INSA Toulouse, Département GEI

Hardware Security Lab - Introduction to Fault Injection - Clock Glitching Attack

November, 2025

1 Introduction

In this practical work we will perform an invasive fault-injection attack on a target to alter its runtime behaviour. The lab's objective is not to provide a recipe for exploitation but to let you observe how faults affect real systems, measure fault conditions, and assess the effectiveness of countermeasures. You will record timing parameters, success rates, and observable symptoms (skipped or corrupted instructions, wrong outputs, crashes) and draw conclusions from reproducible data.

Modern systems implement multiple countermeasures to reduce the risk and impact of fault attacks: hardware techniques (clock filters, glitch detectors, brown-out/voltage supervisors, redundant clock domains), architectural or microarchitectural measures (instruction and control-flow integrity checks, pipeline interlocks), and software protections (redundant checks, time redundancy, cryptographic shielding, secure boot). Many defenses combine detection (raise an exception/log when abnormal timing or states occur) with mitigation (abort, reset, or switch to a safe mode).

All experiments must be performed only on lab hardware you own or are explicitly authorized to use. Fault injection against devices you do not own or have explicit permission to test is illegal and may carry civil and criminal penalties.

2 Fault Injection by Clock Glitching, the theory

In the previous lab, we exploited a side-channel leakage that correlated power traces with the secret code to recover the digicode and then proposed a hardened implementation. In this lab we show a different threat: by injecting faults you can bypass the password authentication itself, even if the code has been made resilient to side-channel attacks. The goal is to observe how hardware-level faults can defeat software countermeasures designed for SCA.

Fault injection attacks are intrusive techniques designed to disrupt the normal operation of an algorithm or device. Understanding a fault model, what faults can occur, where, and with what probability, is essential because it frames experiments, helps interpret results, and guides the design of effective countermeasures.

Faults can be induced by many means (clock glitches, electromagnetic or laser pulses, voltage glitches, etc.); each method has different capabilities and constraints. This lab focuses on clock glitching.

All electronic chips require a stable and reliable clock source to function correctly and execute their instructions as intended. Modern architectures implement a fetch–decode–execute pipeline that runs on each clock cycle to optimize execution. This allows the next instruction to be loaded while the current one is still being executed.

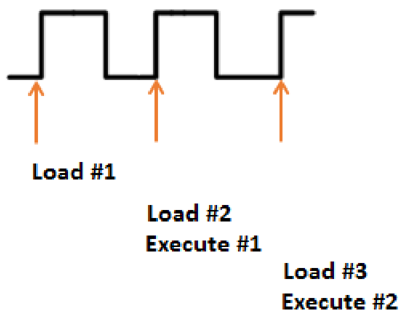

Figure 2.1: Nominal functionning of a pipeline with a correct clock signal.

If we perturb the clock signal, the CPU may not have enough time to complete an instruction. In that case, the instruction can be effectively skipped or only partially executed, leading to altered program behavior.

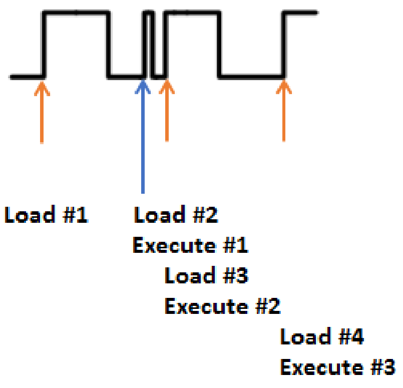

Figure 2.2: Execution of the instructions with a glitched clock signal.

In practice, these attacks can have severe and practical impact. If a glitch is precisely timed, an instruction may not complete or may be partially executed, resulting in a changed software behavior. For example, a carefully placed glitch during a password-check routine can skip or corrupt the instruction that enforces the check, allowing authentication to be bypassed without breaking the protocol, simply by preventing the intended branch or comparison from executing.

3 Evaluation Platform

A system clock, provided either by the ChipWhisperer or the Device Under Test (DUT), is used to generate glitches. These glitches are typically reintroduced into the clock signal, although they can also

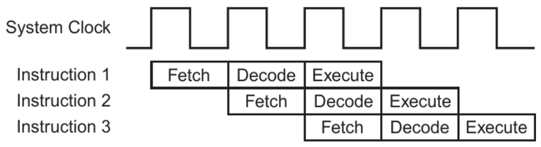

Figure 2.3: Pipeline Overview

be applied independently for other types of fault injection, such as voltage or electromagnetic glitches.

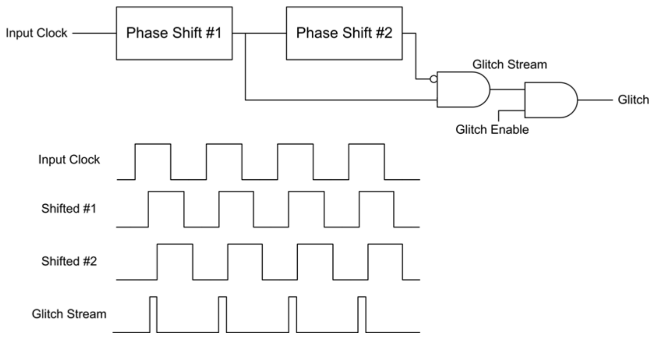

Figure 3.1: Glitch Generation with two variable phase shift modules

The glitched clock is then fed into the DUT's clock input to execute the attack: by shortening or distorting one or more clock periods the CPU's pipeline can miss or partially execute instructions, producing skipped branches, corrupted registers or unexpected control-flow changes. In the lab you will apply these glitched pulses at specific program points, observe the device's reactions (wrong outputs, crashes, or bypassed checks), and record timing parameters and success rates to correlate glitch timing with observed faults.

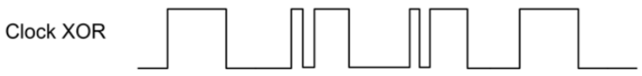

Figure 3.2: Glitched Clock Signal.

4 Provided Material

The archive `Ressources_CGFI` contains:

- `setup.sh` – installs the ARM toolchain.
- `Makefile` – compiles the victim firmware.
- `bin/` – contains precompiled binaries and Python scripts.
- `target_prog/` – support files for programming the DUT.
- `firmware/` – source code of the victim program.

Before starting:

```
> ./setup.sh
> make
```

This generates `glitch_loop.hex`, the firmware used during the first experiments.

5 Task 0 — Glitching the Loop Function (`'G'`)

The command `'g'` runs a short loop surrounded by a trigger signal. This provides a stable timing reference for injecting clock glitches.

In a typical workflow, a Python script would sweep the glitch parameters and generate a two-dimensional fault map. **For this lab, you are not required to run the script yourself. A precomputed glitch heatmap is provided to you.**

Your task

Using the supplied heatmap:

- interpret the meaning of each region (normal behaviour, partial faults, crashes);
- identify clusters where faults occur consistently;
- determine which ranges of `ext_offset` correspond to "sensitive" timing windows.

These observations will guide your parameter selection for the password bypass in Task 2.

Parameter meanings The glitch module exposes four main parameters:

- `ext_offset`: coarse timing, in clock cycles after the trigger.
- `offset`: fine sub-cycle adjustment.
- `width`: duration of the glitch.
- `repeat`: number of consecutive cycles affected.

Although you will not run the sweep script, understanding these parameters is essential for interpreting the results and planning the next attack.

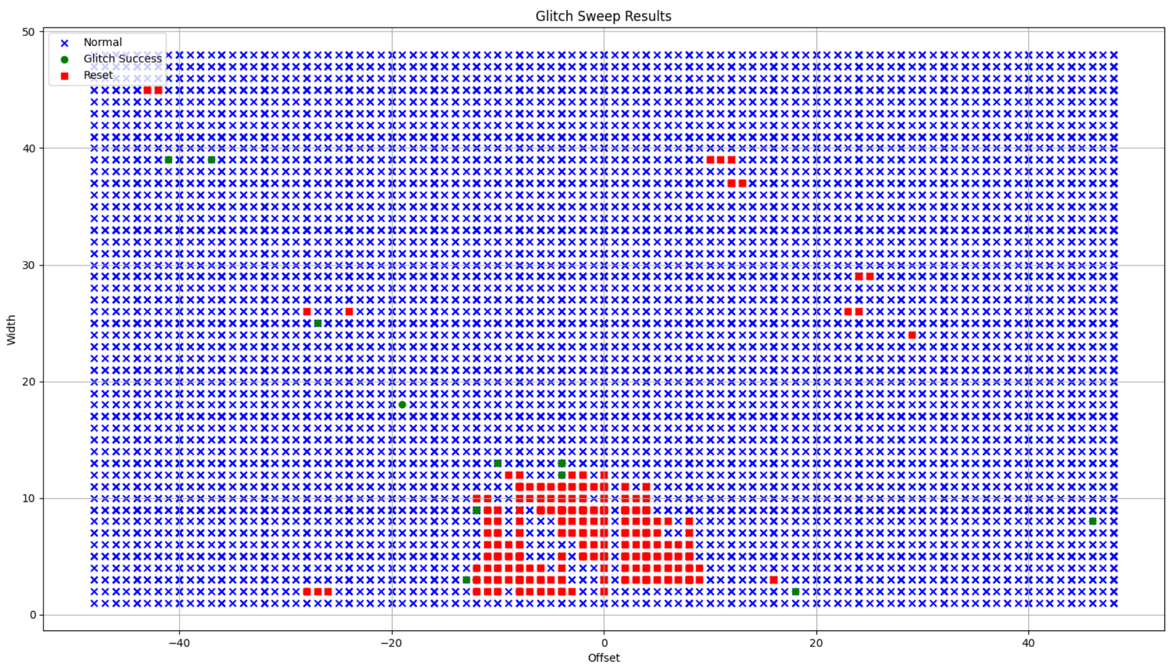

Figure 5.1: Glitch Sweep Results

6 Task 1 — Exploring the Glitch Terminal

Before performing any fault-injection experiments, you will interact with the target using the script `glitch_terminal.py`. This script opens an interactive shell in which you can type commands such as `normal`, `glitch`, and `reset`. Its purpose is to let you experiment with the glitch parameters in real time and observe how the target firmware behaves under different conditions. By manually trying parameter combinations and watching the system's response, you will become familiar with the difference between normal execution, successful glitches, and target crashes.

This script provides an interactive shell that lets you :

- run the loop function normally,
- apply glitch parameters manually,
- reset the device,
- observe how the target responds.

Running the Terminal

Launch the interface with:

```
> ./glitch_terminal.py
```

A prompt `>` will appear, indicating that the script is ready to receive commands.

Available Commands

The terminal understands the following commands:

`normal` Runs the loop function without glitching. Prints whether the behaviour is normal (expected loop counter) or if a reset occurred.

`glitch -o <offset> -e <ext_offset> -w <width> -r <repeat>` Runs the loop while applying a clock glitch with the specified parameters. Outputs either:

- a normal behaviour,
- a successful glitch (`GLITCHED`),
- or a crash/reset.

`reset` Resets the target board.

`quit` Exits the terminal.

All parameters correspond to the firmware's glitch configuration:

- `-e / -ext_offset` : coarse timing offset,
- `-o / -offset` : fine timing adjustment,
- `-w / -width` : glitch width,
- `-r / -repeat` : number of affected cycles.

Your Task

Use this terminal to familiarise yourself with the output format:

- Run `normal` several times and observe the expected loop counter.
- Try a few arbitrary glitch commands, e.g.

  ```
  glitch -o -12 -e 910 -w 8 -r 1
  ```

  and observe whether the target behaves normally, resets, or shows a glitch.
- Use `reset` when the device becomes unresponsive.

The objective is for students to gain intuition about how fault injection parameters influence the timing and reliability of glitch attacks. This hands-on exploration will help you correctly interpret behaviour during the glitch parameter analysis (Task 1) and the password bypass (Task 2).

7 Task 2 — Glitching the Password Check (`'p'`)

The firmware also contains a password-verification routine triggered by the command `'p'` . A correct password yields a success response, while any incorrect password normally leads to an authentication failure.

Your objective is to induce a fault during this verification process so that the firmware accepts an incorrect password.

Important Note

The script `glitch_terminal.py` *cannot* send the `’p’` command. It is only meant for triggering the loop function (`’g’`) and manually testing glitch configurations.

To attack the password check, you must use the provided password-attack script, which sends:

```
p <wrong password>
```

while applying the glitch parameters you specify.

Objective

Use the glitch windows identified in Task 1 to bypass the password check while supplying a known incorrect password.

Instructions

1. Open the password-attack script supplied with the lab. This script uploads the firmware, applies glitch parameters, sends the `’p’` command, and prints whether the password was accepted.
2. Select a handful of candidate parameter sets using the heatmap from Task 1. Focus on regions where the loop function showed reliable faults.
3. For each chosen set of parameters, run the password-attack script and record:
   - whether the device behaved normally (password rejected),
   - whether a glitch occurred (password accepted despite being wrong),
   - or whether the target crashed/reset.
4. Adjust the parameters within the sensitive region (e.g., small variations in `offset` or `ext_offset`) until you obtain at least one successful bypass.
5. Once a successful bypass occurs, record the parameter configuration that led to it.

Notes

- You do *not* need to brute-force or search the full parameter space. A few targeted tests based on the heatmap are sufficient.
- Crashes are expected; use the script’s reset function whenever the target stops responding.
- The goal is to glitch the control flow so that the firmware skips the password-check logic. You are *not* recovering the correct password.

8 Task 3 — Implementing Countermeasures

Fault injection attacks can often be mitigated by adding redundancy or verification steps in firmware. In this task, you will modify the firmware to make the password check more resilient to glitches.

Objective

Your goal is to prevent a single glitch from bypassing the password verification. You will implement software-level countermeasures and verify that the target is no longer vulnerable with the same glitch parameters used in Task 2.

Instructions

1. Identify critical operations in the password verification function that could be skipped or corrupted by a glitch.
2. Modify the firmware to protect these operations. Possible strategies include:
   - storing multiple copies of critical variables and comparing them before making decisions,
   - repeating the password check or branching logic,
   - adding cross-checks between independent values,
   - inserting consistency checks on computation results.
3. Recompile and flash the updated firmware.
4. Using `glitch_terminal.py`, attempt to apply the same glitch parameters that previously caused a bypass.
5. Record the behaviour: the target should now reject incorrect passwords consistently.

Expected Outcome

After implementing countermeasures:

- Previously successful glitch parameters should no longer bypass the password check.
- Target resets or invalid responses may still occur, but the authentication logic remains secure.
- You should be able to explain why redundancy or repeated checks reduce vulnerability to fault injection.

Note: The purpose of this task is to demonstrate practical mitigation strategies, not to create an invulnerable system. Focus on improving robustness against simple clock glitch attacks.