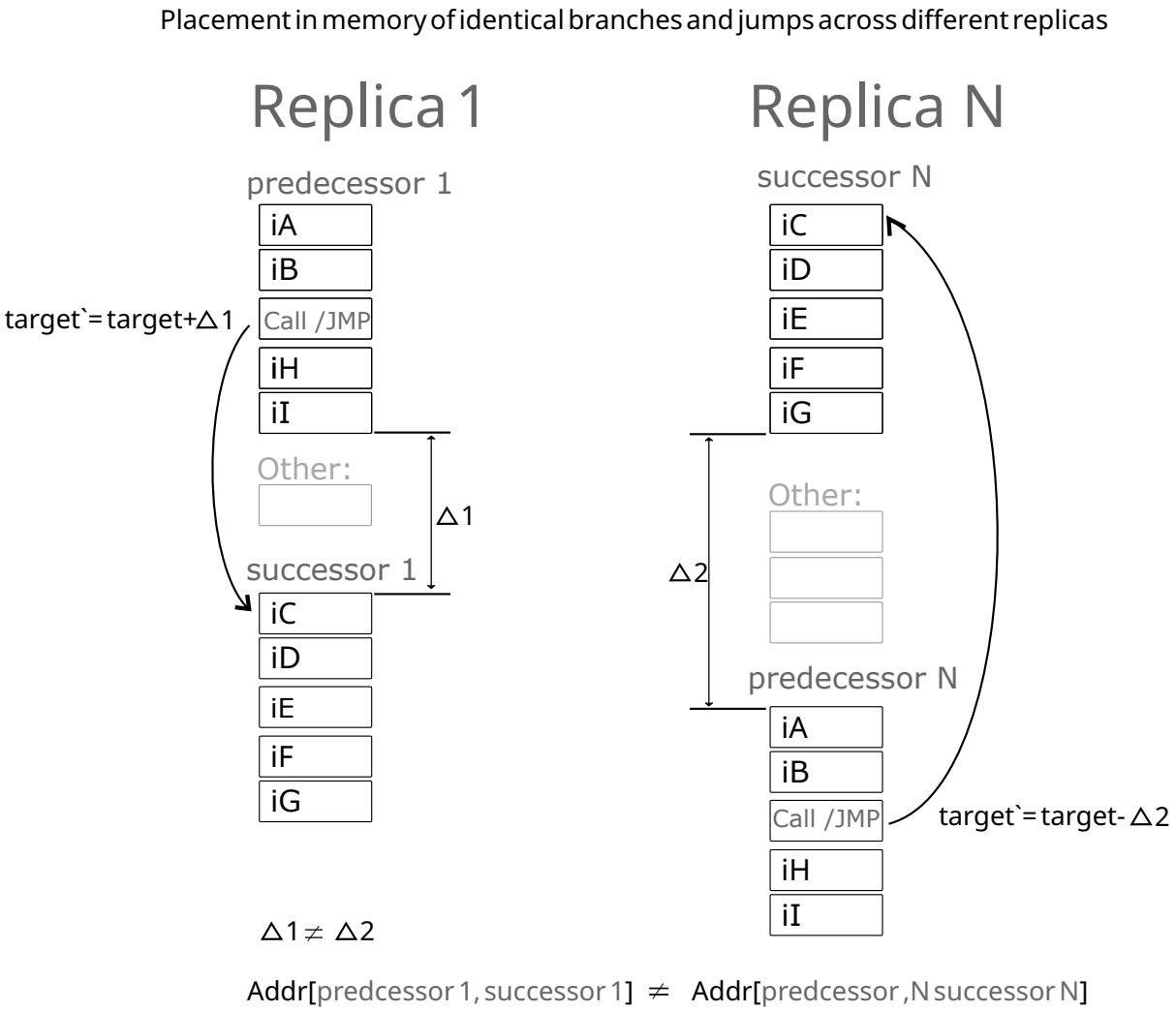

Figure 4: Example of compiler placement of three identical logical blocks and functions in memory. For instance, if a branch in the first replica has a positive offset (PC+ = 64), the same branch in a second replica may have a negative offset (PC− = 96). The delta between branch targets also differs across replicas, ensuring structural address-space diversity.

and its execution results:

$$H_n^{t+1} = F(H_n^t, C(I_n^t, s_n^t)),$$

where F is a deterministic collision-resistant hash function.

The canonical element $C(I_n^t, s_n^t)$ includes:

- instruction opcode and qualifiers,
- register identifiers,
- immediate operands,
- *loaded values* from memory,
- *computation results* written to registers or memory,
- *condition flags* affected by the instruction.

This comprehensive hashing ensures that any fault affecting either the control flow or the data

path produces an immediate mismatch. After every logical instruction, hashes are synchronously compared across replicas.

5 Control-Flow and Data Fault Analysis

Address-space decorrelation ensures that identical numerical program counters map to different logical blocks across replicas. For a given replica set, let:

- L be the number of valid block entry points,
- R the address-space size,
- $\rho = L/R$ the entry density.

The probability that all N replicas alias to the same incorrect block is

$$P_{\text{aliasing}} \leq (\rho)^{N-1}.$$

Thus correlated control-flow aliasing decreases exponentially with the number of replicas. Even if aliasing occurs, matching canonical instructions for k consecutive steps requires identical opcode and operand patterns, with probability decaying exponentially in k.

However, DME also detects data-path faults through execution-result hashing, providing protection beyond control-flow errors.

Theorem 5.1 (Unified fault detection)**.** *For any instruction I that computes a result r or accesses memory:*

1. *If a control-flow fault causes divergence to different logical blocks, the canonical traces diverge immediately.*
2. *If a data-path fault causes $r_i \neq r_j$ across replicas i and j, then $C(I, s_i) \neq C(I, s_j)$ with probability $1-\epsilon$, where ϵ is the hash collision probability.*

Thus DME provides unified detection of both control-flow and data-path faults within a single instruction window, with detection latency bounded by one instruction and failure probability bounded by the hash collision rate.

6 Correctness

Theorem 6.1 (Trace divergence detection)**.** *Under the stated assumptions:*

1. *If $\hat{T}_i^t = \hat{T}_j^t$, then $H_i^t = H_j^t$.*
2. *If $\hat{T}_i^t \neq \hat{T}_j^t$, then $\Pr(H_i^t = H_j^t) \leq \epsilon$.*
3. *Detection latency is at most one instruction.*

Proof. Equality follows from determinism of F. Inequality follows from the hash collision bound. Hashes are compared after every instruction, yielding unit latency. □

7 Implementation

We implemented DME as a lightweight register-based virtual machine executing multiple replicas in interleaved lockstep on ARM Cortex-M microcontrollers. The system uses the proprietary 3o|||sheet Compiler (project details available at `https://zoshytlogic.github.io/`) to generate structurally diverse binaries. Although the tests were performed on the 3o|||sheet virtual machine, this architecture and method are capable of operating at the native level.

Independent compilations randomize code and data placement to ensure structural diversity. Canonicalization is performed during decode, and hashing uses a 64-bit incremental function. The runtime requires approximately 8 KB of RAM for three replicas.

8 Evaluation

The system was tested on the 3o|||sheet virtual machine executing LD and ST instructions typical for industrial controller applications. Deliberate faults were injected into the program counter, return addresses, and data values to test for silent data corruption. Detection latency depends on the type of operation and specific instruction.

Tests were performed on a low-power STM32G030 microcontroller:

- Control-flow faults were detected within **0.5 microseconds**.

- Data corruption was detected within **2–13 microseconds**, depending on the instruction type and memory access patterns.

The longest detection latencies occur for instructions that read and write multiple values from and to memory, such as multi-variable operations like `var1 = varA = varB`. The fastest instructions are single load or store operations of one memory variable. These results demonstrate that canonical trace comparison is practical even on resource-constrained embedded platforms, with detection latency varying by fault type but remaining within microseconds for all tested scenarios.

9 Related Work

Hardware lockstep and TMR compare architectural state but preserve identical layouts, leaving them susceptible to correlated control-flow faults. Software duplication techniques increase instruction count substantially, while control-flow integrity methods protect only specific classes of errors. DME differs by introducing structural diversity and comparing semantic instruction trajectories entirely in software.

10 Limitations

DME predicts errors and physical faults only if they affect the program execution flow or cause data corruption. Systematic design bugs in the software itself remain undetectable if they do not affect the program execution flow and do not corrupt data. The method does not protect against fully correlated physical failures that simultaneously affect all replicas (e.g., power supply glitches). Memory overhead scales linearly with the number of replicas, and correctness relies on compiler transformations preserving semantic equivalence.

11 Conclusion

Divergent Multi-Version Execution reframes redundancy from comparing physical state to comparing logical execution trajectories. By decorrelating address layouts and hashing canonical instruction traces, DME detects divergence within a single instruction without specialized hardware.

This approach provides a practical, formally analyzable, and low-overhead mechanism for fault detection in embedded safety-critical systems. Future work includes native implementation on bare-metal targets, evaluation under broader fault models, and exploration of DME in multi-core execution environments.

References

[1] S. Manoni, E. Parisi, R. Tedeschi, D. Rossi, A. Acquaviva, A. Bartolini. CVA6-CFI: A First Glance at RISC-V Control-Flow Integrity Extensions. *arXiv preprint arXiv:2602.04991*, February 2026.

[2] J. von Neumann. Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components. *Automata Studies*, pages 43–98, Princeton University Press, 1956.

[3] Tomislav Lovrić. Error detection by systematic diversity in design diversified time redundant computer systems and their evaluation using error injection. Ph.D. Dissertation, University of Essen, Germany, 1996.

[4] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[5] J. C. Laprie, C. Béounes, and K. Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *IEEE Computer*, 23(7):39–51, 1990.

[6] Improved address space layout randomization. *Google Patents DE112017002277T5*, 2017.

[7] Jakub Breier, Štefan Kučerák, Xiaolu Hou. EMMap: A Systematic Framework for Spatial EMFI Mapping and Fault Classification on Microcontrollers. *arXiv preprint arXiv:2602.16309*, February 2026.

[8] Adam Caulfield, Muhammad Wasif Kamran, N. Asokan. Resolving Availability

Table 1: Comparison of fault-detection approaches in redundant and software-based protection systems

Criterion	HW Lockstep	TMR	CFI (SW)	SWIFT / EDDI	ILR	DME
Software-only	✗	✗	✓	✓	✓	✓
Control-flow protection	✓	✓	✓	★	✓	✓
Data-path protection	✓	✓	✗	✓	✓	✓
Structural decorrelation	✗	✗	✗	✗	✗	✓
Correlated fault mitigation	✗	✗	✗	✗	✗	✓
Detection latency	1 cycle	N cycles	1 instr.	1–2 instr.	N instr.	1 instr.
RAM overhead	low	high	low	medium	medium	linear
CPU overhead	∼0%	∼200%	∼5–15%	∼80–120%	∼50%	linear in N

and Run-time Integrity Conflicts in Real-Time Embedded Systems. *arXiv preprint arXiv:2511.14088*, November 2025.

[9] Nan Chen, Xiaotian Dai, Tong Cheng, Alan Burns, Iain Bate, Shuai Zhao. LEFT-RS: A Lock-Free Fault-Tolerant Resource Sharing Protocol for Multicore Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS 2025)*, December 2025.

[10] Arsalan Ali Malik, Harshvadan Mihir, Aydin Aysu. Honest to a Fault: Root-Causing Fault Attacks with Pre-Silicon RISC Pipeline Characterization. *arXiv preprint arXiv:2503.04846*, March 2025.

[11] Arsalan Ali Malik, Harshvadan Mihir, Aydin Aysu. CRAFT: Characterizing and Root-Causing Fault Injection Threats at Pre-Silicon. *arXiv preprint arXiv:2503.03877*, March 2025 (revised October 2025).

[12] Roozbeh Siyadatzadeh, Mohsen Ansari, Muhammad Shafique, Alireza Ejlali. RL-TIME: Reinforcement Learning-based Task Replication in Multicore Embedded Systems. *arXiv preprint arXiv:2503.12677*, March 2025.

[13] Nishant Chinnasami, Rye Stahle-Smith, Rasha Karakchi. ML-Enhanced AES Anomaly Detection for Real-Time Embedded Security. *arXiv preprint arXiv:2507.04197*, July 2025.

[14] Abhishek Tyagi, Charu Gaur. SLAM-Based Navigation and Fault Resilience in a Surveillance Quadcopter with Embedded Vision Systems. *arXiv preprint arXiv:2504.15305*, April 2025.

[15] David Jannis Schmidt, Grigory Fridman, Florian von Zabiensky. Offloading tracing for real-time systems using a scalable cloud infrastructure. In *Proceedings of the ECRTS 2025 RT-Cloud Workshop*, July 2025.

[16] Patrik Velčický, Jakub Breier, Mladen Kovačević, Xiaolu Hou. DeepNcode: Encoding-Based Protection against Bit-Flip Attacks on Neural Networks. *arXiv preprint arXiv:2405.13891*, May 2024.

[17] Bing Xue, Mark Zwolinski. Using Formal Verification to Evaluate Single Event Upsets in a RISC-V Core. *arXiv preprint arXiv:2405.12089*, May 2024.

[18] Mohammadreza Amel Solouki, Shaahin Angizi, Massimo Violante. Dependability in Embedded Systems: A Survey of Fault Tolerance Methods and Software-Based Mitigation Techniques. *arXiv preprint arXiv:2404.10509*, April 2024.

[19] Abhishek Tyagi, Reiley Jeyapaul, Chuteng Zhu, Paul Whatmough, Yuhao Zhu. Characterizing Soft-Error Resiliency in Arm's Ethos-U55 Embedded Machine Learning Accelerator. *arXiv preprint arXiv:2404.09317*, April 2024.

[20] Boyu Chang, Binbin Zhao, Qiao Zhang, Peiyu Liu, Yuan Tian, Raheem Beyah, Shouling Ji. FirmRCA: Towards Post-Fuzzing Analysis on ARM Embedded Firmware with Efficient Event-based Fault Localization. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P) 2025*, 2024.