POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

# A Cross-Layer Kernel-Level Fault Emulation Framework for Processor-based Systems

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

**Author:** Andrea Carbonetti

**Advisor:** Prof. Luca Maria Cassano

**Co-advisor:** Elia Lazzeri

**Academic year:** 2025-2026

---

## 1. Introduction

The aerospace and automotive industries are currently undergoing a fundamental paradigm shift from proprietary, radiation-hardened legacy hardware to high-performance Commercial Off-The-Shelf (COTS) components, often referred to as "New Space" [3]. While modern COTS architectures offer the computational power necessary for advanced autonomous algorithms, they introduce a critical reliability gap. Unlike their radiation-hardened predecessors, they are highly susceptible to Single Event Upsets (SEUs) [9][12][4]. These transient bit-flips, caused by ionizing radiation, capable of triggering catastrophic system failures, introduce severe risks to safety-critical systems. Ensuring compliance with rigorous safety standards (such as ISO 26262 and DO-254) requires validating software resilience against these faults, but the current verification methodologies force engineers into a difficult trade-off. Physical Radiation Testing provides realism but it can be destructive, it is prohibitively expensive and late-stage dependent. Conversely, Simulation-based approaches offer extremely high controllability and repeatability but suffer from negligible throughput, often requiring hours to simulate mere seconds of execution [11]. Existing software-based injection tools frequently fail to bridge this gap, often being too intrusive, limited to kernel-space verification, or burdened by significant performance overheads. To address this challenge, this thesis proposes and implements a novel Cross-Layer Kernel-Level Fault Emulation Framework. Designed as a Linux Loadable Kernel Module (LKM), this tool bridges the gap between accurate hardware testing and fast software simulation. By using kernel privileges to inject faults directly into the physical hardware used by user-space processes, the proposed framework enables the validation of software resilience at near-native speeds, eliminating the abstraction layer of simulators without the costs (literal costs and low controllability) of physical irradiation.

## 2. Background and Related Works

To understand the mechanics and necessity of the proposed framework, it is essential to analyze how physical radiation phenomena manifest as software errors and why existing methodologies are insufficient for verifying modern COTS binaries.

## 2.1. Radiation Effects and Software Manifestation

The reliability of microelectronics in high-altitude and extraterrestrial environments is dictated by the interaction between ionizing radiation and semiconductor materials. Charged particles, originating from Galactic Cosmic Rays (GCR) and Solar Particle Events (SPE), interact with silicon atoms to deposit charge. As transistor's sizes decrease to nanometer scales, the "Critical Charge" required to alter the state of a transistor drops significantly [13]. Consequently, a single particle strike can induce a Single Event Upset (SEU), which translates as a spontaneous bit-flip in a memory cell or register [8].
While these faults are transient hardware events, their impact on the system is determined by the software layer. In the context of the Linux Operating System, the propagation of a fault depends on which segment of the Process and Memory Model is affected:

- **Instruction Memory (Text Segment):** A bit-flip here can alter opcodes or operands, leading to immediate illegal instructions or drastic changes in control flow.
- **Data Memory (Heap/Stack):** Faults in these regions may corrupt variables or return addresses. These errors often remain latent (Silent Data Corruption) until the corrupted data is consumed, making them difficult to detect.
- **Architectural State (Registers):** Registers represent the most timing-sensitive storage. A fault in a register (e.g., the Program Counter or Stack Pointer) often results in immediate crashes (Segmentation Faults) or hangs.

User-space applications are isolated from the underlying hardware by the kernel's privilege model. A standard process cannot arbitrarily modify its own instruction pointer or inspect the architectural state of other processes. This isolation, while necessary for security, creates a barrier for observing and emulating faults, necessitating a solution that operates with the elevated privileges of the Linux Kernel.

## 2.2. Limitations of Existing Fault Injection Methodologies

The validation of safety-critical software requires a Fault Injection (FI) methodology that balances three competing metrics: *Controllability* (spatial/temporal precision), *Repeatability*, and *Throughput*. A review of the state of the art reveals that no single existing approach satisfies perfectly all requirements for COTS validation, especially the *Throughput*:

### 2.2.1 Physical and Hardware-Based Injection

Irradiation testing (e.g., heavy-ion bombardment) serves as the ground truth for reliability. However, it suffers from high cost and low controllability since researchers cannot target specific logic gates or clock cycles [13][6].

### 2.2.2 Microarchitectural Simulation

Tools such as Gem5 [2] and GemFI [10] allow for perfect observability by simulating the hardware in software. While this maximizes controllability, it introduces a severe performance bottleneck. Simulators run orders of magnitude slower than native hardware, making them unsuitable for validating full operating system stacks or heavy workloads. Furthermore the injection campaign is executed on a model of the architecture rather than the architecture itself, which can be a limiting factor since the results can depend on the model used [5].

### 2.2.3 Software-Implemented Fault Injection (SWIFI)

SWIFI techniques attempt to emulate faults by modifying software state, but current tools exhibit distinct downsides:

- **Compile-Time Tools (e.g., LLFI [7]):** These operate on the source code or intermediate representation (IR). They are effective but inapplicable to proprietary COTS software where source code is unavailable.
- **Debugger-Based Tools (e.g., UN-FIT [1]):** These rely on user-space interfaces like GDB or `ptrace`. The use of debuggers introduces significant overhead, making the realization of large campaigns time consuming.
- **Kernel-Level Injectors (KITO [13]):** While tools like KITO leverage kernel privileges for speed, they primarily target kernel structures (e.g., `task_struct`) to test OS

stability. They lack the specialized mechanisms to inject faults specifically into the virtual memory and register context of user-space applications.

### 2.3. The Identified Gap

The analysis highlights a clear gap: there is a lack of tools capable of injecting faults into **user-space COTS binaries** with **kernel-level privileges**. A framework is required that can bypass the overhead of debuggers and the opacity of radiation testing, utilizing the kernel's visibility to emulate hardware faults at near-native speeds.

## 3. The proposed methodology

The proposed framework tries to fill the gap by working as a fault emulator. This means that the framework executes binary applications as if they were in a hostile environment by emulating via software the faults that may happen in the hardware used by the running binary. Unlike distributed setups requiring external host machines, this solution operates entirely on a single Linux node (Kernel 6.14+). The architecture is divided into three logical components: a **C++ Userspace Controller** for orchestration and analysis, a **Loadable Kernel Module (LKM)** for direct hardware manipulation, and the **Victim Process**, which is executed natively to ensure realistic timing behavior.

### 3.1. Userspace Orchestration and Parallelism

The Controller serves as the campaign orchestrator, parsing JSON configurations to define targets and fault models. To enhance the throughput, the Controller utilizes **OpenMP** to spawn multiple worker threads, enabling parallel injections on multiple experiments. Each thread maintains an isolated session with the kernel via a unique file descriptor, preventing race conditions during concurrent testing. A critical feature of the implementation is the **Software Watchdog**. Since fault injection may cause processes to hang, the Controller first obtains the average execution time of the target. If a run exceeds a safety margin (set to 10× the baseline), the Controller issues a specific `IOCTL_KILL` command to the kernel, forcefully reaping the zombie process to ensure the campaign continues uninterrupted.

### 3.2. Kernel-Level Fault Emulation

The core innovation resides in the custom LKM, which exposes a character device for communication. To ensure portability across COTS hardware, the module implements a **Hardware Abstraction Layer**. This maps generic register requests to architecture-specific structures (e.g., mapping to rax on x86-64 or x0 on ARM64), allowing the same tool to validate different platforms without code modification. The injection mechanism follows a **Stop-Flip-Resume** protocol to guarantee atomicity:

1. **Process Control:** The module uses the *User Mode Helper* API to spawn the target, immediately freezing it in a TASK_STOPPED state via SIGSTOP. This is done to allow the spawn of the necessary `monitor_thread` and `bitflip_thread` by LKM. After their initialization the process is resumed.
2. **Fault Injection Mechanisms:**
   - **Register Injection:** The module accesses the pt_regs structure on the kernel stack, flipping a single bit in the target register.
   - **Memory Injection:** The module traverses the process's Virtual Memory Areas (VMAs), bypassing page protections to inject faults directly into the virtual address space. For file-backed memory the system employs a "dirty page restoration" mechanism to prevent permanent corruption of the binary on the disk.
3. **Resumption:** Once the state is modified, the module issues SIGCONT, allowing the processor to consume the corrupted data naturally.

### 3.3. Triggering Modes

The framework supports two injection modalities to cover different validation needs: **Non-Deterministic**, which uses timers for exploratory testing of random asynchronous interrupts and **Deterministic**, which uses Linux **Uprobes** to inject faults at precise instruction offsets. This dual approach allows for both the discovery of failures in the code and the testing of specific vulnerability points.

### 3.4. Automated Analysis and SDC Detection

To close the verification loop, the framework integrates an automated post-processing analyzer that categorizes the outcome of every injection run. The Controller compares the artifacts generated by the victim process against a "Golden Output" established during the profiling phase. This analysis logic autonomously classifies failures into three categories:

- **Hangs:** Identified by the Software Watchdog when execution times out.
- **Crashes:** Detected via non-zero exit codes.
- **Silent Data Corruption (SDC):** Detected through strict bit-wise comparison of output files. For visual outputs the system generates differential image maps, rendering discrepancies in white to visualize the specific impact of the bit-flip on the data processing algorithm.

## 4. Validation and Results

The experimental evaluation of the proposed framework demonstrates that it successfully bridges the gap between the accuracy of hardware testing and the accessibility of software simulation. The validation methodology focused on three critical parameters: fault model accuracy, runtime performance, and scalability.

### 4.1. Accuracy against Hardware Ground Truth

To ensure statistical robustness, results were averaged over five campaigns of 10,000 injections per benchmark. Tables 1 and 2 present the comparative results. The framework achieved SDC rates highly comparable to the hardware ground truth (e.g., 3.18% FPGA vs. 3.82% XL-FI for `coremark`), confirming that the software-based bit-flips accurately emulate transient hardware errors. The maximum difference ($\Delta$) in "Hang+Crash" rates reached 3.91%, but this can be explained by the architectural differences. The $\chi^2$ tests returned $p$-values $< 0.001$ indicating that the distributions are not statistically identical. In addition to this, the calculated Cramér's V (0.031 and 0.049) shows that the association between the distributions is negligible, confirming accurate emulation.

Table 1: Validation results for `coremark`. Counts are averages over five 10000-injection campaigns. Statistical relevance is reported via $\chi^2$ and Cramér's V (effect size).

| | **FPGA** (Med. BOOM) | **XL-FI** (HiFive P550) |
|---|---|---|
| Runs | 10000 | 10000 |
| Benign/No-effect | 7647 | 7798 |
| SDC | 318 | 382 |
| Hang+Crash | 2035 | 1820 |
| *Statistical Analysis* | | |
| Diff. (Max $\Delta$) | – | 2.15 pp |
| $\chi^2$ ($p$-value) | 19.31 ($p < 0.001$) | |
| Cramér's V | 0.031 (Negligible) | |

Table 2: Validation results for `audiomark`. Counts are averages over five 10000-injection campaigns. Statistical relevance is reported via $\chi^2$ and Cramér's V (effect size).

| | **FPGA** (Med. BOOM) | **XL-FI** (HiFive P550) |
|---|---|---|
| Runs | 10000 | 10000 |
| Benign/No-effect | 7790 | 8177 |
| SDC | 6 | 10 |
| Hang+Crash | 2204 | 1813 |
| *Statistical Analysis* | | |
| Diff. (Max $\Delta$) | – | 3.91 pp |
| $\chi^2$ ($p$-value) | 48.42 ($p < 0.001$) | |
| Cramér's V | 0.049 (Negligible) | |

### 4.2. Performance and Overhead Analysis

A primary objective of this thesis was to overcome the throughput bottlenecks of microarchitectural simulators. The implementation results confirm that the Kernel-Level approach operates at near-native speeds.

- **Minimal Latency:** As shown in Table 3, the framework introduces a runtime overhead of less than 10% across most workloads. For computationally intensive tasks like `coremark`, the overhead drops to approximately 2.74%, as the fixed costs of injection are amortized over the execution time.
- **Linear Scalability:** Leveraging the proposed parallel orchestration architecture, the framework demonstrated linear speed-up relative to the number of worker threads.

Table 3: Performance Overhead of Fault Injection (FI) Tool across 1000 runs Measured on a Ryzen 9 5900

| Target | Native (ms) | FI Tool (ms) | Overhead (ms) | Overhead (%) |
|---|---|---|---|---|
| MNIST | 169 | 186 | 17 | 10.06% |
| YOLO | 288 | 300 | 12 | 4.16% |
| Image Proc. | 197 | 220 | 23 | 11.67% |
| CoreMark* | 11,617 | 11,935 | 318 | 2.74% |

*CoreMark based on 100 runs.

Tests on high-core-count platforms (Ryzen 9 and Threadripper) showed that injection campaigns scale efficiently up to 32 concurrent workers, enabling the execution of 100,000 injections in significantly reduced timeframes compared to serial execution.

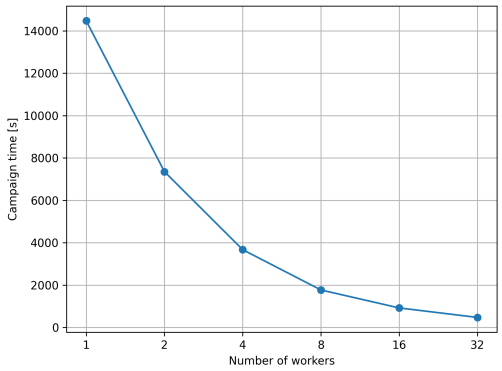


Figure 1: Campaign time versus number of workers on the Threadripper 9970X platform for a 1000-injection `coremark` campaign.

### 4.3. Insights from Large-Scale Campaigns

The high throughput of the framework enabled extensive campaigns (up to 100,000 runs) on complex workloads, yielding statistical convergence and actionable insights into software reliability:

- **Statistical Convergence:** Data comparisons between 10,000 and 100,000 injection runs (summarized in Table 4) revealed consistent vulnerability profiles, indicating that the framework provides reliable reliability estimates even with smaller, faster to execute campaigns.
- **AI Workload Vulnerability:** Comparative analysis between Neural Networks of varying sizes (MNIST vs. YOLO) confirmed a direct correlation between model "footprint" and SDC probability. Larger models like YOLO exhibited a significantly higher rate of misclassifications (0.20%) compared to smaller models (0.03%), validating the hypothesis that larger memory footprints increase the surface area for silent data corruption.

In conclusion, the results validate the framework as a robust, high-performance tool capable of executing massive fault injection campaigns on COTS binaries with the fidelity required for safety-critical certification standards.

## 5. Conclusions

This thesis addresses the critical verification bottleneck impeding the adoption of Commercial Off-The-Shelf (COTS) hardware in the aerospace and automotive sectors. By developing and implementing a novel Cross-Layer Kernel-Level Fault Emulation Framework, this work successfully bridges the dichotomy between the high fidelity of physical radiation testing and the accessibility of software simulation.

The proposed solution changes the validation workflow by eliminating the reliance exclusively on intrusive debuggers or source-code instrumentation. Instead, it leverages the Linux kernel's privileged access to inject faults directly into the architectural state of binary executables. The experimental validation confirms that the framework satisfies the main components for an effective fault injection:

- **Fidelity:** It reproduces physical hardware fault distributions with high accuracy, validated against FPGA ground truth.
- **Performance:** It operates at near-native speeds with negligible overhead ($< 10\%$), significantly outperforming traditional microarchitectural simulators.
- **Scalability:** It supports massive, parallelized campaigns, enabling the detection of

Table 4: Summary of injection campaigns executed for general tool evaluation. Percentages are relative to the number of injections in each campaign.

| Workload (campaign size) | Time [s] | Benign | SDC | Crash | Hang |
|---|---|---|---|---|---|
| `coremark` (10 000) | 13210.0 | 7798 (77.98%) | 382 (3.82%) | 1807 (18.07%) | 13 (0.13%) |
| `coremark` (100 000) | 110615.0 | 77728 (77.73%) | 4285 (4.29%) | 17752 (17.75%) | 235 (0.23%) |
| `audiomark` (10 000) | 6879.59 | 8177 (81.77%) | 10 (0.10%) | 1811 (18.11%) | 2 (0.02%) |
| `audiomark` (100 000) | 67761.7 | 81205 (81.20%) | 87 (0.09%) | 18683 (18.68%) | 25 (0.03%) |
| MNIST inference (100 000) | 3820.25 | 75333 (75.33%) | 25 (0.03%) | 22910 (22.91%) | 1732 (1.73%) |
| YOLO inference (100 000) | 15369.8 | 76438 (76.44%) | 199 (0.20%) | 21643 (21.64%) | 1720 (1.72%) |
| Blur Filter (100 000) | 2925.39 | 74531 (74.53%) | 139 (0.14%) | 23501 (23.50%) | 1829 (1.83%) |

rare Silent Data Corruption events in complex AI workloads.

- **Controllability:** It enables fine-grained, deterministic injection, allowing the targeting of exact architectural registers at precise instruction boundaries.

Ultimately, this research provides a viable framework for certifying high-performance COTS systems for safety-critical missions. By enabling rapid, low-cost, and accurate reliability profiling, the framework supports the "New Space" paradigm, allowing engineers to confidently deploy advanced autonomous software on non-hardened hardware in radiation-prone environments.

## References


[1] Alexander Aponte-Moreno et al. “Reliability Evaluation of RISC-V and ARM Microprocessors Through a New Fault Injection Tool”. In: *2021 IEEE 22nd Latin American Test Symposium (LATS)*. 2021, pp. 1–6.

[2] Nathan Binkert et al. “The gem5 simulator”. In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7.

[3] Giuseppe Brunetti et al. “COTS Devices for Space Missions in LEO”. In: *IEEE Access* 12 (2024), pp. 76478–76514.

[4] Nikoloas Chatzivangelis et al. *Analysis of Devices in Radiation Harsh Environments*. Tech. rep. 2025.

[5] Manolis Kaliorakis et al. “Differential fault injection on microarchitectural simulators”. In: *Proceedings - 2015 IEEE International Symposium on Workload Characterization, IISWC 2015*. Institute of Electrical and Electronics Engineers Inc., Oct. 2015, pp. 172–182.

[6] Johan Karlsson et al. *Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms*. Tech. rep.

[7] Qining Lu et al. “LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults”. In: *Proceedings - 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015*. Institute of Electrical and Electronics Engineers Inc., Sept. 2015, pp. 11–16.

[8] Eugene Normand. *Single-Event Effects in Avionics*. Tech. rep. 2. 1996.

[9] Geffrey Ottman et al. “An Algorithmic Approach to Observed Single Event Effects in Van Allen Probes Solid State Recorders”. In: *IEEE Transactions on Nuclear Science* 66.12 (Dec. 2019), pp. 2408–2416.

[10] Konstantinos Parasyris et al. “GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates”. In: *Proceedings - 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014*. Institute of Electrical and Electronics Engineers Inc., Sept. 2014, pp. 622–629.

[11] Paolo Rech. “Artificial Neural Networks for Space and Safety-Critical Applications: Reliability Issues and Potential Solutions”. In: *IEEE Transactions on Nuclear Science* 71.4 (Apr. 2024), pp. 377–404.

[12] Devesh Tiwari et al. “Understanding GPU errors on large-scale HPC systems and the implications for system design and operation”. In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 331–342.

[13] Alejandro David Velasco et al. “KITO tool: A fault injection environment in Linux kernel data structures”. In: *Microelectronics Reliability* 60 (May 2016), pp. 153–162.