# POLITECNICO DI TORINO

Master's Degree in Cybersecurity


Politecnico
di Torino

Master's Degree Thesis

# Fault Attack Injection strategies for RISC-V Microprocessors in Simulated Environments

Supervisors

Prof. Alessandro SAVINO

Prof. Stefano DI CARLO

Candidate

Giorgio FARDO

October 2025

# Summary

Hardware and software security are becoming increasingly critical due to the widespread proliferation of computing systems and the exponential growth of the IoT and embedded domains, both of which demand robust protection mechanisms. Addressing the rising sophistication and impact of attacks requires reducing the complexity of software testing against specific classes of vulnerabilities. In this context, access to simulator software capable of evaluating software robustness against such attacks can significantly lower the cost of security assessments and shorten the time to market of final products. This thesis presents modifications to the gem5 architectural simulator through the integration of a fault injection module for the RISC-V architecture. The proposed module enables the injection of register level single and multi-bit faults during simulation, supporting both fine-grained, deterministic fault injection and general fault testing through randomized spatial and temporal fault distributions. The work begins with an overview of the current state of the art in fault injection, fault attacks and emerging research trends. That review drove the design choices made to ensure the injection module realistically mimics real world fault modalities and capabilities. In the second part, the thesis details the modifications applied to gem5, the design decisions underlying these changes, and the challenges encountered due to the simulator's architecture. Furthermore it describes the auxiliary tools developed to support this work, including the Campaign Manager, which provides an entry point for managing and launching automated test campaigns. The final part of the thesis presents the evaluation of the proposed tool and outlines directions for future development. The evaluation uses security-focused code from the FISSC collection curated by the Université Grenoble Alpes. This dataset comprises multiple versions of test software incorporating varying levels of countermeasures and hardening, allowing assessment across different attack strengths and granularities. Particular attention is given to testing different versions of VerifyPIN, a program designed to mimic a card PIN verification routine. As future work, the thesis proposes extending fault injection support to caches and main memory to broaden the range of possible attacks and widen testing coverage.

# Table of Contents

# List of Tables

# List of Figures

# Acronyms

**AI**
: artificial intelligence

**SEE**
: single event effects

# Chapter 1

# Introduction

Testing software against hardware-injected faults has long been an important topic in both reliability and security research. Historically, this type of testing began with studies on the effects of radiation on hardware and how such environmental factors influenced software execution. These early efforts focused on reliability testing in harsh or exposed environments, where radiation could alter the normal behavior of hardware components. Over time, researchers realized that similar fault effects could also be intentionally induced by adversaries. This realization shifted the focus toward understanding, modeling, and mitigating hardware fault attacks.

With the emergence of embedded systems and the Internet of Things (IoT), the relevance of such attacks has become even more pronounced. As these devices are often deployed in unprotected environments, they present attractive targets for fault injection attacks. Consequently, research in this area has expanded rapidly, addressing both novel attack techniques and improved countermeasures. At the same time, as hardware designs grow increasingly complex, there is a growing need for alternative approaches to study fault effects without requiring access to physical devices.

The main objective of this thesis is the development of a fault injection simulator, called **InjectV**, built upon the **gem5** architectural simulator for the **RISC-V** architecture. The simulator is capable of injecting transient faults into the registers of the simulated CPU. The secondary objectives include validating the simulator through test scenarios and demonstrating its operational modes.

**RISC-V** was chosen as the target instruction set architecture (ISA) due to the limited number of existing gem5-based fault injectors supporting it, as well as its growing adoption in both research and industry, particularly in the IoT and embedded domains.

The importance of this work lies in the demand for fast, flexible, and cost-effective methods to test the reliability and security of software when access to physical hardware is limited or infeasible. Hardware-based fault injection setups are often prohibitively expensive and time-consuming to manage. Moreover, as systems become more interconnected and security grows in importance, understanding the impact of hardware faults on software reliability is critical for designing robust and secure systems. In this context, **InjectV** provides a software-based alternative that facilitates early testing and prototyping, while also serving as a valuable educational and research tool given the widespread use of gem5 in the academic community.

This thesis is structured into five main chapters. Following this introduction, **Chapter 2** presents an overview of physical fault injection techniques and discusses the types of effects that InjectV aims to replicate. It also includes a brief introduction to the RISC-V architecture, examples of its use in real-world applications, and a description of existing architectural simulators. The chapter concludes with a detailed technical overview of gem5 and its components, justifying its selection as the foundation for InjectV. **Chapter 3** describes the architecture and operation of InjectV, outlining its internal structure and fault injection workflows. **Chapter 4** focuses on the experimental setup used to validate InjectV, including the test software, simulation configuration, and injection scenarios. Finally, **Chapter 5** summarizes the main findings of this work and suggests possible directions for future development.

# Chapter 2

# Background : State of the art

## 2.1 Fault Injection techniques

This is a brief taxonomy of the current state-of-the-art of fault injection techniques and the current research trends on the matter. The most common physical fault injection techniques that can be found used in research and the actual industry are : Laser Fault Injection, EM Fault Injection, Voltage/Power Glitching and Clock Glitching.

### 2.1.1 Laser Fault Injection

Laser fault injection, or LFI, is a fault injection technique that uses a focused laser beam to inject faults on the silicon die. The theory behind it is as follows, physical characteristics of semiconductor transistors influence their sensitivity to ionizing radiation, and laser radiation can ionize an IC's semiconductor regions if its photon energy exceeds the semiconductor band gap. The ionizing power and thus the applicability to generate alteration in the transistor behavior depend on the type and power of the laser used. Different wavelengths result in different penetration power, dispersion, and precision.

These transients at the gate level, if timed correctly, can induce bit flips and/or bit resets.

#### History

The first paper that discusses laser use to introduce SEE in silicon is from 1965 by D. H. Habing [1], where they used lasers to simulate the effects of SEE caused by

exposure of silicon to intense gamma-ray sources. The used neodymium laser was able to induce transients that resemble those from a flash X-ray machine.

Following, the first paper on LFI is S. P. Skorobogatov et al. in 2002 [2], where they introduced a new class of attacks on secure microcontrollers and smart cards. The "low-cost" setup, based on a laser pointer and flashgun, highlighted the shift from infrared, which had good penetration depth and good spatial ionization but lacked precision for modern semiconductor devices, to visible red or green lasers, which also benefited from much higher photon absorption. The thinning of semiconductors and reduced scale meant also that lower-power lasers were now sufficient to introduce faults.

With semiconductor process node scaling, the precision of the laser beam became much more important to obtain high accuracy and localization; especially gate-level accuracy became harder as the process node was shrunk.

Recent research also described multi-spot laser fault injection, where multiple locations on the semiconductor are subjected to the influence of the laser in tight temporal proximity, meaning we can induce bit flips and/or bit resets in multiple locations almost simultaneously [3]. Brice Colombier et al. describe multi-spot laser fault injection as a new technique, backed by real-world test cases, that is able to overcome the shortcomings of traditional laser fault injection techniques, such as targeting more complex protected designs that have preventive countermeasures in place.

### 2.1.2 Electromagnetic Fault Injection

EM fault injection uses a transient, localized electromagnetic disturbance (a pulse or burst) placed near a packaged chip to produce unwanted electrical transients inside the target integrated circuit. The advantages over other types of fault injection are that de-packaging is not always needed, making the attacks less destructive, while still allowing relatively high localization to be achieved.

According to the literature, there are two main demonstrated techniques [4]. The first one, called harmonic EM injection, uses continuous EM waves to affect the behavior of critical analog blocks, such as TRNGs and embedded clock generators. The second technique, characterized by inducing transient faults through sudden EM pulses near the target IC, targets digital blocks.

The most recent empirical model, called the sampling fault model, is based on sampling faults that are “direct perturbations of the sampling process of D-type

flip-flops" [4]. In this scenario, EMFI alters the input and control signals of the DFF and provokes erroneous sampling of data if the perturbations are timed just before the rising edge of the clock. This model highlights the presence of a Sampling Fault Window (SFW), which is the time frame during which the DFF is susceptible to this type of perturbation caused by EM.

### 2.1.3 Voltage Glitching

Voltage glitching (voltage fault injection, VFI) is a physical fault injection technique where an attacker transiently perturbs the supply voltage of an integrated circuit (usually by producing a short-duration voltage drop or spike) in order to cause timing violations or undefined internal states. Properly timed and parameterized glitches can induce instruction skips, corrupt register or memory values, bypass software or hardware checks (authentication and signature verification code paths), or create side channels that leak secrets. The attack requires physical access to the power domain (directly or through an injection point) and precise control of glitch timing, amplitude, and duration.

The first significant research on voltage glitching dates back to 1997, when Anderson and Kuhn demonstrated the practical use of voltage manipulation in attacks targeting smart cards. In their analysis, they highlighted the vulnerabilities of some smart cards and processors to voltage spike and fluctuation attacks that could cause instruction skips or data corruption during execution [5].

More recent real-world research on VFI is represented by VoltPillager (Chen et al.) [6]. The authors built a low-cost hardware tool that injects messages on the Serial Voltage Identification (SVID) bus between the CPU and the on-board voltage regulator, allowing an adversary with physical access to directly control the CPU core voltage. By abusing SVID (which lacks cryptographic authentication), they performed hardware-based undervolting/glitching on Intel CPUs, enabling fault-injection attacks against Intel SGX enclaves—even on fully patched systems where the software undervolting interface (MSR 0x150 / Plundervolt mitigations) had been disabled. Using this technique, they reproduced previous attacks (in this case, Plundervolt [7]) and demonstrated key-recovery attacks against cryptographic code (including mbed TLS), as well as novel faults such as briefly delayed memory writes, showing that physical SVID manipulation can bypass software countermeasures and break SGX integrity.

### 2.1.4 Clock Glitching

Clock glitching is a fault injection technique where an attacker deliberately alters a device's clock signal to induce errors in its normal operation. Although this method was once highly popular in the realm of smart card testing, it eventually became less prominent when many smart cards transitioned to internal clock generators. However, clock glitching is regaining relevance, as modern systems-on-chip (SoCs) and other embedded devices often rely on external clocks during critical initialization phases. These windows of opportunity allow attackers to disrupt the clock signal at just the right moments, potentially exposing vulnerabilities that other testing methods might miss.

At its core, clock glitching involves creating short disturbances in the clock line. By timing these with critical sections of the system's operating cycle, the attacker can alter normal execution flow, resulting in unpredictable behavior such as instruction skips, unexpected or corrupted memory reads and writes, or bypasses of security checks.

This technique is particularly useful in cases where the system relies on an external clock source and has been used in more recent proof-of-concept attacks on the MediaTek BootROM [8], where researchers from the NCC Group used clock glitching to bypass the signature verification of the BootROM in the MediaTek MT8163V SoC. This demonstrates that even modern SoCs are still susceptible to these types of attacks and that they remain a valid area of research.

## 2.2 RISC-V

RISC-V is an open-standard instruction set architecture (ISA) that is widely used in microcontrollers and other devices with embedded cores. It is also gaining significant traction as part of AI accelerators and high-performance computing (HPC), thanks in part to major projects like the Chip Act [9], which will most likely boost investments and development toward RISC-V chips and IP. As examples of already deployed use cases of RISC-V in the AI and HPC sectors, we have the *Meta MTIA 2i*, an AI inference accelerator that uses RISC-V cores in the processing elements of the accelerator [10]. In Figure 2.1, we can see the internal architecture of a single processing element, which contains two RISC-V cores.

| Fault Injection Method | Requirements | Example of necessary equipment | Cost | Attack control |
|---|---|---|---|---|
| Clock glitching | Access to internal clock, generation and introduction of different clock waveforms. | • Oscilloscope<br>• Clock Fault Generator | Low | Medium |
| Voltage glitching | Access to efficient ways of effecting the power supply, e.g. switching two or more voltage sources to introduce voltage shapes. | • Oscilloscope<br>• Voltage Fault Generator | Low | Low |
| Electromagnetic Fault Injection | Electromagnetic pulse shape generation at desired location on chip. | • Electromagnetic probes<br>• Probe positioning<br>• Pulse Generator<br>• Oscilloscope | Medium / High | Medium / High |
| Laser Fault Injection | Chip decapsulation and high precision laser spot generation. | • XYZ Table<br>• Oscilloscope<br>• Laser Control<br>• Laser Source | High | High |

**Table 2.1:** Overview of fault injection methods, requirements, and typical equipment.

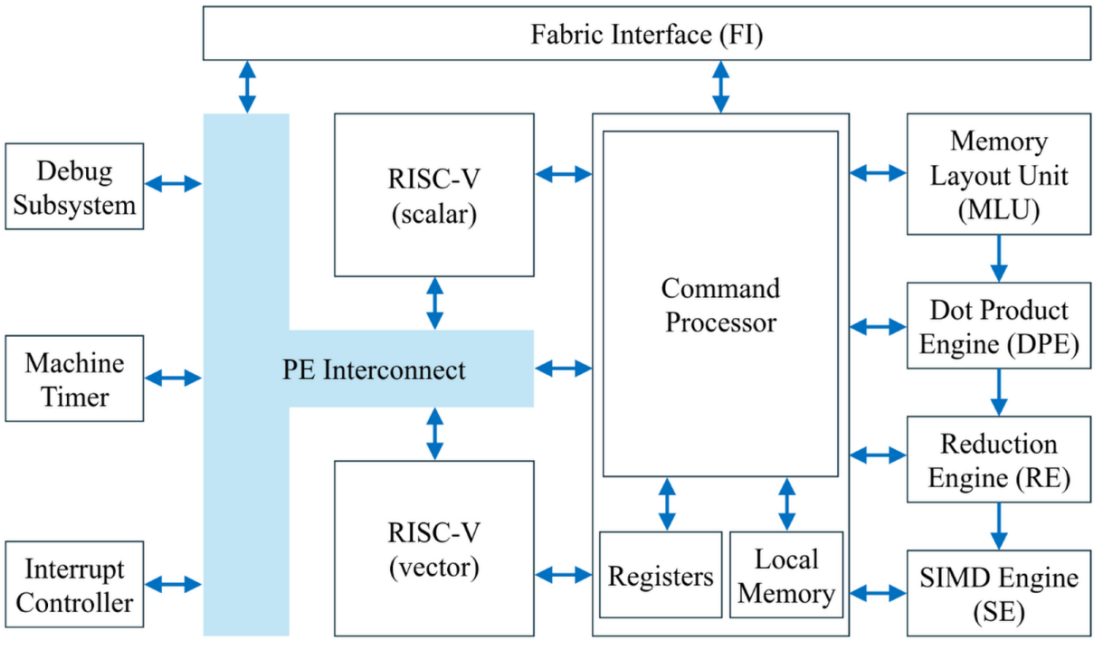



**Figure 2.1:** Meta MTIA 2i PE architecture [10]

### 2.2.1 Base ISA and extensions

RISC-V adopts strongly RISC-oriented design principles, allowing it to be implemented, modified, or extended by anyone without license requirements. Modularity is at the core of the ISA, consisting of a base set that can be extended with special-purpose, custom, and standard extensions.

The base ISA is identified by the 64-bit RV64I and 32-bit RV32I variants, with support for reduced versions such as RV32E and RV32C. The E variant reduces the number of registers, while the C (compressed) extension supports 16-bit instruction variants.

On top of this base, a wide range of standard extensions can be added: M (Integer Multiplication and Division), A (Atomic Instructions), F (Single-Precision Floating-Point), D (Double-Precision Floating-Point), Q (Quad-Precision Floating-Point), L (Decimal Floating-Point), C (Compressed Instructions), B (Bit Manipulation), J (Dynamically Translated Languages), T (Transactional Memory), P (Packed-SIMD/DSP), V (Vector Operations), and Zicsr/Zifencei (Control and Status Register and Instruction-Fetch Fence support). There are also specialized extensions such as H (Hypervisor), K (Cryptography), and N (User-Level Interrupts).

## 2.3 Architectural Simulators

In this section, the available architectural simulators on the market are described, motivating the choice of gem5 as the target for development.

Architectural simulators are critical tools in computer architecture research, providing a controlled and repeatable environment to evaluate new designs, validate performance measures, and study complex hardware–software interactions. These simulators vary in their level of abstraction, accuracy, and supported platforms, ranging from fast functional emulators to cycle-accurate full-system models. Below, we summarize several widely used simulators relevant to modern processor and system-level research.

**gem5** The gem5 simulator [11] is one of the most widely adopted in computer architecture research. It is a modular, flexible, and open-source platform that supports detailed modeling of processor cores (both in-order and out-of-order), memory hierarchies, and interconnects. gem5 supports multiple ISAs, including x86, ARM, RISC-V, and MIPS. It can operate in either system-call emulation or full-system mode, allowing real operating systems to be executed. Its extensibility and accuracy make it the de facto standard for studies on microarchitecture, cache

hierarchies, and heterogeneous systems. In addition, it is highly extendable and relatively well-documented, making it an optimal choice for custom solutions or extensions.

**Simics** Simics is a commercial full-system simulator developed by Intel (originally by Wind River)[12]. Unlike cycle-accurate simulators, Simics focuses on deterministic and high-performance functional simulation of complete hardware platforms, including processors, peripherals, and networks. It is particularly valuable for pre-silicon software development, system validation, and large-scale debugging.

**SPIKE** SPIKE is the RISC-V ISA simulator [13]. It can perform full-system simulation or proxied emulation through HTIF/FESVR. It supports the RISC-V base ISA and RV32IMAFDQCV extensions, privileged specifications, and single-step debugging. Unlike cycle-accurate simulators such as gem5, SPIKE focuses on functional simulation of RISC-V processors, serving as a reference implementation of the ISA specification. Due to its strict adherence to the ISA, SPIKE is widely used for verification and compliance testing, ensuring that hardware implementations and toolchains (compilers, assemblers, debuggers) conform to the RISC-V standard. Although it does not model microarchitectural details such as pipelines, caches, or timing, its simplicity and correctness make it useful for early development, debugging, and validation of RISC-V-based systems.

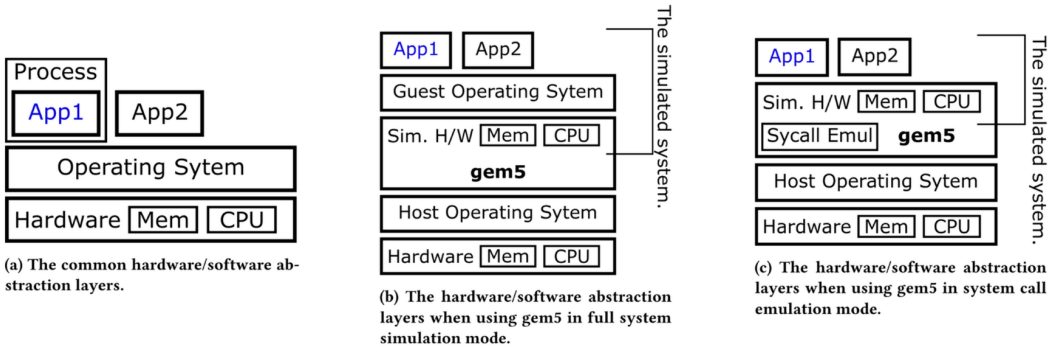
## 2.4 gem5

This section explains the details of the internal structure of gem5, from the simulation modes to the internal programming concepts like SimObjects. The following paragraphs will be key to understand the reasoning behind InjectV structure and the design choices.

gem5 provides four interpretation-based CPU models: a simple one-CPI CPU, a detailed in-order CPU model, a detailed out-of-order CPU model, and a KVM-based CPU that uses virtualization to accelerate simulation. These CPU models use a common high-level ISA description. The module focuses on the detailed in-order CPU to simplify analysis of the results against the test code, but is interoperable with all CPU models.

gem5 also includes a detailed event-driven memory system, including caches, crossbars, and a DRAM controller model, that is capable of simulating current memory types. This subsystem is fully configurable and modifiable in order to define custom cache hierarchies and heterogeneous memories.

Furthermore, gem5 supports a series of ISAs, including Alpha, ARM, SPARC, MIPS, POWER, RISC-V, and x86. Our target in this module is the RISC-V architecture. Porting the fault injector module to other architectures would be possible, keeping in mind the register layout and limits.




(a) The common hardware/software abstraction layers.

(b) The hardware/software abstraction layers when using gem5 in full system simulation mode.

(c) The hardware/software abstraction layers when using gem5 in system call emulation mode.

**Figure 2.2:** gem5 internals [14]

Continuing with its capabilities, gem5 supports the simulation of many-core systems; however, the limits depend on the chosen ISA. These capabilities are part of one of the two modes of operation of the gem5 simulator.

### 2.4.1 Operations modes

#### Syscall-emulation mode

In syscall-emulation mode, there is no OS image to boot, as the simulator itself emulates the operating system. SE mode ignores the timing of many system-level effects, including system calls, TLB misses, and device accesses.

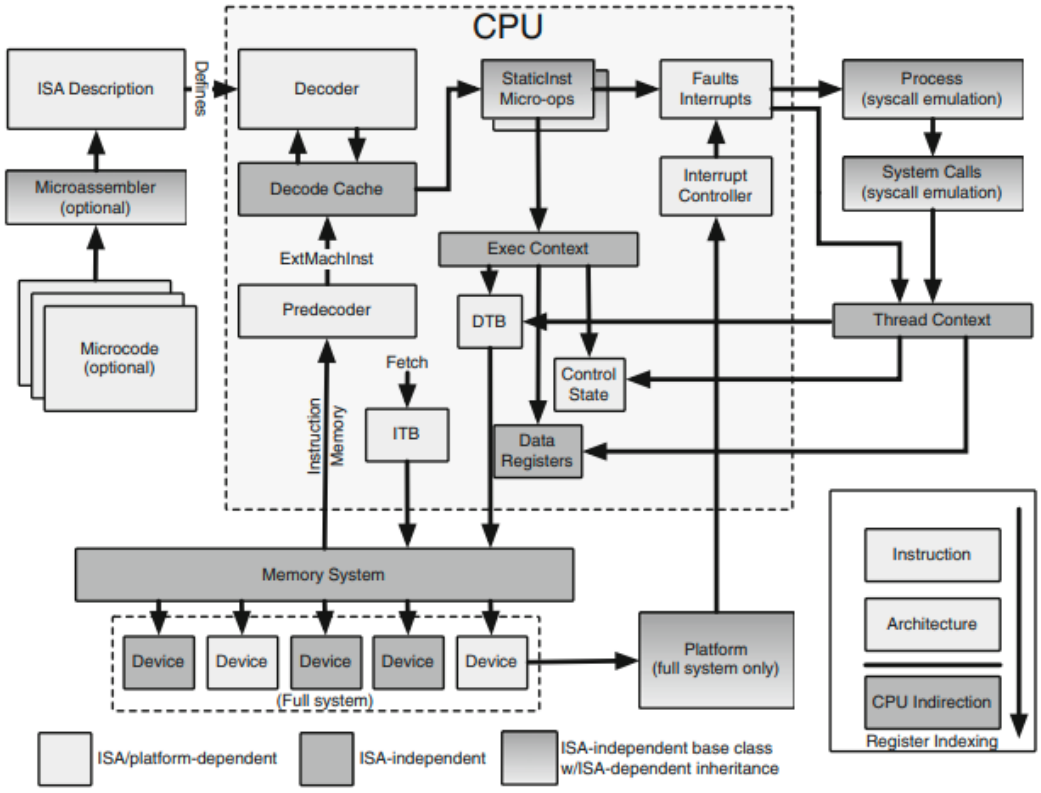



**Figure 2.3:** gem5 components [11]

### Full-System Mode

Full-system mode (FS) can boot a full-fledged Linux-based operating system. After booting, the guest can execute the application used to test the configuration.

## 2.4.2 Simulation Scripts

To configure the simulated system, gem5 relies on Python configuration files, where it is possible to define the ISA, memory hierarchy, number of cores, memory sizes, mode of operation, and type of CPU. Furthermore, from these files it is possible to specify the kernel to be booted, along with the bootloader and disk image. After configuration, the simulator can be initialized and started. These scripts are fed to the compiled gem5 runtime, which initializes the system objects and starts the simulation. At the end of the run, gem5 outputs various statistics that contain information about the completed simulation. Some of these include:

- Number of ticks: the basic unit of measurement representing a single simulated time instant (e.g., a single clock cycle);
- Duration of the simulation;
- Number of instructions executed;
- Other architecture-specific metrics, such as cache misses and hits, TLB misses and hits, memory accesses, and disk accesses.

In the output folder, full instruction traces can also be found if the run is executed with *–debug-flags=ExecAll –debug-file="<path-to-file>"*. These files are usually very large, as they contain all executed instructions for both the operating system and the binary running on the guest. This can complicate analysis of the program's behavior under test, but filtering is possible thanks to gem5's ability to import kernel code symbols, allowing extraction of traces that exclude kernel code.

### 2.4.3 Interaction Between Guest and Simulator

gem5 provides an API for interaction between the guest and the simulator itself. The main tool for interacting with this API is the *m5* utility, which is capable of sending commands to gem5 from within the guest. Some of the most useful commands that can be sent are:

- *exit*: Stops the simulation.
- *resetstats*: Resets simulation statistics.
- *dumpstats*: Saves simulation statistics to a file.
- *dumpresetstats*: Same as *dumpstats*.
- *checkpoint*: Creates a checkpoint.
- *readfile*: Prints the file specified by the configuration parameter *system.readfile*. This is how the rcS files are copied into the simulation environment if they are not already included in the image.
- *switchcpu*: Causes an exit event of type "switchcpu," allowing the Python configuration to switch to a different CPU model if desired.
- *workbegin*: Causes an exit event of type "workbegin," which can be used to mark the beginning of a region of interest (ROI).
- *workend*: Causes an exit event of type "workend," which can be used to mark the end of an ROI.

Some of these commands will be used later in the testing setup for InjectV.

### 2.4.4 Checkpoints

Due to the inherent slowness of the simulator, especially in FS mode, when using the *TIMING* CPU model, which is more accurate than the *SIMPLE* CPU model, the runtime required to boot a full Linux image and load the environment tools can range from several minutes to several hours, depending on the operating system being booted and the hardware of the host machine. We will examine some of these runtime figures in the results chapter.

To overcome this speed limitation, a very useful tool for repeated runs or for simulating applications after the OS has booted is checkpointing. This feature allows the complete state of the simulation to be saved in a checkpoint file, which can later be restored. The restore process restarts the simulation from the tick time at which it was stopped.

This is particularly useful to skip the boot process entirely (after the first simulation run), which is typically slow due to the initialization subsystem and the final stages of the bootloader. After the initial checkpoint is saved post-boot, done by inserting the *m5 checkpoint* command at the start of the rcS file, so it executes just after system initialization, the simulator can be restarted any time (even with a different workload) and rerun from just after the complete Linux boot.

### 2.4.5 ThreadContext

ThreadContext is the external interface to all the state of a thread, for anything outside the CPU model. It provides accessor methods to state that might be needed by external objects, ranging from register values to kernel statistics. In our specific case, it exposes methods such as *getReg()* and *setReg()*, which enable direct access to register values.

### 2.4.6 Event-Driven Programming Model

gem5 is an event-driven simulator. In this model, each event has a *callback function* responsible for processing the event when it is triggered. Whenever an event is fired, the simulator calls the defined callback function to execute it.

Scheduling of events can be done in two ways:

- Before the simulation starts: using the *startup()* call. Inside its body, events can be scheduled with the *schedule(<event>, <tick>)* function. This allows all events to be scheduled before the simulation starts, setting triggers for them in advance.

- Dynamically during simulation: a first event can be scheduled in the *startup()* function, and subsequent events can be scheduled within the callback of the previous event. This allows successive events to be scheduled based on conditions that are not known at the start of the simulation, providing greater flexibility.

### 2.4.7 SimObjects

Almost all objects in gem5 are SimObjects, which represent physical components and can be specified and instantiated in the configuration file. For example, a ClockedObject is a superset of SimObject that adds a clock and functions to interact with it (e.g., *nextCycle*, *clockEdge*) in relation to ticks, which are the base unit of time in the simulation.

# Chapter 3

# InjectV

InjectV is the proposed gem5 module that provides support injecting register transient faults in full-system simulations based on the RISC-V architecture. To better understand the reasoning behind the design choices, the following section analyzes the architecture of gem5 and how it can be extended.

Currently, InjectV focuses on transient single and multi-bit faults in the registers.

## 3.1 Why RISC-V

The target architecture is RISC-V because, upon reviewing the available fault injectors for gem5, there was no support specifically for the RISC-V ISA. Fault injection modules for other architectures exist; for example, projects like GemFI [15] support x86, ARM, and Alpha. There is also gem5-MARVEL, which is a much more complex framework that primarily focuses on heterogeneous systems.

## 3.2 Architecture

The two fundamental components of InjectV are the gem5 module *FaultInjector* and the *CampaignManager*.

### 3.2.1 FaultInjector Module

The gem5 module is responsible for:

- Parsing the fault settings and the mode of operation defined by the runners in the *CampaignManager*.
- In *Precise* mode: mapping register names to actual RISC-V gem5 registers and checking bit indices for bounds.

- In *Random* mode: generating random fault locations and bit indices according to the settings provided by the caller.

- Reverse-mapping register names and register IDs for both modes.

- Preparing, planning, and executing the faults defined by the *CampaignManager*.

Below we show the flow of a fault from its definition in the *CampaignManager* to execution in the gem5 *FaultInjector* module.

For CPU state access, the *FaultInjector* uses the *ThreadContext* interface, which provides the ability to inspect and modify the state of the running CPU model. Register reads and writes are the two operations used by the *FaultInjector*'s *flipBit()* method to inject faults.

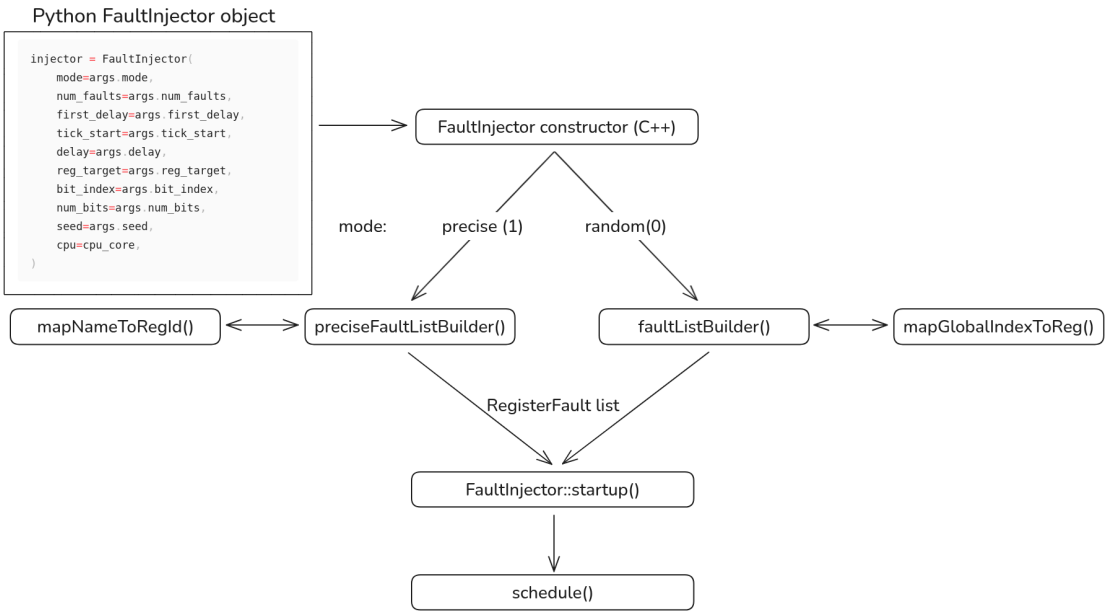



**Figure 3.1:** FaultInjector class

The creation of the *FaultInjector* object (which inherits from *SimObject*) is a two-step process. The simulation script (written in Python) creates a Python wrapper object for the underlying C++ class and calls the C++ *FaultInjector* constructor with the provided parameters. From that point onward the implementation is in C++. One of the constructor parameters specifies the mode (0 = Random mode, 1 = Precise mode). This choice affects the number of parsed parameters and the code paths used to process the faults to be injected. In particular:

### Random mode

When the mode is set to Random, the constructor calls the internal method *faultListBuilder()*, which, given the parameters for the run, generates a list of *RegisterFault* objects. *faultListBuilder()* uses an internal random generator, which can be seeded externally via the *FaultInjector* constructor, to select the target register and the bit index for each fault injection. The current version of InjectV uses a uniform distribution.

The random generator produces an integer representing a flat register index, this index is mapped to a real RISC-V register in gem5 by the function *mapGlobalIndexToReg()*. That function translates the flat index used by the generator to the corresponding register (selected from the integer and floating-point register classes in gem5's RISC-V ISA implementation). After translation, the fault tick time, the bit index, the fault width, and the resulting register are packed into a *RegisterFault* object.

The list of *RegisterFault* objects produced by *faultListBuilder()* is then passed to the *startup()* method, which creates the *FaultEvent* objects and schedules them for execution with *schedule()* [16]. The *startup()* function is called by the gem5 simulator when the simulation begins, it prepares the *SimObjects* and schedules their associated events.

### Precise mode

In Precise mode, the *FaultInjector* calls *preciseFaultListBuilder()*. Unlike *faultListBuilder()* in Random mode, *preciseFaultListBuilder()* uses *explicit* parameters supplied to the *FaultInjector* constructor, no parameters are randomly generated. This method calls *mapNameToRegId()*, which maps a register name string (e.g. `x12`, `f5`) to the corresponding register ID that gem5 methods can use. If the constructor parameter *num_faults* is greater than one, the *RegisterFault* entry is copied with the specified parameters and the tick target is incremented appropriately to repeat the same fault at regular intervals. The resulting *RegisterFault* list is processed the same way as in Random mode.

### Fault execution

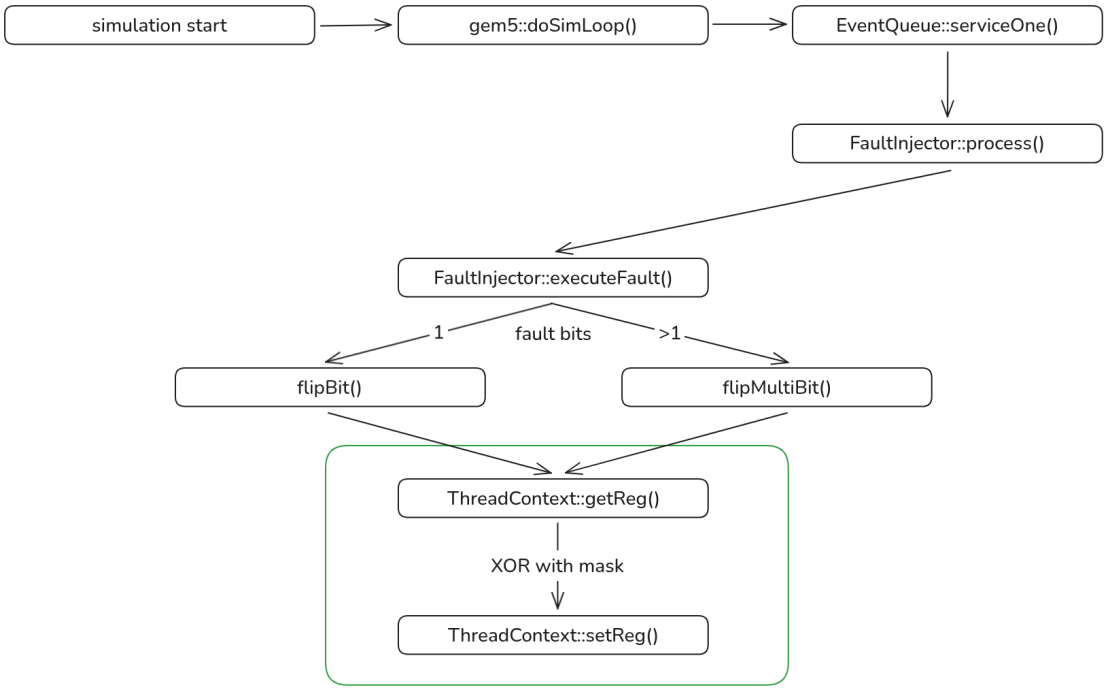



**Figure 3.2:** FaultInjector fault execution

As shown in Figure 3.2, the planned *FaultEvent* objects are stored in the event queue. The gem5 main simulation loop calls *serviceOne()* at each iteration, which extracts and processes the first event in the queue. Processing an event invokes the class's *process()* method, every class that inherits from *SimObject* implements this interface to generate or handle events. In our case, *FaultInjector::process()* calls the internal function *executeFault()*.

*executeFault()* performs the bit flip. Depending on the fault width, it calls either *flipBit()* for single-bit injections or *flipMultiBit()* for multi-bit injections. Both methods operate similarly: they obtain the current value of the target register via the *ThreadContext* interface (using *getReg()*), construct an appropriate mask for the bit flip, XOR the mask with the register content, and write the updated value back using *ThreadContext::setReg()*, which updates the gem5 CPU object state.

This process is repeated for all *FaultEvent* objects scheduled during the preparation phase.

### 3.2.2 CampaignManager

The CampaignManager is responsible for the interface with the user and exposes different modes and settings to define, plan, and execute testing campaigns. The tool can launch five modes:

- *Prepare mode*: This loads the user-defined boot image, kernel, and bootloader, runs until the first user-defined checkpoint (for example, just after boot), creates a checkpoint, and returns. This is used to prepare the system for testing runs and speeds up simulation, as the boot process is usually the slowest. The checkpointing after boot is based on guest-side checkpointing defined in the *init.d* script. To further accelerate booting, this part of the simulation is executed using an *ATOMIC* CPU.

- *Golden-run mode*: This mode uses the user-defined workload (which must be loaded in the specified disk image), restores the previously made checkpoint, and runs until the workload is completed. This run is used to determine the baseline timing for the simulation (used later to define timeouts for successive simulations). Furthermore, if enabled, this mode runs the simulation with the debug flag *ExecAll*, exporting the entire run trace to a file. By manually passing the generated trace to the debugTrace.py script, the trace is parsed and the user-specified function entry point is identified, enabling specific tick targeting for precise testing.

- *Random-campaign mode*: As the name suggests, this mode launches a random fault injection campaign, leveraging the random mode of the FaultInjector module. Parameters such as the number of faults per run, number of bits per fault, delays between faults, number of parallel runs, and total number of runs can be specified. After each simulation, the run output is processed to determine the outcome. This process is customizable by defining custom AttackParsers tailored for each tested binary.

- *Precise-campaign mode*: This mode runs the FaultInjector in precise mode, taking the fault plan from a file. Multiple fault patterns can be tested in a single campaign. The outcome is handled in the same manner as in random-campaign mode.

### 3.2.3 Flowgraph

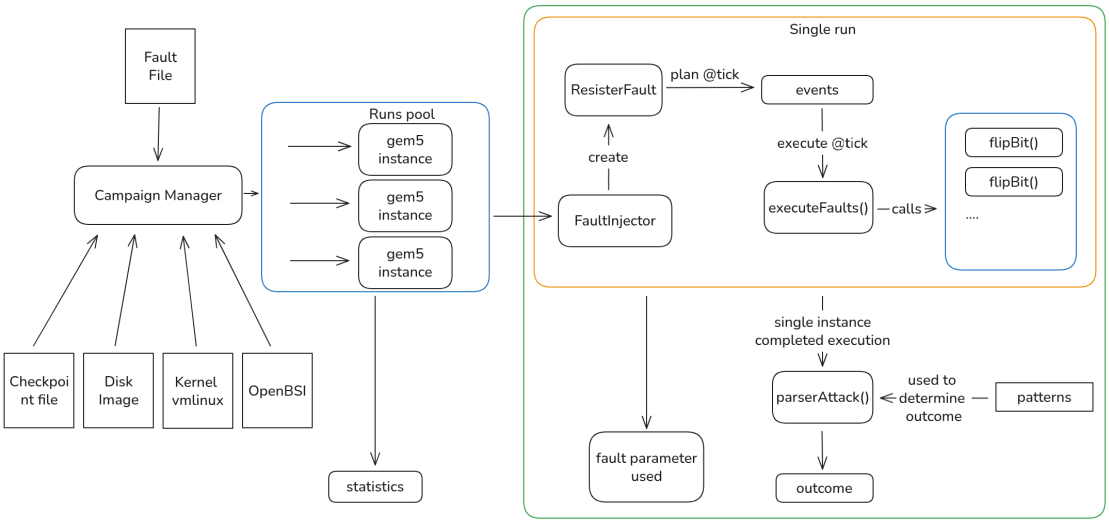



**Figure 3.3:** Precise campaign mode

Fault flow for the precise campaign mode is as follows:

1. Faults are defined in the fault plan file, each line tells InjectV to create a simulation with those parameters, that are:
   - **Number of faults to insert**: this drives the number of consecutive faults, delayed by Delay each one with the same values as the one described by this line in the fault file.
   - **First delay**: this sets the delay of the fist injection from the specified tick time
   - **Tick time**: this defines the moment where the fault will be injected, if the faults are multiple then the second fault will be injected at Tick Time + Delay where delay is the next parameter.
   - **Delay**: this as mentioned just above is the interval between insertions.
   - **Register target**: used to define where the fault will be injected.
   - **Bit index target**: bit target for the bit-flip, in case of multiple bits fault, this is the first bit, the others will be on the left, until register boundary.
   - **Fault width**: drives the amount of bits that are faulted, will be truncated when going over the register boundary
2. The fault file is parsed and the number of runs defined in the tool are set up, each with the defined faults.

3. For each run, the fault list is sent to the simulation script, which takes these as input parameters.

4. The simulation script creates the FaultInjector object with the input parameters, mapping the register name to the actual register index for gem5 and checking bit boundaries.

5. The simulation is started, restoring from the checkpoint after boot that has been created previously.

6. The *startup()* function is called, setting the triggers for the defined faults and the parameters for the faults to be executed.

7. The simulator starts executing. When it reaches the planned tick where the FaultEvent is scheduled, it executes the *faultExec()* function. This function retrieves the specified register value from the ThreadContext, XORs the content with the mask defining the bits to flip, and then writes back the modified register content using the ThreadContext.

8. Execution continues until one of two things happens:

   - Simulation concludes and stops: the binary executed and reached the simulation exit. The outcome of the finished execution is classified later.
   - Simulation hangs: this is checked by the CampaignManager, which assigns each gem5 process a timeout proportional to the time of the golden run. If the simulation process reaches the timeout, it is terminated and the outcome is recorded as **HANG**.

9. If the execution completed, the outcome is determined based on a user-provided script, which analyzes the terminal output of the simulation to classify the program as:

   - **CRASH**: if, for example, a kernel panic occurred or the binary crashed but the simulation ended.
   - **NO_EFFECT**: if the fault was correctly injected but the program outcome did not change.
   - **SUCCESS**: if the fault was injected and the binary oracle indicates that the attack was successful.

10. Stats are generated from the campaign, highlighting the successful injections that generated attacks, and the distribution of the faults injected.

# Chapter 4

# Testing and Results

To verify the functionality of InjectV and to demonstrate its capabilities, we created the following demonstration. The demonstration architecture is intended to mimic a typical InjectV use case, highlighting the steps and procedures required to set up the experiment and collect results. Because the primary goal of the simulator is to emulate a real-world physical fault-testing campaign, the objectives of the simulated campaign closely match those of an actual campaign.

The necessary steps in the campaign are:

1. **Select the test target**: In this case we use two versions of **VerifyPIN**; more details follow below.
2. **Prepare the image**: Because the gem5 simulator boots Linux, we prepare the disk image, the compiled kernel, and the bootloader so the test target can be booted and executed.
3. **Identify timing and triggers.** Determine the timing of the code sections to be tested and the triggers for those sections. In our case we identify function entry points and the execution time of individual instructions using the `debugTrace` script (see Section 4.1.3).
4. **Create the fault plan(s)**: Define the fault-injection plan(s) that will drive the simulations and the injections.
5. **Set up outcome parsers**: Each binary produces different terminal outputs that correspond to different exit states; for example, a binary that performs authentication will emit a confirmation string on success. Parsers are configured to capture those outputs.
6. **Run the simulator**: Execute the simulator with the selected fault plan and mode.

7. **Analyze results**: Interpret the statistics and logs generated by the tool.

## 4.1 Testing setup

This section describes the components needed for the evaluation.

### 4.1.1 VerifyPIN

As mentioned earlier, the chosen test program for the capability evaluation of InjectV is `VerifyPIN`, this binary is part of the FISSC collection. The FISSC collection [17] is *the Fault Injection and Simulation Secure Collection*, a collection of C codes with countermeasures against fault injections associated with attack scenarios.

From this collection, we focused our analysis on the `VerifyPIN` program, which mimics a card PIN verification routine. In the FISSC there are seven versions of the VerifyPIN program. These versions differ in the types of hardening that are in place.

| Version | HB | FTL | INL | BK | SC | DT |
|---|---|---|---|---|---|---|
| v0 | No | No | No | No | No | No |
| v1 | Yes | No | No | No | No | No |
| v2 | Yes | Yes | Yes | No | No | No |
| v3 | Yes | Yes | Yes | Yes | No | No |
| v4 | Yes | Yes | Yes | Yes | Yes | No |
| v5 | Yes | Yes | Yes | Yes | No | Yes |
| v6 | Yes | Yes | Yes | Yes | No | Yes |
| v7 | Yes | Yes | Yes | Yes | Yes | No |

**Table 4.1:** FISSC `VerifyPIN` versions and their applied hardenings. HB: Hardened Booleans, FTL: Fixed-Time Loop, INL: Inlined Calls, BK: Backup Copy, SC: Step Counter, DT: Double Test.

The `VerifyPIN` program is divided into three main functions: `initialize()`, `verifyPIN()`, and `oracle()`. The `initialize()` function creates two PINs that are deliberately different so that a normal execution results in a failed authentication. The `verifyPIN()` routine performs the verification; this is the component that varies across versions, we discuss the two chosen versions in detail later. Finally, `oracle()` triggers the oracle based on the program outcome.

The oracle is not strictly necessary when using the `attackParser` script, since the parser directly checks the program outcome; therefore the oracle would be

redundant for our tests. All tests reported here use the default authentication oracle. There are two oracle types:

- **auth** oracle checks if at the end of the execution the result is *g_authenticated = 1*, that means that authentication has been bypassed and no countermeasure was triggered.
- **ptc** oracle checks if at the end of the execution the result is *g_ptc*>= 3, that means that the try counter has been manipulated, as from a normal execution, as we can see from Appendix 6.1 on line 31, *g_ptc* is decremented.

Table 4.1 are listed all the VerifyPIN versions from the v0 that has no hardenings all the way to v7 that is the most hardened and secure.

For the testing phase of InjectV we focus on version 0, the base one without hardenings, and version 4, which introduces hardened bools, a fixed-time loop, inlined functions, PTC decremented first, PTC backup, and a loop counter.

### VerifyPIN_0

VerifyPIN_0 splits the main *verifyPIN* function in two, separating the *byteArray-Compare()* function that is responsible for the comparision of the two pins, byte by byte. This returns 1 only if all the bytes of the pins match. Following Listing 6.1 on Appendix 6.1 we can examine the rest of the function body of *verifyPIN*, notable section are the reset and decrement of the *g_ptc* counter that is reset to the start value of 3 in case of successful pin comparison, and decremented whenever the comparison fails.

### VerifyPIN_4

VerifyPIN_4, full code is in Appendix 6.2, presents the following hardenings :

- **Hardened Booleans**: to prevent vulnerabilities where a single-bit fault could invert a logical value, booleans are encoded using values with the maximum Hamming distance, making it extremely difficult for a single fault to change `True` into `False` (or vice versa).
- **Fixed-Time Loop**: ensures that the PIN verification process always takes the same number of iterations, regardless of when a mismatch is detected. This prevents timing attacks that could reveal information based on execution time.
- **Inlined Calls**: all function calls are inlined to eliminate call/return instructions that could be exploited by instruction-skipping or fault-injection attacks targeting control-flow changes.

- **Backup Copy**: critical variables (such as the tries counter and loop counters) are stored in redundant backup copies. These are checked for consistency during execution to detect and mitigate transient faults.

- **Step Counter**: a dedicated counter keeps track of the number of executed steps or iterations, allowing the program to detect anomalies such as skipped instructions or early termination caused by fault injection.

As we can see in listing 6.2 in this version the *byteArrayCompare* function is embedded in the code and not in a separate function. After that the *g_ptc* decrement is not vulnerable as is backed up and not isolated ad the end of the verification.

### 4.1.2 Image configuration

The VerifyPIN binary was loaded into a custom disk image built with RootFS as the base, disabling the TC module to reduce size and initial boot times. For basic utilities, we used BusyBox. The kernel used was Linux 6.8, compiled with *defconfig* as a base, and OpenSBI 1.3.1 was used as the bootloader, configured to bootstrap the vmlinux. After creating the disk with an Ext2 file system, the VerifyPIN binary was inserted and called from the *init.d* scripts to allow automatic execution at boot.

To skip the boot process for subsequent runs, the init script was structured as follows:

1. The kernel boots and executes the *init rcS* script. The first instruction echoes the full boot and immediately calls the m5 utility to create a checkpoint and stop the simulation.

2. When the checkpoint is restored, the init script execution resumes. The next instruction calls the VerifyPIN binary.

3. After the VerifyPIN binary execution completes, the m5 utility *exit* command is executed, terminating the simulator.

### 4.1.3 Fault plan

We have prepared two fault plans, one for random injections that will use the Random mode of InjectV, this plan will target generally the VerifyPIN binary, and a run with custom tailored fault file for the precise mode.

### Random campaign

For the random campaign 4 different injection point have been choosen for each of the versions on `VerifyPIN`.

For v0 the injection points are :

- `main`: positioned at the start of the *main* function of the binary.
- `verifyPIN`: positioned at the start of the *verifyPIN* function.
- `byteArrayCompare`: positioned at the start of the *byteArrayCompare* function.
- `dec_tries`: positioned a couple ticks before the decrement of the tries counter (`g_ptc` $--$).

For v4 they are:

- `main`: positioned at the start of the *main* function of the binary.
- `verifyPIN`: positioned at the start of the *verifyPIN* function.
- `byteArrayCompare`: positioned at the start of the section of *verifyPIN* that does the PIN comparison byte by byte function.
- `verifyPIN_2`: positioned a couple ticks after the pin comparison section.

Each campaign for each location was composed by 400 to 600 simulations, depending on the location, each one with 8 faults injected with a delay of 2000 ticks between them. Each fault was 1 bit wide, the register target and the bit in the register to target were randomly selected by the simulator. We used as starting seed 123456789 to make the run repeatable, as the seed is linearly incremented for each instance of the run. Considering the 8 faults per run, 4 for each version of VerifyPIN the total amount of injected faults is 3100 simulations for a total of 24800 faults injected.

### Precise campaign

Then the precise fault plan will be used for the Precise mode, in this case the objective is not general testing but is to try to prove that with only register injected faults that cause bit flips we can stop the decrement of the *g_ptc* counter or infer arbitrary values to increment the number of possible tries. The fault file used for VerifyPIN_0 is listed in listing 4.1. This fault file was redacted after the analysis of the normal execution trace of VerifyPIN_0 4.1 compared with the compiled binary, in order to identify the instructions responsible for the write back of the updated value of *g_ptc*.

The trace of the golden run highlighted a probable placement for the injection, as at tick 641435737000, register a4 (x14) contains the already update value of *g_ptc* (2). Thus injecting a bit flip on the first bit will set the register content to value 3.

**Listing 4.1:** `verifyPIN_0` precise fault list file

```
1 1,0,641435737000,500,x12,0,1
2 1,0,641435737000,500,x14,0,1
3 1,0,641435737000,500,x14,63,1
```

```
641435724000: board.processor.cores.core: A0 T0 : 0x10670    : ld a5, 1500(a5)          : MemRead :  D=0x000000000007c8e1 A=0x7ac48  flags=(IsInteger|IsLoad)
641435726000: board.processor.cores.core: A0 T0 : 0x10674    : lb a5, 0(a5)             : MemRead :  D=0x0000000000000003 A=0x7c8e1  flags=(IsInteger|IsLoad)
641435728000: board.processor.cores.core: A0 T0 : 0x10678    : andi a5, a5, 255         : IntAlu :  D=0x0000000000000003  flags=(IsInteger)
641435729000: board.processor.cores.core: A0 T0 : 0x1067c    : c_addiw a5, -1           : IntAlu :  D=0x0000000000000002  flags=(IsInteger)
641435731000: board.processor.cores.core: A0 T0 : 0x1067e    : andi a5, a5, 255         : IntAlu :  D=0x0000000000000002  flags=(IsInteger)
641435733000: board.processor.cores.core: A0 T0 : 0x10682    : sliw a4, a5, 24          : IntAlu :  D=0x0000000002000000  flags=(IsInteger)
641435735000: board.processor.cores.core: A0 T0 : 0x10686    : sraiw a4, a4, 24         : IntAlu :  D=0x0000000000000002  flags=(IsInteger)
641435737000: board.processor.cores.core: A0 T0 : 0x1068a    : auipc a5, 106            : IntAlu :  D=0x000000000007a68a  flags=(IsInteger)
641435739000: board.processor.cores.core: A0 T0 : 0x1068e    : ld a5, 1470(a5)          : MemRead :  D=0x000000000007c8e1 A=0x7ac48  flags=(IsInteger|IsLoad)
641435742000: board.processor.cores.core: A0 T0 : 0x10692    : sb a4, 0(a5)             : MemWrite :  D=0x0000000000000002 A=0x7c8e1  flags=(IsInteger|IsStore)
```

**Figure 4.1: VerifyPIN_0** trace around the *g_ptc* decrement.

**How to obtain the tick time of a function:** To obtain the tick time where the simulation starts executing a function, we developed a Python script `debugTrace` that, given the binary to be tested and the debug trace of the golden run (obtained by extracting the output of the *–debug-flags=ExecAll* run), uses *riscv64-linux-gnu-addr2line* to match symbols in the binary to the tick times when they are executed during the simulation. This script can parse the usually very large full trace file and extract the selected function symbols contained in the binary under test.

This can also be used to obtain precise tick times for single-instruction execution to test in tailored precise fault-injection campaigns.

## 4.2 Simulation execution

The tests were executed on a workstation with the following specifications: AMD Ryzen 9 7945HX (16 cores / 32 threads), 64 GB DDR5, and a 1 TB NVMe disk. The cumulative time for all simulations was approximately five hours, running four parallel simulations at a time. Each individual simulation run required roughly 28 seconds.

The primary constraint on parallelism was memory usage: four concurrent runs peaked at about 34 GB. Accounting for system caching and other services, running four to five concurrent simulations was reasonable on this machine. With further memory optimizations or on systems with greater memory capacity, a larger degree of parallelism would be possible; a single run consumes about 8.5 GB at peak. CPU

cores were not the limiting factor because gem5 simulations are single-threaded, so a single simulation cannot be parallelized across multiple cores.

To illustrate an example InjectV run, we analyze one of the campaign setups. Listing 4.2 shows the random campaign configuration passed to the campaign manager. The arguments appear in the following logical order:

- run mode (here: `random`);
- bootloader, disk image and kernel directives (`-opensbi`, `-disk`, `-kernel`);
- the base checkpoint to restore from (`-checkpoint-dir`);
- fault-injection parameters: number of faults per run (`-num-faults`), delay before the first injection in ticks (`-first-delay`), the starting tick (`-tick-start`), delay between successive faults in ticks (`-delay`), and number of bits per fault (`-num-bits`);
- parallelism and total workload control: number of concurrent runs (`-parallel`) and total number of simulations (`-num-sims`);
- output management: output directory prefix (`-outdir-prefix`) and the golden-run output directory used for reference timing (`-golden-outdir`);
- and finally the seed (`-seed`).

**Listing 4.2:** `InjectV` random campaign setup

```
python3 ./runner/campaign_manager.py random \
    --opensbi ./image/fw_jump.elf \
    --disk ./image/riscv_disk \
    --kernel ./image/vmlinux \
    --gem5 ./gem5/build/RISCV/gem5.opt \
    --checkpoint-dir test_run/verifyPIN_0/boot_checkpoint \
    --num-faults 8 \
    --first-delay 100 \
    --tick-start 641434903000 \
    --delay 2000 \
    --num-bits 1 \
    --parallel 4 --num-sims 400 \
    --outdir-prefix test_run/verifyPIN_0/random_main \
    --golden-outdir test_run/verifyPIN_0/m5out_golden/ \
    --seed 123456789
```

## 4.3 Results

This section will contain the summary of the result from the simulations and for the resulted successful injections, the ones that generate an attack on the two binary versions.

### 4.3.1 Random campaign

In this section are presented the results from the random campaign, with detailed results for each of the injection points for VerifyPIN_0 and VerifyPIN_4. Each table has the counts for the occurrences for the observed outputs, classified based on the outcome.

#### VerifyPIN_0

In this section are presented the detailed results coming from the 4 random campaigns that have been illustrated before.

**Table 4.2:** v0 main results

| **v0 main** | **count** |
|---|---|
| `crash` | |
| seg_fault | 92 |
| illegal instruction | 3 |
| `no_effect` | |
| [@] g_countermeasure = 0, g_authenticated = 0, g_ptc = 2 | 290 |
| `countermeasure` | |
| [@] g_countermeasure = 3, g_authenticated = 0, g_ptc = 0 | 1 |
| `other` | |
| [@] g_countermeasure = 0, g_authenticated = 0, g_ptc = 0 | 14 |

The `main` campaing resulted in 95 crashes, 290 runs that had no effect and the program output remained the default one, one case where the countermeasure is reported as a number that should not be normally possible and, the tries counter is zeroed.

**Table 4.3:** v0 verifyPIN results

| **v0 verifyPIN** | **count** |
|---|---|
| `crash` | |
| seg_fault | 31 |
| `no_effect` | |
| [@] g_countermeasure = 0, g_authenticated = 0, g_ptc = 2 | 369 |

The `verifyPIN` campaign resulted in just crashes in 31 cases and 369 *no_effect*.

**Table 4.4:** v0 byteArrayCompare results

| **v0 byteArrayCompare** | **count** |
|---|---|
| `crash` | |
| seg_fault | 82 |
| no exec | 1 |
| `no_effect` | |
| [@] g_countermeasure = 0, g_authenticated = 0, g_ptc = 2 | 496 |
| `success` | |
| [@] g_countermeasure = 0, g_authenticated = 1, g_ptc = 3 | 18 |
| `other` | |
| [@] g_countermeasure = 0, g_authenticated = 0, g_ptc = 0 | 3 |

For the `byteArrayCompare` function campaign we have 83 crashes, of those 1 returns no binary output. In the other non crashed instances we have we have a successful injection that produces an attack, for our testing and InjectV features an attack is a case where the program returns that the authentication is 1 and the countermeasure are not triggered (*g_countermeasure = 0*).

**Table 4.5:** v0 dec_tries results

| **v0 dec_tries** | **count** |
|---|---|
| `crash` | |
| seg_fault | 54 |
| no exec | 2 |
| `no_effect` | |
| [@] g_countermeasure = 0, g_authenticated = 0, g_ptc = 2 | 441 |
| `success` | |
| [@] g_countermeasure = 0, g_authenticated = 0, g_ptc = 34 | 2 |
| `other` | |
| [@] g_countermeasure = 0, g_authenticated = 0, g_ptc = -126 | 1 |

For the `dec_tries` test we have two ghost executions, 441 *no_effects* 2 successful attacks as two runs managed to increase the try counter path the original 3 value. The run that produces the negative number as try counter is labeled as other, even though due to the initial check in the code of *verifyPIN* function, returns immediately if the number of tries is negative. Going deeper on the two successful runs we can analyze the InejctV FaultInjector log to see where the fault was injected.

```
1   641831922100: board.faulter: Injecting fault at tick
    641831922100: reg=integer:integer[15] bit=5 numBits=1
2   641831922100: board.faulter: Current register content: 3
3   641831922100: board.faulter: Updated register content: 35
```

If we observe the above listing we can see that the target register was the int register number 15, the content when the register is read was 3 and the result with the applied bit flips is 35. Due to the fact that binary returned 34 as the tries counter status, this means that the fault was injected when the *g_ptc* value is read from memory, hence the *"Current register content: 3"*. Then the bit is flipped and the value is then decremented.

### VerifyPIN_4

For v4 we have to keep in mind that due to the now implemented boolean hardening, false will be `0x55` and true will be `0xaa`, this is useful to read some of the tables to better understand the results.

**Table 4.6:** v4 main results

| **v4 main** | **count** |
|---|---|
| `crash` | |
| seg_fault | 91 |
| illegal_instr | 2 |
| Aborted (fatal glibc error) | 1 |
| `no_effect` | |
| [@] g_countermeasure = 0, g_authenticated = 55, g_ptc = 2 | 291 |
| `countermeasure` | |
| [@] g_countermeasure = 3, g_authenticated = 55, g_ptc = 84 | 1 |
| [@] g_countermeasure = 85, g_authenticated = 55, g_ptc = 0 | 2 |
| `other` | |
| [@] g_countermeasure = 0, g_authenticated = 55, g_ptc = 0 | 12 |

For v4 `main` we have 94 crashes, 291 case where the fault did not result in

change in the behavior of the binary output. Then we have two different variants where the countermeasures report random values, and the tries counter has also variations.

**Table 4.7:** v4 verifyPIN results

| **v4 verifyPIN** | **count** |
|---|---|
| `crash` | |
| seg_fault | 49 |
| `no_effect` | |
| [@] g_countermeasure = 0, g_authenticated = 55, g_ptc = 2 | 329 |
| `countermeasure` | |
| [@] g_countermeasure = 1, g_authenticated = 55, g_ptc = 2 | 20 |
| `success` | |
| [@] g_countermeasure = 0, g_authenticated = 55, g_ptc = 3 | 2 |

In v4 `verifyPIN` runs, the crashes are all due to segmentation faults, the *no_effect* are 329. They are them followed by 20 instances where the countermeasure is triggered but no authentication is done. And then most notably, 2 instances where the countermeasures are not triggered, the authentication is not done but the tries counter is reset to 3. This can enable brute-force attacks to guess the PIN. Further analysis is needed on the run outcome to identify the root cause that enabled the fault injection to bypass the double check on the *g_ptc* counter, that would have prevented the reset to the original value of 3.

**Table 4.8:** v4 byteArrayCompare results

| **v4 byteArrayCompare** | **count** |
|---|---|
| `crash` | |
| seg_fault | 39 |
| `no_effect` | |
| [@] g_countermeasure = 0, g_authenticated = 55, g_ptc = 2 | 356 |
| `countermeasure` | |
| [@] g_countermeasure = 1, g_authenticated = aa, g_ptc = 3 | 1 |
| [@] g_countermeasure = 1, g_authenticated = 55, g_ptc = 2 | 4 |

For the injections in `byteArrrayCompare` section we have 39 segmentation faults, 356 that produces no variation of the output. After those we have 1 run where the authentication is successful but the countermeasures are triggered. The 4 remaining ones triggered countermeasures but did not produce any successful authentication.

**Table 4.9:** v4 verifyPIN_2 results

| **v4 verifyPIN_2** | **count** |
|---|---|
| `crash` | |
| seg_fault | 38 |
| `no_effect` | |
| [@] g_countermeasure = 0, g_authenticated = 55, g_ptc = 2 | 326 |
| `countermeasure` | |
| [@] g_countermeasure = 1, g_authenticated = aa, g_ptc = 3 | 1 |
| [@] g_countermeasure = 1, g_authenticated = 55, g_ptc = 2 | 35 |

Lastly we have the results from the campaigns on the second insertion point in *verifyPIN*. This resulted in 38 segmentation faults and 326 that produced no variation. The rest of the runs resulted in activations of the countermeasures.

Here follows the summary table with the general statistics for the `VerifyPIN_0` campaigns.

**Table 4.10:** v0 overall statistics

| **Category** | **Count** | **Percentage (%)** |
|---|---:|---:|
| crash | 265 | 13.9 |
| no_effect | 1596 | 84.0 |
| countermeasure | 1 | 0.05 |
| success | 20 | 1.1 |
| other | 18 | 0.9 |
| **Total** | 1900 | 100.0 |

And here the one for the `VerifyPIN_4` campaigns.

**Table 4.11:** v4 overall statistics

| **Category** | **Count** | **Percentage (%)** |
|---|---:|---:|
| crash | 220 | 13.8 |
| no_effect | 1302 | 81.4 |
| countermeasure | 64 | 4.0 |
| success | 2 | 0.1 |
| other | 12 | 0.8 |
| **Total** | 1600 | 100.0 |

Overall we can see the on both the versions the faults generated no observable

difference in the binary output, crash figures remained somewhat consistent between the tho binary versions. But countermeasures triggers where much more present in the v4 version, this has to be expected due to the presence of the hardenings. Also based on this we can observe the drastically reduced number of successful attacks, that went from 20, in v0, to 2 in v4.

### Precise fault campaigns

**VerifyPIN_0** The precise campaign using the fault file 4.1 produces the following simulation output:

```
Instance 0 finished. Result code: ('NO_EFFECT', {})
Instance 1 finished. Result code: ('SUCCESS', {'matched': '[@] g_countermeasure = 0, g_authenticated = 0, g_ptc = 3'})
Instance 2 finished. Result code: ('NO_EFFECT', {})
```

**Figure 4.2: VerifyPIN_0** precise campaign results

In particular, run number 2 (corresponding to line 2 of the fault file 4.1) produced the expected outcome, restoring *g_ptc* to `3`. In practice, if an attacker can consistently inject this fault, they can effectively brute-force the PIN, gaining unlimited attempts without triggering the countermeasures.

**VerifyPIN_4** From the limited analysis done on the `VerifyPIN_4` binary we were not able to identify a feasible attack path with a single fault injection, or a number of fault injections with the same parameters during the execution. Because of the structure of version 4, locating a fault injection point that bypasses the *g_ptc* decrement is extremely hard with our current modeling capabilities, or would require substantially longer simulations to exhaust the search space of possible fault locations and timings.

Building from the successful injection run observed in the random campaign would require developing accessory tools that can instrument the code and compare the full execution trace of the single run to the golden run. To implement this we would need to introduce alternative solutions for full trace collection, as full traces require considerable storage space, the golden run trace was around 7GB in size, meaning that saving all traces is not possible. More conceptualization work has to be done on this front.

Regarding the precise fault injections, due to the current limitation of InjectV, in precise mode, only faults with the same parameters can be run at intervals; we cannot insert two or more faults with different parameters at different temporal locations in the precise mode.

# Chapter 5

# Conclusions and Future Work

This chapter discusses the main takeaways of this work and outlines possible future developments to advance it further.

## 5.1 Main Takeaways

As described in the previous chapters, there are many classes of fault injection techniques. Each presents subtle differences in operating procedures, available instrumentation, current technology, and manufacturing processes that can influence the outcomes of testing scenarios. Combined with the growing importance of security-related testing, where attack complexity can be immense and hardware complexity increases exponentially as technology evolves, this highlights the need to find alternatives for the initial stages of product development that do not rely on direct access to hardware.

Creating, managing, and instrumenting hardware-based testing platforms can be extremely expensive, as they require specialized equipment and strict procedures to ensure reliable testing mechanisms for product evaluation. In cases where only software testing or limited types of injection scenarios are needed, simulated testing environments can be beneficial, especially due to their speed and relatively low effort requirements.

Projects like **InjectV** provide a fast prototyping solution for testing software against different classes of injections, allowing researchers to study their effects on software. In particular, InjectV's current focus on register-based injection makes it a valuable tool for analyzing the impact of such injections without the need for a

full hardware platform. Furthermore, its extensibility and configurability, features inherited from **gem5**, can be instrumental in adapting it for comprehensive security testing pipelines on a larger scale.

In addition to its extensibility and configurability, **InjectV** also positions itself as a relatively easy-to-use teaching tool for demonstrating the effects of injections on software running on RISC-V. This makes it useful for introducing hardening techniques for both embedded and non-embedded code, forming a foundation for enhancing software robustness against various classes of injections.

## 5.2 Future Developments

By correlating practical physical injections with the feature set of InjectV, we can identify several necessary additions that would greatly expand its capabilities and the range of possible testing. A primary enhancement would be adding the ability to target memory for injections. This would enable testing memory reliability and examining the effects of software behavior on unreliable memory. The gem5 interfaces can support this by applying the same base concept used in the current version of InjectV, using *SimObjects* for planning and execution. By combining gem5's internal API with precisely placed hooks into the memory subsystem, it would be possible to issue timed memory reads and writes, effectively simulating fault injections in memory objects. Consequently, this also opens the possibility of inserting hook points in the cache. These additions would significantly expand the capabilities of InjectV, enabling diverse fault models and allowing the simulation of more complex scenarios and injection workflows.

One aspect to consider is that even with these additions, InjectV would still be limited to injecting faults at the ISA level, meaning it cannot simulate cases where faults occur within the processor's control logic. However, recent literature presents other projects that, starting from RTL-level simulations, allow exploration of this type of injection.

Even with these limitations, InjectV provides an extensible and efficient framework for testing software against fault injection campaigns.

# Chapter 6

# VerifyPIN

## 6.1 VerifyPIN_0

**Listing 6.1:** `verifyPIN_0` code body

```
1  extern SBYTE g_ptc;
2  extern BOOL g_authenticated;
3  extern UBYTE g_userPin[PIN_SIZE];
4  extern UBYTE g_cardPin[PIN_SIZE];
5
6  #ifdef INLINE
7  __attribute__((always_inline)) inline BOOL byteArrayCompare(UBYTE* a1
       , UBYTE* a2, UBYTE size)
8  #else
9  BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2, UBYTE size)
10 #endif
11 {
12     int i;
13     for(i = 0; i < size; i++) {
14         if(a1[i] != a2[i]) {
15             return 0;
16         }
17     }
18     return 1;
19 }
20
21 BOOL verifyPIN() {
22     g_authenticated = 0;
23
24     if(g_ptc > 0) {
25         if(byteArrayCompare(g_userPin, g_cardPin, PIN_SIZE) == 1) {
26             g_ptc = 3;
27             g_authenticated = 1; // Authentication();
28             return 1;
```

```
29          } else {
30              g_ptc--;
31              return 0;
32          }
33      }
34
35      return 0;
36  }
```

## 6.2 VerifyPIN_4

**Listing 6.2:** `verifyPIN_4` code body

```
1  extern SBYTE g_ptc;
2  extern BOOL g_authenticated;
3  extern UBYTE g_userPin[PIN_SIZE];
4  extern UBYTE g_cardPin[PIN_SIZE];
5
6  BOOL verifyPIN() {
7      int i;
8      BOOL status;
9      BOOL diff;
10     int stepCounter;
11     SBYTE ptcCpy = g_ptc;
12     g_authenticated = BOOL_FALSE;
13
14     if(g_ptc > 0) {
15         if(ptcCpy != g_ptc) {
16             countermeasure();
17         }
18         g_ptc--;
19         if(g_ptc != ptcCpy-1) {
20             countermeasure();
21         }
22         ptcCpy--;
23
24         status = BOOL_FALSE;
25         diff = BOOL_FALSE;
26         stepCounter = 0;
27         for(i = 0; i < PIN_SIZE; i++) {
28             if(g_userPin[i] != g_cardPin[i]) {
29                 diff = BOOL_TRUE;
30             }
31             stepCounter++;
32         }
33         if(stepCounter != PIN_SIZE) {
34             countermeasure();
```

```
35          }
36          if(i != PIN_SIZE) {
37              countermeasure();
38          }
39          if (diff == BOOL_FALSE) {
40              status = BOOL_TRUE;
41          } else {
42              status = BOOL_FALSE;
43          }
44
45          if(status == BOOL_TRUE) {
46              if(ptcCpy != g_ptc) {
47                  countermeasure();
48              }
49              g_ptc = 3;
50              g_authenticated = BOOL_TRUE; // Authentication();
51              return BOOL_TRUE;
52          }
53      }
54
55      return BOOL_FALSE;
56  }
```

`numpy`

# Bibliography


[1] D. H. Habing. «The Use of Lasers to Simulate Radiation-Induced Transients in Semiconductor Devices and Circuits». In: *IEEE Transactions on Nuclear Science* 12.5 (Oct. 1965), pp. 91–100. ISSN: 1558-1578. DOI: 10.1109/TNS.1965.4323904. URL: https://ieeexplore.ieee.org/document/4323904/ (visited on 09/20/2025) (cit. on p. 3).

[2] Sergei P. Skorobogatov and Ross J. Anderson. «Optical Fault Induction Attacks». en. In: *Cryptographic Hardware and Embedded Systems - CHES 2002*. Ed. by Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Burton S. Kaliski, Çetin K. Koç, and Christof Paar. Vol. 2523. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 2–12. ISBN: 978-3-540-00409-7 978-3-540-36400-9. DOI: 10.1007/3-540-36400-5_2. URL: http://link.springer.com/10.1007/3-540-36400-5_2 (visited on 09/20/2025) (cit. on p. 4).

[3] Brice Colombier, Paul Grandamme, Julien Vernay, Émilie Chanavat, Lilian Bossuet, Lucie De Laulanié, and Bruno Chassagne. «Multi-Spot Laser Fault Injection Setup: New Possibilities for Fault Injection Attacks». en. In: *Smart Card Research and Advanced Applications*. Ed. by Vincent Grosso and Thomas Pöppelmann. Vol. 13173. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 151–166. ISBN: 978-3-030-97347-6 978-3-030-97348-3. DOI: 10.1007/978-3-030-97348-3_9. URL: https://link.springer.com/10.1007/978-3-030-97348-3_9 (visited on 08/11/2025) (cit. on p. 4).

[4] M. Dumont, M. Lisart, and P. Maurine. «Modeling and Simulating Electromagnetic Fault Injection». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.4 (Apr. 2021), pp. 680–693. ISSN: 1937-4151. DOI: 10.1109/TCAD.2020.3003287. URL: https://ieeexplore.ieee.org/document/9120350/ (visited on 09/21/2025) (cit. on pp. 4, 5).

[5] Ross Anderson and Markus Kuhn. «Tamper Resistance - a Cautionary Note». en. In: () (cit. on p. 5).

[6] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. «VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface». en. In: () (cit. on p. 5).

[7] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. «Plundervolt: Software-based Fault Injection Attacks against Intel SGX». en. In: *2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2020, pp. 1466–1482. ISBN: 978-1-72813-497-0. DOI: 10.1109/SP40000.2020.00057. URL: https://ieeexplore.ieee.org/document/9152636/ (visited on 09/22/2025) (cit. on p. 5).

[8] *There's A Hole In Your SoC: Glitching The MediaTek BootROM*. en. URL: https://www.nccgroup.com/research-blog/there-s-a-hole-in-your-soc-glitching-the-mediatek-bootrom (visited on 09/23/2025) (cit. on p. 6).

[9] *CEO interview: Chips Act boost for RISC-V ...* URL: https://www.eenewseurope.com/en/ceo-interview-chips-act-boost-for-risc-v/ (visited on 09/27/2025) (cit. on p. 6).

[10] Joel Coburn et al. «Meta's Second Generation AI Chip: Model-Chip Co-Design and Productionization Experiences». en. In: *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. Tokyo Japan: ACM, June 2025, pp. 1689–1702. ISBN: 9798400712616. DOI: 10.1145/3695053.3731409. URL: https://dl.acm.org/doi/10.1145/3695053.3731409 (visited on 09/27/2025) (cit. on pp. 6, 7).

[11] Jason Lowe-Power et al. *The gem5 Simulator: Version 20.0+*. en. arXiv:2007.03152 [cs]. Sept. 2020. DOI: 10.48550/arXiv.2007.03152. URL: http://arxiv.org/abs/2007.03152 (visited on 09/24/2025) (cit. on pp. 8, 11).

[12] *Intel® Simics® Simulator*. en. URL: https://www.intel.com/content/www/us/en/developer/articles/tool/simics-simulator.html (visited on 10/07/2025) (cit. on p. 9).

[13] *riscv-software-src/riscv-isa-sim*. original-date: 2011-08-26T20:00:24Z. Oct. 2025. URL: https://github.com/riscv-software-src/riscv-isa-sim (visited on 10/07/2025) (cit. on p. 9).

[14] gem5bootcamp. *gem5 Bootcamp 2022*. en. URL: https://gem5bootcamp.github.io/gem5-bootcamp-env (visited on 09/26/2025) (cit. on p. 10).

[15] Konstantinos Parasyris, Georgios Tziantzoulis, Christos D. Antonopoulos, and Nikolaos Bellas. «GemFI: A Fault Injection Tool for Studying the Behavior of Applications on Unreliable Substrates». en. In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Atlanta, GA, USA: IEEE, June 2014, pp. 622–629. ISBN: 978-1-4799-2233-8. DOI: 10.1109/DSN.2014.96. URL: https://ieeexplore.ieee.org/document/6903616/ (visited on 09/26/2025) (cit. on p. 15).

[16] *gem5: Event-driven programming*. URL: https://www.gem5.org/documentation/learning_gem5/part2/events/ (visited on 10/07/2025) (cit. on p. 17).

[17] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. «FISSC: A Fault Injection and Simulation Secure Collection». In: *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings*. 2016, pp. 3–11 (cit. on p. 23).