INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Referent: *Vincent Migliore*
vincent.migliore@insa-toulouse.fr

Hardware Security Lab - Practical Hardening of a Software Implementation against Power-Based Side-Channel Attacks -

Objectives

This tutorial aims at discussing the practical implementation of mitigations against side-channel attacks based on power or electromagnetic field observation. In particular, we will try to secure a simple algorithm called the One Time Pad, a straightforward encryption algorithm, in c and apply security assessment tools. In a second part, the impact of the c compiler when applying mitigations will also be discussed.

Introduction

The increasing adoption of cyber-physical and connected objects promises to address a large variety of societal challenges, from simplifying human communications (with legal authorities, public and private communications) to supporting next generation industry (so called Industry 4.0), smart cities and transportations. Since these computing objects become more and more integrated into our daily lives, their physical access is also easier for malicious users. However, such systems, in particular cost-driven ones such as Internet-of-Things (IoT) nodes, are vulnerable to a wide range of cyber-attacks. In addition to usual networked systems' threats, computations are vulnerable to a large range of physical attacks, which exploit some characteristics of the target system including both software and hardware. Among these, side-channel attacks (SCA) can infer secret information from physical observations measured during the execution of sensitive computations. SCA typically exploit physical quantities such as electromagnetic signature and power consumption, behaviour of cache memories, and effects due to speculative execution. SCA have gained momentum with the increased use of cryptography because they represent, with fault injection attacks, the most effective way to break cryptographic implementations. For example, the AES encryption, which is widely used in most computing systems nowadays, is considered secure by traditional cryptanalysis but is highly vulnerable to side-channel attacks. SCA based on power and electromagnetic observations are particularly harmful because they have been successfully applied many different kinds of secured computations; furthermore, they are non-intrusive, hence difficult to detect.

Target and leakage modelling

Source of leakage is in general twofold: One contribution is due to the inversion of the internal state of flip-flops; and another due to different propagation delays into combinatorial logic that produces a variation on logic gates output, partially leaking information about inputs.

To assess the security of a target or provide software/hardware protections, a sufficiently simple model of the target and the leakage is required to produce efficient and automatable techniques. The abstraction model of the target is in general based on Instruction Set Architecture (ISA), which basically does not take into account processors' micro-architecture (i.e. only consider General Purpose Registers, ALU and memory, ...).

For the leakage modelling, a popular approach is the **Hamming Distance Model**, which considers that the power consumption P is linear with the number registers' bits that have changed state (i.e. 0→1 and 1→0) at a given time. P is estimated by applying the Hamming Distance between the current value of the register and the futur one after the clock tick. Since this model requires a relatively important knowledge on the target architecture, a simpler model is often used instead (the **Hamming Weight Model**). In the **Hamming Weight Model**, the power consumption is only evaluated with the number of 1 of variables (which is equivalent to compute the Hamming weight on variables).

Important note: the leakage quantification accuracy depends on the finess of the processor's architecture and micro-architecture description. Applied to a too abstract processor's architecture model, the accuracy of the leakage quantification can miss important features of the underlying hardware and reduce the security level when applying mitigations. A contrario, a too fine-grained model would be too complex for security assessment or hardening methods and tools.

Security assessment using leakage simulators

Recently, leakage simulators have gain momentum as a fast way to model the leakage by simulating code execution on a simplified version of the target. MAPS, the simulator that will be used in this tutorial, simulates the ARM Cortex-M3 processor's architecture and quantify the leakage when an internal register or buffer is modified by applying the Hamming Distance Model.

In practice, the simulator works as follows: First, the source code (typically written in c) is compiled using the ARM gcc compiler. The resulting file is called the firmware. Second, the firmware is simulated using the internal ARM cortex-M3 simulator of MAPS. During the simulation, when an internal register or buffer is modified, the hamming distance between the current and the next value of the register/buffer is calculated then stored into a vector. The estimated power consumption is called a **sample**, and the vector of all samples a **trace**. Third, security evaluation is performed using the **test vector leakage assement (TVLA)** method.

Note that TVLA is not an attack, it is used to evaluate potential leaks of an implementation. In TVLA, the designer will capture two subsets of traces, one subset with fixed input data and one with varying input data. Then the security assessment is performed by evaluating if a correlation exists between those two subsets. If there is a correlation, then the power consumption is dependent of the input data.

In practice, the correlation is evaluated iteratively starting from the first sample of each trace, then computed for the second sample of each trace, then computed for the third sample of each trace, and so on. This methodology has the additional benefit to help pointing out more precisely where there is a potential leak.

For this experimentation, the correlation evaluation we will be based on the **Welch's *t*-test**:

$$t = \frac{m_0 - m_1}{\sqrt{\frac{s_0}{N} + \frac{s_1}{N}}}$$

Where m_0 (resp. m_1) is the mean and s_0 (resp. s_1) the standard deviation of the fixed data subset (resp. the varying data subset) applied at the same sample of each trace; and N the number of traces. If the *t*-test is between -4.5 and 4.5 for all samples, then the design has sucessfully passed the security assessment.

Note: The proposed algorithm that will be implemented and evaluated (i.e the OTP) has two inputs (a message and a key). For the TVLA, we need to compare fixed inputs to varying inputs. To simplify the study, the TVLA will only be applied on the first input, and the second input will always be random.

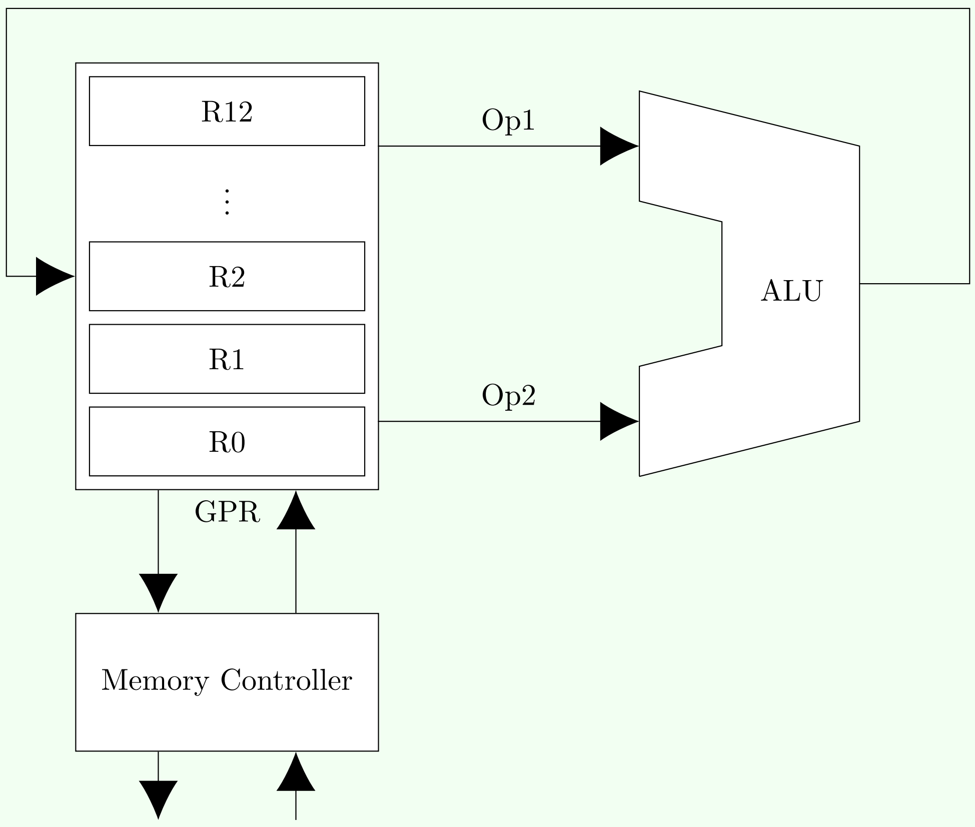
Preliminary questions

Question 1

We consider a 32 bits ARM Cortex-M3 micro-controller. 176 is stored in R1 and 81 in R2. The processor executes the instruction ADD R1,R2 (i.e. R1 = R1 + R2). Quantify the power consumption using the Hamming Distance Model.

Question 2

Below the simplified architecture of the ARM Cortex-M3 micro-controller:



From your point of view, which part of the architecture can potentially leak information? Which kind of leak is involved?

Setting-up the experimentation

First of all, open a terminal, move to the root of the tutorial folder and run the following command:

```
tar jxvf arm-gcc.tar.bz2
```

The tutorial folder is split into 3 sub-folders:

→ gcc-arm-none-eabi-X-XXXX-XX-XXXXX: ARM-GCC toolchain to compile a firmware from source code for ARM platforms. We will use this to compile the firmware that will be run by the MAPS simulator;

→ firmware: Folder used to manage firmwares, with one subfolder per firmware. The c code of the firmware is provided in *firmware.c*.

→ bin: Folder containing a command interpreter called *maps* that embeds MAPS simulator; and *maps* documentation.

Now you are invited to read the documentation of *maps* for a quick starting guide and a description of available commands.

Part 1 - Security Assessment on an unprotected implementation

This first part is dedicated to the security assessment of an unprotected code using MAPS.

Move to the folder */firmware/part-1/* and open *firmware.c*. For now, the file is empty. We would like to implement a very simple encryption algorithm called the One Time Pad (OTP). For a message m, a key k and a ciphertext c:

$$c = m \oplus k$$

Where $\oplus$ is the eXclusive-OR (XOR). In the code, the key k is stored at address 0x30000200, the message m at address 0x30000300, and the ciphertext c at address 0x30000400. m, k and c are 32 bits integers.

Question 3

Using only 1 line of code, implement the OTP.

Verification

To verify if the stream-cipher is correctly implemented, from *maps*:

→ Run make and load commands to compile and load the part-1 firmware into MAPS;

→ Run test_stream_cipher command.

Question 4

Run t_test command. Does your implementation successfully pass the t-test security assessment?

Question 5

By carefully analysing the desassembly code using dump command, fill the table below by mentioning the value of the registers at a given instruction:

Address	Instruction	R1	R2	R3

Why the code is leaky?

Part 2 - Hardening the code with masking

Masking is a well known contremeasure to prevent side-channel leakage. Masking aims at making the leakage independent from a sensitive information s. To do so, a random and unpredictible "mask" r is applied to s using the XOR operator. The procedure provides two values called shares:

$$\begin{aligned} s_0 &= s \oplus r \\ s_1 &= r \end{aligned}$$

Question 6

From s_0 and s_1, how I can recover secret s ?

Question 7

For the OTP, describe elementary operations to compute c from m and k with masking. For m (resp. k), you will consider shares (m_0,m_1) (resp. (k_0,k_1)).

We will now implement and test a masked version of the OTP. Move to the folder */firmware/part-2/* and open *firmware.c*. To simplify the implementation, the masked shares (m_0,m_1) and (k_0,k_1) are already computed inside MAPS and stored at the following addresses:

→ k_0 at address 0x30000200;

→ k_1 at address 0x30000204;

→ m_0 at address 0x30000300;

→ m_1 at address 0x30000304;

→ c_0 at address 0x30000400;

→ c_1 at address 0x30000404;

Question 8

Using only 2 lines of code, implement the masked OTP. Note that the expected output is a masked version of c (i.e. c_0 and c_1).

Verification

Within *maps*:

→ Run make and load commands to compile and load the part-2 firmware into MAPS;

→ Run masking yes to enable masking;

→ Run t_test.

Question 9

Does your implementation successfully pass the t-test security assessment?

Question 10

From the desassembly code of part-1 and part-2 firmwares provided by the dump command, compare the overhead of masking by counting the number of instruction.

Part 3 - Impact of the compiler on masking

In this section, we will evaluate a functionnally equivalent implementation of masked code of the OTP, but which is not compiled in the same way by the compiler.

Move to the folder */firmware/part-3/* and open *firmware.c*.

Question 11

Describe the difference with your code. Verify if the code is functionnally equivalent.

Verification

Within *maps*:

→ Run make and load commands to compile and load the part-3 firmware into MAPS;

→ Run masking yes to enable masking;

→ Run t_test.

Question 12

Does your implementation successfully pass the t-test security assessment?

Question 13

From the desassembly code of part-3 firmware provided by the dump command, determine why the design is leaky.

Part 4 - Impact of the micro-architecture

Even after hardening against side-channel leakage, a reduction of the expected security level has been recently exposed by Barenghi et al. [1]: hardened systems are vulnerable to side-channel attacks due to leakage at the micro-architecture layer. Authors of MAPS [2] investigated the RTL of a Cortex-M3 Arm processor, and demonstrated that internal pipeline registers, not visible at the Instruction Set Architecture level, can lead to a potential leak. They integrated some of them into the leakage simulator. Figure 1 provides partial representation of Arm Cortex-M3 architecture, including these internal registers (called R_a and R_b). In particular, these registers are modified during:

- ALU operations (R_a for left operand, R_b for right operand);
- a register value is stored in memory (R_b).

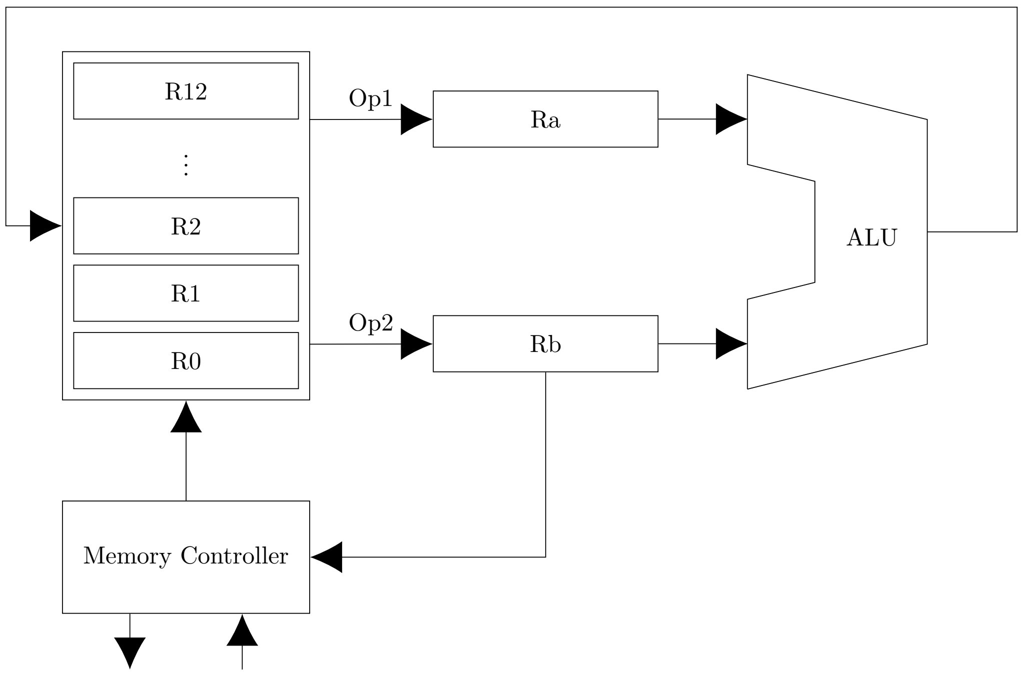


Figure 1: Partial representation of the Arm Cortex-M3 micro-architecture

Now we will demonstrate that if these registers are not taken into account, the security assessment can provide misleading information about the actual security level of the target. Move to the folder */firmware/part-4/* and open *firmware.c*.

Verification

Within *maps*:

→ Run make and load commands to compile and load the part-4 firmware into MAPS;

→ Run masking yes to enable masking;

→ Run t_test in the two following configutations:

- When pipeline registers are not taken into account (pipeline_leakage no);
- When pipeline registers are tracked (pipeline_leakage yes).

Question 14

Which configuration pass the t-test security assessment? What is the consequence on the overall security evaluation?

Question 15

From the desassembly code of part-4 firmware provided by the dump command, determine why the design is leaky.

Question 16

Propose a modification of the code to avoid a leak.

References

[1] Alessandro Barenghi and Gerardo Pelosi. "Side-channel Security of Superscalar CPUs: Evaluating the Impact of Micro-architectural Features". In: *DAC*. 2018. DOI: `10.1145/3195970.3196112`.

[2] Yann Le Corre, Johann Großschädl, and Daniel Dinu. "Micro-architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors". In: *COSADE*. 2018.