EXPERIMENTAL EVALUATION OF TWO SOFTWARE COUNTERMEASURES AGAINST FAULT ATTACKS

Nicolas Moro\textsuperscript{1,3}, Karine Heydemann\textsuperscript{3}, Amine Dehbaoui\textsuperscript{2}, Bruno Robisson\textsuperscript{1}, Emmanuelle Encrenaz\textsuperscript{3}

\textsuperscript{1} CEA
Commissariat à l’Energie Atomique et aux Energies Alternatives

\textsuperscript{2} ENSM.SÈ
Ecole Nationale Supérieure des Mines de Saint-Etienne

\textsuperscript{3} LIP6 - UPMC
Laboratoire d’Informatique de Paris 6
Sorbonne Universités - Université Pierre et Marie Curie

\textit{Amine Dehbaoui is now with Serma Technologies}

IEEE HOST 2014 – MAY 6-7, ARLINGTON, USA
Concern: **Security of embedded programs against fault attacks**

- Many **software** countermeasures
- Defined by respect to a **fault model**
- Often based on **redundancy principles**
- Some recent schemes propose to **add this redundancy at assembly level**

---

Can we **evaluate the practical effectiveness** of some assembly-level countermeasures against fault attacks?

---

1. Provide an experimental evaluation **on single isolated instructions**
2. Provide an experimental evaluation **on complex codes**
I. Experimental setup

II. Preliminaries about the fault model

III. Evaluation on simple codes

IV. Evaluation on a FreeRTOS implementation

V. Conclusion
Pulsed electromagnetic fault injection

- **Transient and local** effect of the fault injection
- **Standard circuits not protected** against this technique
- **Solenoid** used as an injection antenna
- Up to **210V** sent on the injection antenna, pulses width longer than **10ns**

**Microcontroller based on an ARM Cortex-M3**

- 130nm CMOS technology, ARMv7-M architecture
- Frequency 56 MHz, clock period 17.8 ns
- **16/32 bits Thumb-2** RISC instruction set
- Keil ULINKpro JTAG probe to **debug the microcontroller**
- **3-stage pipeline** (Fetch – Decode – Execute), no prefetch

**The Definitive Guide to the ARM Cortex-M3** – Joseph Yiu, Newnes, 2009
EXPERIMENTAL SETUP

- The experiment is **driven by the computer**
- The target code is **runned on the microcontroller**
- The pulse generator **sends a voltage pulse**
- The microcontroller is stopped
- The microcontroller’s internal data is harvested

**Main experimental parameters**

- **Position** of the injection antenna (fixed for this work)
- **Electric parameters** of the pulse (fixed for this work)
- **Injection time** of the pulse
- **Executed code** on the microcontroller

**Hardware exceptions**

**UsageFault** exceptions for illegal instructions are triggered

⇒ Used to **identify the impacted instruction** for a given injection time
I. Experimental setup

II. Preliminaries about the fault model

III. Evaluation on simple assembly codes

IV. Evaluation on a FreeRTOS implementation

V. Conclusion
**Fault Injection on a Single 16-Bit LDR Instruction**

The instruction `ldr r0, [pc,#40]` loads a 32-bit word into a register from the Flash memory.

- **Instruction Fetch**
- **Instruction Decode** (data fetch)
- **Hamming Weight in r0**

The chart shows the injection time (ns), by steps of 200ps, with the pulse voltage (V) indicated by color.
**Fault Injection on a Single 16-bit LDR Instruction**

\[ \text{ldr } r0, [pc, #40] \rightarrow \text{loads a 32-bit word into a register from the Flash memory} \]

---

**Consequences regarding the instruction flow** (instruction fetch)
- Instructions *replacements*
- Instruction *skips under certain conditions* (~ 20-30% of time)

**Consequences regarding the data flow** (instruction decode)
- Corruption of the *ldr* instructions from the Flash memory

---

**Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller**

I. Experimental setup

II. Preliminaries about the fault model

III. Evaluation on simple assembly codes

IV. Evaluation on a FreeRTOS implementation

V. Conclusion
Two fault injection attempts, every 200 ps
During a time interval defined by hardware instructions

150 ns
1500 fault injection attempts
180 faulty outputs

300 ns
3000 fault injection attempts
210 faulty outputs / 50 faulty outputs

Relevant metric to evaluate the countermeasures?
Replacement sequences add some instructions \( \Rightarrow \) longer execution time
\( \Rightarrow \) more fault injections to do \( \Rightarrow \) different number of results to compare

From a security point of view, \textit{effectiveness} = \textit{reduction of faulty outputs}
- Fault tolerance against one **instruction skip**
- Formally verified using model-checking tools
- A **replacement sequence** for every instruction
- No protection for the data flow
- Experiment performed on the `bl` instruction
- In the tested code, the subroutine modifies `r0`

---

**Formal verification of a software countermeasure against instruction skip fault attacks**
Fewer faults by forcing the 32-bit encoding of instructions (orange curve)
The countermeasure is not effective with 16-bit instructions (blue curve)
The combination 32-bit inst + countermeasure is very effective (green curve)
- Detects **any single fault** (instruction skips, replacements, data flow)
- Proposed **for a restricted set of instructions** (ALU, load-store)
- Tested for a `ldr` instruction from the Flash memory

```assembly
ldr  r0, [pc, #34]
ldr  r1, [pc, #38]
cmp  r0, r1
bne  <error>
```

**Countermeasures against fault attacks on software implemented AES**
- Faults for 16-bit and 32-bit encodings, some due to the corruption of the data transfer

- The FD countermeasure is **not effective with a 16-bit encoding** (blue curve)

- However, **countermeasure + 32-bit encoding ➔ very effective** (green curve)
I. Experimental setup

II. Preliminaries about the fault model

III. Evaluation on simple assembly codes

IV. Evaluation on a FreeRTOS implementation

V. Conclusion
Portable RTOS written in C, **multitasking** operating system

**Fault tolerance countermeasure**

- At the OS initialization
- The systems starts in privileged mode
- Then it switches to unprivileged mode

---

<table>
<thead>
<tr>
<th>MSR Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msr control, r3</td>
<td>Changes priv. mode</td>
</tr>
<tr>
<td>msr psp, r0</td>
<td></td>
</tr>
<tr>
<td>mov r0, #0</td>
<td></td>
</tr>
<tr>
<td>add lr, r1, #1</td>
<td></td>
</tr>
<tr>
<td>msr basepri, r0</td>
<td></td>
</tr>
<tr>
<td>ldr lr, =0xffffffff</td>
<td></td>
</tr>
</tbody>
</table>

**prvRestoreContextOfFirstTask** function

An attacker may try to **stay in privileged mode**

To evaluate the effectiveness, we observe the **number of faults in the control register**
Not very good effectiveness for the fault tolerance countermeasure on this code

The protected `msr` instruction is maybe too specific or the fault model too simplistic

Further experiments are required to deeply analyze the effectiveness of this CM
Fault detection countermeasure

• During task creation
• Each task has its own priority level
• The priority level is loaded from the Flash

Arguments for the function

An attacker may try to change a priority level

To evaluate the effectiveness, we observe the number of faults in this priority level

(in the xTaskGenericCreate function)
The countermeasure when only applied to ldr instructions still misses some faults

The countermeasure is very effective on this code when applied to every instruction

However, not all the instructions can be protected with this countermeasure

This countermeasure must be combined with other techniques against faults
I. Experimental setup

II. Preliminaries about the fault model

III. Evaluation on simple assembly codes

IV. Evaluation on a FreeRTOS implementation

V. Conclusion
The effectiveness of both CM can be nullified if not well implemented
On this platform, we need to check that the 32-bit encoding of instructions is used

The fault tolerance CM can significantly reinforce an isolated instruction

However, it was not very effective on the FreeRTOS tested code
The instruction skip fault model may be too simplistic

The fault detection CM was very effective on all the tested codes
But its applicability is limited since it cannot be applied to several instructions

Perspectives

• Further experiments are required for the fault tolerance countermeasure
• Can we combine those countermeasures to secure an assembly code?
• What about side-channel leakages on cryptographic implementations?
Any questions?

Nicolas MORO
PhD student, CEA

Graduation expected in Sep. 2014

- www.nicolasmoro.net
- nicolas.moro [at] gmail.com
- +33.(0)4.42.61.67.13