FAME: Fault-attack Aware Microprocessor Extensions for Hardware Fault Detection and Software Fault Response

Bilgiday Yuce, Nahid Farhady Ghalaty, Chinmay Deshpande, Conor Patrick, Leyla Nazhandali, Patrick Schaumont

Virginia Tech

HASP 2016

This research was supported in part through the NSF Grant 1441710, and in part through the SRC.
• Threat model expands from software into hardware.
• Inject engineered faults into with a specific security objective in mind

Cryptographic Algorithm

① Fault Injection
- Clk
- Vdd
- Laser
- EM

② Faulty Instructions
Inject engineered faults with a specific security objective in mind.

Analyze fault response of the software to break the security.
Why are Faults a Security Issue?

• May enable bypass of security checks/actions

```c
int pinCheck (int userPin) {
    if (userPin == devicePin) {
        unlockPhone();
        return 0;
    } else {
        lockPhone();
        return -1;
    }
}
```

Instruction skip

Phone is unlocked even if userPin is wrong
Why are Faults a Security Issue?

- May enable leakage of secret information
- Even correct output may leak the secret information.

```plaintext
// Elliptic Curve Cryptography
// (Simplified) Scalar Multiplication
...
Q[0] = 2Q[0]; // Inject a fault into P
Q[1] = Q[0] + P;
Q[0] = Q[key_bit];
return Q[0];
```

If the fault does not affect the output, `key_bit` is 0.
• **Fault handling** can be separated into **Fault Detection** and **Fault Response**

• **Fault Detection:**
  • It must be low-latency
  • It must be hard-to-bypass

  → Hardware Fault Detection
How should We Handle Faults?

- Fault handling can be separated into Fault Detection and Fault Response

  - **Fault Detection:**
    - It must be low-latency
    - It must be hard-to-bypass

  - **Fault Response:**
    - It must be application-specific
    - It must be adaptive

  - Hardware Fault Detection
  - Software Fault Response
How should We Handle Faults?

• Fault handling can be separated into Fault Detection and Fault Response

• Fault Detection:
  • It must be low-latency
  • It must be hard-to-bypass

• Fault Response:
  • It must be application-specific
  • It must be adaptive

• **Communication** between Fault Detection and Response
  • SW Fault Response must be aware of HW fault status
  • Processor HW must have features to support fault-attack resistant execution of SW Fault Response

⇒ Hardware Fault Detection
⇒ Software Fault Response
⇒ HW/SW Approach
Overview of FAME

• Combination of HW/SW extensions
• Captures faults using fault detectors in HW level
• User-defined fault policy in SW level
• Fault-attack resistant execution of fault policy
• Status & recovery information to fault handler
Nominal Mode

... User Application

PC-1 : LD MEM[key_address], R1
PC   : LD MEM[state_address], R2
PC+1 : XOR R1, R2, R2
PC+2 : ST MEM[state_address], R2
...
Operation of FAME - 2

Nominal Mode

User Application

PC-1 : LD MEM[key_address], R1
PC   : LD MEM[state_address], R2
PC+1 : XOR R1, R2, R2
PC+2 : ST MEM[state_address], R2
...

Software

Hardware

Vdd
Clk
Data
Operation of FAME - 4

- Locks down FRRs
- Aborts Memory/Register File Writeback
- Annuls Instructions in the Pipeline
- Switches processor to safe mode
- Initiates a non-maskable trap
Operation of FAME - 4

Software

Hardware

Fault Detection Unit (FDU)

Fault Control Unit (FCU)

Fault Response Registers (FRRs)

1. User Application
   - PC-1 : LD MEM[key_address], R1
   - PC  : LD MEM[state_address], R2
   - PC+1 : XOR R1, R2, R2
   - PC+2 : ST MEM[state_address], R2

2. Vdd
   - Clk
   - Data

3. Save Fault Recovery Information
   - Memory
   - Register File
   - Restart Trap Handler

Hardware/Software Extensions

STOP

Nominal Mode

Safe Mode
Operation of FAME - 5

Software

Hardware

Fault Detection Unit (FDU)

Fault Control Unit (FCU)

Vdd
Ck
Data

alarm

Save Fault Recovery Information

Fault Response Registers (FRRs)

Software Trap Handler

READ FRRs

APPLY USER-DEFINED POLICY

Transfer the control to the software trap handler

STOP

Nominal Mode

Safe Mode

1. ... (User Application)
   PC-1 : LD MEM[key_address], R1
   PC  : LD MEM[state_address], R2
   PC+1 : XOR R1, R2, R2
   PC+2 : ST MEM[state_address], R2
   ...

2. Fault Detection Unit (FDU)

3. Fault Control Unit (FCU)

4. Pipeline Registers
   Memory
   Register File

FIRST Hardware/Software Extensions

If “back-to-back fault injections”

Restart Trap Handler
Minimal Trap Handler

; AES Start
...
; Round 10, addRoundKey

LD   MEM[key_address], R1
LD   MEM[state_address], R2
XOR  R1, R2, R2
ST   MEM[state_address], R2
...

; Trap handler
; Read FRRs to determine the valid FRR
; and get the trap return address
call  40001fcc <read_asr>

; Restore the pre-fault state of R2
call  40001f4c <write_asr>

; Return from trap
retn  <next instruction>
• Protects against setup *time violation* attacks
  • Clock/voltage glitching
  • Voltage underfeeding

• Extends a Leon3 processor to FAME
  • 32-bit, 7-stage RISC Pipeline

• Implemented and tested on a Spartan6 FPGA of a SAKURA-G board
Case Study: The Cost of FAME

• Hardware Overhead (9% logic, 15% regs)

<table>
<thead>
<tr>
<th>Component</th>
<th># LUTs</th>
<th># Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON3 (baseline)</td>
<td>3,435</td>
<td>1,275</td>
</tr>
<tr>
<td>FCU and FRR</td>
<td>256 (7.5%)</td>
<td>181 (14%)</td>
</tr>
<tr>
<td>FDU</td>
<td>53 (1.5%)</td>
<td>3 (1%)</td>
</tr>
</tbody>
</table>

• Software Overhead (application dependent)

<table>
<thead>
<tr>
<th>Application</th>
<th># Cycles</th>
<th>Footprint (Byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES (baseline)</td>
<td>17,631</td>
<td>25,964</td>
</tr>
<tr>
<td>AES + fault-Resume</td>
<td>17,810 (+1%)</td>
<td>26,116 (+0.6%)</td>
</tr>
</tbody>
</table>
Conclusions

• FAME provides a HW/SW solution to handle fault attacks:
  • Low-cost
  • Performance-efficient
  • Adaptive
  • Backward-compatible

• FAME is generic
  • Can support multiple fault detectors/sources
  • Can support multiple CPU architectures
Thank you!
How do Existing Methods Handle Faults?

- **Full fault-tolerance**
  - Information, temporal, or spatial redundancy
  - Either in Software or Hardware

- **Detect-and-Despair**
  - Mute/Lock the device
  - Initiate a hard-reset event
  - Kill/Destroy the device
Related Work

- **Software countermeasures**
  - Instruction Duplication, Application-specific redundancy, Concurrent Error Detection
  - Performance Hit and increased footprint

- **Fault tolerant design**
  - Redundant hardware design (similar overhead)

- **Secure Processors**
  - Memory integrity, confidentiality, attestation, isolation, ...
  - Do not address faults
How do FRRs Maintain the Backup State?

• FRR allow to rewind the selected processor state one clock cycle, just before the fault was injected.

• Minimum Content of FRR:
  • Return address to the interrupted program
  • Processor status register
  • Register file inputs of write-back stage
How do FRRs Maintain the Backup State?

• FRR allow to backtrack *selected* processor state one clock cycle, before the fault was detected.

Minimum Content of FRR:
• Return address to the interrupted program
• Processor’s status register
• Register file inputs of write-back stage
Fault Detection Unit

- **Timing Violation Detector**
  - caused by glitches
  - caused by voltage starving

- **Not limited to glitches**
  - Optical, EM, overvoltage, .. detectors
  - Memory/Register checksum
Operation of FAME - 5

Software

Nominal Mode → Safe Mode

Software

Hardware

Fault Detection Unit (FDU) → Fault Control Unit (FCU)

FIRST Hardware/Software Extensions

Fault Detect → Fault Control

Fault Response Registers (FRRs)

READ FRRs
APPLY USER-DEFINED POLICY

Transfer the control to the software trap handler

SAVE Fault Recovery Information

If “back-to-back fault injections”

Restart Trap Handler
Why are Faults a Security Issue?

- May enable leakage of secret information by altering the data flow

```c
// (Simplified) AES AddRoundKey
...
state = secretKey ^ state;
ciphertext = state;
return ciphertext;
```

Zeroize the state

Ciphertext will be equal to secretKey