Hardware Fault Injection and Co-Simulation

Tutorial

Nikhil

Contents

1 Introduction: Why This Matters

Imagine you're designing a bank vault. You'd want to test if someone could break in, right? Hardware fault injection is similar, but instead of testing physical doors, we're testing digital security systems.

In modern computing, we have:

- **Hardware** (physical chips and circuits)
- **Software** (programs running on that hardware)

Attackers can exploit weaknesses in *both*. Fault injection lets us simulate attacks before our devices reach customers, helping us build more secure systems.

Key Point

The Goal: Learn how to simulate hardware attacks using software tools, without needing expensive physical equipment.

2 Foundation Concepts

2.1 What is RTL? (Hardware Description)

RTL stands for **Register Transfer Level**. Think of it as the "blueprint" for computer chips.

Example

Just like an architect uses blueprints to design a house, hardware engineers use RTL to design chips. The most common RTL language is called **Verilog**.

A Simple Analogy:

- Building a house: Blueprint → Construction → House
- Building a chip: RTL/Verilog → Fabrication → Physical Chip

Here's what RTL code looks like:

Listing 1: Simple RTL Example - A Security Key

```
// This is a hardware register that stores a secret key
always @(posedge clock) begin
    if (reset)
        secret_key <= 128'h0;  // Clear the key
    else if (write_enable)
        secret_key <= new_key_value;  // Store new key
end
```

This code describes a piece of hardware that remembers a secret key, just like a password safe.

2.2 What is QEMU? (Software/Hardware Emulation)

QEMU is like a "virtual computer" that runs on your actual computer.

Example

Imagine you want to test an iPhone app but only have an Android phone. An emulator lets you run iOS on your Android. QEMU does this for entire computer systems.

Why QEMU is Powerful:

1. **Fast:** Test without building physical hardware
2. **Safe:** Mistakes won't break real devices
3. **Flexible:** Simulate any computer system
4. **Controllable:** Inject faults precisely when needed

Listing 2: QEMU Attack Simulation Example

```
// Attacker code running in QEMU
if (attack_mode_enabled) {
    // Trigger a fault at a specific time
    inject_voltage_glitch();
    send_fault_signal_to_hardware();
}
```

2.3 What is Fault Injection?

Fault injection means deliberately causing errors to test system behavior.

Real-World Analogy:

- **Car safety testing:** Crash test dummies (fault injection) help design safer cars
- **Hardware security:** Simulated attacks (fault injection) help design more secure chips

Key Point

We're not trying to break things maliciously. We're testing defenses by simulating what an attacker might do.

3 Types of Fault Attacks

3.1 Voltage Glitching

What it is: Temporarily changing the power supply voltage to cause errors.

Physical Analogy: Imagine flickering the lights in a house. For a brief moment, some electronics might malfunction.

How it attacks security:

Listing 3: Vulnerable Security Check

```
// This should check if the password is correct
if (password == correct_password) {
    grant_access = 1;
} else {
    grant_access = 0;
}
```

During voltage glitch:

Listing 4: What Happens During Attack

```
// The comparison might fail or skip
if (password == correct_password) {  // <- Glitch here!
    grant_access = 1;  // Might execute anyway!
}
```

Important

A voltage glitch can make a chip skip instructions or compute wrong values, potentially bypassing security checks.

3.2 Bit Flip Attacks

What it is: Changing a single bit (0 to 1, or 1 to 0) in memory or a register.

Real-World Example:

Correct password hash: `0xFF81A2B3`

After 1-bit flip: `0xFF81A2B2` (last bit changed)

Listing 5: Bit Flip Attack on Privilege Level

```
// User privilege level (should be 0 for normal user)
user_privilege = 0;  // Binary: 0000

// After bit flip attack
user_privilege = 1;  // Binary: 0001 (now admin!)
```

3.3 Clock Glitching

What it is: Manipulating the clock signal that synchronizes chip operations.

Analogy: Like speeding up a metronome. Musicians might miss beats; chips might skip instructions.

Listing 6: Timing-Dependent Security

```
always @(posedge clock) begin
    step1: verify_signature();
    step2: check_permissions();  // <- Might skip if clock too
       fast
    step3: execute_command();
end
```

4 Co-Simulation: Connecting RTL and QEMU

4.1 Why Co-Simulation?

Modern systems have:

- **Hardware** (chip designs in RTL)
- **Software** (operating systems, applications in QEMU)

Co-simulation means running both simultaneously, allowing:

1. Software to trigger hardware faults
2. Hardware to report results to software
3. Complete system testing before manufacturing

Example

Think of co-simulation like testing a self-driving car:

- **QEMU:** Simulates the car's computer and software
- **RTL:** Simulates the sensors and control circuits
- **Together:** Complete system test without a real car

4.2 The Communication Bridge

RTL and QEMU communicate through a "bridge":

Listing 7: RTL Side - Receiving Fault Commands

```
// Hardware listens for fault signals
always @(posedge clock) begin
    if (fault_injection_signal) begin
        // Simulate voltage glitch
        key <= 128'h0;  // Clear the secret key!
    end
end
```

Listing 8: QEMU Side - Sending Fault Commands

```
void trigger_hardware_attack() {
    // Software sends signal to hardware
    write_to_hardware_register(FAULT_ADDR, FAULT_ENABLE);

    // Wait for hardware to process
    wait_cycles(10);

    // Read result
    int result = read_from_hardware_register(STATUS_ADDR);
}
```

5 Practical Workflow

5.1 Step 1: Design Your Hardware (RTL)

Create a Verilog module with fault injection capability:

Listing 9: Secure Hardware with Fault Injection

```
module secure_module (
    input wire clock,
    input wire reset,
    input wire fault_inject,  // Fault injection input
    input wire [127:0] input_key,
    output reg [127:0] stored_key,
    output reg access_granted
);

always @(posedge clock) begin
    if (reset) begin
        stored_key <= 128'h0;
        access_granted <= 0;
    end else if (fault_inject) begin
        // Simulate fault: corrupt the key
        stored_key <= 128'hDEADBEEF;
    end else begin
        // Normal operation
        if (input_key == stored_key)
            access_granted <= 1;
        else
            access_granted <= 0;
    end
end

endmodule
```

5.2 Step 2: Simulate Hardware (Verilator)

Verilator converts your RTL into a fast C++ simulation:

Listing 10: Compiling RTL with Verilator

```
# Convert Verilog to C++ simulation
verilator --cc secure_module.v --exe testbench.cpp

# Compile the simulation
make -C obj_dir -f Vsecure_module.mk

# Run simulation
./obj_dir/Vsecure_module
```

5.3 Step 3: Set Up QEMU

QEMU runs your attack software:

Listing 11: Attack Software in QEMU

```c
#include <stdio.h>
#include <stdint.h>

// Memory-mapped hardware addresses
#define HW_KEY_ADDR    0x40000000
#define HW_FAULT_ADDR  0x40000010
#define HW_STATUS_ADDR 0x40000020

void perform_fault_attack() {
    printf("Stage 1: Normal access attempt...\n");
    uint32_t wrong_key = 0x12345678;
    *(volatile uint32_t*)HW_KEY_ADDR = wrong_key;

    int status = *(volatile uint32_t*)HW_STATUS_ADDR;
    printf("Access granted: %d (should be 0)\n", status);

    printf("Stage 2: Injecting fault...\n");
    *(volatile uint32_t*)HW_FAULT_ADDR = 1;  // Trigger fault

    printf("Stage 3: Retry access...\n");
    status = *(volatile uint32_t*)HW_STATUS_ADDR;
    printf("Access granted: %d (might be 1!)\n", status);
}

int main() {
    printf("=== Fault Injection Attack Simulation ===\n");
    perform_fault_attack();
    return 0;
}
```

5.4 Step 4: Connect Them Together

The bridge code links QEMU and RTL:

Listing 12: Co-Simulation Bridge

```c
#include "Vsecure_module.h"  // Verilator-generated
#include <stdio.h>

// QEMU device callbacks
void qemu_hardware_write(uint64_t addr, uint64_t value) {
    if (addr == 0x40000010) {  // FAULT_ADDR
        // Tell RTL to inject fault
        rtl_model->fault_inject = (value & 1);
    }
}

uint64_t qemu_hardware_read(uint64_t addr) {
    if (addr == 0x40000020) {  // STATUS_ADDR
        // Read RTL status
```

```
        return rtl_model->access_granted;
    }
    return 0;
}
```

6 Complete Example: Breaking a Password Check

Let's walk through a complete attack simulation:

6.1 The Vulnerable System

Listing 13: Vulnerable Password Checker (RTL)

```
module password_checker (
    input wire clock,
    input wire [31:0] input_password,
    input wire check_enable,
    input wire fault_enable,  // For testing
    output reg access_granted
);

localparam CORRECT_PASSWORD = 32'hABCD1234;

always @(posedge clock) begin
    if (fault_enable) begin
        // Fault: Skip password check entirely
        access_granted <= 1;
    end else if (check_enable) begin
        if (input_password == CORRECT_PASSWORD)
            access_granted <= 1;
        else
            access_granted <= 0;
    end
end

endmodule
```

6.2 The Attack

Listing 14: Attack Code

```
int main() {
    // Step 1: Try without fault (should fail)
    printf("Attempt 1: Wrong password, no fault\n");
    set_password(0xWRONG);
    enable_check();
    printf("Result: %d\n", read_access());  // Outputs: 0

    // Step 2: Try with fault injection
```

```
    printf("Attempt 2: Wrong password, WITH fault\n");
    enable_fault();  // <- Attack happens here
    set_password(0xWRONG);
    enable_check();
    printf("Result: %d\n", read_access());  // Outputs: 1 (Hacked!)

    return 0;
}
```

6.3 The Results

```
=== Simulation Output ===
Attempt 1: Wrong password, no fault
Result: 0 (Access Denied - Correct behavior)

Attempt 2: Wrong password, WITH fault
Result: 1 (Access Granted - Security bypassed!)
```

Important

This demonstrates why we need fault-resistant designs. Real attackers can inject similar faults using physical equipment.

7 Real-World Applications

7.1 What You Can Build

After mastering these techniques, you can work on:

1. **Virtual System-on-Chip (SoC)**
   - Complete chip systems in simulation
   - Test interactions between CPU, memory, peripherals
   - Example: ARM processor + custom accelerators
2. **Driver Testing**
   - Test Linux device drivers before hardware exists
   - Simulate hardware errors safely
   - Example: Testing PCIe drivers with fault injection
3. **Secure Enclave Development**
   - Design protected execution environments
   - Test against fault attacks
   - Example: TEE (Trusted Execution Environment) security

4. **Hardware Security Research**
    - Publish research papers on attack/defense techniques
    - Discover new vulnerabilities
    - Develop countermeasures

7.2 Academic and Industry Impact

Key Point

Skills from this tutorial translate directly to:

- Hardware security conferences (CHES, HOST, IEEE S&P)
- Industry positions at AMD, Intel, ARM, Apple
- Security consulting and penetration testing
- Academic research positions

8 Tool Setup Guide

8.1 Minimal Toolchain

Purpose	Tool	Installation
Virtual Hardware	QEMU	`apt install qemu-system`
RTL Design	Verilog	(text editor - any)
RTL Simulation	Verilator	`apt install verilator`
Software Compilation	GCC	`apt install build-essential`
Debugging	GDB	`apt install gdb`
Waveform Viewing	GTKWave	`apt install gtkwave`

Table 1: Essential Tools for Fault Injection

8.2 Installation Steps (Ubuntu/Debian)

Listing 15: Quick Setup Script

```
#!/bin/bash

# Update package list
sudo apt update

# Install core tools
sudo apt install -y \
    qemu-system \
    verilator \
    build-essential \
```

```
    gdb \
    gtkwave \
    git

# Verify installations
echo "=== Verifying Installations ==="
qemu-system-arm --version
verilator --version
gcc --version
gdb --version

echo "Setup complete!"
```

9 Building Your First Project

9.1 Project Structure

```
fault_injection_project/
 rtl/
    secure_module.v       (Your hardware design)
    testbench.v            (Test harness)
 software/
    attack.c               (Attack simulation)
    Makefile
 bridge/
    qemu_rtl_bridge.cpp   (Communication layer)
 scripts/
     build.sh               (Build automation)
     run.sh                 (Run simulation)
```

9.2 Build Script

Listing 16: Automated Build Script

```
#!/bin/bash

echo "Step 1: Compile RTL with Verilator..."
verilator --cc rtl/secure_module.v --exe bridge/qemu_rtl_bridge.
   cpp
make -C obj_dir -f Vsecure_module.mk

echo "Step 2: Compile attack software..."
cd software
make
cd ..

echo "Step 3: Set up QEMU environment..."
qemu-system-arm -M virt -m 256M \
    -kernel software/attack.bin \
```

```
    -nographic -monitor none

echo "Build complete!"
```

10 Advanced Topics Preview

10.1 Countermeasures

Once you understand attacks, you can design defenses:

Listing 17: Fault-Resistant Design Example

```
module secure_checker (
    input wire clock,
    input wire [31:0] password,
    output reg access
);

// Redundant checking
reg check1, check2, check3;

always @(posedge clock) begin
    // Triple redundancy
    check1 <= (password == CORRECT_PASS);
    check2 <= (password == CORRECT_PASS);
    check3 <= (password == CORRECT_PASS);

    // Majority voting - harder to fault
    access <= (check1 & check2) |
              (check2 & check3) |
              (check1 & check3);
end

endmodule
```

10.2 Power Analysis

Combine fault injection with power analysis for advanced attacks:

- Measure power consumption during operations
- Identify when cryptographic keys are used
- Inject faults at precise moments

11 Learning Path

11.1 Beginner Projects (Weeks 1-4)

1. Build a simple password checker in Verilog

2. Simulate it with Verilator
3. Write basic fault injection testbench
4. Observe security failures

11.2 Intermediate Projects (Weeks 5-12)

1. Set up QEMU with custom device
2. Bridge QEMU and Verilator
3. Implement voltage glitch simulation
4. Test against AES encryption module

11.3 Advanced Projects (Weeks 13+)

1. Build complete SoC with CPU + custom accelerator
2. Implement secure boot with fault resistance
3. Write research paper on new attack/defense
4. Contribute to open-source security tools

12 Common Pitfalls and Solutions

12.1 Timing Issues

Problem: QEMU and RTL run at different speeds.
Solution: Use synchronization signals:

```
// Wait for hardware to be ready
while (!rtl->ready_signal) {
    rtl->clock = 1;
    rtl->eval();
    rtl->clock = 0;
    rtl->eval();
}
```

12.2 Memory Mapping

Problem: QEMU addresses don't match RTL addresses.
Solution: Create address translation layer:

```
uint64_t translate_address(uint64_t qemu_addr) {
    // Map QEMU 0x40000000 -> RTL 0x0000
    if (qemu_addr >= 0x40000000 && qemu_addr < 0x40010000) {
        return qemu_addr - 0x40000000;
    }
    return 0xFFFFFFFF;  // Invalid
}
```

13 Resources and Further Reading

13.1 Online Resources

- **Verilator Documentation:** `https://verilator.org`
- **QEMU Documentation:** `https://www.qemu.org/docs/`
- **ChipWhisperer:** Hardware fault injection toolkit
- **Riscure Training:** Professional hardware security courses

13.2 Recommended Papers

1. "Fault Attacks on Cryptographic Devices" - Boneh et al.
2. "Clock Glitch Attacks: Methods and Countermeasures" - Yuce et al.
3. "Electromagnetic Fault Injection" - Dehbaoui et al.

13.3 Practice Platforms

- **CryptoHack:** Cryptography challenges (some involve fault attacks)
- **HackTheBox:** Some hardware security challenges
- **IEEE CSAW ESC:** Annual embedded security challenge

14 Conclusion

You now understand:

- What RTL and QEMU are and why they're used together
- How fault injection simulates real-world attacks
- The workflow from RTL design to attack simulation
- Practical applications in industry and research

Key Point

Next Steps:

1. Install the minimal toolchain
2. Build the password checker example
3. Experiment with different fault types
4. Start your first security research project

Remember: This is a *defensive* skill. Use it to build more secure systems, not to attack real devices without authorization.