POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

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

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Andrea Carbonetti**

Student ID: 10731413
Advisor: Prof. Luca Maria Cassano
Co-advisors: Elia Lazzeri
Academic Year: 2025-2026

# Abstract

The paradigm shift toward "New Space" has made necessary the adoption of Components-off-the-shelf (COTS) for space applications, including SoCs. While these systems offer the necessary computational power to execute modern algorithms and neural networks, they lack the radiation tolerance of legacy radiation-hardened hardware, shifting the responsibility of reliability from the hardware to the software layer. In safety-critical sectors such as the automotive and aerospace environments, faults induced by ionizing particles, specifically Single Event Upsets, present a significant threat to system integrity since they are capable of causing critical failures. The current state of the art struggles to address this challenge effectively. Physical radiation testing provides realistic data but is costly and can be destructive. On the other hand Simulation-based approaches such as RTL simulation offer great accuracy and reproducibility, but lack the throughput to execute complex software stacks. The Existing Software implemented Fault Injection (SWIFI) tools attempt to bridge this gap but often face a fundamental tradeoff: they either provide cycle-accurate simulation at an extremely low throughput, or employ invasive techniques that introduce high intrusiveness on a system, hindering the execution of large scale injection campaigns.

This thesis proposes a Cross-Layer Kernel-Level Fault Emulation Framework for processor based systems designed to bridge this gap. By operating as a Loadable Kernel Module (LKM) on Linux it achieves near-native speeds, demonstrating a performance overhead of less than 12% across diverse workloads, while maintaining high controllability over the injection process. The framework was validated against traditional FPGA-based hardware injectors, showing consistent fault outcome distributions within a 4% margin. Furthermore, scalability tests confirm a linear speed-up when parallelized across multi-core architectures, enabling 100k injection campaigns to be completed in only a few hours. This provides developers a high-performance open-source solution to analyze software radiation resilience on processor based systems.

**Keywords:** Fault Injection, SWIFI, Single Event Upsets, SEU, Linux Kernel, New Space

# Abstract in lingua italiana

Il cambio di paradigma verso il "New Space" ha reso necessaria l'adozione di componenti COTS (Components-off-the-shelf) per applicazioni spaziali, inclusi i SoC. Sebbene questi sistemi offrano la potenza di calcolo necessaria per eseguire moderni algoritmi e reti neurali, mancano della tolleranza alle radiazioni tipica dell'hardware legacy "radiation-hardened", spostando la responsabilità dell'affidabilità dal livello hardware a quello software. In settori critici per la sicurezza come quelli automobilistico e aerospaziale, i guasti indotti da particelle ionizzanti, nello specifico i Single Event Upset (SEU), rappresentano una minaccia significativa per l'integrità dei sistemi poiché in grado di causare malfunzionamenti critici. L'attuale stato dell'arte fatica ad affrontare con efficacia questa sfida. I test fisici di irraggiamento forniscono dati realistici, ma sono costosi e possono essere distruttivi, mentre gli approcci basati sulla simulazione, come la simulazione RTL, offrono grande accuratezza e riproducibilità, ma mancano del throughput necessario per eseguire programmi complessi. Gli strumenti esistenti di Software Implemented Fault Injection (SWIFI) tentano di colmare questo divario ma spesso affrontano un compromesso fondamentale: o forniscono una simulazione cycle-accurate con un throughput basso, oppure impiegano tecniche invasive che introducono un'elevata intrusività nel sistema, rendendo difficoltosa l'esecuzione di campagne di iniezione su larga scala.

Questa tesi propone un framework di emulazione dei guasti a livello kernel e cross-layer per sistemi basati su processore, progettato per riempire questo vuoto implementativo. Operando come Loadable Kernel Module (LKM) su Linux, il framework permette di eseguire processi con velocità prossime a quelle native, dimostrando un overhead prestazionale inferiore al 12% su diversi carichi di lavoro, pur mantenendo alta la controllabilità sul processo di iniezione. Il framework è stato validato rispetto ai tradizionali iniettori hardware basati su FPGA, ottenendo dei risultati simili che rientrano entro un margine del 4% nel peggiore dei casi. Inoltre, i test di scalabilità confermano un aumento delle prestazioni lineare quando parallelizzato su architetture multi-core, consentendo di completare campagne da 100.000 iniezioni in poche ore. Ciò fornisce agli sviluppatori un framework open-source ad alte prestazioni per analizzare la resilienza alle radiazioni di software su sistemi basati su processore.

**Parole chiave:** Fault Injection, SWIFI, Single Event Upsets, SEU, Linux Kernel, New Space

# Contents

# 1 | Introduction

The contemporary landscape of computing is vast and diverse, spanning from trivial embedded tasks, such as activating a simple timer, to complex, high-performance operations executing within safety-critical environments. In sectors such as the automotive and aerospace industries, the role of software has shifted from auxiliary to foundational. Modern software stacks embedded in vehicles must now comply with rigorous reliability requirements, such as ISO 26262, while aerospace software is governed by standards like DO-254.

The necessity for these strict standards arises from the potentially catastrophic consequences of failure. In safety-critical domains, a system fault is not merely a bug that requires a patch, it is a liability that can lead to silent data corruption, system hangs, or crashes, ultimately endangering human life or causing massive economic loss. As reliance on autonomous systems and complex on-board processing grows, ensuring the resilience of these systems against environmental hazards has become a primary engineering challenge.

## 1.1. The Hardware Dilemma: Performance vs. Reliability

Historically, systems deployed in harsh environments, particularly out-of-atmosphere applications, relied on radiation-hardened hardware. The development of such hardware is inherently slow due to the extensive design and testing phases required to guarantee immunity to environmental factors. A prominent example of this technological lag is the on-board system of the James Webb Space Telescope, which utilizes a decades old processor based on 180nm technology with a clock frequency of just 200MHz [31].

In stark contrast, consumer and industrial markets have seen rapid advancements in processing power. For comparison, a standard general-purpose CPU from 2021, such as the AMD Ryzen 9 5900, utilizes 7nm process technology and achieves base clock speeds of 3GHz [1]. This disparity highlights a critical dilemma: while general-purpose Commercial Off-The-Shelf (COTS) hardware offers the performance necessary for modern algorithms,

such as AI and image processing, its smaller transistor sizes make it significantly more susceptible to radiation-induced faults than older, radiation-hardened legacy hardware.

To mitigate the risks associated with radiation induced faults, architectures like HARV-SoC [18] have been developed. These architectures employ architectural redundancy such as Triple Modular Redundancy (TMR) and Error Correction Codes (ECC) to improve fault tolerance and enable the reliable use of COTS FPGAs in radiation prone environments.

As the industry pushes toward using COTS hardware in space [9] (the "NewSpace" paradigm), the risk of radiation induced errors increases. Specifically, electronics that work in the space environment are prone to Single Event Upsets (SEUs) [34][47][10]. A SEU occurs when charged particles, typically from cosmic rays or radiation belts, ionize the silicon medium, leaving a path of free charge carriers (electron-hole pairs) along its trajectory. The charge carriers then migrate driven by gradients in electric potential [15] that induce a transient current.

## 1.2. The Challenge of Verification

Validating a system's resilience against SEUs is a complex undertaking. The most realistic method is Radiation Testing, which consists in bombarding physical circuits with heavy ions or neutrons [44]. While this provides realistic data [40], it is prohibitively expensive, requires the physical device to be available, which is impossible during early design phases, and carries the risk of permanently destroying the test hardware [48][4].

Alternatively, engineers use Simulation-Based approaches, such as Register Transfer Level (RTL) simulations. While these offer high visibility into the system state without physical damage [40], they suffer from extremely low throughput. Simulating a few seconds of a modern workload on an RTL model can take hours or even days [40], making it unsuitable for validating complex software stacks or operating systems since state of the art simulators run as slow as 1KHz [26].

Bridging the gap between Radiation Testing and RTL Simulation are Fault Injection and Fault Emulation techniques. These methods aim to artificially introduce faults into systems to observe their behavior, balancing the need for realism with accessibility [4]. Common techniques include:

- **Pin-level injection**, which forces voltage levels on the pins of an integrated circuit [29]. It is a way of injecting faults into a physical system without using radiation.

- **Debug Port injection** is a fault injection method that utilizes built in debugging interfaces to halt the processor and modify internal states [39]. As stated in [48] this type of solution has as drawback a high implementation cost, a high time of campaign execution caused by the need of restarting the system after every experiment.

This creates a need for Software Implemented Fault Injection (SWIFI). SWIFI emulates hardware faults by modifying registers and memory via software. This category includes Microarchitectural Simulation, a technique that unlike RTL simulation models the processor at a higher level of abstraction using simulators such as Gem5 [8]. Then, using frameworks like GeFIN and MaFIN [23] makes possible the injection of faults into specific hardware structures. This approach is significantly faster than RTL simulation, but still slower than native speeds and require detailed hardware simulations, but even then this might not be enough to perfectly match the final silicon.

However, existing SWIFI tools, both simulation based and native, often suffer from limitations:

- **Scope limitations:** Some tools, like KITO [48], inject faults primarily into kernel data structures, leaving user-space applications less verified.
- **Intrusiveness:** Techniques like compile-time injection (modifying Intermediate Representation) require access to source code and recompilation, which may not always be possible with third-party binaries.
- **Performance Overhead:** Many simulators or debuggers introduce significant performance overhead, making testing a very time consuming operation for large campaigns. One of the many tools that will be analyzed in the chapter 3 is UN-FIT [4], a tool that uses QEMU [7] and the debugger GDB, that introduces breakpoints and inherently, overhead.

## 1.3. Contribution of the Thesis

This thesis proposes and implements a novel Cross-Layer Kernel-Level Fault Emulation Framework for Processor-based Systems. The proposed tool is designed to bridge the gap between accurate hardware testing and fast software simulation to perform large injection campaigns only on systems based on General Purpose CPUs that can run Linux as OS. By operating as a Loadable Kernel Module (LKM), the tool utilizes the Linux kernel's elevated privileges to access and manipulate the virtual address space and CPU registers of specific user-level processes by emulating the effects of hardware faults on the

architectural state of the process. Using a LKM that emulates faults directly on the CPU it is possible to maintain near-native speeds while still affecting all the hardware parts used during the execution of a user process.

Unlike existing kernel-injectors that target the OS itself, this tool is specifically designed to assess the robustness of user-level applications. It allows for the emulation of transient hardware faults in the memory and registers of a running process with minimal execution overhead. This means that the tool emulates Hardware faults and analyzes its effects in the Software.

In the context of this thesis, these faults are modeled as transient, non-permanent bitflips (changing a logic 0 to 1, or vice versa). While they do not permanently damage the hardware, they can corrupt software logic, leading to unpredictable behavior.

The key contributions of this work include:

- **Kernel Space Injection Mechanism:** A module that executes programs, monitors them and interrupts their execution to flip bits in registers or memory to simulate SEUs dynamically.
- **User Space Automation Suite:** A controlling application that manages injection campaigns using a configuration file (JSON). It allows users to define parameters such as the target thread, injection timing (random or specific), and target registers.
- **Automated Analysis:** A built-in analysis engine that compares campaign outputs against a fault-free execution which is referred to as a "Golden Run". It categorizes results automatically into Silent Data Corruptions (SDC), Crashes, or Hangs, generating statistical summaries and CSV reports for the user.

There is currently a critical gap to fill in the verification landscape: designers must choose between a realistic black-box non reproducible approach, given by the nature of physical irradiation, or the abstraction of simulation-based SWIFI. While architectural simulators still provide functional insight, they do so relying on idealized models, validating the model of the silicon rather that the silicon itself. This thesis bridges that gap. By embedding fault emulation directly into the OS kernel of the physical target hardware, the proposed tool eliminates the abstraction layer of simulators.

The objective of this thesis is to provide an open-source, easily deployable tool that accelerates the development and testing of safety-critical applications on Linux systems, allowing developers to easily analyze a software's radiation resilience.

## 1.4. Thesis Organization

The remainder of this thesis is organized as follows:

- **Chapter 2: Background** This chapter provides the theoretical foundation necessary to understand the work, detailing Operating System concepts, memory management, and the physics of how radiation interacts with microelectronics.
- **Chapter 3: Related Works** This chapter analyzes the state of the art in fault injection. It details the distinctions between Hardware Injection, Simulation-based methods, and other SWIFI approaches, placing the proposed tool within the current research panorama.
- **Chapter 4: Tool Implementation** This chapter describes the architecture of the developed framework, explaining the interaction between the kernel module and the userspace controller, as well as the implementation of the injection logic.
- **Chapter 5: Validation and Results** This chapter presents the experimental campaigns conducted using the tool, analyzing the impact of SEUs on various workloads and validating the tool's effectiveness.
- **Chapter 6: Conclusions and Future Developments** Finally, this chapter discusses the strengths and limitations of the current implementation and proposes directions for future expansions of the framework.

# 2 | Background

This chapter is going to provide the reader the knowledge needed to understand what the proposed tool emulates, how it does that and where it operates.

## 2.1. Linux Process and Memory Model

Modern operating systems provide abstractions that decouple software execution from the underlying hardware. These abstractions are essential for performance, security, and portability, but they also define how hardware faults propagate to the software layer. Since this thesis focuses on software-level manifestations of radiation-induced faults, it is necessary to understand how Linux represents execution, memory, and isolation boundaries.

### 2.1.1. Process Abstraction

A process is the fundamental abstraction used by the Linux operating system to represent a running program. Conceptually, a process encapsulates all the information required for execution, including the program code, its data, the execution context, and the associated system resources. Each process is uniquely identified by a Process Identifier (PID), which allows the kernel to manage scheduling and resource allocation. Internally, Linux represents a process using a data structure known as the `task_struct`, also referred to as the Process Control Block (PCB). This structure contains critical information such as the process state, scheduling priority, memory mappings, open file descriptors, and the saved CPU register context [12][48].

#### Processes and Threads

While the terms *process* and *thread* are often used interchangeably, they represent distinct execution abstractions. A process defines an isolated virtual address space and a set of resources, whereas a thread represents an independent execution flow within that address space.

In Linux, threads are implemented as lightweight processes that share the same address space, file descriptors, and other resources, but maintain independent register states and stacks. From the kernel perspective, both processes and threads are managed through the same `task_struct` abstraction. As a result, a fault affecting the architectural state of a single thread may remain localized or propagate to other threads depending on the shared resources involved.

This distinction is particularly relevant in the context of fault injection, as a bit-flip affecting shared memory can impact all threads within a process, whereas a fault in a thread-local register may only influence a single execution flow.

### 2.1.2. Virtual Address Space Layout

To provide isolation and flexibility, Linux employs virtual memory. Each process operates within its own virtual address space, which is translated to physical memory through the Memory Management Unit (MMU). This abstraction ensures that processes cannot directly access or corrupt each other's memory, while allowing the kernel to manage memory efficiently [6].

The virtual address space of a Linux process is logically divided into several regions, each serving a distinct purpose.

#### Text Segment and File-Backed Mappings

The text segment (`.text`) contains the executable instructions of the program. This region is typically marked as read-only and executable to prevent accidental or malicious modification.

Text segment and shared libraries are "file-backed memory", meaning they map a file directly into the virtual address space. To optimize physical memory usage, the Linux kernel employs a **Unified Page Cache**. If multiple processes map the same shared library (e.g., `libc`), they share the same physical page frames in RAM while retaining distinct virtual addresses.

From the perspective of radiation-induced faults, corruption in this region is critical. A single bit-flip can alter control flow, generate illegal instructions, or silently change computational behavior. Furthermore, because these pages may be shared via the page cache, faults in shared library code can potentially propagate errors across multiple processes relying on the same resource.

## Data and BSS Segments

The data segment (`.data`) stores initialized global and static variables, while the BSS segment (`.bss`) contains uninitialized global and static variables that are zero-initialized at program startup. These regions persist for the entire lifetime of the process.

Faults affecting these segments may corrupt configuration parameters, counters, or state variables, often leading to subtle and hard to detect errors that persist throughout execution.

## Heap and Anonymous Memory

The heap is a dynamically managed memory region used for runtime allocation. Its size can grow or shrink during execution depending on the program's allocation patterns.

The heap represents "anonymous memory", meaning it is not backed by any file on the filesystem. Consequently, when the operating system needs to reclaim physical RAM used by these pages, it must write the data to the swap area (if available) or discard it if the process terminates.

Radiation-induced faults in the heap are purely transient in the context of the running process but may corrupt dynamically allocated data structures. This can lead to incorrect computations, invalid memory accesses, or delayed failures when the corrupted data is eventually consumed.

## Stack

The stack stores function call frames, local variables, and return addresses. Like the heap, the stack utilizes anonymous memory, but it is automatically managed by the kernel to support control flow. Each thread maintains its own private stack.

Corruption of stack data is particularly dangerous. A single bit-flip affecting a return address or a stack pointer can redirect execution to unintended locations, potentially causing crashes or undefined behavior.

## Instruction and Data Memory Perspective

From a software fault model standpoint, it is useful to classify the virtual address space into two broad categories:

- **Instruction memory**, primarily represented by the text segment and shared libraries, which governs control flow and execution semantics.

- **Data memory**, encompassing the data, BSS, heap, and stack regions, which stores program state and intermediate results.

This distinction is central when analyzing the impact of radiation-induced faults, as errors in instruction memory often lead to immediate and catastrophic failures, whereas data memory corruption may result in delayed or silent errors.

### 2.1.3. Context Switching and Register State

Modern multitasking operating systems rely on context switching to allow multiple processes and threads to share the same physical CPU. A context switch occurs when the kernel suspends the execution of a running task and transfers control to another one. This mechanism is fundamental for scheduling, but it also plays a critical role in how transient hardware faults propagate to software-visible behavior.

#### Context Switch Mechanism

When a context switch is triggered, either voluntarily (e.g., a blocking system call) or involuntarily (e.g., a timer interrupt), the kernel must preserve the complete execution context of the currently running task. This context includes the architectural state of the processor, such as the general-purpose registers, the program counter or instruction pointer, the stack pointer, and architecture-specific status or flag registers.

The saved state is stored within the task's kernel data structures, typically inside the process control block. Once the state is preserved, the kernel restores the previously saved context of the next scheduled task and resumes its execution as if it had never been interrupted.

Although the exact set of registers and the switching procedure are architecture-dependent, the conceptual model remains consistent across modern processors. For example, on x86-like architectures, registers such as `RIP`, `RSP`, and the general-purpose register file are saved and restored, while other architectures expose analogous constructs.

#### Register State as Software-Visible Architectural State

Registers represent the fastest and most frequently accessed storage elements in a processor. They hold intermediate computation results, function parameters, return values, and control information. As such, they are a critical component of the architectural state that directly influences program execution.

A radiation-induced bit-flip affecting a register may immediately corrupt the execution

of an instruction, or it may remain latent until the corrupted register is later used. If a fault occurs shortly before a context switch, the corrupted register state may be saved by the kernel and later restored, effectively extending the lifetime of the fault beyond the original execution window.

This behavior highlights how transient faults can persist at the software level even though the underlying physical disturbance was momentary.

### Timing Sensitivity of Faults

The impact of a fault is strongly dependent on its timing relative to program execution and operating system activity. A bit-flip occurring in a register may have vastly different consequences depending on whether the affected register is actively used, overwritten shortly after, or preserved across a context switch.

For instance, a fault injected into a register that is overwritten in the next instruction may be masked, whereas a fault affecting a register holding a return address or loop counter may lead to crashes or silent data corruption. If the fault is captured during a context switch, it can reappear much later when the task is rescheduled, making the root cause difficult to trace.

This temporal sensitivity explains why fault injection experiments must consider not only the location but also the timing of the fault, and why the observable effects of radiation-induced errors may be delayed with respect to the physical event that caused them.

## 2.1.4. Kernel Privilege Model

To ensure isolation and security, modern operating systems enforce a privilege model that restricts access to critical system resources. In Linux, this model separates execution into at least two privilege levels: user mode and kernel mode.

### User Mode and Kernel Mode

User mode is the least privileged execution level, where application code runs. In this mode, software is restricted from performing operations that could compromise system integrity, such as directly accessing hardware devices, modifying page tables, or manipulating processor control registers.

Kernel mode, on the other hand, is a highly privileged execution level reserved for the operating system kernel. Code executing in kernel mode has unrestricted access to system resources, including physical memory, processor registers, and hardware interfaces.

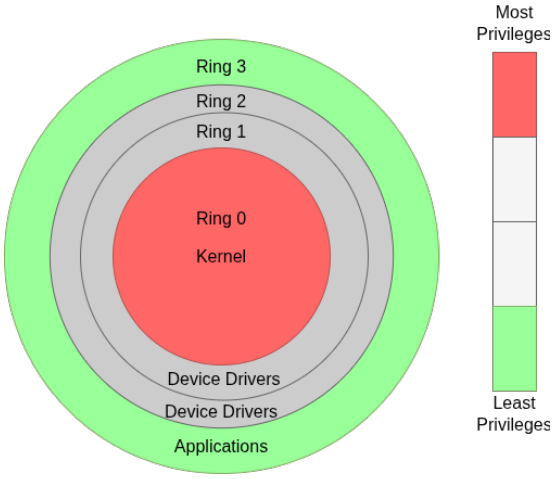


Figure 2.1: Privilege rings

Transitions from user mode to kernel mode occur through well-defined mechanisms such as system calls, interrupts, or exceptions.

Although the number and naming of privilege levels differ across processor architectures, the conceptual separation between unprivileged application code and privileged kernel code is a common design principle. The standard x86 privilege ring is shown in Figure2.1

## Access Restrictions in User Space

Due to the enforced privilege separation, userspace applications are inherently limited in their ability to observe or manipulate execution state. Specifically, a user-level process cannot directly access the memory of other processes, as each process operates within its own isolated virtual address space. This is the reason of the dreaded Segmentation faults obtained in failed executions of C code. The process attempted to access a virtual memory address that was not allocated to the process or had incompatible permissions, making the Operating System terminate the process.

Similarly, direct access to processor registers is restricted. While user code can read and write registers implicitly through instruction execution, it cannot arbitrarily inspect or modify the complete register state of itself or another process. Such operations require kernel mediation.

These restrictions are fundamental for system stability and security, but they also imply that user-level software alone is insufficient for observing or altering the full architectural state affected by radiation-induced faults.

### Kernel-Level Visibility and Control

Because the kernel operates in a privileged execution mode, it has full visibility over process execution state. It can inspect and modify memory mappings, access and alter register contents during context switches, and manage execution across all running tasks.

As a result, only the kernel is capable of observing and manipulating the complete process state in a controlled and system-wide manner. This property makes kernel-level mechanisms uniquely suited for studying and emulating hardware faults that affect architectural state, while preserving the isolation and correctness guarantees of the operating system.

## 2.1.5. Kernel Space - User Space Communication

To allow communications between userspace applications and kernel space applications the Linux Kernel provides specific interfaces. The most common methods, which are the ones used in the developing of the tool, involve the use of character devices and the Input/Output Control (IOCTL) system call.

### Character device

A character device is one of the fundamental device types in Linux. It is represented as a file in the `/dev` directory and allows data to be accessed as a stream of bytes. Drivers expose functionality through standard file operations such as `open()`, `read()`, `write()` and `close()`. When a user-space program interacts with a character device file, the Virtual File System (VFS) routes these calls to the corresponding functions defined in the kernel module's `file_operations` structure. This creates a bridge allowing the user to trigger kernel-side logic.

### IOCTL

While the `read` and `write` are suitable for data transfer, they are often insufficient for device control operations. The `ioctl` system call solves the problem of triggering specific actions by providing a versatile channel for sending control commands. An `ioctl` call typically takes a file descriptor, a command number and an optional argument which is usually a pointer to a data structure. This allows complex data structures to be passed bi-directionally between userspace and kernel space.

### 2.1.6. Kernel's Virtual File System

Linux is an operating systems that was developed by following Unix's philosophy. One of the paradigms is that everything is a file. The Virtual File System (VFS) is the kernel subsystem that implements this abstraction. It provides a common interface for all file system operations, regardless of the underlying physical file system.

When a userspace application calls `open()` it does not interact directly with the hardware driver, but with the VFS. The VFS handles the system call and redirects the request to the specific functions implemented by the mounted file system.

### 2.1.7. High resolution timer vs. Timer

Precise timing is a requirement for accurately injecting faults at specific moments during execution. Linux offers two primary timing mechanisms with distinct granularities.

#### Standard Kernel Timers

Standard timers (`kernel/timers.c`) in Linux are based on the system tick (jiffies). The resolution of these timers is determined by the kernel configuration constant `HZ`. Common values for `HZ` are 100, 250, or 1000, resulting in a timer granularity of 10ms, 4ms, or 1ms, respectively. While sufficient for the majority of the instructions, this granularity is too coarse for short fault injection in short lived programs, where a processor operating at gigahertz speeds executes millions of instructions within a single millisecond.

#### High resolution timers

To overcome the limitations of jiffies-based timers, Linux offers High Resolution Timers (HRT) [2]. HRTs rely on 64 bit logic instead of the 32 bit logic used in the standard kernel timers. HRTs use the `ktime` infrastructure, allowing time to be specified in nanoseconds. Internally, they are based on data structures that are not bound to jiffies or any other type of granularity[2].

### 2.1.8. Linux Kernel Uprobes

Uprobes (User-space Probes) is a Linux kernel feature that enables the dynamic instrumentation of userspace applications. It allows the kernel to intercept execution at any virtual address within a user process without requiring the program to be recompiled or restarted.

When a uprobe is registered for a specific address, the kernel modifies the target process's memory by replacing the instruction at that address with a breakpoint instruction. When the CPU executes this breakpoint:

1. A trap occurs, switching the CPU from user mode to kernel mode.
2. The kernel's exception handler catches the trap and locates the associated uprobe handler.
3. The custom handler (defined by the tracing tool or kernel module) is executed. This is where register states can be inspected or faults can be injected.
4. Once the handler finishes, the kernel single-steps the original instruction and resumes normal execution of the program.

This mechanism provides a non-intrusive way to pause execution flow and inject faults exactly before specific instructions are executed.

### 2.1.9. Read-Copy-Update (RCU)

Concurrency control is a major challenge in kernel programming. Traditional locking mechanisms like mutexes can degrade performance significantly in scenarios where data is read frequently but updated rarely. RCU is a synchronization mechanism designed to solve this problem.

RCU allows multiple readers to access data concurrently without using locks, ensuring extremely low overhead for read operations. When a writer needs to update a data structure:

1. **Read:** Readers access the data via pointers.
2. **Copy:** The writer creates a copy of the data structure and modifies the copy.
3. **Update:** The writer atomically swaps the global pointer to point to the new, modified copy.

The old data cannot be freed immediately because pre-existing readers might still be referencing it. The memory is reclaimed only after a "grace period," which guarantees that all readers accessing the old data have finished. This mechanism is extensively used in the Linux kernel for traversing task lists and routing tables, ensuring that a monitoring tool can iterate over processes without blocking the system.

### 2.1.10. Watchdog Timers

In embedded systems reliability is paramount. Embedded Systems might be deployed in conditions or locations that are difficult to access, hence there's a need to reset their operations if a failure is detected. A watchdog timer is a mechanism used to detect and recover from computer malfunctions. Fundamentally a watchdog is a timer that counts down from a specific initial value. The running software is responsible for resetting it periodically ("kicking" it) before it reaches 0. If the timer expires the watchdog generates a timeout signal which typically triggers a corrective action such as rebooting the system or terminating the faulty process.

## 2.2. Radiation Effects on Microelectronics

Utilizing microelectronics in high altitude or out-of-atmosphere conditions can affect the systems used. Radiation sources, such as high energy cosmic particles and alpha particles constantly interact with semiconductor materials[35][48].

### 2.2.1. Space and High-Altitude Radiation Environment

Radiation is a naturally occurring phenomenon, understanding its sources and spatial variations is critical for the reliability of modern electronics. The sources of radiation are different and the three main ones are:

#### Sources of Radiations

- Galactic Cosmic Rays (GCR): These radiations consist of charged ions constantly released by stars and major celestial event that happen in the universe. Because of Earth's magnetic field, the majority, but not all, of the particles are deviated. When these rays collide with nuclei in Earth's atmosphere, they produce a variety of secondary particles including alphas, protons, gammas and neutrons[40].
- Solar Particle Events (SPE): These events involve particles released from the sun. Solar flares can cause significant increase in radiation intensity particularly in the upper atmosphere and space[13].
- Van Allen Belts: while other radiations come from deep space, the Earth's magnetic field traps certain particles, primary protons and electrons, in belts surrounding the planet. The sources of protons are especially critical for the reliability of satellites in Low Earth Orbit (LEO)[34]

## Why altitude and latitude matter

The intensity of radiation exposure is not uniform, in fact it is heavily dictated by physical positioning.

- **Altitude**: The Earth's atmosphere acts as a protective shield, but the flux of particles does not simply increase linearly with height. Instead, the intensity is determined by a competition between two processes: the *production* of secondary particles created when cosmic rays collide with atmospheric nuclei, and the *removal* or attenuation of those particles as they travel deeper into the atmosphere.

  This interaction results in a peak radiation intensity known as the Pfotzer Maximum. For both neutrons and protons (specifically in the 100-750 MeV energy range), this maximum flux occurs at an altitude of approximately 60,000 ft (roughly 18 km). Below this altitude, the atmosphere absorbs the particles; above this altitude, the air is too thin to generate the maximum volume of secondary particle cascades [33].

- **Latitude**: The Earth's magnetic field provides levels of protections that vary depending on the latitude. The field in fact is strongest at the equator, requiring particles to have a higher energy to penetrate compared to the weakest points that are at the poles[33]. A more concrete example takes the neutron flux at the equator and compares it to the one at the poles, the result is a flux 6 times lower at the equator compared to the ones at the poles [33].

## Why COTS Systems are increasingly exposed

The constant drive for higher performance has led to reduced transistor sizes and increased transistor density other than lower operating voltages [40]. This physical scaling has significant implications for system reliability. As predicted by Lu et al. [28], modern computer systems are increasingly likely to expose hardware faults to the software layer rather than masking them entirely. Consequently, relying on hardware reliability is no longer sufficient, it is now critical to design and evaluate software-level error resilience mechanisms to tolerate these inevitable faults. In addition to the shrinkage of the systems, emerging technologies such space based AI require massive parallel processing power to execute Deep Neural Networks (DNNs) in real time. Using parallel and complex hardware devices for DNN execution can increase the vulnerability of these systems to radiation [40].

A lack of radiation hardened (rad-hard) components is another reason that contributes to COTS being chosen and to the need of testing if software handles correctly failures.

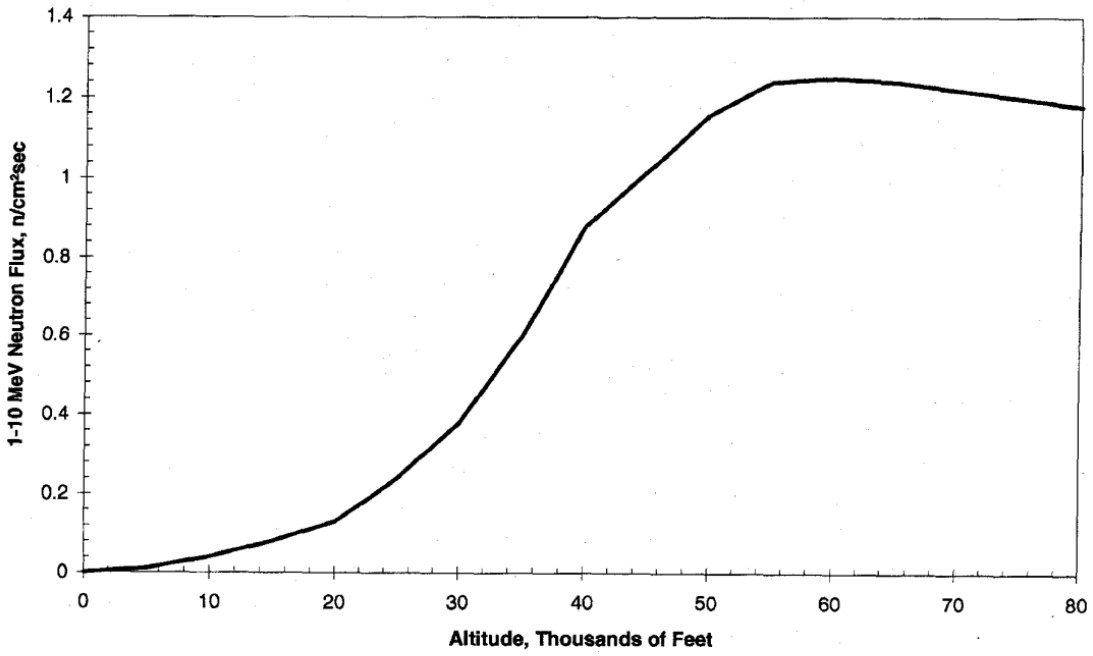


Fig. 2. The 1–10 MeV atmospheric neutron flux as a function of altitude based on aircraft and balloon measurements [2].

Figure 2.2: Neutron flux variation with altitude from [33]

Rad-hard components are designed to withstand harsh environments, but they are often based on technology that is years old and is not powerful enough to execute modern AI models [40]. This brings to only one possibility which is to choose COTS or soft-core implementations of FPGAs when possible, enabling modern applications ranging from high performance computing to efficient TinyML inference [26][20].

### 2.2.2. Physical Mechanism of SEUs and SEEs

The interaction between high energy particles and silicon circuits leads to transient disruptions collectively known as Single Event Effects (SEE). A common manifestation is the Single Event Upset (SEU), but the physical mechanism is a multistage process that begins with the interaction of a subatomic particle and a semiconductor.

#### Charge Deposition

When a ionizing radiation particle, passes through a semiconductor, it interacts with the silicon atoms [32]. Upon traversing the device, the charged particle deposits energy, generating electron-hole pairs along its trajectory. These carriers subsequently migrate via drift and diffusion, driven by gradients in electric potential and carrier concentration [15]. If the particle intersects with a sensitive node the local electric field rapidly collects

the generated carriers. This collection manifests as a transient current pulse [32].

## Critical Charge

The critical charge is defined as the minimum amount of charge that must be collected at a sensitive node to flip the state of a storage element[15]. If the charge deposited by the particle strike exceeds this threshold it can force a transistor state to change from ON to OFF or vice versa [40]. As technology continues to scale downward the transistor sizes shrink and operating voltages decrease, which causes the Critical Charge to drop [48]. A consequence of this lowering of the Critical Charge value is an increased sensitivity to upsets of modern chips caused even by low energy particles.

## Components vulnerability

When a particle strikes in proximity to a sensitive node, the effects can vary depending on the component and the circuit topology.

**Single Event Upsets (SEU)** In a storage cell such as a memory cell, a latch, or a flip-flop, when a sufficiently strong charge is collected, the result is a bit-flip, known as a Single Event Upset (SEU) [32].

**Single Event Multiple Upsets (SEMU)** Beyond single bit-flips, the high density of modern transistors means a single particle track can influence multiple adjacent nodes. This leads to Single Event Multiple Upsets (SEMU), where several memory cells are corrupted simultaneously by a single strike [6].

**Single Event Transients (SET)** When the collecting node belongs to a logical gate, the transient current pulse is transformed into a voltage glitch known as a Single Event Transient (SET). This pulse can propagate through combinational logic paths to reach latches or flip-flops [32]. Crucially, SETs can also affect clock distribution networks; a glitch here is severe as it can lead to erroneous data values globally or inadvertently clear memory content [6].

**Permanent and Frequency Effects** Finally, the charge from radiation can modify the characteristics of a transistor, possibly reducing the operation frequency of the device or even causing permanent failures[40].

## 2.3. Fault Abstraction and Classification

As already seen in the sections above the interactions between particles and semiconductors create a variety of fault behaviors that can be categorized by their duration and impact on the system.

### 2.3.1. Transient and Permanent Fault Models

Before describing the differences between transient and permanent faults the definition of fault is needed. A fault is an event that can cause a change to a system's state. When a fault affects the state of the system it becomes an error. If an error causes a change of the system's external state it becomes a failure [16]. In the previous section 2.2 we have already seen how the radiations, which are non other than faults, may cause errors and consequently failures by opening or closing a logic gate. In the scientific community it almost looks like the categorization of fault is loosely applied. The SEU is treated as a fault because it is the cause of a potential computation error.

This thesis concentrates on the effects of the radiation induced errors on software, hence it is needed to understand how an error translates to higher layers, hence the fault is going to be considered at a high level.

#### Transient Faults

Transient faults manifest in the software as Single Event Upsets (SEUs), where a stored bit in the memory or registers is flipped. Being "transient" it does not cause disruption to the system in a permanent way, meaning that a simple restart of the system or an overwrite of affected bit can be enough to make it disappear. These types of faults are called "soft" errors. Reason being the fact that the faulty internal state of the system can be reset or overwritten.

**Recovery** A possibility to recovery the correct internal state of the system consist in a simple re-execution or rollback from a known golden checkopoint [32][30].

#### Permanent Faults

A permanent fault is a long-lasting defect where a specific hardware resource no longer functions correctly regardless of software intervention. Permanent faults are also known as "hard" errors. As the name suggests they are long-lasting or permanent defects in the hardware structures. These faults typically result from manufacturing defects or physical

damage. Hard errors can have serious consequences for the system's stability and it is likely that there will be data corruptions, or even severe system failures [11]. Unlike transient faults these errors persist through reboots and power cycles.

**Recovery** Software can not fix a permanent fault. To fix a permanent fault either an hardware repair or replacement is needed [11].

### 2.3.2. Single Bit-Flip Abstraction

A fundamental step in abstracting the effects of radiations on microelectronics is to consider the interaction of particles with the silicon as a localized inversion of the binary state. This approximation is used to mimic the impact of particle strikes in both storage elements and computational logic [42]. As described in a 1994 paper [24], single bit-flips are the most common result obtained from striking silicon using heavy ions radiation. From the paper it can be understood that, even though it is possible to affect multiple bits, the most likely result is having only one affected bit.

#### Limitations

While single bit-flip fault models have their uses, the single flip model represents a limitation. In complex parallel accelerators like GPUs a single strike in control logic, such as a hardware scheduler, can make the modeling of the fault not accurate since the programmer doesn't have access to those elements [43]. Another limitation is given by the fact that even though single and double-bit flips are accurate for the main memory structures such as register file and caches, they might be unrealistic to model faults in the computing cores or control logic as shown in [46].

## 2.4. Fault Injection Methodology

Fault injection is a fundamental method to understand the behavior and weaknesses of systems[37].

#### Principles of Fault injection

**Acceleration of Rare Events** In a natural environment the probability of a strike to the silicon coming from a particle is extremely low[45][47][27] To evaluate how systems handle faults their occurrence has to be artificially accelerated, allowing researchers to observe in a few hours or days what might take years to occur naturally[48].

**Need for Large Campaigns** Because the "fault injection space", which is the combination of every possible bit at any possible instruction injected at any possible time, is large, researchers must conduct extensive campaigns. These campaigns can involve hundreds of thousands of individual experiments to ensure that the results are representative of the actual hardware behavior[48].

## Fault Injection Metrics

**Controllability** The ability of the tool to specify the time and location of the fault [24]. High controllability allows researchers to select freely the injection point. This means that the fault has to be injected with the utmost spacial and temporal precision. Simulation-based fault injections offer the highest controllability compared to the heavy ion radiation, where the random nature of the particles strikes is impossible to control and predict.

**Repeatability** The ability of the tool to repeat the exact same experiments multiple times [48]. This is one of the main metrics used in the validation phase of a framework. It is a metric highly interconnected with the Controllability, which is a prerequisite. The reason is that, being the Repeatability the ability of a tool to perform the exact same experiment multiple times with identical parameters and conditions. It is used to check if a software fix works. For example if an injected fault causes a system crash, the developer might try to fix the software to handle the fault. To verify the fix the developer has to be able to inject the exact same fault, spatially wise, at the exact same cycle.

**Reproducibility** The ability of the tool to recreate the same result [48] obtained in a previous experiment. As the other two metrics this one's intertwined with the previously two. In particular the main difference between Reproducibility with the Repeatability is the focus that the aforementioned focuses on the output.

**Intrusiveness** The impact that the tool has on the system not taking into consideration the fault injection [22][26]. Usually the intrusiveness is measured by the performance overhead introduced by the injection mechanism. Tools that rely on debuggers that use serial communication, such as KGDB [22] introduce significant runtime overhead. On the other hand tools that use hardware breakpoints are considered non intrusive because they allow the system to run at native speeds with negligible overhead [22].

## Fault Outcome Classification

There can be mainly 4 types of fault outcomes during the execution of a process.

**Masked Fault/Benign exit (No Effect)** These are faults that haven't been propagated during the execution. This can have mainly two reasons. The first is that the fault landed in the memory, but the bit-flip happened in a piece of code on the stack or heap that got already used, hence it won't modify in any way the execution of the process unless there's a read from that zone. The second possibility is that the fault happened in a register that is not being used for any kind of calculation in that moment because it already used the data written on it, hence a simple update of the data on the register with an overwrite caused by a new calculation is sufficient to not propagate the fault.

**Silent Data Corruption (SDC)** The program terminates normally without any error signals, but the output is not the expected one.

Consider a simple integer addition, 2+2, which should return 4. The value 2 is represented in binary as `0010`. A single bit-flip in a register holding this operand could invert the least significant bit, changing the value to `0011`. If the fault happens before the execution of the arithmetic operation, the CPU computes 3+2 and outputs 5.

These faults usually are the worst type of fault in safety-critical applications because they can lead to catastrophic failures of the systems without the system ever realizing it was in an erroneous state[16].

**Crashes** These errors are detected unrecoverable errors. In this case the program terminates with an exit code that specifies what error the program encountered during the execution. An example is the injection of a fault inside the RSP register, which will likely lead the program to either an exit by **segmentation fault**.

**Hangs** Hangs are the types of errors that occur when the program fails to terminate within a predefined time limit [42], making the programs execute indefinitely.

# 3 | Related Works

The validation of reliability in safety-critical systems requires rigorous testing methodologies to assess how hardware faults propagate to the software layer. Over the last decades, the research community has developed a classification of Fault Injection techniques ranging from physical irradiation of chips to high-level software emulation. This chapter reviews the state of the art in fault injection, categorizing methodologies into Physical Injection, Simulation-based Injection and Software-Implemented Fault Injection (SWIFI) to provide a comprehensive context for the proposed kernel-level framework.

## 3.1. Physical and Hardware-Based Injection

### 3.1.1. Radiation Testing

The most accurate method for reproducing Single Event Effects (SEEs) involves exposing a system to radiation sources. Standard procedures include heavy-ion bombardment in particle accelerators or neutron irradiation [24]. While this methodology captures the true physical behavior of the device, including latch-ups and permanent damage, it is prohibitively expensive [15]. Furthermore it suffers from low controllability, as researchers cannot precisely target specific logic gates or clock cycles. Moreover radiation testing only reveals errors when they manifest at the system output, not allowing researchers to track the fault's propagation through the microarchitecture [43].

### 3.1.2. Pin-Level Injection

Alternative hardware techniques such as MESSALINE [5] and RIFLE [29] involve Pin-Level injection, where faults are induced by manipulating the voltage on accessible pins. However, in the modern context there's a lack of access to allow effective Pin-Level injections since not all of the pins are reachable in modern ICs [48], rendering these techniques less viable for validating complex System-on-Chip (SoC) architectures.

## 3.2. Microarchitectural Simulation-Based Injection

To overcome the cost and accessibility issues of physical testing and the extreme slowness of RTL simulation, researchers employ full-system microarchitectural simulators [11][40]. These tools allow for high controllability and consequently repeatability and reproducibility, enabling the analysis of how faults propagate from specific hardware structures up to the operating system. Microarchitectural simulation in general has a higher fault coverage compared to the software output [40].

### 3.2.1. Cycle-Accurate Simulation

Several frameworks have been developed to balance the trade-off between simulation speed and hardware fidelity.

#### PTLsim

PTLsim [50] is a cycle accurate simulator designed exclusively for the x86-64 instruction set, capable of modeling superscalar out of order cores with a configurable level of detail ranging from RTL-level up to full-speed native execution on the host CPU [50].

PTLsim operates as a virtual machine, allowing it so switch between a "native mode" that allows it to run directly on the hosts' CPU and a cycle accurate mode that uses PTLsim's processor core models [50]. This framework allows developers to simulate a system with high precision. With PTLsim it is not necessary to emulate billions of instructions during the initialization phase of a benchmark, in fact it is sufficient to specify a point on top of the benchmark's main loop before which the simulation will execute the code in native mode until the trigger point is hit [50].

While highly detailed, the computational cost for RTL-level simulations is still very high and this framework lacks the ability to model other architectures in addition to x86.

#### GemFI

GemFI [37] is a fault injection tool built on top of the widely used gem5 [8] simulator.

GemFI injects faults into specific pipeline stages and can inject at cycles specified by the user. To mitigate the slow speed of simulation, GemFI implements a checkpoint mechanism. The user can first decide a checkpoint by running the simulation up to the point when fault injection is activated. Then for the following experiments the checkpoint is used as a starting point for the simulation, skipping the initialization of the process to

speed up the experiment.

Despite introducing a very small overhead on top of gem5 of just 3.3%, GemFI is still a simulation based on gem5, which naturally still introduces a significant overhead over native execution.

### MaFIN and GeFIN

These two frameworks have been developed to compare the two most popular simulators, gem5 [8] and MARSS [38]. MaFIN is based on the MARSS simulator, which uses QEMU [7] for emulation, while GeFIN uses gem5. MaFIN adds on the MARSS simulator from extra data arrays to cache models in order to enable precise memory injections [23]. While this made possible the comparison between MARSS and gem5 on x86, it reduced the MARSS throughput of 40% [23].

GeFIN on the other hand already models data arrays, thanks to gem5, and didn't need extra work to allow precise injections [23].

The comparative study revealed that internal implementation cause significant divergence in results. For instance MaFIN uses the QEMU hypervisor to handle system calls. When QEMU is invoked memory accesses go to main memory instead of caches, masking L1D cache faults [23]. On the other hand, gem5 doesn't use an hypervisor and this sort of masking of faults does not happen [23]. This highlights that simulation results are approximations heavily dependent on the model implementation decisions.

### gem5-MARVEL

Gem5-MARVEL [11] addresses the problem of heterogeneous computing by creating a framework for injecting faults into Systems on Chip that contain both CPUs and Domain Specific Accelerators. Gem5-MARVEL combines gem5 [8] for CPU modeling with gem5-SALAM [41] to accelerate the modeling of accelerators to have the ability to inject into the entire SoC.

Naturally the downsides are still caused by the simulation approach to the problem, but because Gem5-MARVEL uses pre-RTL simulation for accelerators the simulation itself is significantly faster than an RTL-based one [11].

### SOFIA

SOFIA [16] is a framework designed to automate soft error analysis and mitigation for complex software stacks running on multicore systems. It integrates gem5 [8] for applica-

tions and architectures profiling and M*DEV [21] for high-speed instruction-accurate fault injection [16]. This allows it to run massive injection campaigns at a faster throughput compared to pure cycle accurate tools. The M*DEV module trades accuracy for speed leading to a mismatch of 7.5% when faults are injected in general purpose registers in RTL-based simulations.

## AVGI

AVGI [36] is a statistical fault injection tool built upon an already cited tool, GemFI [37]. The primary objective of this tool is to solve the main bottleneck in fault simulation campaigns, which is the required time to run exhaustive Fault Injection campaigns. In particular AVGI achieves speedups up to 337× by not finishing the simulation for every experiment of the fault injection campaign.

1. First of all there's a configuration phase in which the fault list and target hardware are defined
2. Secondly the microarchitecture is simulated on top of gem5 [8] considering all the architecture details.
3. Then the fault is injected and the simulation is continued until the fault becomes visible to the software. As soon as the fault is visible to the software it is categorized.
4. In this phase the fault effects are automatically categorized as Masked, Crash or Silent Data Corruption. This is achieved without full program execution by relying on the observation that the final outcome of a specific fault manifestation is relatively uniform across different workloads. AVGI applies pre-calculated statistical weights to the detected IMM class to predict the final outcome immediately [36]

Even though this tool can reach speedups of up to 337× this is not always the case. In fact the use of a simulation still causes a huge bottleneck in the execution time. As stated in [36]

> *since Benign faults do not corrupt any microarchitectural or software resource, there is no deviation in the commit stage (i.e., there is no IMM categorization). Therefore, the simulation should run until the end of the program. The number of Benign faults is relatively large, although it is different across workloads and hardware structures. This means that there are many injected faults that still require end-to-end simulated executions.*

## 3.3. Software-Implemented Fault Injection (SWIFI)

SWIFI techniques emulate hardware faults by modifying the architectural state (memory and registers) of a running system via software. These methods offer a balance between the speed of native execution and the controllability of simulation.

### 3.3.1. Compile-Time and Assembly Instrumentation

One approach is to modify the program before the execution. This allows for fine grained control where faults are injected relative to the program structure.

#### LLFI

LLFI is an open source fault injection tool that operatees at the LLVM [25] compiler's Intermediate Representation (IR) level [28]. Thanks to the integration with LLVM it is compatible with a variety of programming languages and architectures [25].

LLFI first takes the program's source code, then it converts it to the IR using LLVM and only at this point LLFI injects a fault injection function at specific program points specified by the user [28]. A profiling pass is then executed to determine the total execution count of these points and only after there's the execution used to inject faults into selected targets at runtime.

While being configurable, high-level IR injection is less accurate than low level assembly instructions because IR instructions do not map one on one to machine code [28], or as stated in another paper [17] the accuracy of IR FI compared to assembly level FI is significantly less accurate due to assembly level binary instructions and optimizations executed in the back end that are not available at the IR level. Furthermore it is found that for Wei et al. [49] LLFI is accurate for emulating SDCs, but it is not accurate for emulating crashes.

#### FERRUM

FERRUM [19] is a software based fault detection technique that implements error detection by duplicating instruction (EDDI) at the assembly level.

In particular this work uses EDDI by replicating instruction with the use of under utilized SIMD registers to improve the performance over traditional EDDI [19].

A great limitation is given by the fault model used in FERRUM, in fact the focus is only on single bit flips caused by transient hardware faults in computing components of the

processor, without considering the faults in memory or cache because it is assumed that they are already protected by ECC. This limitation is common among EDDI techniques as stated in [19].

### 3.3.2. Runtime and Debugger-Based Injection

#### GOOFI

The Generic Object Oriented Fault Injection Tool (GOOFI [3]) is a portable tool designed to adapt to various target systems. GOOFI is rendered portable by the use on Java as a programming language and an SQL portable database. The tool, although it is possible to expand the used injection technique, in the cited version can use a pre-runtime SWIFI, which means that faults are injected into the target program before it begins the execution.

GOOFI is composed of 3 layers:

- **GUI layer**: provides a graphical interface for the user to configure and monitor campaigns
- **Operational Layer**: contains the core injection logic to perform fault injections
- **SQL Database**: stores the target system data, campaign configurations and logged system state

The use of pre-runtime SWIFI can inject faults into memory locations that do not hold live data, meaning that they will be overwritten before being read, making the experiment useless [3]. Furthermore GOOFI lacks an automated data analysis tool for analyzing the data collected during the campaigns, the user is in fact required to write a tailor made SQL query to the DB [3].

#### UN-FIT

UN-FIT [4] is an open source fault injection tool designed for ARM and RISC-V processors that uses the QEMU emulator. It connects to the QEMU simulation via the GNU Debugger (GDB).

The tool manages the injection campaign by setting breakpoints at specific addresses. When a breakpoint is triggered it pauses the execution, modifies the processor register via GDB commands and resumes execution.

The fact that it relies on QEMU rather than more accurate models is a limiting factor since not all the details of the processor's microarchitecture are captured by it, therefore, the

results are approximations of architectural state corruption rather than precise hardware behavior [4].

In conclusion UN-FIT, while being faster than RTL still has overhead cause by both QEMU and the handling of GDB signals and breakpoints that make it slower than native execution.

### 3.3.3. Kernel-Level Injectors

Kernel-level injectors operate as drivers or kernel modules, granting them privileged access to system resources to test operating system stability.

#### KITO

KITO [48] is a tool designed to inject faults into the Linux kernel data structures. It is the most similar tool compared to the one that is being presented in this thesis not only because of the implementation via loadable kernel module (LKM), but it uses software timers to trigger an interrupt, at which point it performs an atomic bit-flip in specific memory locations.

The main difference with the proposed kernel module is that KITO performs fault injections into memory locations belonging to the kernel space (e.g. `task_struct`), while the one that is being proposed injects into user applications both in memory and registers. Another similarity is in the time resolution being restricted to the granularity of software interrupts/timers, meaning faults cannot be injected into specific pipeline stages of a single instruction.

The main disadvantage that KITO has is the fact that it mainly tests faults only in memory locations that belong to the kernel space, making it not suited for fault injection campaign where the target program is a user application.

#### FIFA

FIFA [22] is a kernel level fault injection framework specifically for ARM based embedded Linux systems. It uses two methods:

- The first one is a **KGDB based injection**, which uses the kernel debugger to insert breakpoints and manipulate data without kernel modification
- The second one uses a **Hardware Breakpoint Injection**, which uses the ARM processor's debug resources to stop execution and inject faults.

The KGDB method, which uses two devices, the target device and the host machine of FIFA, has as a benefit the fact of not introucing much overhead, but it is slow because of serial port communication with the host machine [22]. On the other hand the hardware breakpoint method is limited by the number of available debug registers on the ARM chip [22], limiting the complexity of injection campaigns.

## 3.4. Discussion and Gap Analysis

The review of the state of the art highlights a fundamental difficulty in the verification of safety-critical systems: there's always a tradeoff between Controllability, Throughput and how close the test is to physical reality. In fact no single existing methodology satisfies the requirements for validating complex software stacks running on COTS hardware that at the same time give a near native throughput for the experiment's execution.

### 3.4.1. The Throughput vs. Accuracy Dilemma

Physical irradiation and pin-level injection provide the highest level of accuracy regarding hardware behavior. However, their stochastic nature and high cost make them unsuitable for the iterative debugging required during the software development life cycle. Conversely, simulation-based approaches (RTL, GemFI, PTLsim) offer excellent observability but suffer from distinct performance bottlenecks. With state of the art simulators running orders of magnitude slower than native silicon, validating a full OS boot or a complex image processing algorithm becomes computationally unfeasible.

### 3.4.2. The Gap in Software Implemented Fault Injection (SWIFI)

To achieve the throughput required for large-scale campaigns, SWIFI is the preferred approach. However, current SWIFI tools leave a specific gap regarding intrusiveness and scope:

- **Intrusiveness of Instrumentation:** Tools like LLFI operate at the Intermediate Representation (IR) level. This creates a dependency on source code availability considering that it is not always possible to access the source code of a software, especially if it is proprietary.
- **Overhead of Debuggers:** Tools relying on GDB or ptrace (like UN-FIT) incur significant overhead due to the constant context switching between the debugger and the target process.

- **The Kernel-User Disconnect:** This is the most critical gap. Existing kernel-level injectors (like KITO) utilize the speed and privileges of the kernel to inject faults, but they focus almost exclusively on verifying the Operating System itself (e.g., corrupting `task_struct` or kernel slabs). They are not optimized to target the virtual memory space or register context of a specific user application.

### 3.4.3. Positioning the Thesis

This thesis positions itself specifically to bridge the gap between Kernel-Level Privileges and User-Space Verification to enable the verification of COTS binaries.

Unlike KITO, which targets the kernel, the proposed framework uses the kernel only as a delivery mechanism to inject faults into user applications.

Unlike GDB-based tools, it operates within the kernel space, eliminating heavy context switches and enabling near-native execution speeds.

Unlike LLFI, it operates on the running binary, removing the need for source code.

This unique positioning allows the framework to generate a reliability profile for standard COTS software, enabling the "NewSpace" paradigm of using high-performance, non-hardened hardware with robust software mitigation.

# 4 | Implementation

## 4.1. Architecture Overview

The architecture of the proposed fault injection framework is designed as a hybrid system that bridges two distinct execution domains: the User Space and the Kernel Space. This separation of roles allows the tool to leverage the flexibility of high-level application logic for orchestration, while utilizing higher privileges of the operating system's kernel for low-level fault injection.

Unlike distributed fault injection setups such as FIFA [22], which require separate host and target machines, this system is developed to run on a single machine, requiring only a Linux environment (Kernel version 6.14 or higher) and standard dependencies.

The core design philosophy of this tool is trying to maximize the portability of the program while still allowing for granular operations during the fault injection.

To achieve this, the architecture is split into 3 logical components:

- **Controller**: Acts as the orchestration engine. It handles tasks such as parsing JSON configuration files, managing campaigns and analyzing results.
- **Kernel Module**: This component, implemented as a Loadable Kernel Module (LKM) acts as a bridge to the hardware. It allows modifications of the memory and registers of the target application directly.
- **Victim Process**: This process is executed normally, unaware that it is being monitored and manipulated by the injector.

## 4.2. Userspace Controller

The controller serves as an orchestrator for the fault injection framework. It is implemented in C++ and is responsible of parsing the campaign configurations, communicating injection parameters via the `ioctl` interface and managing the lifecycle of hanged processes.

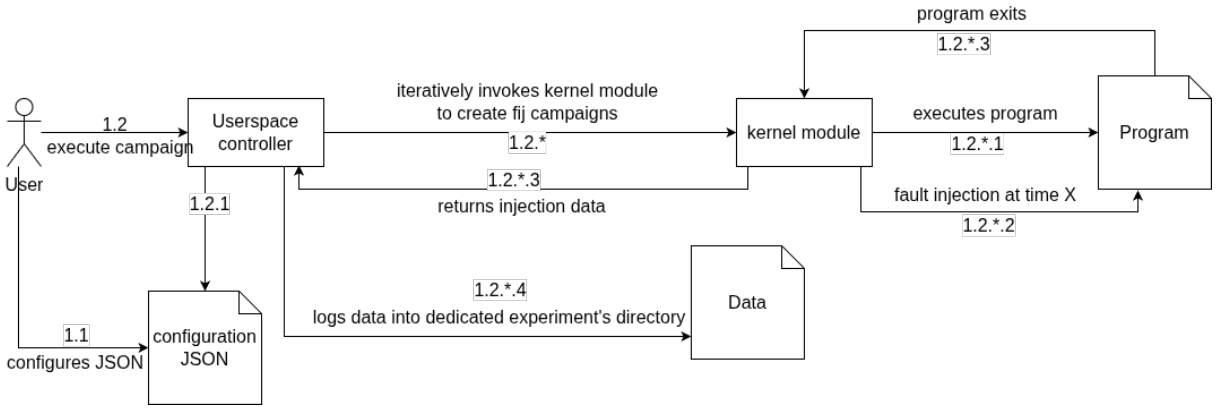


Figure 4.1: Collaboration Diagram

The controller execution flow is divided into four main logical stages:

- **Stage 1: Configuration Loading.** The system ingests and validates the user-defined parameters for the campaign.
- **Stage 2: Baseline Profiling.** The target is executed without faults to establish a performance baseline.
- **Stage 3: Injection Campaign Execution.** This is the core phase where faults are injected (in parallel if specified by the user).
- **Stage 4: Data Analysis.** The aggregation and interpretation of the logs generated during execution.

### 4.2.1. Configuration and Architecture Abstraction

The Controller operates by ingesting a JSON configuration file which defines the specific targets and parameters for the injection. A major challenge in fault injection is handling the specific register naming conventions of different processor architectures. To ensure cross-architecture portability, the system incorporates an abstraction layer for register mapping.

The function `reg_name_to_id` is pivotal for this abstraction, it maps human readable register names such as "rax" on x86-64, "x0" on ARM64 or "a0" on RISC-V, to internal identifiers defined in the shared header between Userspace application and Kernel Module.

The decision to implement the Userspace Controller in C++, rather than using other high-level languages like Python or Java, was driven by the necessity for strict data structure alignment. By using C++, the system can maintain a single shared header file (`.h`) for the majority of the system's critical data structures, streamlining maintenance and new

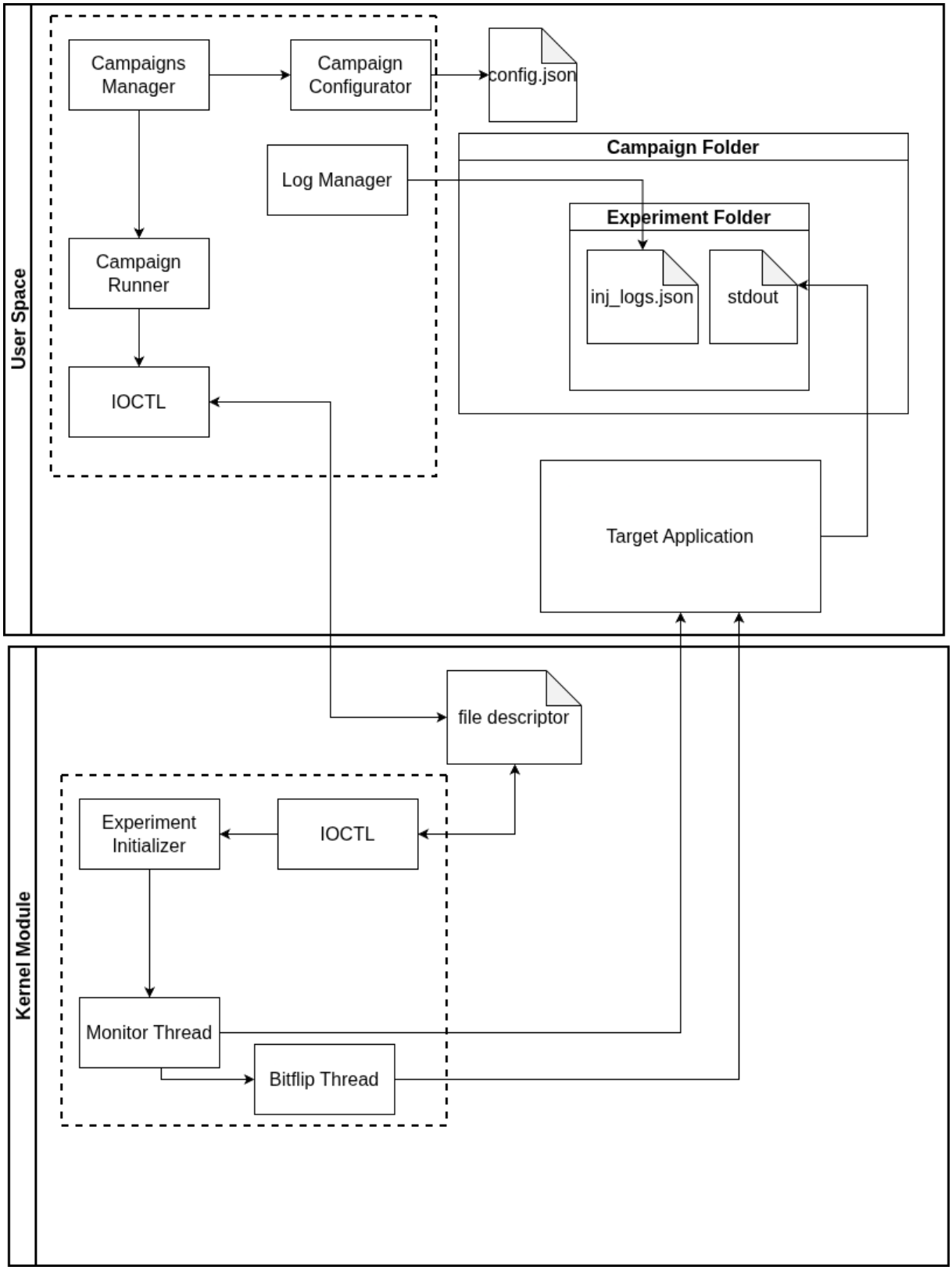


Figure 4.2: Architecture Overview

feature implementations. If the Userspace Controller were implemented in a different language, the developer would be required to create object definitions that exactly mirror the C structures used in the Kernel Module. This duplication would not only increase the complexity of software maintenance but also heighten the risk of serialization bugs, padding issues, and data misalignment. Furthermore it would introduce an unnecessary translation layer when marshaling data between the user space and the kernel via IOCTL.

To strictly enforce this binary compatibility, the project employs a modular build system reliant on multiple Makefiles. A specific header file, `fij.h`, acts as the "Single Source of Truth" for the entire system. This file is explicitly referenced as a dependency in both the Kernel Module's build configuration and the Userspace Controller's Makefile. By compiling both components against the exact same physical header file, the system guarantees that the memory layout of shared structures (such as `fij_params` and `fij_result` shown in Figure 4.6) remains identical across the User/Kernel boundary, eliminating the possibility of version mismatches during compilation.

The configuration loader also supports template substitution (e.g., replacing `{base_path}` placeholders), enabling flexible path management for test executables.

### 4.2.2. Configuration Specification

To allow different testing scenarios, the framework exposes a set of configurable parameters via a JSON-formatted file `config.json`.

#### Global and Target Parameters

The primary configuration fields control the campaign orchestration and target selection. These are shown in Table 4.1.

Table 4.1: Global and Target Configuration Parameters

| Parameter | Type | Description |
|---|---|---|
| `workers` | Integer | Number of parallel threads used to execute injections concurrently. |
| `base_path` | String | A helper variable for constructing absolute paths (e.g., `{base_path}` placeholder). |
| `baseline_runs` | Integer | Number of safe executions (Phase 1) used to profile execution time. Defaults to 100 if omitted. |
| `defaults` | Object | A set of injection parameters applied to all campaigns unless overridden. |
| `targets` | Array | A list of victim programs to test. |
| `path` | String | The absolute path to the target executable. |
| `args` | Array | A list of argument objects, allowing multiple input sets per target. |

## Injection Logic and Probability

The framework uses a weighting system to determine the fault model (Register vs. Memory) for each run. This is controlled by the `weight_mem` parameter. The selection logic is defined as follows:

- **Register Weight ($W_{reg}$)**: Fixed at 1.
- **Memory Weight ($W_{mem}$)**: User-defined via `weight_mem` (defaults to 0).

The probability of a memory injection $P(Mem)$ is calculated as:

$$P(Mem) = \frac{W_{mem}}{1 + W_{mem}}$$

Conversely, the probability of a register injection $P(Reg)$ is:

$$P(Reg) = \frac{1}{1 + W_{mem}}$$

For example, setting `weight_mem` to 19 results in a 95% probability of memory injection ($\frac{19}{20}$). Table 4.2 provides common configuration values.

Table 4.2: Injection Probability Distribution

| weight_mem | Register % | Memory % | Ratio (Reg:Mem) |
|---|---|---|---|
| 0 | 100% | 0% | Only Registers |
| 1 | 50% | 50% | 1 : 1 |
| 3 | 25% | 75% | 1 : 3 |
| 9 | 10% | 90% | 1 : 9 |
| 19 | 5% | 95% | 1 : 19 |

## Advanced Injection Controls

The parameters in Table 4.3 can be used for granular control over the experiment. These can be placed in a `defaults` section.

| Parameter | Type | Description |
|---|---|---|
| `runs` | Integer | The number of injection iterations to perform. |
| `weight_mem` | Integer | The weight of memory used to choose where to inject the fault as explained in 4.2.2 |
| `only_mem` | Boolean | Flag (0 or 1). If 1, bypasses probability logic and forces 100% memory injections. |
| `reg` | String | Specifies a target register by name (e.g., "rax", "sp"). Architecture dependent. |
| `reg_bit` | Integer | Forces the bit-flip at a specific index (0-63). |
| `pc` | Integer | (Deterministic Mode) The relative instruction pointer offset in bytes where the fault should trigger. |
| `min_delay_ms` | Integer | The lower bound for the random time delay window. |
| `thread` | Integer | The specific thread index (e.g., 0 for main thread) to target. |
| `all_threads` | Boolean | Flag. If 1, the injector attempts to fault all accessible threads. |

Table 4.3: Fault Injection Configuration Parameters

### 4.2.3. Campaign Orchestration

The core logic resides in the `run_injection_campaign` function. To ensure the injection in the process, the controller utilizes a two phase execution strategy:

1. **Phase 1**: Before starting the injection campaign the controller executes the target process a number of times specified by the parameter (`baseline_runs`), sending the execution to the kernel module with the flag `no_injection` set to true. This phase serves the purpose of warming up the system for the execution of chosen process and to calculate the average execution time duration of the victim process. This average execution time is used to compute the `max_delay_ms`, which defines the temporal window within which the random injection delay must occur, making more likely that faults are injected during the active life of the process.

2. **Phase 2**: The controller iterates through the requested number of injection runs. It uses **OpenMP** to manage worker threads, allowing for parallel execution of campaigns.

   A feature of the implementation is the guarantee of a number of successful injections in the system equal to the number of runs that the user selects at the beginning of the campaign. Inside this phase the controller checks the result of every execution.

   During the phase 1 an average execution time of the application is computed in order to have an injection temporal window. A common issue in dynamic fault injection is the race condition where the target program exits **BEFORE** the kernel module acts. While the specific kernel-side mechanism is detailed in Section 4.3, here it is sufficient to know that the Controller accounts for this by checking the `fault_injected` flag.

   If the kernel module reports that the injection was not successful by setting the flag `fault_injected` to 1, the controller discards that run and immediately retries the iteration until a valid fault injection is confirmed.

   For each iteration, a software timer is started based on the `max_delay_ms`. As detailed in Section 2.4, one of the possible outcomes of a bit-flip is a system HANG, where the process becomes unresponsive. In embedded scenarios, a hardware watchdog would reset the board. However, for this testing framework, a full system reboot is unfeasible as it would halt the campaign and corrupt the dataset.

   The solution found in this software is a custom "software home-made" watchdog. For each iteration, as soon as the function that starts the injection experiment is

ran, a timer is started. Using the profiling data from Phase 1, the timeout threshold is set to 10 times the `max_delay_ms`. A factor of 10 was selected to provide a conservative safety margin, accounting for potential system overhead during injection without triggering false positives. If this time elapses without process completion, the Userspace Controller sends a specific `KILL` command to the kernel module, ensuring the zombie process is terminated and the kernel module resources are released for the next run.

### 4.2.4. Kernel Communication Interface

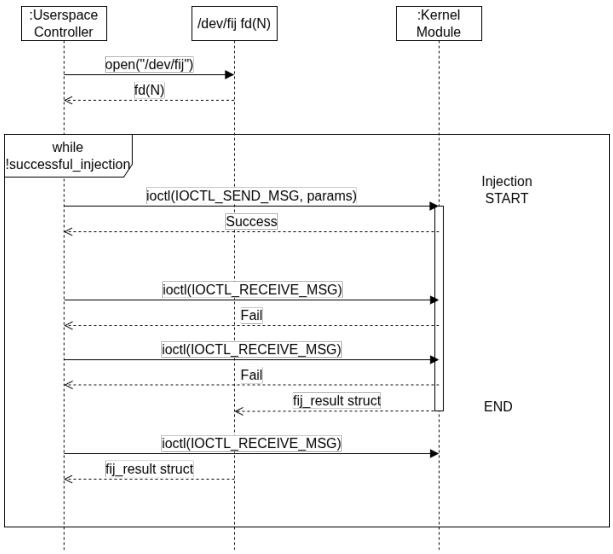


Figure 4.4: Userspace-Kernelspace Communication Sequence Diagram

Communication with the kernel module is handled via `ioctl`, a mechanism that allows communication between user space and kernel space already explained in Subsection 2.1.5.

The introduction of multi-threading to parallelize injection campaigns necessitated a redesign of the IOCTL mechanism. Initially, the system operated sequentially, relying on a single synchronous blocking IOCTL function. However, parallel execution required an asynchronous approach to prevent thread contention. The Userspace Controller now utilizes a distinct set of IOCTL messages to manage this concurrency:

**IOCTL_SEND_MSG** This message is transmitted from the Userspace Controller to the kernel module to initiate a session. It opens a unique file descriptor for each thread's IOCTL message. This design allows for true parallelization without the risk of data overwrites or race conditions between threads. It utilizes the `fij_params` structure (see

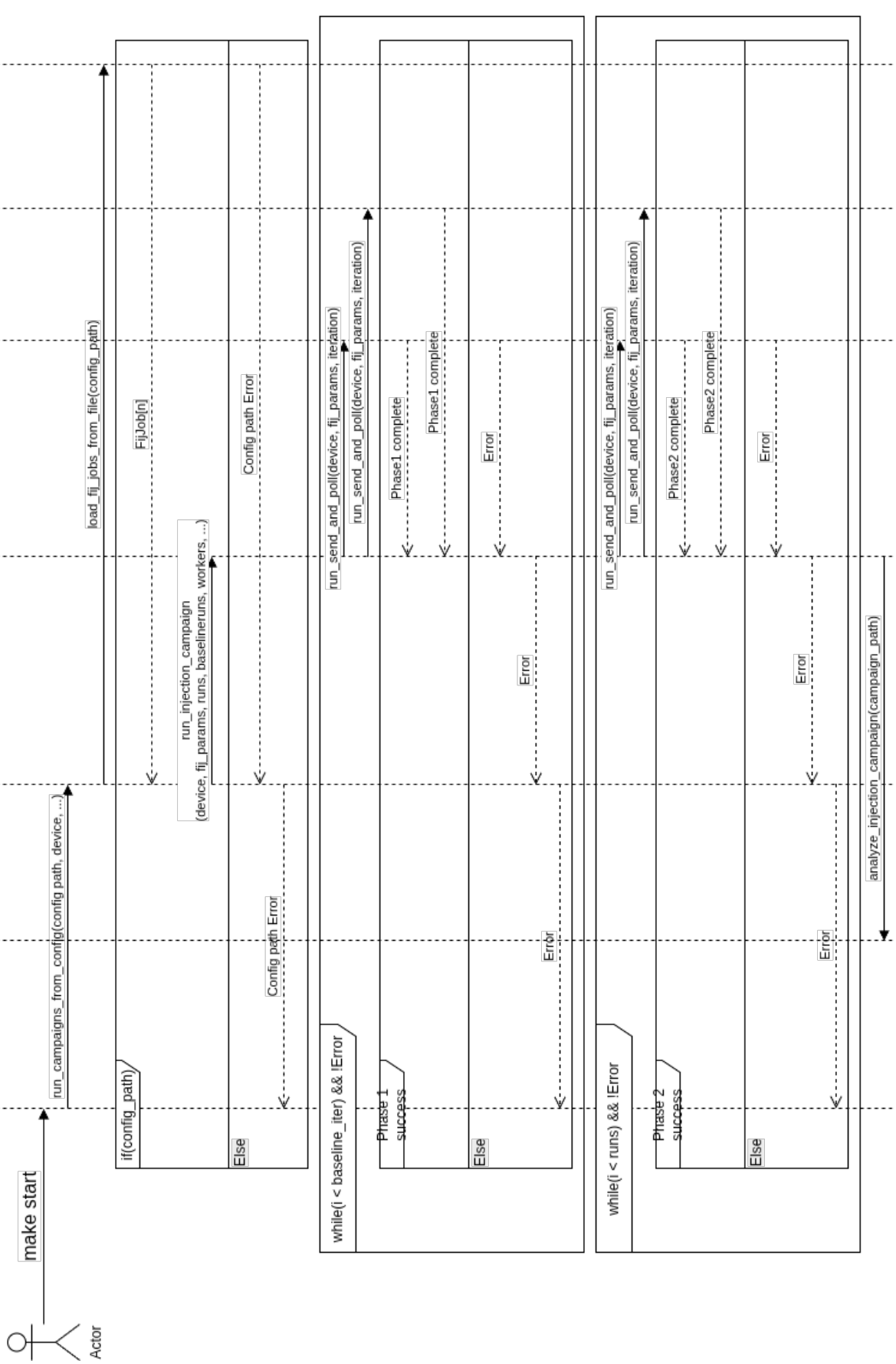


Figure 4.3: Userspace Sequence Diagram

Figure 4.6) to encapsulate and pass all necessary parameters (PID, register target, delay, etc.) required for the injection

**IOCTL_RECEIVE_MSG** This message is used to retrieve the `fij_result` (see Figure 4.6) structure from the kernel after execution. The retrieve of the result is performed by the Controller with a Client-Server like communication protocol. After the `IOCTL_SEND_MSG` is sent, the Controller starts polling on the assigned file descriptor to retrieve the `fij_result` structure. This structure serves as the primary log for the experiment, containing the detailed CPU register and memory states both before and after the bit-flip, a boolean success flag, and the final exit status of the process (e.g., whether it crashed, exited normally, or required termination).

**IOCTL_KILL_TARGET** This is the message that is sent from the Userspace to the Kernel module to kill a process. This message is triggered specifically when the software watchdog detects a timeout, instructing the kernel to forcefully terminate a process identified as being in a hanged state. An example of use is shown in Figure 4.5

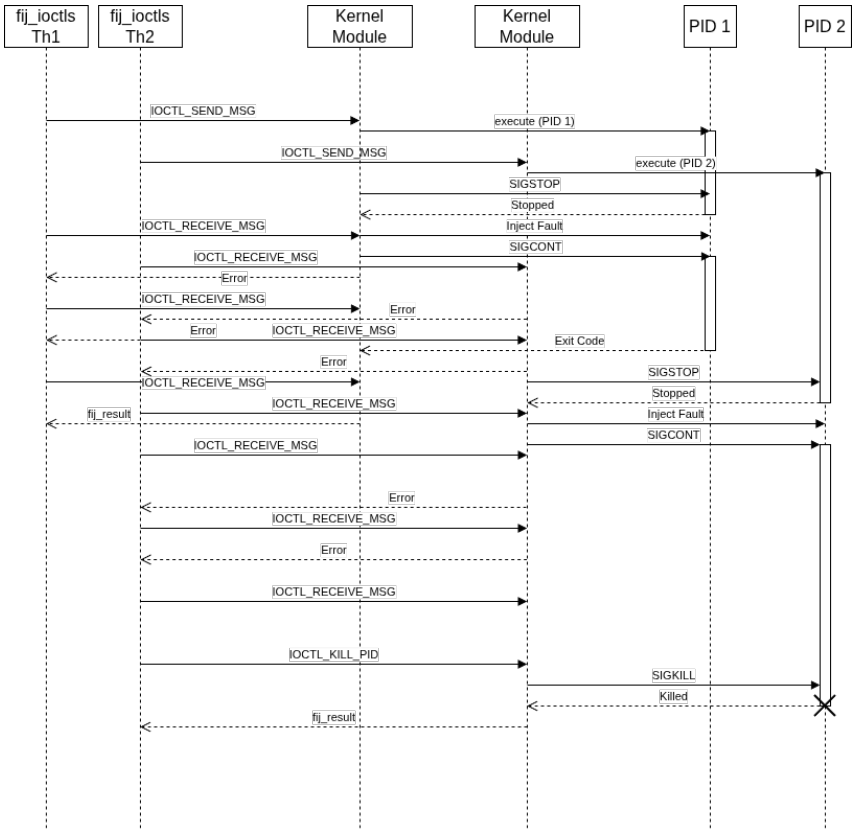


Figure 4.5: Process Kill Sequence Diagram

### 4.2.5. Data Logging

To facilitate rigorous post-campaign analysis, the Controller generates a deep, structured directory hierarchy derived from the target binary name and its input arguments. The tool automatically provides a dedicated folder where all logs regarding a single experiment's execution are archived. In this context, an "injection campaign" is rigorously defined as the execution of a specific number of runs, on a specific program, with a fixed set of input arguments.

The logging architecture is organized as follows:

- The root logs folder contains sub-directories for each unique campaign configuration.
- **Run Directories:** Inside a campaign folder, directories named `injection_X-1` store data for each specific run $X$.
- **Baseline Data:** A `no_inj` directory stores the logs from the baseline profiling runs (Phase 1), serving as the control group.
- **Differential Analysis:** A `diff` directory is generated to highlight anomalies. This folder mimics the campaign structure but only contains sub-directories (`diff_Y`) for runs where a failure was detected (see Section 2.4).

Within the `diff` directory, the system generates a comprehensive CSV summary file. This file aggregates the campaign results, listing every run that resulted in a failure, the specific classification of the failure, and the exact injection location (register or memory address). The file concludes with a statistical summary that aggregates the total failure count by type and further categorizes them by the fault location, enabling rapid statistical analysis of the campaign's impact.

#### Data Analysis

After the fault injection campaign terminates a fundamental part that improves the framework's quality of life in its use is the automatic analysis of the just executed campaign. During this part of the execution the Controller automatically analyzes all the experiments that resulted in a failure. For each and every one of the experiment that result in an actual failure a `diff_Y` directory is created, as it was already explained in this section.

The analysis of the experiment uses multiple logs and files to determine if there was a failure during the execution.

1. **Hang Detection**: First of all the log.json file is checked. Here there are multi-

ple parameters to check if the experiment exited with an error. The first checked parameter is the `hanged` parameter, which is a boolean that is flipped to 1 if the controller sent the **IOCTL_KILL** signal. Obviously if this is true the experiment is cosidered as hanged.

2. **Crash Detection**: Another parameter considered is the `exit_code`. If it is different from 0 the experiment is considered a crashed.

3. **SDC Detection**: Lastly, the system identifies *Silent Data Corruptions* (SDCs). This is the most complex failure mode to categorize, as the process terminates successfully (exit code 0) but produces incorrect results. To detect this, the Controller compares the artifacts generated by the injection run against the "Golden Output" files created during the Baseline Profiling (Phase 1).

   The comparison logic operates automatically by doing a strict bit-to-bit comparison if the file is a standard file, such as the stdout.txt. If the output that is being analyzed is an image, the system generates a differential image map to visualize the corruption. This mask renders pixels as **black** where the injection output matches the Golden Output, and **white** where discrepancies occur. This allows for an immediate visual assessment of how the bit-flip impacted the data processing algorithm.

### 4.2.6. Parallel Execution and Session Isolation

As mentioned in Section 4.2.3, the framework utilizes **OpenMP** to spawn multiple worker threads in Userspace, allowing for high-throughput injection campaigns. However, implementing this parallelization required a specific design choice regarding the User-Kernel communication channel to avoid race conditions.

If the Userspace Controller were to open the character device `/dev/fij` once globally and share the file descriptor among all threads, the kernel module would be unable to distinguish which thread was requesting which injection. This would lead to a cross-talk scenario where parameters set by thread A could be overwritten by thread B before execution, or results form experiment A could be returned to thread B.

To solve this, the implementation enforces a **session per thread** architecture, encapsulated within the `run_send_and_poll` function. The isolation is achieved by executing the `open()` system call inside the scope of the worker function rather than at the process initialization in order to ensure that every concurrent injection thread acquires a unique private file descriptor (fd). In this way each time a fd is opened, a fresh `fij_ctx`, whose content is shown in Figure 4.6, is allocated for the communication of data be-

tween userspace and kernelspace, after which every subsequent IOCTL communication is performed on this thread-local file descriptor.

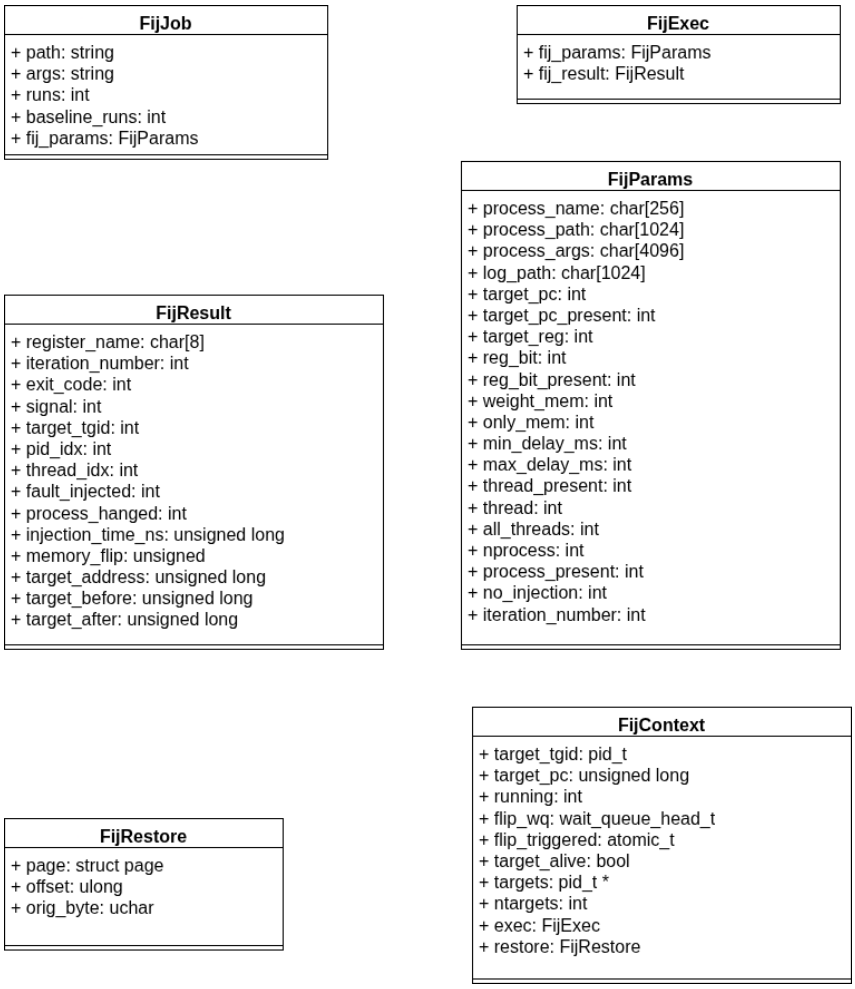


Figure 4.6: Class (Struct) Diagram

When the experiment concludes, the `close(fd)` call triggers a function in the kernel that cleanly destroys the context associated with that specific run.

## 4.3. Kernel Module

The Kernel Module is the essential part of this tool. It is that part of the proposed software that takes some parameters as input, parameters that will be later discussed, and executes programs to then do random injections in them. It's the heart of the project and many software engineering design choices were taken in order to produce a module that satisfied all the needed requisites for a fault injector.

### 4.3.1. Module Architecture and Lifecycle Management

The kernel component is implemented as a standard Linux Character Device Driver (see Subsection 2.1.5) utilizing the `miscevice` framework. Upon initialization via the `fij_init` function, the module registers the device node `/dev/fij` within the kernel's Virtual File System (VFS). This registration allows the Userspace Controller to interact with the kernel module using standard POSIX file operations such as `open`, `close` and `ioctl`.

A critical aspect of the module's lifecycle management is the handling of the injection context, represented by the `fij_ctx`. When a user opens the device file, the `fij_open` function allocates a dedicated context instance. This structure is the central repository for the session's state, containing the target process parameters, synchronization primitives and workqueue structures. Specific attention was paid to the clean shutdown of these resources. The `fij_release` function ensures stability by enforcing a strict cleanup order:

1. **Thread Termination**: The monitor and bit-flip threads are explicitly stopped.
2. **Work queue Flushing**: Calls to `cancel_work_sync` are issued for both the `inject_work` and the `uprobe_disarm_work`. This was a mandatory action needed to prevent the "use after free" kernel panics, where a scheduled work item attempts to access memory freed during the module unload.
3. **Resource Deallocation**: Finally the uprobes are disarmed and the context memory is freed.

### 4.3.2. User-Kernel Communication Interface

The communication bridge between the Userspace Controller and the Kernel Module relies entirely on the `ioctl` (Input/Output Control) system call. This interface is defined in the `fij_unlocked_ioctl` function, which acts as a dispatcher for the commands sent by the controller.

To ensure system stability, all data marshaling between the two execution domains is guarded by the `copy_from_user` and `copy_to_user` primitives. These kernel functions verify that the memory pointers provided by the userspace application are valid and accessible, preventing segmentation faults within the kernel space.

The interface implements three primary commands:

**IOCTL_SEND_MSG: Configuration and Execution Trigger** This command represents the "Start" signal for an injection run. When the kernel receives this opcode, it performs two atomic actions:

1. **State Ingestion**: It copies the `fij_params` structure from userspace into the kernel context. This structure contains all the necessary metadata, such as the target binary path, the chosen fault model (register vs. memory), and the delay parameters.
2. **Execution Launch**: It immediately triggers `fij_start_exec`, which spawns the target process and initializes the monitoring infrastructure.

**IOCTL_RECEIVE_MSG: Asynchronous Polling** As discussed in the Userspace Controller section, the architecture supports parallel campaign execution. To make this possible, the Receive command is non blocking in nature. When the controller queries for results, the kernel checks the state of the internal completion variable `ctx->monitor_done`.

- If the injection and monitoring cycle is not yet finished, the module immediately returns `-EAGAIN`. This signals the userspace thread to yield and retry later, implementing an efficient polling mechanism.
- If the cycle is complete, the kernel copies the populated `fij_result` structure back to the user, containing the final exit code, the exact injection timestamp, and the before/after state of the modified resource.

**IOCTL_KILL_TARGET: Intervention with Hanged Process** This command provides the implementation for the "software watchdog" described in Section 4.2.3.

If the Userspace Controller detects that a target process has exceeded its allowed execution window (indicating a hang), it issues this command. Internally, the handler executes `fij_send_sigkill`, which looks up the `task_struct` associated with the stored Target Group ID (TGID) and issues a `SIGKILL`. This ensures that zombie processes are forcefully reaped and that threads, which might be sleeping waiting for a target that will never wake up, are manually stopped.

### 4.3.3. Target Execution and Process Control

Another challenge found trying to realize this kernel-based fault injection tool was trying to control the execution of the target application. The main problem was about the initialization of the project. A requisite was allowing a user to execute a given program, allowing bit-flips to be inserted from the instant 0 until the very end of the execution of

the program. At first it was thought of allowing the user choose a specific PID to inject in, but this added a non-controllable component since processes can spawn with PIDs of unknown value. The only solution was to pass the kernel module the executable of the program to test, making the module execute the process all by itself.

This method allows for more control, allowing for the immediate interception fo the execution flow upon creation, arming probes and initializing all the needed timers before the application logic begins.

To achieve this the module utilizes a Linux Kernel's API, User Mode Helper. Specifically the `call_usermodehelper_setup` and `call_usermodehelper_exec` functions. The implementation defines a custom initialization callback, `helper_child_init`, which is executed in the context of the new process **before** the binary image is loaded via exec. This callback is necessary for especially two setup tasks:

1. **I/O Redirection**: It opens the log file path provided by the Controller using the `filp_open`. It then manipulates the file descriptor table of the new process, utilizing the `fd_install` to map Standard Output (stdout) and Standard Error (stderr) to this log file. This captures all application output transparently, without requiring changes to the target source code.
2. **Execution Freeze**: To prevent the newly spawned process from running before the injector is ready, the helper issues a `SIGSTOP` signal. This places the newly created process in a `TASK_STOPPED` state immediately, giving time to the other parts of the injector to be properly initialized.

This solves the fundamental problem of allowing the injector to inject the fault right before the start of the program. The used method makes possible the injections at *any* time during the process, even at the very beginning, before the execution of any instruction.

Once the target is successfully launched and frozen, the module spawns a dedicated kernel thread via `fij_monitor_start`. This Monitor Thread acts as the lifecycle manager for the victim process and of the Bitflip Thread. It enters a sleep loop on a waitqueue, waking only when the target process changes state (exits or dies) or when the module receives a `KILL` command. Considering that the process might exit before the Bitflip Thread has the opportunity to inject the fault, the Monitor Thread handles it's death too as shown in Figure 4.7.

This design decouples the injection logic from the process supervision. If the target crashes or exits normally, the monitor thread captures the exit code into the shared result structure and performs the necessary cleanup, such as disarming probes and reverting memory

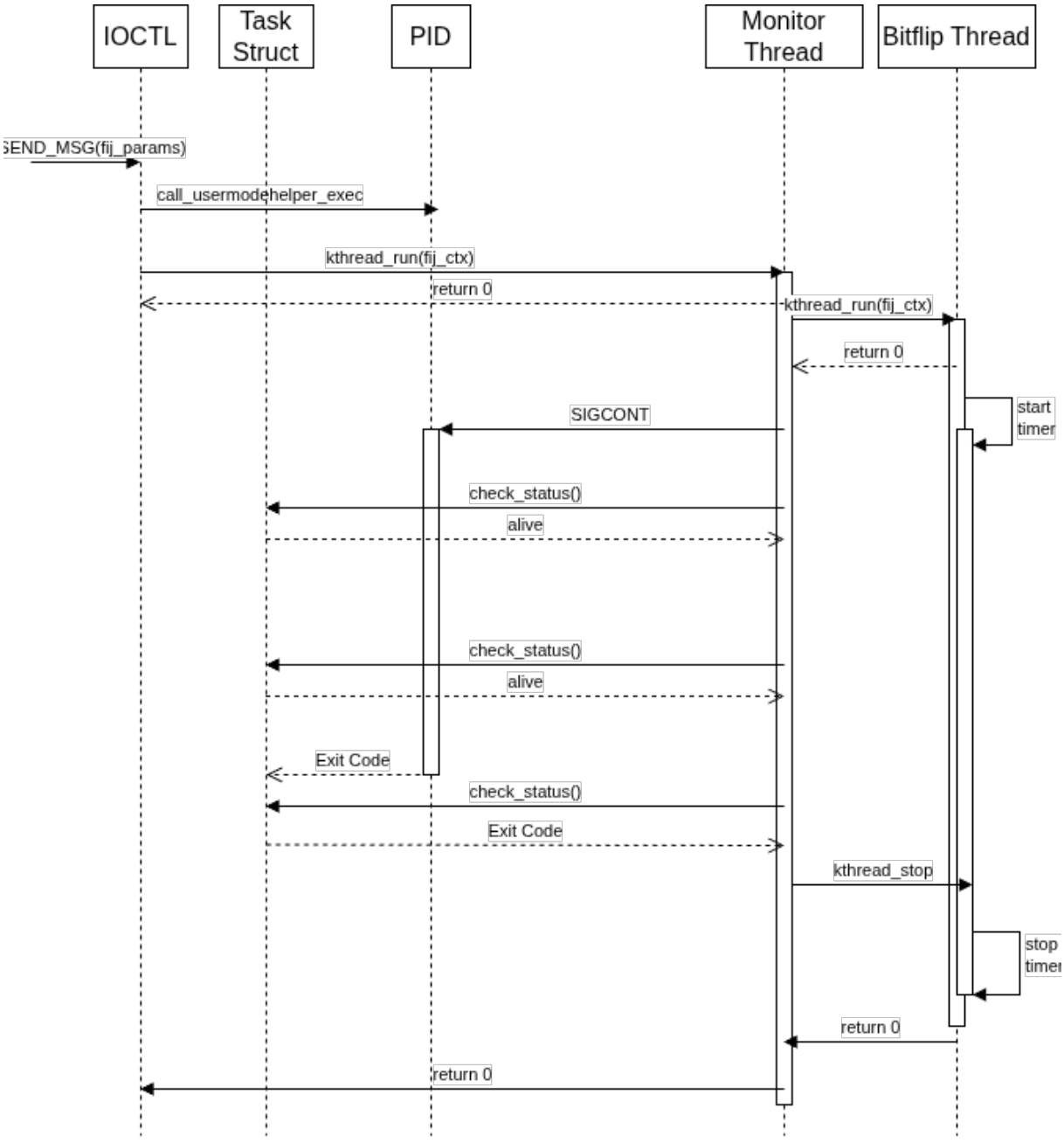


Figure 4.7: Sequence Diagram of an Unsuccessful Injection

changes before signaling the Userspace Controller via the parameter `monitor_done`.

Only after the monitor thread and the remaining injection logic (bit-flip timers or probes) are fully initialized does the module issue a `SIGCONT` to the target, allowing the victim process to begin its execution.

### 4.3.4. Target Traversal and Thread Selection

During the first few iterations of the development, the requirement was to follow the execution of just the process directly spawned by the kernel module during the execution, as explained in 4.3.3.

This meant being able to save the PID of only the spawned process and allowing the fault injection at runtime of only that specific PID. Obviously the idea was more complex than this, but because a SCRUM approach was followed for the development of this tool, it was necessary to first build the MVP functions.

After the main requirements were satisfied, the focus shifted in making the injection land not only in a register of the thread 0 or the memory of only the spawned PID, but in a register of *any* thread of *any* process that got executed by the specified program. This requirement was deemed necessary because modern workloads are rarely composed of a single execution thread. Ofter multiple threads are spawned and even new processes are spawned via `fork()`. To create an effective fault injector it was necessary to make it have a complete view of the victim's process hierarchy.

Navigating this hierarchy in the kernel is full of concurrency hazards. A target process might spawn a new thread or terminate exactly at the moment the injector is attempting to list them. To manage this dynamic environment without incurring the heavy performance penalty of locking the global task list, the implementation relies heavily on the **Read-Copy-Update** (RCU) synchronization mechanism.

**Hierarchy Traversal** The traversal logic is encapsulated in the `fij_collect_descendants` function. This function is responsible for flattening the target's process tree into a linear array of Target Group IDs stored in the injection context (`ctx->targets`). The traversal algorithm operates under the `rcu_read_lock()`, which guarantees that the `task_struct` pointers remain valid while being read, even if the tasks are concurrently exiting. The system performs a recursive pre-order walk of the process tree:

1. It resolves the root task using the target PID.
2. It iterates over the `parent->children` list using `list_for_each_entry_rcu`, which

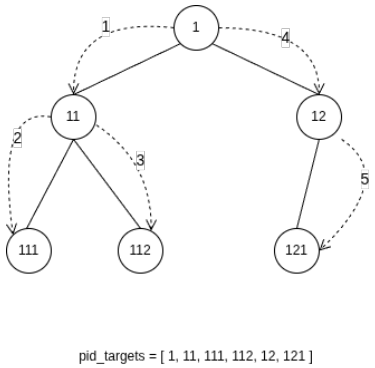


Figure 4.8: Descendants Collection Pre-Order Walk

safely follows the linked list pointers protected by RCU.

3. For every child found, it recursively collects its descendants.

Now that the processes are flattened into an array it is possible to easily choose one by selecting a random index in that array.

Once the target process is chosen the system must select a specific thread for injection. The code implements a lot of filtering checks to ensure system stability.

Similarly to the selection of the process the thread selection algorithm employs a two pass approach.

- **Pass 1**: The code iterates through the thread group under RCU protection to count the number of valid, living userspace threads.
- **Pass 2**: It generates a random index k and iterates the list a second time to locate the kth eligible thread.

Once the victim thread is identified the function calls `get_task_struct` to increment the task's reference count. This pins the task structure in memory, preventing it from being freed by the kernel before the injection logic can execute.

### 4.3.5. Injection Triggers (Deterministic vs. Non-Deterministic)

A core requirement for the fault injection tool was the ability to support both deterministic and non-deterministic injections. This means being able to support exploratory testing, where a fault is injected randomly to discover new bugs and regression testing, where a specific fault must be reproduced exactly. To address this the kernel module implements two distinct triggering mechanisms that dictate *when* the fault is injected.

### Non-Deterministic: Time based injection

This mode is the one that allows for exploratory testing. This part is the one that is thought of being the most useful for general users since it is the mode in which the fault injector works by default.

In this mode, the injection occurs after a random delay within a temporal window defined by `min_delay_ms` and `max_delay_ms`. The parameter `min_delay_ms`, which defaults to 0, is user specifiable in the config.json file, while the `max_delay_ms` is found in by the userspace controller as already explained in 4.2.3.

This logic is handled by the `bitflip_thread_fn`, a kernel thread dedicated to the timing and execution of the fault.

To ensure high precision without incurring unnecessary system overhead, the implementation employs an adaptive sleep strategy based on the duration of the chosen time window.

- **Window $<$ 500ms**: For thight temporal windows the module utilizes a High Resolution Timer. The function `fij_sleep_hrtimeout_interruptible_ns` puts the thread into a `TASK_INTERRUPTIBLE` state and schedules a high precision wakeup. This minimized oversleeping, ensuring the fault occurs when intended. This design choice was found to be critical to be able to successfully inject faults in short-lived processes.
- **Window $>=$ 500ms**: For longer durations, the module reverts to the standard `msleep_interruptible`. This relieves pressure on the high resolution timer subsystem as the granularity of a millisecond is sufficient for long running workloads.

### Deterministic: Event based Injection via Uprobes

For scenarios requiring exact reproducibility, the module leverages the Linux Kernel Uprobes. This allows the tool to pause execution and inject a fault precisely when the instruction pointer reaches a specific Virtual Address in the target binary. To be able to inject at a specific instruction, the user is requested only the "pc" parameter in the config.json file. The "pc" is the byte offset represented as an integer number of the difference between the first instruction of the program and the instruction the execution has to reach to start the fault injection sequence.

Implementing this feature within a kernel module presents a specific address translation challenge: Uprobes are registered agains a *physical file offset* (inode + offset), whereas userspace tools typically operate on *Virtual Addresses*. To solve this incompatibility an

interface function `fij_va_to_file_off` was made.

The `fij_va_to_file_off` function iterates through the Virtual Memory Areas of the target process to find the memory mapping backed by the executable file. Then it calculates the file offset as

$$\text{Offset} = (\text{Target\_VA} - \text{VMA\_Start}) + (\text{VMA\_Page\_Offset} \times \text{PAGE\_SIZE})$$

Once the offset is resolved, the module registers the probe using `uprobe_register`. When the target execution hits this address, the `uprobe_hit` callback is triggered in interrupt context.

In one of the first iterations of this fault injector the bit-flip happened inside this interrupt handler because it was a simple and atomic operation, but with the addition of the process traversing explained in 4.3.4, the operations became more complex, making the execution of the bitflip operation unsafe. Instead the handler executes a "handoff" pattern by delegating the bit-flip task:

1. The `uprobe_hit` sets an atomic flag and wakes up the sleeping bit-flip thread via queue.
2. The bit-flip thread, which was blocked waiting on this queue, wakes up.
3. Now the bit-flip thread proceeds to safely stop the threads and perform the injection, ensuring the fault coincides exactly with the execution of the target instruction.

This part of the code that once was completely deterministic now has a small variable delay. In fact the only problem with the deterministic approach to the injection is made by multiple constraints.

First of all the wake up of the thread is not instant, meaning that a delay might be added in the fault injection. Secondly the bit-flip operation on the process tree itself is not atomic and can't be atomic by definition. The reason for this affirmation is caused by the fact that first of all there are the process traversal operations, so the processes continue their execution while the kernel module traverses them, then a process is chosen, a thread of that process is chosen and only after all these operations have been completed does the execution of the chosen thread stop.

A possible modification to this logic could be done by allowing deterministic injections only on the process that is being launched, rendering again the bitflip operations small and atomic, allowing true deterministic injections.

### 4.3.6. The fault Injection Mechanism

The core requirement of this entire tool was the ability to introduce a single bit error into the execution state of the victim process. This implementation supports the *Single Event Upset* fault model capable of targeting both the CPU registers and the virtual memory space.

The injection logic is encapsulated within two primary functions:

`fij_flip_register_from_ptregs` and `fij_perform_mem_bitflilp`. Both of these operations require the target process to be in a STOPPED state. This atomiticy ensures that the execution context is stable and saved on the kernel stack before any modification is attempted.

#### Register Injection

Register injection was selected as the fault model because modifying the processor's execution state directly is highly effective at inducing system errors. Prior research utilizing QEMU based injection platforms has demonstrated that bit upsets in registers result in significant error rates. A clear example of this has been shown in [14], where faults injected into the Stack Pointer returned errors more than 80% of the time.

Register injection involves modifying the saved state of the processor's registers while the target task is paused. in the Linux kernel, when a task enters kernel mode, its user-space register values are pushed onto the kernel stack in a structure called `pt_regs`. The module operates by modifying this structure. When the process is resumed, the processor restores these corrupted values, effectively realizing the fault.

A significant engineering challenge is portability, as already discussed in 4.2.1. The layout of the `pt_regs` differs drastically between architectures. To solve this, the module implements a hardware abstraction layer via the `fij_arch_map` function.

This function acts as a translation unit. It accepts a generic architecture agnostic identifier and returns a pointer to the specific memory location within the `pt_regs` struct that holds that register's value. For example a request to corrupt the "accumulator" is mapped to:

- `regs->ax` on X86_64
- `regs->regs[0]` on ARM64
- `regs->a0` on RISC-V

This design decouples the injection logic from the underlying hardware, allowing for porta-

bility across different architectures.

After the register was accessed, being the tool a Single Bit Fault injection framework, a random bit of the register is chosen and flipped with a XOR operator. An example of this operation is illustrated in Table 4.4. If we take a binary value `1001` and apply a mask of `0100` targeting the bit 2, the XOR operation flips only the targeted bit $0 \oplus 1 = 1$, resulting in `1101`, while leaving all other bits unchanged.

Table 4.4: Example of Single Bit-Flip using XOR Operation

| **Description** | **Bit 3** | **Bit 2** | **Bit 1** | **Bit 0** |
|---|---|---|---|---|
| Original Register Value | 1 | 0 | 0 | 1 |
| Injection Mask | 0 | 1 | 0 | 0 |
| Operation | $\oplus$ | $\oplus$ | $\oplus$ | $\oplus$ |
| **Resulting Value** | **1** | **1** | **0** | **1** |

## Memory Injection and Page Restoration

Memory injection targets the Virtual Address Space of the victim process. Unlike register injection, which has a fixed target set, memory injection requires navigating the complex Virtual Memory Area structures of the Linux kernel.

The function `fij_perform_mem_bitflip` executes the following sequence:

1. **VMA Traversal**: It acquires the memory descriptor lock and iterates throug the process' VMAs. It explicitly filters out mapped I/O regions to prevent accidental writes to hardware peripherals, which could cause system instability.
2. **Random Selection**: A valid VMA is selected at random, followed by a random offset withing that area, determining the target Virtual Address.
3. **Bit-Flip Execution**: The module utilizes `access_process_vm` with the memory flags `FOLL_WRITE | FOLL_FORCE`. This kernel API allows the injector to bypass standard page protections to emulate hardware memory corruption.

**Handling File Backed Mappings** A critical aspect that is taken into consideration in this module is the possibility of executing the bit-flip in a file backed memory region. Being a file backed memory region the fault poses a severe risk: the operating system's dirty page write back mechanism could permanently persist the corruption to the disk file, ruining the binary for future runs and potentially crashing other system processes that share the same library.

To mitigate this, the module detects if a target VMA is file backed. As explained in 2.1.2 a fault injection in a file backed memory poses severe risks one of which is the Cross Process Contamination.

Since the physical page is shared, a bit-flip injected into a shared library for one target process is immediately visible to all other processes running on the system that use that library. If multiple fault injection experiments are running in parallel, a fault in a shared library in one experiment will corrupt the execution of simultaneous experiments, invalidating the isolation assumption.

The solution for this problem was solved using the following algorithm. If such region is targeted then, before performing the injection, the module picks the target physical page in memory, saves its unmodified content, performs the bit flip on a random bit of the chosen page and replaces that page with the new faulty page. After the target process has terminated, the scheduled restoration of the previously modified page begins and replaces that dirty page with its unmodified version.

In this way the corruption fault still affects the single run, ensuring that the fault can happen in any part of the virtual memory region dedicated to the process, while still allowing for tens of thousands of runs to not fail for a previously injected fault.

### 4.3.7. Concurrency and Synchronization Strategy

The kernel module operates in a highly concurrent environment, managing interactions between four distinct execution contexts: the Userspace Controller (via IOCTL), the internal Monitor Thread, the internal Bit-flip Thread, and the external Target Process. Ensuring data consistency and preventing race conditions, such as injecting a fault into a process that has already exited, was a non-trivial task that required a synchronization strategy.

The implementation relies on a combination of atomic state variables, wait queues, and Linux signal subsystem to coordinate these actors

#### Inter-Thread Coordinator

The lifecycle of an injection run involves a handoff of control between threads, managed primarily through wait queues and completion primitives:

- **Monitor Sychronization**: The monitor thread serves as the heartbeat of the session. Instead of busy-waiting for the target to finish, a `wait_event_killable_timeout` is used to puth the thread to sleep for short intervals (1ms). This puts the thread

to sleep periodically, without putting unnecessary stress on the CPU until specific events occur. The events that can wake the monitor thread are a change of state of the target or `KILL` request from the module.

- **Injection Handoff (`flip_wq` & `flip_triggered`):** In the deterministic Uprobe mode, the injection trigger occurs in an interrupt context (the probe handler), which cannot perform complex logic. To bridge this gap, the probe handler atomically sets the `flip_triggered` flag and wakes the `flip_wq`. The Bitflip Thread, which is blocked on this queue, wakes up immediately to perform the injection.
- **Result Signaling (`monitor_done`):** To synchronize with the Userspace Controller, the module employs a `completion` variable. When the Monitor Thread finishes its cleanup (detecting exit codes, restoring memory), it calls `complete(&ctx->monitor_done)`. The IOCTL handler checks this completion status before returning results to userspace, ensuring that incomplete or partial data is never reported.

### Stop, Flip, Resume Sequence

The most critical synchronization challenge involves the actual modification of the target's registers and virtual memory. To guarantee this consistency, the module implements a strict **Stop-Flip-Resume** protocol as shown in Figure 4.9:

1. **Group Stop**: The injector issues a `SIGSTOP` signal to the entire process group via `fij_group_stop`. This ensures that not only the main thread but all sibling threads are paused, preventing race conditions where one thread might modify the memory another is executing.
2. **State Stabilization**: Simply sending the signal is insufficient; the kernel takes time to deliver it and deschedule the task. The module enters a polling loop in `fij_wait_task_stopped`, checking the task state until it stabilizes at `TASK_STOPPED`. This acts as a barrier, guaranteeing that all the needed structures for the bit-flip are stable and safe to access.
3. **Atomic Injection**: With the world stopped, the module performs the bit-flip.
4. **Resume**: Finally, the module issues `SIGCONT` via `fij_group_cont`, returning the process to the `TASK_RUNNING` state and allowing the CPU to load the corrupted context.

This approach ensures that the fault is injected atomically with respect to the instruction stream of the victim application.

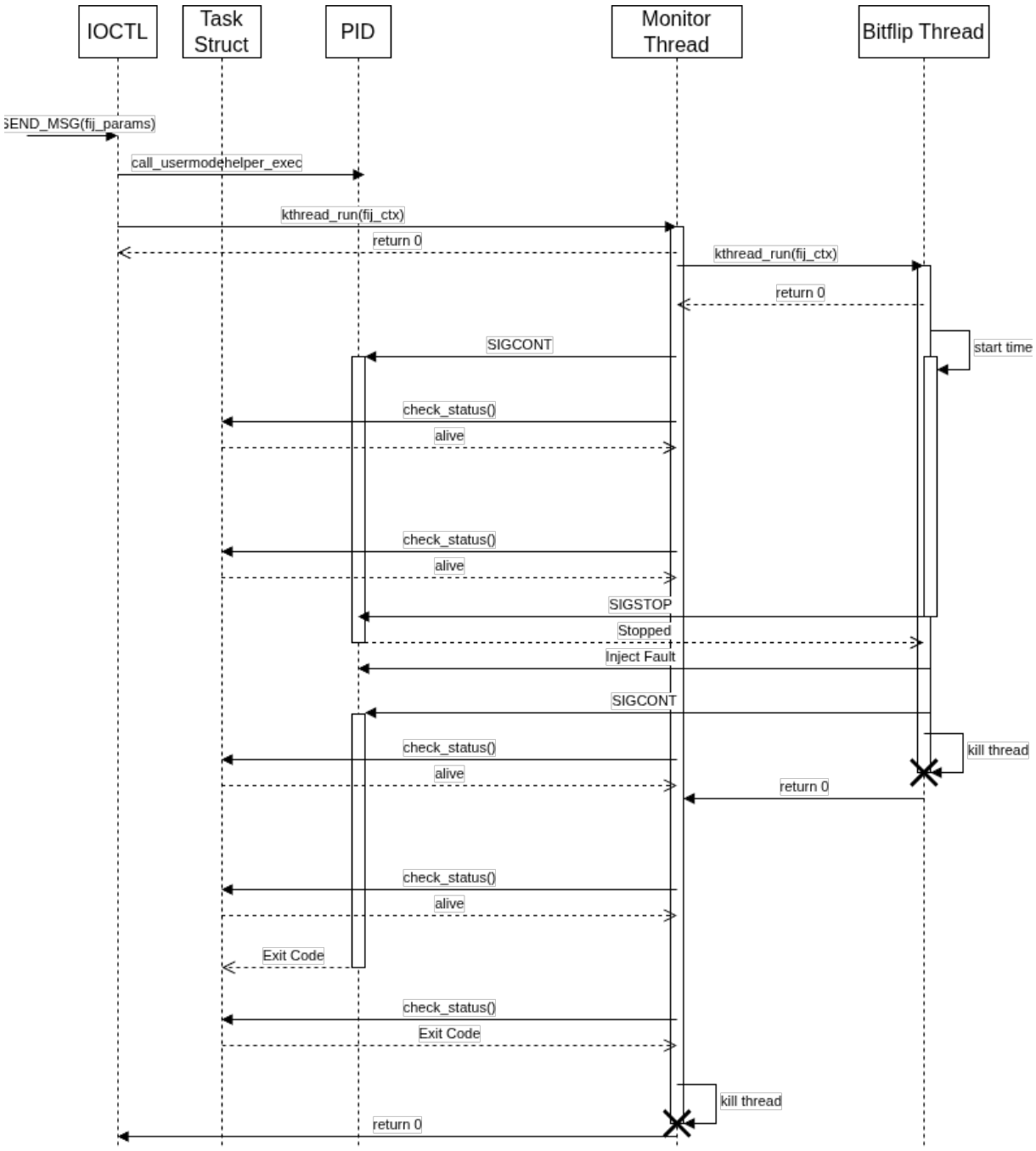


Figure 4.9: Sequence Diagram of a Successful Injection

# 5 | Validation and Results

This chapter contains the results that validate the proposed framework for fault injection. As it has already been explained in the rest of the thesis this work aims at reducing large campaign injection time on COTS systems by providing a framework that uses a Loadable Kernel Module as injection tool, this creates low overhead while still enabling enough controllability and reproducibility to execute successfully exploratory fault injection campaigns on specified executables.

In this chapter different results will be provided to demonstrate the accuracy, scalability and usability of the framework. Among the results there will be a demonstration of the accuracy, scalability and usability by comparison with an FPGA system that was tested on the same benchmarks (coremark and audiomark), the results about how well the framework's performance speeds up with the assignation of multiple workers to parallelize the injection campaigns and other examples about other Silent Data Corruption outcomes that have been found in the injection campaigns.

## 5.1. Validation

To validate the accuracy of the framework, the same benchmarks were subjected to fault injection using both the presented framework (targeting the HiFive P550) and a traditional FPGA-based hardware injector (targeting a BOOM architecture). To ensure statistical robustness the experiments where averaged over five campaigns of 10,000 injections per benchmark.

Tables 5.1 and 5.2 present the comparative results for the `coremark` and `audiomark` benchmarks, respectively. In both tables, the columns distinguish between the ground-truth hardware environment (FPGA) and the proposed software framework (XL-FI). The rows detail the specific outcome counts (Benign, SDC, and Hang+Crash), followed by a section dedicated to the statistical analysis metrics, including the maximum percentage point difference ($\Delta$), the Chi-squared value ($\chi^2$), and Cramér's V effect size.

The results indicate a strong correlation between the hardware ground truth and the

software framework. For `coremark`, the SDC rates remained highly comparable, with the FPGA recording 3.18% and `XL-FI` recording 3.82%. Similarly, in the `audiomark` benchmark, SDC events were rare in both environments with 0.06% for the FPGA vs 0.10% for `XL-FI`.

The largest difference was observed in the "Hang+Crash" category. In `audiomark`, the FPGA observed a failure rate of 22.04%, while `XL-FI` recorded 18.13%, resulting in a maximum difference ($\Delta$) of 3.91 percentage points. In `coremark`, this difference was even lower at 2.15%.

To rigorously quantify this similarity, a $\chi^2$ test was performed and Cramér's V effect size was calculated. The $\chi^2$ tests returned $p$-values $< 0.001$ indicating that the distributions are not statistically identical. The Cramér's V values were calculated at 0.031 for `coremark` and 0.049 for `audiomark`, which denote negligible associations.

This confirms that while a minor variance exists, probably caused by the use of different architectures, the proposed framework provides an accurate emulation of hardware injected faults with negligible deviation from the hardware baseline.

Table 5.1: 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) | |

## 5.2. Overhead Analysis

A small overhead over native execution of processes has been the main point in proposing this framework. Table 5.3 summarizes the performance comparison. The rows represent the different target workloads benchmarked (MNIST, YOLO, Image Processing, and CoreMark), while the columns report the execution time for both Native and Fault Injection (FI) tool scenarios, followed by the calculated overhead in absolute milliseconds and percentages. As it can be seen in the Table, the overhead is here measured as added

Table 5.2: 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) | |

latency to the execution of a process over natively vs. by the fault injection LKM.

It can be noticed that the added latency is on average less than 10%.

For shorter tasks it seems like the LKM impacts the performance of the process more since the operations executed on longer processes like `coremark` are amortized during the longer execution time.

This demonstrates the high performance of the framework. For shorter and lighter tasks it introduces a small overhead, mainly produced by fixed operations of spawning processes, but especially for computationally intensive workloads it demonstrates effective because the relative cost of these fixed operations becomes negligible.

This low level of intrusion on the system allows the execution of very large fault injection campaigns at near native speeds.

Table 5.3: 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.

## 5.3. Scalability Analysis

As shown in Table 5.4, the scalability of the framework is analyzed by varying the level of parallelism. The table lists the number of workers in the rows, while the columns present the total execution time and the relative speed-up calculated for two distinct hardware platforms: the Ryzen 9 7900X and the Threadripper 9970X.

The data shows that the framework's speed-up is linearly dependent on the number of workers used for the execution of the campaign time, meaning that the parallelization of the experiments works as it should.

Having tested the framework on both the Ryzen 9 7900X and Threadripper 9970X as shown in Figure 5.1 for the former and 5.2 for the latter, the results show similar curves regarding the diminishing time needed to execute a campaign relative to the number of workers used to execute it.

Table 5.4: Scalability results for a 1000-injection `coremark` campaign. Speed-up is computed with respect to the 1-worker configuration on the same platform.

| **Workers** | **Ryzen 9 7900X** | | **Threadripper 9970X** | |
|---|---|---|---|---|
| | **Time [s]** | **Speed-up** | **Time [s]** | **Speed-up** |
| 1 | 19430.80 | 1.00 | 14469.30 | 1.00 |
| 2 | 9398.14 | 2.07 | 7350.69 | 1.97 |
| 4 | 4826.51 | 4.03 | 3668.84 | 3.94 |
| 8 | 2470.32 | 7.87 | 1767.23 | 8.19 |
| 16 | 1321.02 | 14.71 | 920.20 | 15.72 |
| 32 | – | – | 471.86 | 30.66 |

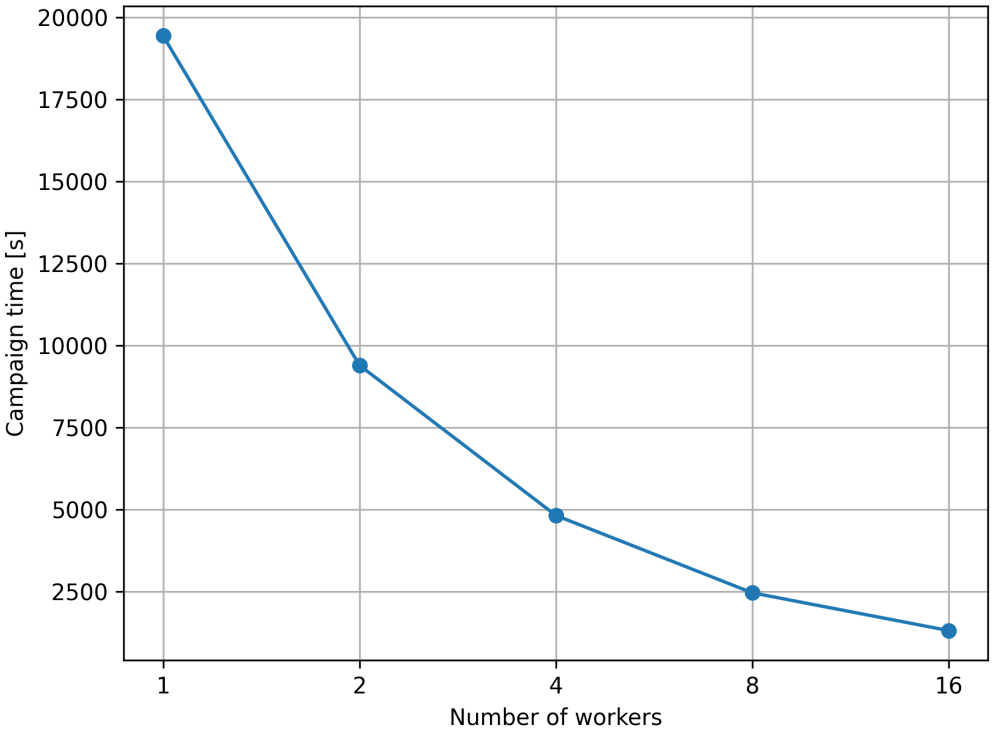


Figure 5.1: Campaign time versus number of workers on the Ryzen 9 7900X platform for a 1000-injection `coremark` campaign.

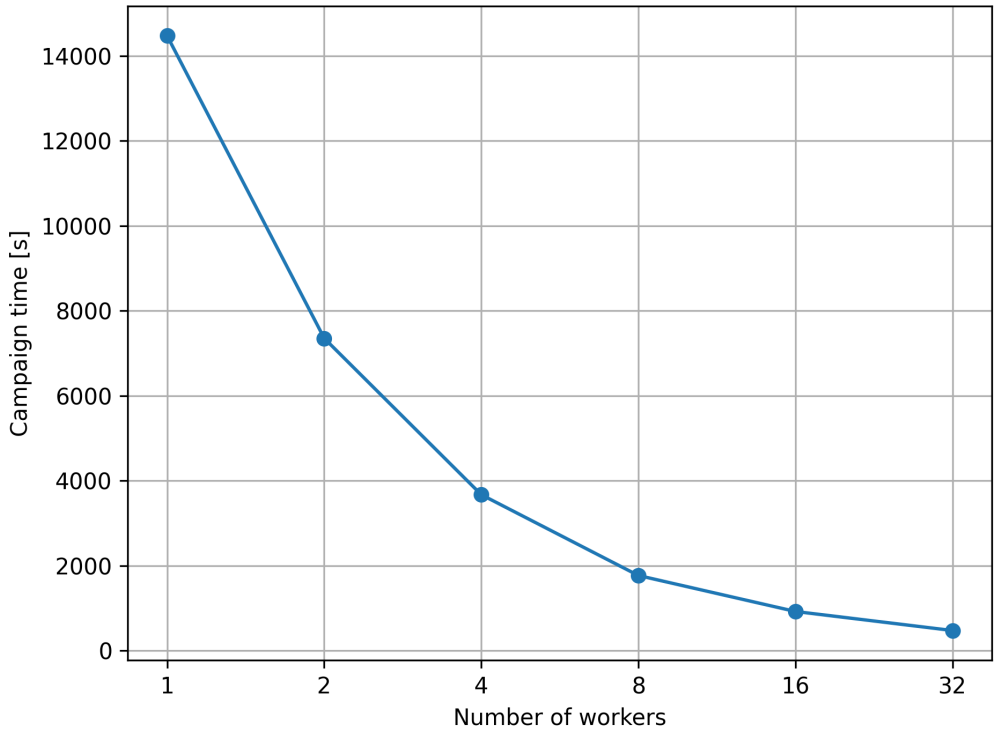


Figure 5.2: Campaign time versus number of workers on the Threadripper 9970X platform for a 1000-injection `coremark` campaign.

## 5.4. Large-Scale Campaign Analysis

The data collected in the large-scale campaigns is summarized in Table 5.5. The table is organized with the different workloads (executed for 100,000 runs) in the rows, while the columns detail the total execution time and the classification of the fault injection outcomes (Benign, SDC, Crash, and Hang)

An interesting insight that can be grasped from this data is the differences of results between MNIST and YOLO, two Neural Networks. In both of them the "No-effect" outcomes are very similar, with them happening in 75.33% of the times in MNIST and 76.44% in YOLO. The number of crashes and hangs are very similar too, being the aforementioned around 22% and the latter around 1.7%. The main difference is between the Silent Data Corruptions in the two Neural Networks.

This preliminary data shows that there might be some correlation in how big the Neural Network is and in how likely it is the fault injection in a portion containing live data. Basically the higher the "footprint" of the model on the hardware, the more likely it is for it to have cases of misclassifications. This reasoning is caused by the analysis

Table 5.5: 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` (100 000) | 110615.0 | 77728 (77.73%) | 4285 (4.29%) | 17752 (17.75%) | 235 (0.23%) |
| `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%) |

of the outcome breakdown by injection location shown in Table 5.6. In both YOLO and MNIST the outcomes are almost identical if the SDCs are not considered. The Silent Data Corruptions in MNIST happened for merely 0.03% times, which translates in just 25 misclassifications, while in YOLO, even though the difference is not by entire percentage points the SDCs happened in 0.2% of the times, meaning that in 199 cases there have been misclassifications.

This confirms what was found in [43], which is that the bigger the Neural Network the more likely it is to present Silent Data Corruptions. In the Santos et al. paper this is deducible from the PVF factor and it finds a practical confirmation in these data.

Table 5.6: Outcome breakdown by injection location.

| Workload | Run | Type | Total | Reg | Mem |
|---|---|---|---|---|---|
| `coremark` | 100 000 | Crash | 17752 | 16838 | 914 |
| | | Hang | 235 | 205 | 30 |
| | | SDC | 4285 | 3512 | 773 |
| | | Benign | 77728 | 29484 | 48244 |
| `audiomark` | 100 000 | Crash | 18683 | 18145 | 538 |
| | | Hang | 25 | 12 | 13 |
| | | SDC | 87 | 60 | 27 |
| | | Benign | 81205 | 31916 | 49289 |
| MNIST inference | 100 000 | Crash | 22910 | 21507 | 1403 |
| | | Hang | 1732 | 1719 | 13 |
| | | SDC | 25 | 15 | 10 |
| | | Benign | 75333 | 28903 | 46430 |
| YOLO inference | 100 000 | Crash | 21643 | 20774 | 869 |
| | | Hang | 1720 | 1719 | 1 |
| | | SDC | 199 | 46 | 153 |
| | | Benign | 76438 | 30256 | 46182 |
| Blur Filter | 100 000 | Crash | 23501 | 22325 | 1176 |
| | | Hang | 1829 | 1820 | 9 |
| | | SDC | 139 | 95 | 44 |
| | | Benign | 74531 | 26970 | 47561 |

## 5.5. Silent Data Corruption: Case Study

This section analyzes the impact of Silent Data Corruptions (SDCs) on the output quality of an image processing workload. In particular this case focuses on the blur filter applied to a satellite image.

Figure 5.3 shows a visual breakdown of a specific SDC event. The figure is organized into three images to facilitate a direct comparison between the expected and actual results. The left image displays the original input satellite image fed into the blur algorithm. The center image shows the corrupted output generated during the fault injection experiment. The right image presents a differential mask calculated by comparing the corrupted output against the golden (expected) output.

In the differential mask, black pixels indicate a perfect match between the experimental

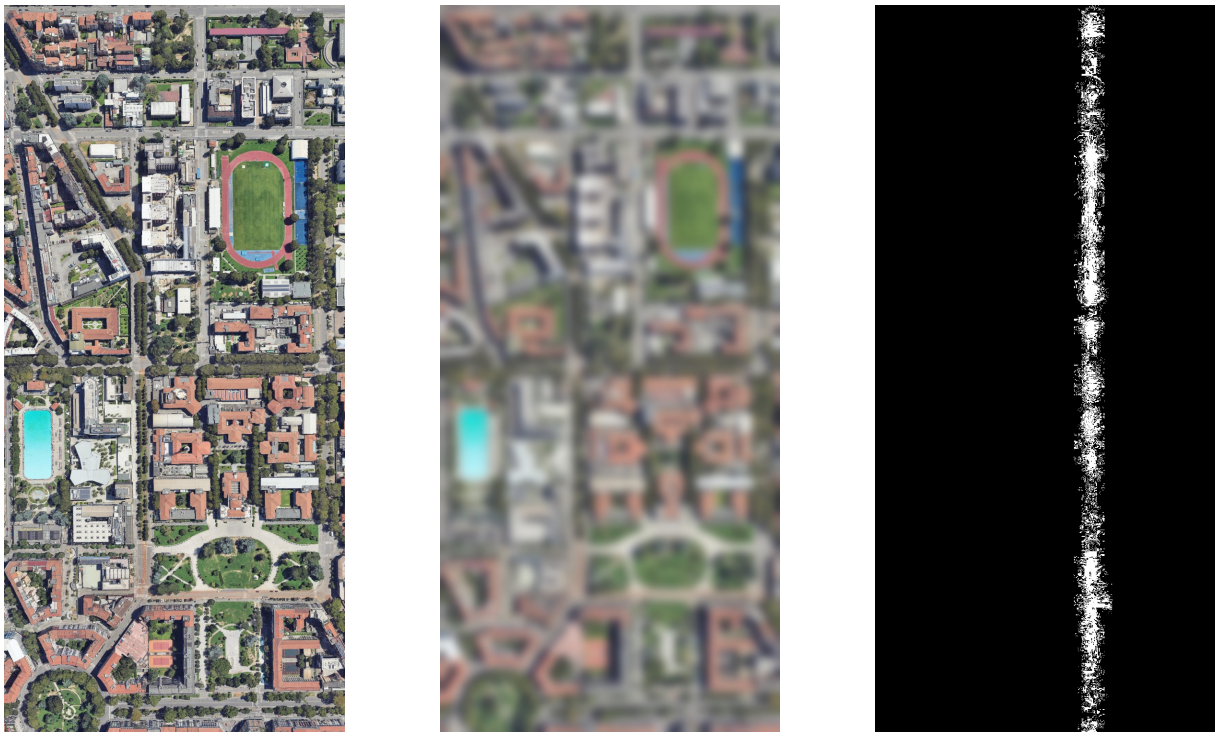

Figure 5.3: Blur filter campaign placeholder. Left: Original satellite input image. Middle: Example output classified as SDC when compared to the golden output after a bit-flip injection. Right: The different pixels highlighted

and the golden execution, while white pixels highlight the specific areas where the pixel values diverged. As observed in the figure, the fault injection did not crash the process but introduced a localized error. This visualization demonstrates how the injected fault propagates through the convolution kernel, resulting in specific visual artifacts rather than global noise. This type of analysis allows for a qualitative assessment of how critical a specific fault is to the final application output.

# 6 | Conclusions and future developments

The verification of reliability in safety-critical systems running on Commercial Off-The-Shelf (COTS) hardware presents a complex challenge: balancing the high accuracy of hardware-based testing with the throughput required for modern, complex software stacks. This thesis presented a novel kernel-level fault injection framework designed to bridge the gap between kernel-level privileges and user-space verification.

By operating as a Loadable Kernel Module (LKM), the proposed framework successfully utilizes the kernel as a mechanism to inject faults directly into user applications by using only their executables without the use of more invasive tools that base their injection capabilities GDB or on instruments invasive in other ways such as being source-code dependent like LLFI [28].

## 6.1. Summary of Contributions

The experimental results presented in Chapter 5 demonstrate that the framework achieves its primary objectives of accuracy, performance, and scalability.

- **Validation against Physical Hardware:** When compared to traditional FPGA-based hardware injection on standard benchmarks (`coremark` and `audiomark`), the framework demonstrated high fidelity. The distribution of fault outcomes (SDC, Crash, Hang) remained consistent with the FPGA reference, with statistical discrepancies remaining within a 4% margin (Tables 5.1 and 5.2). This validates the framework as a reliable substitute for physical injection.
- **Minimization of Overhead:** A critical contribution of this work is the reduction of injection latency. The overhead analysis confirmed that the framework introduces less than 10% added latency on average compared to native execution (Table 5.3). For computationally intensive workloads, this overhead becomes negligible, enabling the execution of massive injection campaigns that would be infeasible with cycle-

accurate simulators.

- **Scalability and Throughput:** The framework demonstrated linear scalability when parallelized across multiple worker threads. As evidenced by tests on Ryzen and Threadripper CPUs (Table 5.4), the speed-up is directly proportional to the available core count, allowing for rapid iteration during large-scale campaigns.
- **Insights into Modern Workloads:** The application of the framework to Neural Networks (MNIST and YOLO) revealed a correlation between the model's memory footprint and its susceptibility to Silent Data Corruption (SDC), confirming theoretical predictions found in the state of the art [43].

In conclusion, this thesis establishes that a kernel-mode SWIFI approach can generate a reliability profile for standard processor-based COTS system's binaries, enabling the "NewSpace" paradigm of using high-performance hardware with robust software mitigation.

## 6.2. Future Developments

While the current implementation provides a robust platform for fault injection, several features to develop in future research remain.

### 6.2.1. Enhanced Determinism and Configurability

As discussed in Section 4.3.5, the current version (v1.0) maintains a balance between speed and determinism. However, minor non-deterministic behaviors persist due to several design choices already addressed in the previously cited chapter. Future work could focus on extending the configurability of the deterministic injection trigger mechanism. By allowing users to strictly define the deterministic parameters of the injection context, the reproducibility of specific failures can be further heightened.

### 6.2.2. Cross-Architecture Portability

Finally, the current framework has been validated primarily on the x86-64 architecture, with initial support for ARM and RISC-V. A key future objective is to mature the implementation for these embedded architectures. Given the dominance of ARM in embedded automotive systems and the rising popularity of RISC-V in space applications, native support for these ISAs will significantly broaden the framework's applicability.

# Bibliography

[1] AMD Ryzen 9 5900 specifications. URL `https://www.amd.com/en/support/downloads/drivers.html/processors/ryzen/ryzen-5000-series/amd-ryzen-9-5900.html`.

[2] Linux Kernel. URL `https://kernel.org/`.

[3] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. GOOFI : Generic Object-Oriented Fault Injection Tool. Technical report.

[4] A. Aponte-Moreno, F. Restrepo-Calle, and C. Pedraza. Reliability Evaluation of RISC-V and ARM Microprocessors Through a New Fault Injection Tool. In *2021 IEEE 22nd Latin American Test Symposium (LATS)*, pages 1–6, 2021. doi: 10.1109/LATS53581.2021.9651874.

[5] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: a methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, 1990. doi: 10.1109/32.44380.

[6] S. Azimi, C. De Sio, D. Rizzieri, and L. Sterpone. Analysis of single event effects on embedded processor. *Electronics (Switzerland)*, 10(24), 12 2021. ISSN 20799292. doi: 10.3390/electronics10243160.

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–55. California, USA, 2005.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, 8 2011. ISSN 0163-5964. doi: 10.1145/2024716.2024718. URL `https://doi.org/10.1145/2024716.2024718`.

[9] G. Brunetti, G. Campiti, M. Tagliente, and C. Ciminelli. COTS Devices for Space

Missions in LEO. *IEEE Access*, 12:76478–76514, 2024. ISSN 21693536. doi: 10.1109/ACCESS.2024.3405373.

[10] N. Chatzivangelis, N. Zazatis, W. Grignani, G.-I. Paliaroutis, D. A. Santos, C. Imianosky, M. Kastriotou, C. Cazzaniga, F. Wrobel, A. Veronesi, C. Sotiriou, M. Andjelkovic, F. L. Vargas, D. Bertozzi, and L. Dilillo. Analysis of Devices in Radiation Harsh Environments. Technical report, 2025. URL `https://hal.science/hal-05371489v1`.

[11] O. Chatzopoulos, G. Papadimitriou, V. Karakostas, and D. Gizopoulos. Gem5-MARVEL: Microarchitecture-Level Resilience Analysis of Heterogeneous SoC Architectures. In *Proceedings - International Symposium on High-Performance Computer Architecture*, pages 543–559. IEEE Computer Society, 2024. ISBN 9798350393132. doi: 10.1109/HPCA57654.2024.00047.

[12] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux device drivers*. O'Reilly, 2005. ISBN 9780596005900.

[13] C.S. Dyer, A.J. Sims, J. Farren, and J. Stephen. Measurements of Solar Flare Enhancements to the Single Event Upset Environment in the Upper Atmosphere. 12 1990. doi: 10.1109/23.101211. URL `https://ieeexplore.ieee.org/document/101211`.

[14] F. de Aguiar Geissler, F. L. Kastensmidt, and J. E. P. Souza. Soft error injection methodology based on QEMU software platform. In *2014 15th Latin American Test Workshop - LATW*, pages 1–5, 2014. doi: 10.1109/LATW.2014.6841910.

[15] P. E. Dodd. Device Simulation of Charge Collection and Single-Event Upset. Technical Report 2, 1996.

[16] J. Gava, V. Bandeira, F. Rosa, R. Garibotti, R. Reis, and L. Ost. SOFIA: An automated framework for early soft error assessment, identification, and mitigation. *Journal of Systems Architecture*, 131, 10 2022. ISSN 13837621. doi: 10.1016/j.sysarc.2022.102710.

[17] G. Georgakoudis, I. Laguna, D. S. Nikolopoulos, and M. Schulz. REFINE: Realistic Fault Injection via Compiler-based Instrumentation for Accuracy, Portability and Speed. In *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.

[18] W. Grignani, D. A. Santos, M. Kastriotou, C. Cazzaniga, D. R. Melo, F. Wrobel,

L. Dilillo, and . Harv-Soc. Evaluation on SmartFusion2 and PolarFire FPGAs Under Neutron Radiation. Technical report.

[19] Z. He, H. Xu, and G. Li. A Fast Low-Level Error Detection Technique. In *Proceedings - 2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2024*, pages 90–98. Institute of Electrical and Electronics Engineers Inc., 2024. ISBN 9798350341058. doi: 10.1109/DSN58291.2024.00023.

[20] C. Imianosky, D. A. Santos, and L. Dilillo. Efficient TinyML Inference on a Fault-Tolerant RISC-V SoC with Vector Extension Title: Efficient TinyML Inference on a Fault-Tolerant RISC-V SoC with Vector Extension "Efficient TinyML Inference on a Fault-Tolerant RISC-V SoC with Vector Extension" Efficient TinyML Inference on a Fault-Tolerant RISC-V SoC with Vector Extension. pages 1–6, 2025. doi: 10.1109/IWASI66786.2025.11121981{\"{i}}. URL `https://hal.science/hal-05127892v1`.

[21] O. V. P. Imperas. M*DEV. URL `https://github.com/OVPworld/Information`.

[22] E. Jeong, N. Lee, J. Kim, D. Kang, and S. Ha. FIFA: A Kernel-Level Fault Injection Framework for ARM-Based Embedded Linux System. In *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*, pages 23–34. Institute of Electrical and Electronics Engineers Inc., 5 2017. ISBN 9781509060313. doi: 10.1109/ICST.2017.10.

[23] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulos. Differential fault injection on microarchitectural simulators. In *Proceedings - 2015 IEEE International Symposium on Workload Characterization, IISWC 2015*, pages 172–182. Institute of Electrical and Electronics Engineers Inc., 10 2015. ISBN 9781509000883. doi: 10.1109/IISWC.2015.28.

[24] J. Karlsson, P. Dahlgren, R. Johansson, and U. Gunneflo. Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms. Technical report.

[25] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. doi: 10.1109/CGO.2004.1281665.

[26] D. León, J. C. Fabero, and J. A. Clemente. Non-intrusive study on FPGA of the SEU sensitivity on the COTS RISC-V VeeR EH1 soft processor from Western Digital. *Microprocessors and Microsystems*, 105, 3 2024. ISSN 01419331. doi: 10.1016/j.micpro.2024.105021.

[27] A. Lesea, S. Drimer, J. J. Fabula, C. Carmichael, and P. Alfke. The rosetta

experiment: Atmospheric soft error rate testing in differing technology FPGAs. *IEEE Transactions on Device and Materials Reliability*, 5(3):317–328, 9 2005. ISSN 15304388. doi: 10.1109/TDMR.2005.854207.

[28] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman. 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*, pages 11–16. Institute of Electrical and Electronics Engineers Inc., 9 2015. ISBN 9781467379892. doi: 10.1109/QRS.2015.13.

[29] H. Madeira, M. ‡rio Rela, F. Moreira, and J. G. Silva. RIFLE: A General Purpose Pin-level Fault Injector. Technical report.

[30] I. Marques, C. Rodrigues, A. Tavares, S. Pinto, and T. Gomes. Lock-V: A heterogeneous fault tolerance architecture based on Arm and RISC-V. *Microelectronics Reliability*, 120, 5 2021. ISSN 00262714. doi: 10.1016/j.microrel.2021.114120.

[31] Nadim F. Haddad, Ronald D. Brown, Richard Ferguson, Andrew T. Kelly, Reed K. Lawrence, Daniel M. Pirkl, and John C. Rodgers. Second generation 200MHz RAD750 microprocessor radiation evaluation. 2012. doi: 10.1109/RADECS.2011.6131320. URL `https://ieeexplore.ieee.org/document/6131320`.

[32] M. Nicolaidis. Design for soft error mitigation. *IEEE Transactions on Device and Materials Reliability*, 5(3):405–418, 9 2005. ISSN 15304388. doi: 10.1109/TDMR.2005.855790.

[33] E. Normand. Single-Event Effects in Avionics. Technical Report 2, 1996.

[34] G. Ottman, R. H. Maurer, and E. Mellert. An Algorithmic Approach to Observed Single Event Effects in Van Allen Probes Solid State Recorders. *IEEE Transactions on Nuclear Science*, 66(12):2408–2416, 12 2019. ISSN 15581578. doi: 10.1109/TNS.2019.2953205.

[35] L. Palazzi, G. Li, B. Fang, and K. Pattabiraman. A Tale of Two Injectors: End-to-End Comparison of IR-level and Assembly-Level Fault Injection. Technical report.

[36] G. Papadimitriou and D. Gizopoulos. AVGI: Microarchitecture-Driven, Fast and Accurate Vulnerability Assessment. In *Proceedings - International Symposium on High-Performance Computer Architecture*, volume 2023-February, pages 935–948. IEEE Computer Society, 2023. ISBN 9781665476522. doi: 10.1109/HPCA56546.2023.10071105.

[37] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas. 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*, pages 622–629. Institute of Electrical and Electronics Engineers Inc., 9 2014. ISBN 9781479922338. doi: 10.1109/DSN.2014.96.

[38] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS: a full system simulator for multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 1050–1055, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306362. doi: 10.1145/2024724.2024954. URL `https://doi.org/10.1145/2024724.2024954`.

[39] M. Rebaudengo and M. Sonza Reorda. Evaluating the fault tolerance capabilities of embedded systems via BDM. In *Proceedings 17th IEEE VLSI Test Symposium (Cat. No.PR00146)*, pages 452–457, 1999. doi: 10.1109/VTEST.1999.766703.

[40] P. Rech. Artificial Neural Networks for Space and Safety-Critical Applications: Reliability Issues and Potential Solutions. *IEEE Transactions on Nuclear Science*, 71 (4):377–404, 4 2024. ISSN 15581578. doi: 10.1109/TNS.2024.3349956.

[41] S. Rogers, J. Slycord, M. Baharani, and H. Tabkhi. gem5-SALAM: A System Architecture for LLVM-based Accelerator Modeling. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 471–482, 2020. doi: 10.1109/MICRO50266.2020.00047.

[42] B. Sangchoolie, K. Pattabiraman, and J. Karlsson. An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors in Programs. *IEEE Transactions on Dependable and Secure Computing*, 19(3):1988–2006, 2022. ISSN 19410018. doi: 10.1109/TDSC.2020.3043023.

[43] F. F. Santos, J. E. Condia, L. Carro, M. S. Reorda, and P. Rech. Revealing GPUs Vulnerabilities by Combining Register-Transfer and Software-Level Fault Injection. In *Proceedings - 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021*, pages 292–304. Institute of Electrical and Electronics Engineers Inc., 6 2021. ISBN 9781665435727. doi: 10.1109/DSN48987.2021.00042.

[44] F. F. D. Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech. Analyzing and increasing the reliability of convolutional neural networks on GPUs. *IEEE Transactions on Reliability*, 68(2):663–677, 6 2019. ISSN 00189529. doi: 10.1109/TR.2018.2878387.

[45] V. Sridharan, N. De Bardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi. Memory errors in modern systems the good, the bad, and

the ugly. In *ACM SIGPLAN Notices*, volume 50, pages 297–310. Association for Computing Machinery, 4 2015. ISBN 9781450328357. doi: 10.1145/2694344.2694348.

[46] O. Subasi, C. K. Chang, M. Erez, and S. Krishnamoorthy. Characterizing the impact of soft errors affecting floating-point ALUs using RTL-level fault injection. In *ACM International Conference Proceeding Series*. Association for Computing Machinery, 8 2018. ISBN 9781450365109. doi: 10.1145/3225058.3225089.

[47] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, L. Carro, and A. Bland. 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)*, pages 331–342, 2015. doi: 10.1109/HPCA.2015.7056044.

[48] A. D. Velasco, B. Montrucchio, and M. Rebaudengo. KITO tool: A fault injection environment in Linux kernel data structures. *Microelectronics Reliability*, 60:153–162, 5 2016. ISSN 00262714. doi: 10.1016/j.microrel.2016.02.011.

[49] J. Wei, A. Thomas, G. Li, and K. Pattabiraman. Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 375–382, 2014. doi: 10.1109/DSN.2014.2.

[50] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. Technical report.

# A | Appendix

This appendix serves as a user guide for configuring the proposed framework. We present a series of use cases that cover common experimental requirements: benchmarking, Python virtual environment integration, file I/O redirection, and multi-target orchestration. Each example includes the necessary prerequisite steps, such as setting executable permissions, alongside the specific JSON configuration used to define the injection campaign.

### A.0.1. Example 1: Coremark

**Scenario:** Running a standard C/C++ executable that prints results to STDOUT. Setup: Ensure coremark.exe is compiled and available in your target folder. The example below will consider the target folder in Desktop.

```
{
  "workers": 4,
  "base_path": "/home/andrea/Desktop",
  "defaults": {
    "runs": 1000,
    "weight_mem": 1
  },
  "targets": [
    {
      "path": "{base_path}/coremark-main/coremark.exe",
      "args": [
        { "value": "0x0 0x0 0x66 400000 7 1 2000" }
      ]
    }
  ]
}
```

This `config.json` does an injection campaign of 1000 runs of the `coremark.exe` program using as arguments "0x0 0x0 0x66 0 7 1 2000" and making the probability of injecting

into memory 50% of the times.

### A.0.2. Example 2: Mnist

**Scenario:** Running a Python script inside a virtual environment. This example takes into consideration the case in which we have to run a python script. This program takes executables as input to run injection campaigns. A solution to run python scripts is to add the following line at the top of the script that needs to be executed.

*#! absolute/path/to/specific/python/interpreter/executable*

and then execute the

chmod +x /path/to/python/script.py

Only after these instructions are executed can the script be executed successfully to the injection program.

```
{
  "workers": 4,
  "base_path": "/home/andrea/Desktop",
  "defaults": {
    "runs": 1000,
    "weight_mem": 0
  },
  "targets": [
    {
      "path": "{base_path}/pythonvenv/mnist.py",
      "args": [
        { "value": "{base_path}/pythonvenv/source_image.jpg"
          }
      ]
    },
    {
      "path": "{base_path}/pythonvenv/mnist.py",
      "defaults": { "weight_mem": 1 },
      "args": [
        { "value": "{base_path}/pythonvenv/source_image.jpg"
          }
      ]
```

```
    },
    {
      "path": "{base_path}/pythonvenv/mnist.py",
      "defaults": { "only_mem": 1 },
      "args": [
        { "value": "{base_path}/pythonvenv/source_image.jpg"
          }
      ]
    }
  ]
}
```

The provided `config.json` runs 3 injection campaigns of 1000 runs each. The first campaign does injections only in the registers, the second has a 50% chance of injecting in memory and the last one injects only in memory.

### A.0.3. Example 3: image_filter

**Scenario:** Running code that produces an output in addition to the STDOUT. We are running a program that takes an image as input and outputs the blurred image in the specified path by the user. as in the Example 2 we are running a python script so first of all we need to add the following line to the script `image_filter.py`

*#!home/user/pythonvenv/bin/python3*

then opening the terminal from the folder in which the python script is we execute

chmod +x /path/to/python/image_filter.py

now here's an example of a `config.json` that redirects the output of the image into the experiment's folder.

```
{
  "workers": 4,
  "base_path": "/home/andrea/Desktop",
  "defaults": {
    "runs": 1000,
    "weight_mem": 9
  },
  "targets": [
```

```
    {
      "path": "{base_path}/pythonvenv/image_blur.py",
      "args": [
        { "value": "--input {base_path}/pythonvenv/
           image_to_blur.jpg --output_path {campaign}/
           injection_{run}" }
      ]
    }
  ]
}
```

### A.0.4. Example 4: Multiple Injection Campaigns

**Scenario:** We want to run multiple programs by configuring the `config.json` file with different runs for each program

```
{
  "workers": 4,
  "base_path": "/home/andrea/Desktop",
  "defaults": {
    "runs": 1000,
    "weight_mem": 9
  },
  "targets": [
    {
      "path": "{base_path}/pythonvenv/image_blur.py",
      "args": [
        { "value": "--input {base_path}/pythonvenv/
           image_to_blur.jpg --output_path {campaign}/
           injection_{run}" }
      ]
    },
    {
      "path": "{base_path}/pythonvenv/mnist.py",
      "defaults": {
        "runs": 20000,
        "weight_mem": 1,
```

```
      },
      "args": [
        { "value": "{base_path}/pythonvenv/source_image.jpg"
          }
      ]
    },
    {
      "path": "{base_path}/coremark-main/coremark.exe",
      "defaults": {
        "runs": 500,
      },
      "args": [
        { "value": "0x0 0x0 0x66 400000 7 1 2000" }
      ]
    }
  ]
}
```

in this config file we are running first a campaign of 1000 iterations with 90% chance of injection in memory for the `image_blur` script, then a campaign of the `mnist.py` script of 20000 iterations with 50% chance of memory injections and in the end we are running a campaign of 500 iterations of the `coremark.exe` program with chance of injection in memory equal to 90%.

# List of Figures

# List of Tables