ULB CYBERSECURITY RESEARCH CENTER

life.augmented

Fault injection through hardware simulation

How to evaluate your countermeasures against fault injections with Unicorn

by Cédrick De Pauw

on July 3, 2023

CYBERSECURITY RESEARCH CENTER

ULB

» Context

Sources of faults in a system:

* Harsh environment
* Adversary

Risks?

* Undesired change in a program control flow
* System crash

Solution?

1. Implement countermeasures
2. Test them in practice (expertise is required)
3. Identify new vulnerabilities
4. Repeat

» Fault Injection Simulation

Why would hardware simulation be useful?

- Does not require expensive equipment
- No risk to damage important components
- Allows precise actions on instructions and registers
- Environment is easy to setup
- Simulations are fast to perform and reproducible

Ideal to test countermeasures against fault injections.

» Project Description

Objectives:

* set up simulation environment
* perform firmware vulnerability analysis
    * → **automatic fault injection simulation**
    * → SCA feature: cycle count annotation

Setup:

* ARM GNU Toolchain: C code compilation
* Unicorn: CPU emulation
* Python: simulation tool implementation

CYBERSECURITY RESEARCH CENTER ULB

» Methods

Several methods exist to inject faults in a system:

* Clock fault injection
* Voltage fault injection
* Electromagnetic fault injection
* Optical fault injection
* ...

» Fault Model Characteristics

Two main categories of fault injections:

- Global: affects global parameters (voltage, clock)
- Local: precise fault location (expensive equipment)

Fault models are essentially characterized by:

- Location
  - → Spatial: point or area in the system
  - → Temporal: instant during the execution
  - → Precision level: bit, byte, variable, ...
- Impact: skip, stuck-at, bit-flip, random byte, ...

» Countermeasures: Examples

* Double check important functions execution
* Double check branching conditions
* Verify that loops have not been aborted
* Avoid boolean values to access critical functions

M. Witteman. *Secure application programming in the presence of side channel attacks*. Aug. 2017. URL: `https://www.riscure.com/publication/secure-application-programming-presence-side-channel-attacks/`.

» Unicorn Engine

Overview

* QEMU-based open-source project
* CPU emulator
* Multiple target architectures (ARM, ARM64, MIPS, RISC-V, ...)
* Implemented in C, and many bindings exist
* No requirement regarding emulation context
* Easy to instrument (hooks on specific events)

» Unicorn Engine

Emulation

1. Create Unicorn instance
2. Read binary
3. Map program segments into instance memory
4. Prepare initial state/context (optional)
5. Define start and end addresses
6. Add hooks (syscalls, tracing)
7. Run the simulation

» Unicorn Engine

Inject fault using method **(1)** or **(2)**:

1. → 6. Prepare Unicorn instance (see previous slide)

7. Add fault injection hooks **(1)**

8. Start the simulation

9. Perform fault injection **(2)**:
   * halt simulation
   * inject fault
   * resume

Description

- Based on Unicorn and dedicated to 32-bit ARM architectures
- Using Python binding: easy to adapt to your needs
- Two command line scripts:
  - → one to explore possible fault attacks
  - → one to perform/reproduce a specific fault attack
- User may define external functions:
  - → to set emulation context
  - → to perform fault injection
  - → to extract fault detection flags

» Fault Models

- Skip instruction fault model
- Register-based fault models:
  - → "on instruction" fault (transient)
  - → stuck-at fault (permanent)
  - → bit flip fault (transient)
- Memory-based fault models:
  - → "on write access" fault (permanent)
  - → "on read access" fault (transient)

Fault models are implemented using method **(1)**, i.e. hooks.

Simulated Execution States

State
: Set of values extracted from registers and memory regions at a given point of the emulation

Initial state
: State before the emulation has been started

Final state
: State after the emulation has been completed

Reference state
: Final state of a “sane” emulation

» Status of Fault Attacks

- If fault detection flags are extracted through external functions:
  - → fault is ignored if detection flag was raised,
  - → otherwise, fault injection may be logged.

- Logging of fault injections works as follows:
  - → fault injection is logged if a delta appears between the final state and the reference state,
  - → otherwise, fault injection is ignored.

Exploration of Fault Attacks Space

1. Select desired fault models
2. Try to perform a fault injection attack:
   - → for each fault model,
   - → for each instruction,
   - → depending on written/read registers (if register-based),
   - → for each value of the fault model parameters.
3. Identify the status of the fault attack
4. Log it if necessary

Note: limited set of values for each parameter of the different fault models

Demo.

» Conclusion

- Allows to inject faults in program running on specific architectures
- Should be useful to check if countermeasures work as expected
- Should allow to detect unknown vulnerabilities
- Does not allow to test all specific countermeasures (e.g. delay-based)
- Emulation of peripherals on a high level should be possible