Experimental evaluation of software countermeasures

PROSECCO Workshop

6 novembre 2018
Goal of the presentation

**Goal:** Present the first results of the security evaluation we perform at the Secure Architectures and Systems laboratory (joint team CEA Tech, Mines Saint-Etienne).

This evaluation helps to design **efficient countermeasures** by providing a **feedback to the designer**.

Evaluation carried out for different:

- **Physical threats:**
  - Side-channel analysis
  - Fault-attacks

- **Hardware targets:**
  - 8-bit microcontrollers
  - 32-bit microcontroller ARM Cortex M/A

- **Practical use-cases:**
  - VerifyPIN
  - AES encryption
Evaluation method

Two main axes:

- **Leakage** assessment using statistical tools
  - Attack-independent

- **Attack**-based methodology:

<table>
<thead>
<tr>
<th>Complexity / Cost</th>
<th>Side-channel attacks</th>
<th>Fault attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ / $</td>
<td>Correlation power analysis</td>
<td>Clock glitches</td>
</tr>
<tr>
<td></td>
<td>Template attacks</td>
<td></td>
</tr>
<tr>
<td>+++ / $$$</td>
<td>Machine learning (deep neural networks)</td>
<td>Laser</td>
</tr>
</tbody>
</table>
1. Side-channel leakage assessment

2. Fault attacks
   - VerifyPIN
   - AES-128 encryption

3. Combination of protections

4. Conclusion
Side-channel leakage assessment
Leakage assessment

**Aim**: conduct a statistical study to evaluate the leakages.

**Statistical tests**: reject or not a *null hypothesis* (i.e. the means of the target populations are equal)

Two common tools in SCA context:

- **t-test** [1]: split the traces in two sets w.r.t an intermediate value, see if they differ statistically.
  - The *t* statistic follows a Student law. For sufficient number of traces, $|t| > 4.5$ give a confidence of $99.999\%$ to reject the NH.
  - In our experiments: target at **bit level**.

- **F-test** [2], **SNR**: generalization of *t*-test for multiple sets. Takes the variance into consideration.
  - Ratio of inter-class VS intra-class variance.
  - In our experiments: target at **byte level**.

---


Comparison of unmasked and masked S-boxes

Splitting according to the value of the 8 bits at the 1\textsuperscript{st} S-box output. 20000 traces of 128-bit AES encryption.

\begin{itemize}
  \item \textbf{unmasked}
  \includegraphics[width=\textwidth]{unmasked_graph.png}
  \begin{itemize}
    \item No more 1\textsuperscript{st} order leakage with this masking scheme.
  \end{itemize}
  \item \textbf{masked}
  \includegraphics[width=\textwidth]{masked_graph.png}
\end{itemize}
Identification of new leakage points

The **masks generation process** leaks information as well (F-test).

**Generation of the 6 random masks (4 for MixColumn, 2 for SubBytes):**

In the **worst case scenario** (profiled attacks), these can be **combined** with other leakage points later to perform a **second order attack**.

\[(M \oplus \text{SBOX}(P \oplus K) \oplus M) \Rightarrow \text{SBOX}(P \oplus K)\]
Identification of new leakage points

Interestingly, we can see the masks manipulation during the encryption process. The initial (masked) key schedule can also leak information or be profiled for efficient differential fault attack (DFA):
Identification of new leakage points

Interestingly, we can see the masks manipulation during the encryption process. The initial (masked) key schedule can also leak information or be profiled for efficient differential fault attack (DFA):
Identification of new leakage points

Interestingly, we can see the masks manipulation during the encryption process. The initial (masked) key schedule can also leak information or be profiled for efficient differential fault attack (DFA):
F-test on desynchronised traces

A second order CPA can target – jointly – the **two shares**. Desynchronization-based protections can **reduce this exploitability**.

Leakage evaluation when **simulating desynchronisation** by randomly inserting $n$ blocks of $w$ NOPs during the execution:

- Leakage shrinks and becomes **unexploitable** (20000 traces here).
- Provide hints for protecting the design.
Ongoing works

On protected AES (masking, hiding), powerful template attacks need:

- Strong information compression (PCA, LDA) or
- Detection of points of interest
- Resynchronization techniques

can become rapidly difficult in practice.

Machine Learning-based analysis can be helpful here [3] [4]

- Deep learning-based attacks against masking
- Denoising and resynchronization with autoencoder
- ...

---

[3] Liran Lerman, Romain Poussier, Gianluca Bontempi, Olivier Markowitch, François-Xavier Standaert

[4] Emmanuel Prouff, Remi Strullu, Ryad Benadjila, Eleonora Cagli, Cécile Dumas
Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database. IACR ePrint 2018
Fault attacks
Low cost: Clock glitches on a VerifyPIN

Different **hardened** VerifyPIN have been successfully bypassed:

- ✔ Constant-time
- ✔ Constant-time and inlined functions
- ✔ Constant-time and inlined functions and loop counter
- ✗ Constant-time and inlined functions and double call

**Limitations**

The ChipWhisperer platform **cannot** glitch at two different times.

**Plan to overcome**

We shall shoot with the **laser**!
Laser faults

Preparatory work

- Design a **custom** ChipWhisperer target board:
  - Front-side access
  - Back-side access
- Prepare the target: **decapsulate** the chip to access the die
- **Mechanical setup** of the target on the bench

... Mapping out the faults:

- x-y position,
- power,
- duration,
- delay,
- type of fault (skip, set, reset, flip, ...)
Laser setup

**Characteristics**

- IR (1064nm)
- >30ps
- 0-3W
- 3 objective lenses:
  - x5 (20μm)
  - x20 (5μm)
  - x100 (1μm)
Characteristics

- IR (1064nm)
- >30ps
- 0-3W
- 3 objective lenses:
  - x5 (20μm)
  - x20 (5μm)
  - x100 (1μm)
Laser setup

Characteristics

- IR (1064nm)
- >30ps
- 0-3W
- 3 objective lenses:
  - x5 (20µm)
  - x20 (5µm)
  - x100 (1µm)
8-bit microcontroller results

Instruction skip fault model previously validated experimentally [5]

8-bit microcontroller results

This time, all implementations are vulnerable.

- ✔ Constant-time
- ✔ Constant-time and inlined functions
- ✔ Constant-time and inlined functions and loop counter
- ✔ Constant-time and inlined functions and double call
- ✔ Constant-time and inlined functions and control-flow integrity

Paradox

**Constant-time** implementation makes laser attacks much easier
32-bit microcontroller

A more complex target (32 bits) implies:

- Larger area to cover for cartography (2.5x2.5cm),
- Complex micro-architecture,
- More time variability
32-bit microcontroller

A more complex target (32 bits) implies:

- Larger area to cover for cartography (2.5x2.5cm),
- Complex micro-architecture,
- More time variability
32-bit microcontroller

A more complex target (32 bits) implies:
- **Larger area** to cover for cartography (2.5x2.5cm),
- Complex micro-architecture,
- More time variability
Instruction corruption in Flash memory

A laser shot in flash memory alters the fetched data on-the-fly.

Fault model

**Bit-set** on data (and instructions) fetched from flash memory
Examples of instruction corruption

Modifying a MOVW instruction (32 bits).

<table>
<thead>
<tr>
<th>bits</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
</table>

Reference instructions

<table>
<thead>
<tr>
<th>Generic MOVW</th>
<th>1 1 1 1 0 i 1 0 0 1 0 0</th>
<th>imm4 0</th>
<th>imm3</th>
<th>Rd</th>
<th>imm8</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVW, R0, 0</td>
<td>1 1 1 1 0 i 1 0 0 1 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data corruption

| MOVW, R0, 4  | 1 1 1 1 0 i 1 0 0 1 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 |

Destination register corruption

| MOVW, R1, 0  | 1 1 1 1 0 i 1 0 0 1 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 |

Opcode corruption

| MOVT, R0, 0  | 1 1 1 1 0 i 1 0 0 1 1 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |


Laser fault injection on VerifyPIN

Constant-time implementation with hardened booleans.

1. trials = 3
2. ref_PIN[4] = {1, 2, 3, 4}
3. procedure VerifyPIN(user_PIN[4])
4.   authenticated = FALSE
5.   diff = FALSE
6.   if trials > 0 then
7.     for i ← 0 to 3 do
8.       if user_PIN[i] != ref_PIN[i] then
9.         diff = TRUE
10.       if diff == TRUE then
11.         trials = trials - 1
12.     else
13.       authenticated = TRUE
14.   return authenticated

C code:
if (trials > 0)

Assembly code:
CMP R3, 0
BLE address
Fault injection on the CMP instruction

Performing a bit-set at index 10.
Instead of comparing R3, we compare R7.
By design, R7 stores the frame pointer, always positive.

<table>
<thead>
<tr>
<th>bits</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic CMP</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>Rd</td>
<td></td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP R3, 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Register corruption</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP R7, 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Outcome**

The *trials* value is never compared ➔ unlimited number of trials.

The PIN value can be **brute-forced**.
Laser fault injection on AES-128

C code:
```c
for (int i=0; i<4; i++){
    for (int j=0; j<4; j++){
        ...
    }
}
```

Assembly code:
```assembly
MOV  R0, 0
addr_i:
    MOV  R1, 0
    addr_j:
        ...
    ADD  R1, 1
    CMP  R1, 3
    BLE  addr_j
    ADD  R0, 1
    CMP  R0, 3
    BLE  addr_i
```
Faulting the loop variable increment

Add 5 instead of 1 after executing the body of the loop.

<table>
<thead>
<tr>
<th>bits</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic ADD</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rd</td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R0, 1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data corruption</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R0, 5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Outcome

The for loop exits after **one execution only**.
Faulting the for loops

<table>
<thead>
<tr>
<th></th>
<th>$C_{0,0}$</th>
<th>$C_{1,0}$</th>
<th>$C_{2,0}$</th>
<th>$C_{3,0}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{0,1}$ ⊕ $K_{0,1}^{10}$</td>
<td>$C_{1,1}$</td>
<td>$C_{2,1}$</td>
<td>$C_{3,1}$</td>
<td></td>
</tr>
<tr>
<td>$C_{0,2}$ ⊕ $K_{0,2}^{10}$</td>
<td>$C_{1,2}$</td>
<td>$C_{2,2}$</td>
<td>$C_{3,2}$</td>
<td></td>
</tr>
<tr>
<td>$C_{0,3}$ ⊕ $K_{0,3}^{10}$</td>
<td>$C_{1,3}$</td>
<td>$C_{2,3}$</td>
<td>$C_{3,3}$</td>
<td></td>
</tr>
</tbody>
</table>

Faulting the **inner** for loop on its first execution

Faulting the **outer** for loop

<table>
<thead>
<tr>
<th></th>
<th>$C_{0,0}$ ⊕ $K_{1,0}^{10}$</th>
<th>$C_{1,0}$ ⊕ $K_{1,1}^{10}$</th>
<th>$C_{2,0}$ ⊕ $K_{2,0}^{10}$</th>
<th>$C_{3,0}$ ⊕ $K_{3,0}^{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{0,1}$</td>
<td>$C_{1,1}$ ⊕ $K_{1,1}^{10}$</td>
<td>$C_{2,1}$ ⊕ $K_{2,1}^{10}$</td>
<td>$C_{3,1}$ ⊕ $K_{3,1}^{10}$</td>
<td></td>
</tr>
<tr>
<td>$C_{0,2}$</td>
<td>$C_{1,2}$ ⊕ $K_{1,2}^{10}$</td>
<td>$C_{2,2}$ ⊕ $K_{2,2}^{10}$</td>
<td>$C_{3,2}$ ⊕ $K_{3,2}^{10}$</td>
<td></td>
</tr>
<tr>
<td>$C_{0,3}$</td>
<td>$C_{1,3}$ ⊕ $K_{1,3}^{10}$</td>
<td>$C_{2,3}$ ⊕ $K_{2,3}^{10}$</td>
<td>$C_{3,3}$ ⊕ $K_{3,3}^{10}$</td>
<td></td>
</tr>
</tbody>
</table>

What is left?

Holding one correct and two faulty ciphertexts, the attacker only needs to brute-force the tenth round-key byte $K_{0,0}^{10} \Rightarrow 2^7$. 
## Conclusion on laser faults in flash memory

### Capabilities
- Temporarily alter instruction/data from flash memory,
- Corrupt the **control flow** of a program,
- Weaken security of embedded programs.

### Limitations
- Bit-set only (so far),
- Adjacent bits only,
- Control-flow corruption mostly.

### Future possibilities
- Multispot laser
Combination of protections
Principle of 2nd order CPA: attack two S-box output bytes. Traditionally, target the two shares (mask + masked value) but two consecutive bytes work well:

- $|\text{Leak}(\text{Sbox}(P_i \oplus K_i) \oplus M') - \text{Leak}(\text{Sbox}(P_j \oplus K_j) \oplus M')|$ |
- $\text{HW}(\text{Sbox}(P_i \oplus K_i) \oplus M' \oplus \text{Sbox}(P_j \oplus K_j) \oplus M')$

$= \text{HW}(\text{Sbox}(P_i \oplus K_i) \oplus \text{Sbox}(P_j \oplus K_j)) \Rightarrow$ no more mask!
Combining leakages is easy when traces are perfectly synchronised.

800 traces required to break 1st-roder masked AES on STM32.

A desynchronising countermeasure is very powerful here!
For the worst

Countermeasure against FA or SCA are usually compatible.

Countermeasure against FA and SCA can be incompatible.

Example

**Redundancy**-based protection against Fault Injection Analysis can **enhance** side-channel leakages...

Side-Channel Analysis is not only for key recovering purpose, it also helps in temporaly **profiling** fault injection (bypassing secure boot [6])

**Each case** must be evaluated **separately**.

---

Conclusion

- Inserting protections at **software level** is powerful
- Leakage assessment is a great tool to design protections
  - Provides **metrics** of leakage reduction efficiency
- New attack on **flash memory** of a 32-bit microcontroller
- **Combinations** of protections is a **double-edged** sword
Conclusion

- Inserting protections at **software level** is powerful
- Leakage assessment is a great tool to design protections
  - Provides **metrics** of leakage reduction efficiency
- New attack on **flash memory** of a 32-bit microcontroller
- **Combinations** of protections is a **double-edged** sword

--- Questions ? ---
Contacts: Brice Colombier
brice.colombier@cea.fr

Pierre-Alain Moëllic
pierre-alain.moellic@cea.fr