UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Physical Security Analysis of AES Implementations on 8-bit MCU and Countermeasures

Dissertation presented by
Antoine POUSSART , Jean-Sébastien STAELENS

for obtaining the Master's degree in
Electrical Engineering

Supervisor
François-Xavier STANDAERT

Readers
Laurent FRANCIS, Pierre GÉRARD

Academic year 2016-2017

Abstract

Embedded security is a field studying how the physical implementations of cryptosystems can weaken the integrity of their hidden secrets. Power analysis attacks are one of the most spread side-channel attacks and take advantage of a device's power consumption to retrieve sensitive information about it. One typical countermeasure to thwart those attacks is masking, which consists in splitting the sensitive data into multiple independent shares.

The objective of this master thesis is to propose a physical security analysis of different AES implementations on an 8-bit Atmel micro-controller. This work starts by describing the ChipWhisperer-Lite, the platform that was used to record all the power traces. It continues by studying an unprotected version of the AES, and shows that its execution leaks information that can be easily exploited to recover the key with some basic attacks. It then presents the implementation of several state-of-the-art software masking schemes and compares them in terms of execution time and required randomness. Finally, it analyses their practical security level, based on two different leakage detection tests and univariate attacks.

Acknowledgements

Firstly, we would like to thank Pr. Ir. François-Xavier Standaert for accepting to be our thesis supervisor, giving us many choices for the orientation of this thesis, letting us free to explore the topics that interested us the most and last but not least for his precious help.

The second great thank is for Pierre Gérard, partly for accepting being a reader of this thesis and mostly for his enormous work in removing the Python interface of the ChipWhisperer-Lite and writing the .dll library necessary to use it directly from MATLAB. Almost nothing in this thesis could have been done without his precious help. :-)

We also want to thank Anthony Journault for his fast availability and help on the masking and the leakage detection tests comprehension and Romain Poussier for his help in implementing the profiled ρ-test. We would also like to thank Laurent Francis for accepting to be a reader of this thesis.

Antoine Poussart
Jean-Sébastien Staelens

I personally would like to thank my parents Chantal and Rémy for giving me the opportunity to study at the university and supporting me during my five years of study without ever doubting in me. I would like to thank all my friends for all the good moments spent together and the ones coming. I would also like to thank Valérie for her love and support.

And finally I would like to thank Jean-Sébastien for his work and help during this thesis even if at the beginning, we had not planned to work together. :p

Antoine Poussart

I would mainly like to thank my parents, for their love and support during these five years studying here at UCL. They helped me to become who I am today and I hope that I can continue to make them proud. I would also like to thank Alicia and my room-mates, for their understanding of the particular schedule that I used to follow when writing this thesis.

I would finally like to thank Antoine, who put up with me during this full year, and who I will still be working with next year !

Jean-Sébastien Staelens

Contents

List of abbreviations

ADC	Analog-to-Digital Converter
AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
CMOS	Complementary Metal Oxide Semiconductor
CPA	Correlation Power Analysis
CPU	Central Processing Unit
CW-L	ChipWhisperer-Lite
DPA	Differential Power Analysis
DRAM	Dynamic Random Access Memory
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
ISW	Ishai-Sahai-Wagner
LNA	Low Noise Amplifier
LUT	Look-Up Table
MCU	Micro-Controller Unit
MSB	Most Significant Bit
PAA	Power Analysis Attacks
POI	Point of Interest
PRNG	Pseudo-Random Numbers Generator
RAM	Random Access Memory
RP	Rivain-Prouff
SCA	Side-Channel Attack
SNR	Signal-to-Noise Ratio
SoC	System-on-Chip

List of notations

$x \xleftarrow{\$} \mathcal{X}$	x is randomly drawn from the set $\mathcal{X}$
$\|x\|$	The size in bits of x or the absolute value of x (depending on the context)
$\|\|\mathcal{X}\|\|$	The cardinality of the set $\mathcal{X}$
$x = x_0 \| \| x_1$	x is composed of x_0 and x_1 that are appened together
$\{0,1\}^n$	The set of all the n-bit sequences
0^n	The sequence containing n zeros
$GF(2^8)$	The AES field: $GF(2^8) := GF(2)[x]/(x^8 + x^4 + x^3 + x + 1)$
$GF(2^8)^\times$	The set of invertible elements in $GF(2^8)$: $GF(2^8)^\times = GF(2^8)\backslash\{0\}$
$\mathbb{F}_{2^8}$, $\mathbb{F}_{2^8}^\times$	Other notations for the two previously defined fields
$x = ab$	x is the bitwise `and` operation between a and b
$x = a\|b$	x is the bitwise `or` operation between a and b
$x = a \oplus b$	x is the bitwise `xor` operation between a and b
$x = a \odot b$	x is the field multiplication in $\mathbb{F}_{2^8}$ of a and b
$x = \overline{a}$	x is the bitwise `not` operation of a
$\mathbb{P}[X = x]$	The probability that an iteration of X equals the value x
$X \sim \mathcal{N}(\mu, \sigma)$	The random variable X follows a normal distribution of mean μ and standard deviation σ
$H(x)$	Hamming weight of the word x
$\mathbf{A}^\intercal$	Transposition of the matrix $\mathbf{A}$
$x = \lceil a \rceil$	x is the nearest integer equal or greater than a

Chapter 1

Introduction

In today's world, many cryptographic algorithms are used at every moment to secure the data that we transmit. It could be transiting on your phone, your computer, your television or any other type of communicating device. Those cryptosystems can be implemented either in software or in hardware. In the case of hardware, there are specific chips, ASIC, SoC, that can be used to encrypt and decrypt data very fast but with low flexibility. On the other hand, working with software allows flexibility but is far slower. *Embedded security* is a part of the security halfway between the electrical engineer's competences, the computer scientist's ones and the mathematician's ones. It is the science that will study what an adversary can do to recover the *secret* hidden in a system by having direct access to the said system. It also studies how to secure the device against such an adversary. This secret is called the *key* in the cryptographic domain. We say that a system is broken if any adversary can recover at least partially the key. In embedded security, we are interested in physical adversaries, the ones that are physically present near the chip and that can have interactions such as heating it, measure its power consumption, touch it, tamper it, insert faults, etc. The question that arises is: "What can an untrusted party (i.e. an adversary $\mathcal{A}$) with dishonest intentions do to recover the secret key and then be able to decrypt the messages transiting on the device ?". We will call *victim* or *target* the device that is threatened by an adversary and we will call *attacks* the manipulations that $\mathcal{A}$ can do to try to recover the key.

This master thesis was split in three different parts. The goal of the first part was to adapt the software running on an existing board (the *ChipWhisperer-Lite*) to be plug-and-play and directly callable from MATLAB. Until now, in the framework of the third and fourth exercise sessions of the *LELEC2760*[1] course, the students were using MATLAB to simulate the execution of the AES. It seemed really interesting to use traces measured on a MCU running real AES implementations instead of attacking traces generated by a MATLAB function simulating false data.

Secondly, we will be interested in the theoretical nature of the attacks and how easily they are mountable to recover the target's secret key. To do so, we will study the case of attacks against an 8-bit MCU embedded in the CW-L.

The goal of the third part was to evaluate the level of side-channel security achievable by masking the AES on the CW-L. We implemented different masking schemes in Atmel assembly, assessed their security and attacked them.

This paper is split in chapters as follows: Chapter 2 is the theoretical background that gathers all the theory that is needed to fully understand this thesis, Chapter 3 will be dedicated to the ChipWhisperer-Lite, Chapter 4 will explain the different kinds of attacks that are easily mountable on the unprotected version of AES to recover the whole key, Chapter 5 will explain some technical details about the masking scheme implementations and compare them and finally,

[1] LELEC2760 : Secure Electronic Circuits and Systems, given by François-Xavier Standaert during the academic year 2016-2017, Université Catholique de Louvain, Belgium

Chapter 6 will assess the security of our implementations and show attacks that are mountable to recover information from them.

Chapter 2

Theoretical background

2.1 Cryptography

2.1.1 Introduction

Cryptography is the science of making a communication secure in the presence of third parties called adversaries. Modern cryptography uses mathematics and information theory to create protocols that ensure some cryptographic properties as *confidentiality*, *authentication*, *integrity* and *non-repudiation*.

- *Confidentiality* is the property that ensures that no information about a message is leaked through the computed ciphertext to any adversary $\mathcal{A}$ that does not know a secret which is called the *key*. The confidentiality is not ensured if an adversary can guess information about the message or the key by looking at the ciphertext.

- *Authentication* is the property that allows to verify the origin of a message. A message can be *signed* by its sender and the receiver can then check its validity and be certain of the origin of the message. Authentication is broken if an adversary can *impersonate* an user and remain unnoticed.

- *Integrity* is the property that ensures that a message can not be modified by an adversary $\mathcal{A}$ and remain unnoticed by the receiver. Integrity is broken if an adversary can *forge* a message and remain unnoticed.

- *Non-repudiation* is the property that ensures that the sender of a message can not deny the fact that he wrote and sent it.

There are two main categories of cryptographic algorithms: *asymmetric-key* (or *public-key*) cryptography and *symmetric-key* (or *private-key* or even *secret-key*) cryptography.

Asymmetric-key cryptography contains all the cryptographic protocols and algorithms in which the key k_{enc} that is used to encrypt a plaintext (= a message) and the key k_{dec} that is used to decrypt a ciphertext are not the same. Public-key cryptography mainly uses number theory to compute operations on modular sets (or rings) of numbers (e.g. $\mathbb{Z}/n\mathbb{Z} := \{z + n\mathbb{Z} | z \in \mathbb{Z}\}$). Two big examples of this use of number theory are the RSA and the Diffie-Hellman algorithms that are widely used in the internet security.

Symmetric-key cryptography contains the algorithms that use the same secret key for both the encryption and the decryption: $k = k_{enc} = k_{dec}$. Symmetric-key cryptography is usually faster, cheaper and more secure than the asymmetric-key cryptography, but the price is that all the communicating parties have to use the same secret key. In this thesis, we will only be interested in symmetric-key cryptography, as this is the field of the AES.

2.1.2 Symmetric-key cryptography

The *Symmetric-key cryptography* works as follows: the *encryption function* constructs a *ciphertext* c from a *key* k and a *plaintext* p. The *decryption function* recovers the plaintext p from the key k and the ciphertext c.

The main goals of symmetric-key cryptography are the confidentiality, the integrity and the authenticity of the messages. It works as follows: Alice wants to send a secret message m to Bob using an unsecure channel. It means that there could be an eavesdropper, named Eve, intercepting everything transiting on this channel. How can Alice send her secret message to Bob using symmetric-key cryptography? She decides with Bob of a secret key k using a secure channel (e.g. they meet in a cafe and discreetly decide of the key). She encrypts her message m using the secret key k to obtain the ciphertext c and she sends c on the unsecure channel to Bob. Bob receives c and uses the decryption function using the secret key k to recover the message m. Eve also receives the ciphertext c but since she does not know the secret key k used to encrypt the message m, she will not be able to recover it.

Symmetric-key cryptography ensures a high level of secrecy but if an adversary obtains the secret key, then she will be able to read all the previous ciphertexts encrypted using that key. There are two types of algorithms in symmetric-key cryptography, the *Stream ciphers* and the *Block ciphers*. We will only be interested in the latter.

2.1.3 Block Ciphers

A block cipher is a deterministic algorithm based on fixed-length groups of bits, called *blocks*, used to encrypt or decrypt data in symmetric cryptography. It consists of two functions, $\mathsf{Enc}_k(\cdot)$ and $\mathsf{Dec}_k(\cdot)$, respectively for the encryption and for the decryption with a secret key k. The goal of the encryption function is to hide the plaintext p into a ciphertext c that will give no information about p or k to any adversary $\mathcal{A}$ who does not know the key k. The goal of the decryption function is to recover the plaintext p given the ciphertext c and the secret key k. Both those functions are designed as *one-way functions*, i.e. given the output, it is infeasible to find the input. Here we use the term *infeasible* from a computational complexity point of view, meaning that no adversary $\mathcal{A}$ that fits in our model of security can break the block cipher in a given amount of time with a probability higher than a fixed boundary. We do not seek for *perfect security*, only for *computational security*.

Computational security is a model in which the adversary $\mathcal{A}$ does not have an unbounded computational power and has a small probability of success in breaking the scheme. Nevertheless, those capabilities are bounded by the model and we can make the algorithm as strong as wanted for a given adversary.

Let us denote $\mathcal{M}, \mathcal{K}, \mathcal{C}$ the sets of all the possible plaintexts, keys, ciphertexts, then our functions above are defined as:

$$\mathsf{Enc}_k(\cdot) : \{0,1\}^{||\mathcal{K}||} \times \{0,1\}^{||\mathcal{M}||} \mapsto \{0,1\}^{||\mathcal{C}||} : \mathsf{Enc}_k(p) = c \quad [1]$$

$$\mathsf{Dec}_k(\cdot) : \{0,1\}^{||\mathcal{K}||} \times \{0,1\}^{||\mathcal{C}||} \mapsto \{0,1\}^{||\mathcal{M}||} : \mathsf{Dec}_k(c) = p \quad [2]$$

Let n be our security parameter that we define as the size of the key in number of bits: $n = \log_2 ||\mathcal{K}||$. From now on, we will always consider the case where $||\mathcal{M}|| = ||\mathcal{K}|| = ||\mathcal{C}|| = \{0,1\}^n$. To encrypt an input message m_0 that is shorter than n bits long, we use padding. Padding consists in filling the end of the message with values. From now on, we assume that the padding only consists in adding zeros. Let $|m_0| = l$, then the encryption of m_0 with the key k is $c = \mathsf{Enc}_k(m_0||0^{n-l})$.

[1] An equivalent notation for $\mathsf{Enc}_k(p)$ is $\mathsf{Enc}(k, p)$

[2] An equivalent notation for $\mathsf{Dec}_k(c)$ is $\mathsf{Dec}(k, c)$

There are several solutions, called the *modes* to encrypt an input message m whose size is bigger than n. We cut the message into sub-messages of size n : $m = p_0||p_1||...||p_N$ where $|p_i| = n$ for $0 \leq i \leq N-1$ and $|p_N| \leq n$ with p_N padded if needed.

The Electronic Code Book (ECB) is the most basic mode, is not secure and should not be used to encrypt data. For the encryption, it encrypts all the messages p_i into different ciphertexts c_i that are concatenated together to obtain the whole ciphertext: $c_i = \mathsf{Enc}_k(p_i)$ and $c = c_0||c_1||...||c_N$. For the decryption, we retrieve the message m by decrypting the c_i's of size n: $p_i = \mathsf{Dec}_k(c_i)$ and $m = p_0||p_1||...||p_N$.

The Cipher Block Chaining (CBC) is the most common mode. It is an *iterated* block cipher mode. For the encryption, starting from a known initialization vector IV of size n, the ciphertexts are computed as: $c_0 = \mathsf{Enc}_k(IV \oplus p_0)$ and $c_i = \mathsf{Enc}_k(c_{i-1} \oplus p_i)$. For the decryption, $p_0 = \mathsf{Dec}_k(c_0) \oplus IV$ and $p_i = \mathsf{Dec}_k(c_i) \oplus c_{i-1}$.

The Cipher FeedBack (CFB) mode is another iterated block cipher mode where the Dec function can be implemented using Enc. For the encryption, $c_0 = p_0 \oplus \mathsf{Enc}_k(IV)$ and $c_i = p_i \oplus \mathsf{Enc}_k(c_{i-1})$. For the decryption, $p_0 = c_0 \oplus \mathsf{Enc}_k(IV)$ and $p_i = c_i \oplus \mathsf{Enc}_k(c_{i-1})$.

The Output FeedBack (OFB) mode is nearly the same as the CFB, except that the we feed the next round with the value computed before the `xor` operation. For the encryption, $O_0 = \mathsf{Enc}_k(IV)$, $c_0 = p_0 \oplus O_0$, $O_i = \mathsf{Enc}_k(O_{i-1})$ and $c_i = p_i \oplus O_i$. For the decryption, $O_0 = \mathsf{Enc}_k(IV)$, $p_0 = O_0 \oplus c_0$, $O_i = \mathsf{Enc}_k(O_{i-1})$ and $p_i = O_i \oplus c_i$.

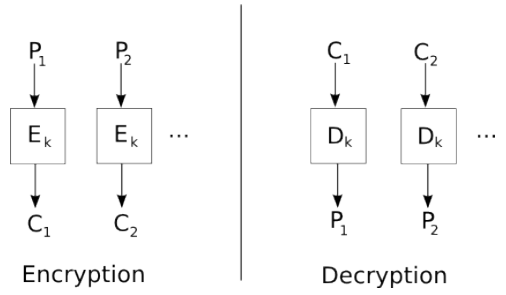


FIGURE 2.1 – ECB

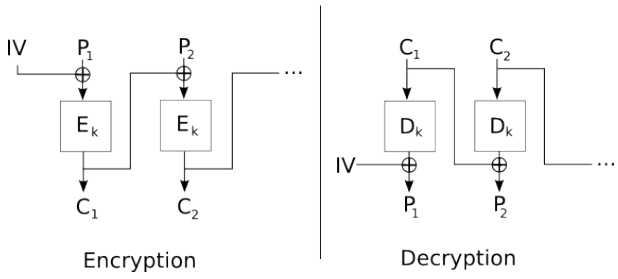


FIGURE 2.2 – CBC

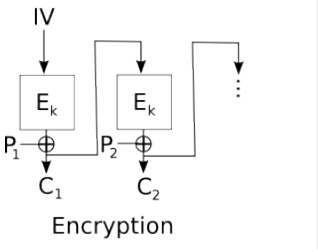


FIGURE 2.3 – CFB

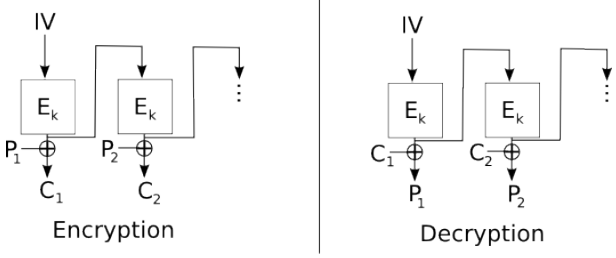


FIGURE 2.4 – OFB

FIGURE 2.5 – The four modes are showed here for indices beginning at 1 instead of 0. The images are from [3]

[3] `http://www.cs.cornell.edu/courses/cs513/2007fa/`

2.1.4 Advanced Encryption Standard

The *Advanced Encryption Standard* (AES) is the current standard for encryption and decryption in symmetric cryptography and has been since 2001. It is the substitude of the *Data Encryption Standard* (DES). The AES implementation is the *Rijndael* algorithm, whose name is the contraction of the names of its inventors, Vincent Rijmen and Joan Daemen, and its full specification is given in [8]. It is a block cipher, implementing a *Substitutions Permutations Network* (SPN). It works with a block size of 128 bits and with three different key sizes : 128 bits, 192 bits and 256 bits. We will furthermore refer to AES-128 for the first one, AES-192 for the second one and AES-256 for the latest one. When not mentioned, we refer by AES to the AES-128 algorithm. The number of rounds N_r depends on the key size, AES-128 has 10 rounds, AES-192 has 12 rounds and AES-256 has 14 rounds. In this thesis, we only work with the most spread AES algorithm, which is the AES-128.

State representation

We usually represent the different values of the plaintext and the subkey during the algorithm by a 4×4 array containing bytes (elements of $\mathbb{F}_{2^8}$) called the *state*. Reminders of operations in $\mathbb{F}_{2^8}$ are presented in the appendix A.

Let $\mathbf{s}^r$ denote the state of the AES at the beginning of the round r:

$$\mathbf{s}^r = \begin{pmatrix} s^r_{0,0} & s^r_{0,1} & s^r_{0,2} & s^r_{0,3} \\ s^r_{1,0} & s^r_{1,1} & s^r_{1,2} & s^r_{1,3} \\ s^r_{2,0} & s^r_{2,1} & s^r_{2,2} & s^r_{2,3} \\ s^r_{3,0} & s^r_{3,1} & s^r_{3,2} & s^r_{3,3} \end{pmatrix}$$

In this matrix, each $s^r_{i,j} \in \mathbb{F}_{2^8}$. The four 32-bit words of p or k are placed columnwise in the state matrix. We denote by $\mathbf{s}$ the instantaneous state at a given step of the AES computation.

At the beginning of the AES, the state contains the plaintext: $\mathbf{s}^0 \leftarrow p$.
Let $p = p_0||p_1||...||p_{15}$ where each $|p_i| = 8$ bits. The bytes of the plaintext are placed in the following order in the initial state (before the first AddRoundKey):

$$\mathbf{s}^0 = \begin{pmatrix} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{pmatrix}$$

Key Expansion

Each round of the AES uses a different *subkey* derived from k, the *master key*. The computation of the a new subkey only requires the last subkey, so each new subkey can be computed at the beginning of each round or all the subkeys can be computed at the same time at the beginning of the algorithm. We call the *key expansion* the computation of each of the subkeys starting from the master key.

Let $\mathbf{k}$ denote the master key k in the state matrix representation:

$$k = k_0||k_1||k_2||...||k_{15} \Leftrightarrow \mathbf{k} = \begin{pmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{pmatrix}$$

Let $\mathbf{k}^r$ denote the subkey associated to the r^{th} round of the AES.
The KeyExpansion algorithm works as follows for the AES-128:

1. Let define $\begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} \end{pmatrix} \leftarrow \mathbf{k} = \begin{pmatrix} \mathbf{w}_{*,1} & \mathbf{w}_{*,2} & \mathbf{w}_{*,3} & \mathbf{w}_{*,4} \end{pmatrix}$.

   We refer to the 4 bytes of the n^{th} column by $\mathbf{w}_{*,n} = \begin{pmatrix} w_{1,n} \\ w_{2,n} \\ w_{3,n} \\ w_{4,n} \end{pmatrix}$

2. For each $5 \leq j \leq 4(N_r + 1)$, compute $\mathbf{w}_{*,j} = \mathbf{w}_{*,j-4} \oplus t$ where

$$t = \begin{cases} \mathsf{RotWord}(\mathsf{SubWord}(\mathbf{w}_{*,j-1})) \oplus \mathsf{Rcon}_{j/4} & \text{if } (j \bmod N_k) == 0 \\ \mathbf{w}_{*,j-1} & \text{otherwise} \end{cases}$$

3. For each $1 \leq r \leq 10$, define $\mathbf{k}^r := \begin{pmatrix} \mathbf{w}_{*,4r+1} & \mathbf{w}_{*,4r+2} & \mathbf{w}_{*,4r+3} & \mathbf{w}_{*,4r+4} \end{pmatrix}$

The $\mathsf{SubWord}(x)$ operation corresponds to the application of the S-box to each of the four bytes $(x_i)_{i=1}^4$ of the input x.

The $\mathsf{RotWord}(x)$ operation rotates the bytes of the input x by one position from the bottom to the top:

$$\mathsf{RotWord}(\mathbf{w}_{*,n}) = \begin{pmatrix} w_{2,n} \\ w_{3,n} \\ w_{4,n} \\ w_{1,n} \end{pmatrix}$$

The Rcon_n value is a vector containing a constant value which is $(\mathtt{0x02})^{n-1}$ in $\mathbb{F}_{2^8}$:

$$\mathsf{Rcon}_n = \begin{pmatrix} \mathtt{0x02}^{n-1} \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Whole AES execution

Each AES round is composed of the same steps, except for the last one:

1. SubBytes: In this step, the same transformation is applied to each byte of the state. It is computed as an affine transformation of x^{-1} in $\mathbb{F}_{2^8}$ for an input byte x. This operation can be precomputed for all the different input values to create the *substitution box* or S-box table. There is a one-to-one relation, i.e. a bijection, between an input byte and the corresponding output byte of the S-box. This operation is non-linear in $\mathbb{F}_{2^8}$.

$$\mathsf{SubBytes}(\mathbf{s}) = \begin{pmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{pmatrix}$$

   In the above operation, $s'_{i,j} = \mathsf{S\text{-}box}(s_{i,j})$. Further explanations on the S-box are available in the section 2.5.3.

2. ShiftRows: In this step, the rows of the state matrix are shifted. The first row stays the same, the second row is shifted one time to the left, the third row two times and the last row three times.

$$\mathsf{ShiftRows}(\mathbf{s}) = \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{pmatrix}$$

3. MixColumns: In this step, the columns are considered as polynomials in $\mathbb{F}_{2^8}$ and are multiplied by a fixed polynomial $a(x) = \texttt{0x03}x^3 + x^2 + x + \texttt{0x02}$ modulo $x^4 + 1$, shifter according the the state column. This corresponds to a matrix multiplication in $\mathbb{F}_{2^8}$ between the state and the following matrix:

$$\mathsf{MixColumns}(\mathbf{s}) = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} * \mathbf{s} \mod x^4 + 1$$

4. AddRoundKey: In this step, the state matrix is xored with the round key (= the subkey) associated to the round.

$$\mathsf{AddRoundKey}(\mathbf{s}) = \mathbf{s} \oplus \mathbf{k}^r = \begin{pmatrix} s_{0,0} \oplus k_0^r & s_{0,1} \oplus k_4^r & s_{0,2} \oplus k_8^r & s_{0,3} \oplus k_{12}^r \\ s_{1,0} \oplus k_1^r & s_{1,1} \oplus k_5^r & s_{1,2} \oplus k_9^r & s_{1,3} \oplus k_{13}^r \\ s_{2,0} \oplus k_2^r & s_{2,1} \oplus k_6^r & s_{2,2} \oplus k_{10}^r & s_{2,3} \oplus k_{14}^r \\ s_{3,0} \oplus k_3^r & s_{3,1} \oplus k_7^r & s_{3,2} \oplus k_{11}^r & s_{3,3} \oplus k_{15}^r \end{pmatrix}$$

The last round is the same as described above without the MixColumns operation. The first round has an extra AddRoundKey layer with the master key at its beginning.

The whole AES pseudo-code is given in the Algorithm 1.

Algorithm 1 AES

Inputs: a plaintext p, a master key k
Output: a ciphertext c

$\mathbf{s}^0 \leftarrow p$
$\mathbf{s}^1 \leftarrow \mathbf{s}^0 \oplus \mathbf{k}$
$(\mathbf{k}^1, \mathbf{k}^2, ..., \mathbf{k}^{N_r}) \leftarrow \mathsf{KeyExpansion}(k)$
for $r = 1$ **to** $N_r - 1$ **do**
    $\mathbf{s} \leftarrow \mathsf{SubBytes}(\mathbf{s}^r)$
    $\mathbf{s} \leftarrow \mathsf{ShiftRows}(\mathbf{s})$
    $\mathbf{s} \leftarrow \mathsf{MixColumns}(\mathbf{s})$
    $\mathbf{s}^{r+1} \leftarrow \mathbf{s} \oplus \mathbf{k}^r$
$\mathbf{s} \leftarrow \mathsf{SubBytes}(\mathbf{s}^{N_r})$
$\mathbf{s} \leftarrow \mathsf{ShiftRows}(\mathbf{s})$
$\mathbf{s} \leftarrow \mathbf{s} \oplus \mathbf{k}^{N_r}$
return $\mathbf{s}$

Remark: There is no need to keep the different $\mathbf{s}^r$ in memory during the whole execution.

2.2 Statistics

2.2.1 Introduction

Statistics are the heart of most of the mathematical attacks mountable on physical devices. In this section, we will briefly recall the basics of statistics needed to understand the principles of the attacks described further.

2.2.2 Notations

Here will be explained the statistical notations used in this thesis. In this section, X is a random variable (r.v.) that can take any value x in the set $\mathcal{X} = \{x_1, x_2, \cdots, x_n\}$ with $n = ||\mathcal{X}||$.

- $\mathbb{P}[X = x]$ denotes the probability that the r.v. X takes the value x.
- $\mathbb{E}[X]$ denotes the expected value of the r.v. X: $\mathbb{E}[X] = \frac{1}{n}\sum_{i=1}^{n} x_i \mathbb{P}[X = x_i]$
- $\mathsf{mean}(\mathcal{X})$ denotes the mean value of the set $\mathcal{X}$: $\mathsf{mean}(\mathcal{X}) = \frac{1}{n}\sum_{i=1}^{n} x_i$.
- $\mathsf{std}(\mathcal{X})$ denotes the standard deviation of the set $\mathcal{X}$, with $\mu = \mathsf{mean}(\mathcal{X})$: $\mathsf{std}(\mathcal{X}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$, also denoted σ_X.
- $\mathbb{V}\mathsf{ar}[X]$ denotes the variance of the r.v. X: $\mathbb{V}\mathsf{ar}[X] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 = \mathbb{E}[(X - \mu)^2]$ where μ is the mean value of X. $\mathbb{V}\mathsf{ar}[X]$ is also denoted σ_X^2.
- $\mathsf{Cov}(X, Y)$ denotes the covariance between the r.v. X and Y: $\mathsf{Cov}(X, Y) = Cov(Y, X) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])]$. $\mathsf{Cov}(X, Y)$ is also denoted σ_{XY}.
- $X \sim \mathcal{N}(\mu, \sigma)$ denotes that the r.v. X follows a normal distribution of mean μ and standard deviation σ: $\mathbb{P}[X = x] = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$
- ρ_{XY} denotes the Pearson's correlation coefficient:

$$\rho_{XY} = \frac{\sigma_{XY}}{\sigma_X \sigma_Y} = \frac{\mathbb{E}[(X - \mu_X)(Y - \mu_Y)]}{\sqrt{\mathbb{E}[(X - \mu_X)^2]\mathbb{E}[(Y - \mu_Y)^2]}}$$

2.2.3 Welch's t-test

Welch's t-test (or unequal variances t-test) is a statistical test that is used in many applications as a distinguisher to check whether two populations have equal means from a limited number of samples under the assumption that both populations follow normal distributions with different variances.

Let A and B be two sets with means μ_A and μ_B, with sample sizes N_A and N_B and with sample variances σ_A^2 and σ_B^2, then the Welch's t-test Δ is defined as:

$$\Delta = \frac{\mu_A - \mu_B}{\sqrt{\frac{\sigma_A^2}{N_A} + \frac{\sigma_B^2}{N_B}}}$$

2.3 Side-Channel Attacks

2.3.1 Introduction

A *Side-Channel Attack* (SCA) is a term used to describe any attack based on the information that leaks from the physical implementation of a cryptosystem. The term *Side-Channel* is used to describe any way that is not part of the mathematical definition of the cryptographic algorithm in which the system can leak information about its secret internal variables.

There are plenty of SCA and we give here a non-exhaustive list of them:

- Power Analysis Attacks (PAA)
- Differential Fault Attacks (DFA)
- Timing attacks

- Electromagnetic attacks
- Data remanence attacks

In this thesis, we will only be interested in the first type of attack listed above: the *Power Analysis Attacks* (PAA).

A system that is secure against the side-channel attacks is said to be *SCA-secure*, *tamper resistant* or *tamper resilient*.

2.3.2 Classification of the SCA

We usually classify the different types of SCA in different groups:

- they can be *invasive*, *semi-invasive* or *non-invasive*
- they can be *active* or *passive*

Non-invasive attacks regroup all the attacks in which no harm is done to the device. We only record and manipulate it. Semi-invasive attacks are attacks where we slightly modify the device in order to mount the attack (e.g. chip depackaging). Invasive attacks is the category in which everything can be done to recover the secrets hidden, no matter how harmful it is for the device (e.g. chip slicing). In term of practical costs, the non-invasive attacks are cheap, the semi-invasive usually are affordable and the invasive attacks are expensive and hard to mount.

Passive attacks regroups all the attacks that do not interfere with the behavior of the chips, i.e. they are unnoticeable (e.g. PAA). Active attacks are the attacks where the attacker interferes with the behavior of the chip to perform the attack (e.g. DFA). The device will not work as expected when an active attack occurs.

2.3.3 Development of countermeasures

Side-channel attacks are a real threat to the security of cryptosystems. Countermeasures have been developed to counter those attacks. The countermeasures that interest us will be explained later, but usually have a cost: a huge overhead. An implementation with countermeasures against SCA on a device usually takes orders of magnitude more time to do a given task. To give some numbers: a basic implementation of AES on a 8-bit MCU needs roughly 3500 cycles to output the ciphertext, while a masked implementation of the same algorithm at the second order usually needs more than 150,000 cycles to output the ciphertext.

2.4 Power Analysis Attacks

2.4.1 Introduction

In 1999, Kocher *et al.* published [19] and showed the world how strong can power analysis attacks be. Monitoring the power consumption of a chip or a smart card was most of the time enough to completely reveal the secret hidden in the system. The leakage does not come from the mathematical properties of the algorithms that are implemented but from the nature itself of the implementation in circuits. Circuits are build based on the laws of physics, the same physics that regulates our world. PAAs are attacks that are both passive (i.e. unnoticeable from the device point of view) and non-invasive (no harm done to the device).

Those attacks are based on a cryptographic model called the *chosen-plaintext attack* model. In this model, the attacker $\mathcal{A}$ has access to an encryption oracle $\mathcal{E}$ (and/or decryption oracle $\mathcal{D}$) that gives him the capability to encrypt (and/or decrypt) any plaintext-ciphertext pair (p, c). This oracle can be seen as a cryptosystem black box with an embedded hidden secret key k that returns a ciphertext $c \leftarrow \mathcal{E}(p) = \mathsf{Enc}(k, p)$ given a plaintext p or a plaintext $p \leftarrow \mathcal{D}(c) = \mathsf{Dec}(k, c)$

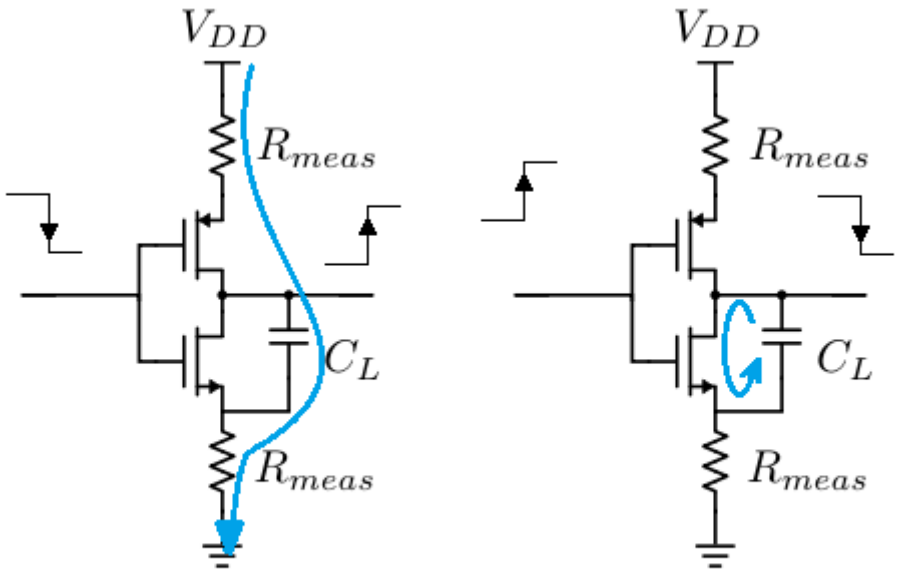


FIGURE 2.6 – Current flow when the voltage input switches from high to low (left) and from low to high (right)

given a ciphertext c. The attacker can take advantage of the information that is leaked from the knowledge of both the plaintext and the ciphertext of a plaintext-ciphertext pair since the goal is to recover k. In a word, $\mathcal{A}$ always knows both p and c to help him recover k.

2.4.2 Origin of the leakage

The leakage comes from the fact that a logic gate in a digital circuit implemented in CMOS-technology will not draw the same amount of current while working with a logical 1 (e.g. a voltage V_{DD}) and a logical 0 (e.g. the ground voltage). To explain it, let us take the example of a CMOS inverter composed of a NMOS and a PMOS transistor as shown in FIGURE 2.6. In the left-hand circuit, the blue arrow shows the path that the current has to flow to charge the capacitance when the input is a logical 0 value. In the right-hand circuit, the capacitance is discharged because the NMOS is conducting current.

The power consumption of a digital circuit can be expressed as

$$P_{inst} = P_{dyn} + P_{stat} \tag{2.1}$$

The P_{stat} factor represents the static power. It does not leak any information, it is simply related to the leakage currents ($I_{substrate}, I_{junction}, I_{gate}$) flowing through the different parts of the transistors due to physics: $P_{stat} = I_{leak} V_{DD}$.

The P_{dyn} factor is the dynamic power. It is the one which depends on the value that is currently treated. It is composed of two parts: the switching power P_{sw} and the short-circuit power P_{sc}.

$$P_{dyn} = P_{sw} + P_{sc} \tag{2.2}$$

The short-circuit power P_{sc} is the power that is leaked during a transition at the input. During this transition, there is a given amount of time Δt_{sc} in which both the PMOS and the NMOS transistors are conducting current. This creates a direct path between the supply voltage

V_{DD} and the ground GND. We denote $I_{sc,avg}$ the average current flowing from the supply to the ground during this time period and we obtain the following relation:

$$P_{sc} = N_{nodes}\alpha_F \Delta t_{sc} I_{sc,avg} V_{DD} f_{clk} \tag{2.3}$$

In the above equation, N_{nodes} stands for the number of nodes in the circuit, α_F is the activity factor and f_{clk} is the clock frequency.

The switching power is the one that will leak the information since it will depend on the value of the bit that is loaded on the load capacitance C_L. The switching power of a digital circuit can be computed as:

$$P_{sw} = \frac{1}{2} N_{nodes}\alpha_F C_L V_{DD}^2 f_{clk} \tag{2.4}$$

We denote $P_{0\to1}$ the switching power from the logic 0 to the logic 1 and $P_{1\to0}$ the switching power from the logic 1 to the logic 0. The power analysis attacks are possible because $P_{0\to1} \neq P_{1\to0}$.

2.4.3 Simple Power Analysis

The simple power analysis (SPA) is the simplest possible mountable attack for an attacker than can record the power consumption of a chip. It consists of directly interpreting power consumption measurements collected during cryptographic operations. It is not directly an attack in itself on the block ciphers but it is very powerful against some applications. It is often a preliminary step prior to other more advanced attacks.

We call a *trace* the power consumption measurements done on the attacked device. A trace measured on the ChipWhisperer-Lite running an unmasked AES, the Rijndael Furious, is shown in FIGURE 2.7. The y-axis represents the power consumption multiplied a variable gain fixed on the LNA of the board. For that reason, units are not shown.

2.4.4 Differential Power Analysis

First-order DPA

The Differential Power Analysis (DPA) as introduced in [19] is a powerful attack that can be used to recover the secret key of many unprotected cryptosystems.

As an illustration, a generic DPA attack is described here, inspired from [16, 25]. Let us say that an adversary $\mathcal{A}$ can record n traces $(\mathbf{T}_{i,*})_{i=1}^{n}$ of m samples each to obtain the set of traces

$$\mathbf{T} = \begin{pmatrix} \mathbf{T}_{1,*} \\ \mathbf{T}_{2,*} \\ \vdots \\ \mathbf{T}_{n,*} \end{pmatrix} = \begin{pmatrix} T_{1,1} & T_{1,2} & \cdots & T_{1,m} \\ T_{2,1} & T_{2,2} & \cdots & T_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ T_{n,1} & T_{n,2} & \cdots & T_{n,m} \end{pmatrix}$$

$\mathcal{A}$ wants to recover a small portion s of the secret data, small enough so that he can do an exhaustive search on all the possible values of s. Let g be a boolean selection function that returns 0 or 1 depending on a boolean condition and let $I(x, s)$ be an intermediate value that only depends on a known data x and s, the small portion of the secret. He then defines, for each possible value $\hat{s}$ of s, the two sets $S_b(\hat{s})$, $b \in \{0, 1\}$ as:

$$S_b(\hat{s}) = \{x \mid g(I(x, \hat{s})) = b\} \text{ for } b \in \{0, 1\}$$

$\mathcal{A}$ then computes the first-order *differential traces* $\Delta_1(\hat{s}, j)$ as

$$\Delta_1(\hat{s}, j) = \left| \langle \mathbf{T}_j \rangle_{x \in S_0(\hat{s})} - \langle \mathbf{T}_j \rangle_{x \in S_1(\hat{s})} \right|$$

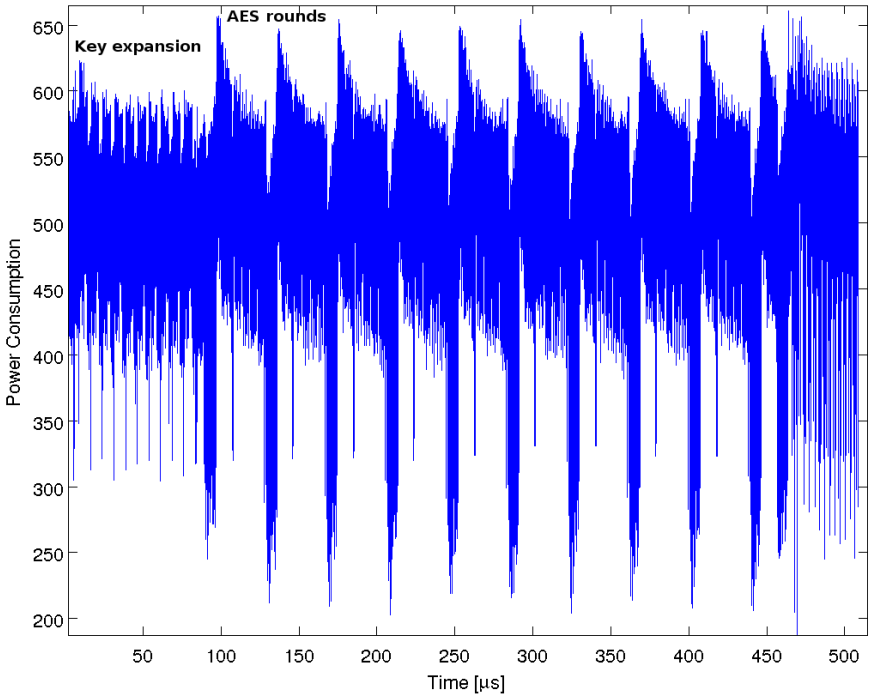


FIGURE 2.7 – Trace from a SPA of the AES-furious running on the ChipWhisperer-Lite target

where $\langle \mathbf{T}_j \rangle_{x \in S_b(\hat{s})}$ denotes the mean over each of the n_b traces in set S_b at the timestep j:

$$\langle \mathbf{T}_j \rangle_{x \in S_b(\hat{s})} = \frac{1}{n_b} \sum_{x \in S_b(\hat{s})} T_{x,j}$$

Hence, $\Delta_1(\hat{s}, j)$ is the difference of the average power consumption between the sets $S_0(\hat{s})$ and $S_1(\hat{s})$. Under the assumptions that (i) the intermediate value $I(x, s)$ always occurs at the same timestep $j = j^*$ and that (ii) there are enough values of x so that $I(x, \hat{s})$ is close to uniform distribution, then the DPA trace $\Delta_1(\hat{s}, j)$ with the highest peak value at timestep j^* is likely to be the one for which $\hat{s} = s$. Why? Because for any $\hat{s} = s' \neq s$, the sets S_0 and S_1 will be filled according to b which has a probability of one half to be correct. It means that for a big number n of traces the limits will tend towards zero: $\lim_{n \to \infty} \Delta_1(s', j) \approx 0$ and the boolean function b will be uncorrelated to the power consumption of the device. On the contrary, if $\hat{s} = s$, then the bit b will always be well computed and the correct values will fill the sets S_0 and S_1. The boolean function g will then be correlated to the power consumption and the DPA trace will show peaks at the timesteps where the correlation between g and $\mathbf{T}$ holds. It allows $\mathcal{A}$ to recover the secret portion s. To recover the whole secret, the same process can be repeated on each secret portion s.

2.4.5 Correlation Power Analysis

Introduction

Correlation Power Analysis (CPA) has been introduced in [4] and is an attack in which the attacker $\mathcal{A}$ assumes that there is a linear correlation between a model of the data and the power consumption. This assumption holds in practice, since the parts of a MCU that consume the

most depend on the data. CPA uses Pearson's correlation coefficient ρ as a distinguisher and the *Hamming distance* as a model. The *Hamming weight* of a given word is its number of bits set. We will use the notation $H(\cdot)$ to write the Hamming weight operation. In a word containing m independent and uniformly distributed bits, the average Hamming weight is $\mu_H = m/2$ and the variance is $\sigma_H^2 = m/4$. The operation $H(D \oplus R)$ is called the *Hamming distance* between D and R. The Hamming distance model assumes that the power consumption of the device is correlated with the Hamming distance between the data word D that is processed and the original state R of the word:

$$W = aH(D \oplus R) + b$$

where W is the power consumption of the device while doing computations using the data word D and R is the original state of the data word. The parameter a is a linear scaling parameter and b is an independent variable depending on all the data-independent power consumptions as the noises or the data-independent routines.

First-order CPA

The first-order CPA is used to recover the key when there is no masking involved. The correlation is computed by an estimator of the Pearson's correlation coefficient $\hat{\rho}$. Let n be the number of traces in the set W of the power consumption traces and let W_i be the i^{th} trace in the set. Let also m be the number of points in each trace, M be a set of random data words of ℓ bits each, let M_i be the i^{th} word in the set and let $H_{i,r} = H(M_i \oplus r)$ be the set of predicted Hamming weights of the random data word M_i. The estimator at the timestep $t \in [1, m]$ is then defined as:

$$\hat{\rho}_{WH}(t, r) = \frac{n \sum_{i=1}^{n} W_i(t) H_{i,r} - \sum_{i=1}^{n} W_i(t) \sum_{i=1}^{n} H_{i,r}}{\sqrt{(n \sum_{i=1}^{n} (W_i(t))^2 - (\sum_{i=1}^{n} W_i(t))^2)(n \sum_{i=1}^{n} H_{i,r}^2 - (\sum_{i=1}^{n} H_{i,r})^2)}}$$

We are only interested in the timestep t^* and the word r^* that maximize the absolute value of the coefficient $\hat{\rho}_{WH}$:

$$(t^*, r^*) = \arg\max_{(t,r)} |\hat{\rho}_{WH}(t, r)|$$

In other words, $\hat{\rho}_{WH}(t, r)$ can be computed for each $0 \leq r < 2^\ell$ and each $1 \leq t \leq m$. The resulting array has a maximum value at the position (t^*, r^*), where t^* is called a *point of interest* (POI) of the trace, i.e. a point where the sensitive data is handled and r^* is the value that is related to the secret (it can be the key word or it is directly related to the key word, such that finding r^* is the same as finding the key word).

2.4.6 Template Attack

Introduction

The template attack is a powerful type of attack in which the adversary $\mathcal{A}$ *profiles* the behavior of the target device by doing a training on a similar device that he can fully control. It is done in two phases, the profiling phase and the online attack phase. The profiling phase is time-expensive, but once it is completed, the online attacks become fast.

Univariate Template Attack

The univariate template attack can be used to recover the key of a device which is not secure at first order. In an univariate attack, the attacker only focuses on one point of interest. This POI can be found by using the timestep maximizing the correlation coefficient of a CPA or by using leakage detection tests that will be described later.

During the profiling phase, the attacker $\mathcal{A}$ uses a device similar to the one under attack that he can use to encrypt data. $\mathcal{A}$ runs N_p encryptions of random plaintexts with random keys

and records the N_p corresponding template traces $\mathbf{T}_p$. $\mathcal{A}$ then constructs for each S-box of the first round 256 templates $\mathcal{L}_i$, one for each value of i, the output of the focused S-box. All those templates are assumed to follow a normal distribution $\mathcal{L}_i \sim \mathcal{N}(\mu_i, \sigma_i)$:

$$\mathcal{L}_i(l) = \mathcal{N}(l|\mu_i, \sigma_i) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{(l-\mu_i)^2}{2\sigma_i^2}\right)$$

The goal of this phase is to evaluate as accurately as possible the values of μ_i and σ_i for all the values $i \in [0, 255]$. The higher the number N_p of template traces, the higher the accuracy of the templates, and the higher the efficiency of the attack. For the online attack, $\mathcal{A}$ runs a small number of encryptions Q on the target device and records the traces $\mathbf{T}_o$. For each key candidate k, he computes the *Bayes* probability:

$$\mathbb{P}[\mathsf{S\text{-}box}(p \oplus k) = i] = \frac{\mathcal{L}_i(l)}{\sum_{j=0}^{255} \mathcal{L}_j(l)} = \frac{\mathcal{N}(l|\mu_i, \sigma_i)}{\sum_{j=0}^{255} \mathcal{N}(l|\mu_j, \sigma_j)}$$

where p and i are the plaintext byte and the intermediate value corresponding to the attacked S-box. Then, by taking the product of these *Bayes* probabilities, he can find the key byte that has the highest chance of being the good one. If Q and N_p are large enough, $\mathcal{A}$ recovers the correct key byte. He can then repeat the same steps and change the focused S-box to attack the whole key.

Towards a lower value of Q

In practice, a trade-off can be made between N_p and Q. The higher is N_p, the more accurate is the profiling, and the more accurate will be the values of the different $\mathcal{L}_i$. If N_p is really huge and if the noise on the attacked device is low enough, we can expect to recover the correct key by only observing one trace of an encryption on the attacked device. This attack could be very interesting to break keys almost in real-time. But the fact that there is intrinsic noises in the electronics of the device will make this one-trace attack hard to put in practice. When recording only one trace for the online attack (i.e. $Q = 1$), we record the single trace $T = T_{pure} + \nu$ where T_{pure} is the perfect signal that we want to recover and $\nu \sim \mathcal{N}(0, \sigma_\nu)$ is the perceived noise which is assumed to be a zero-mean additive white Gaussian noise (AWGN). The attack can only work if the noise power σ_ν^2 is low enough for the trace to not be confused with an incorrect template at the POI.

A higher-value of Q, the number of traces recorded during the online attack is used to attenuate the effects of the noise on the device. Assuming that the noise is a zero-mean AWGN, its expected value is zero $\mathbb{E}[\nu] = 0$, and it will be averaged out.

Multivariate template attack

When attacking masked devices that do not leak at first order, the univariate template attack will not be able to break the key. This is because the sensitive information on the device (i.e. the key) is split into shares that are not directly dependent on the key at one timestep. To successfully attack an implementation with a security level of order d, you need to simultaneously attack $d+1$ points of interest. The profiling phase is the same except that the learning is done for tuples of points. For the online attack, the attacker $\mathcal{A}$ must compute multivariate statistics. When attacking at the order $d > 1$, $\mathcal{A}$ must compute the matrix of covariances, where X_i is the random variable corresponding to the leakage at the i^{th} POI:

$$\mathbf{\Sigma} = \begin{pmatrix} \mathbb{V}\mathsf{ar}[X_1] & \mathsf{Cov}(X_1, X_2) & \cdots & \mathsf{Cov}(X_1, X_{d+1}) \\ \mathsf{Cov}(X_2, X_1) & \mathbb{V}\mathsf{ar}[X_2] & \cdots & \mathsf{Cov}(X_2, X_{d+1}) \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{Cov}(X_{d+1}, X_1) & \mathsf{Cov}(X_{d+1}, X_2) & \cdots & \mathbb{V}\mathsf{ar}[X_{d+1}] \end{pmatrix}$$

Note that this matrix is symmetric w.r.t. its diagonal since $\mathsf{Cov}(A, B) = \mathsf{Cov}(B, A)$. $\mathcal{A}$ must also compute the vector of means of all these variables: $\boldsymbol{\mu} = \begin{pmatrix} \mu_{X_1} & \mu_{X_2} & \cdots & \mu_{X_{d+1}} \end{pmatrix}^T$

The probability function is computed as follows for a vector $\boldsymbol{\ell} = \begin{pmatrix} \ell_1 & \ell_2 & \cdots & \ell_{d+1} \end{pmatrix}^T$ where ℓ_i is the leakage value of the i^{th} POI:

$$\mathcal{L}(\boldsymbol{\ell}) = \frac{1}{\sqrt{(2\pi)^{d+1}\mathsf{det}(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\boldsymbol{\ell} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{\ell} - \boldsymbol{\mu})\right)$$

where $\mathsf{det}(\cdot)$ is the determinant operator. If the value of $\mathcal{L}(\boldsymbol{\ell})$ is high, then the corresponding $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ have a high probability to be the correct ones. The correct key byte is recovered by multiplying these $\mathcal{L}(\boldsymbol{\ell})$ together and by taking the likeliest one.

2.4.7 Countermeasures

Countermeasures aim to make the PAAs harder. To do so, countermeasures must make the power consumption of the device independent from the data that is processed. There is in practice two different kinds of countermeasures to thwart the PAAs: the *hiding* and the *masking*.

Hiding consists in making the power consumption at each cycle either random or constant, e.g. dual rail logic style. This makes the attacks more difficult in practice, but still not infeasible since there will always be a little part of power that is data-dependent.

Masking consists in adding randomness into the process to avoid having correlation between the power and the data. In a masked implementation of a cryptosystem, every sensitive variable is split in different shares such that their recombination gives the correct value.

In this thesis, we will be only interested in masking schemes since they are widely used in practice, can be fully implemented in software and do not need any hardware change. Moreover, hiding is not as studied in the academical world because it is less theoretical.

2.5 Masking the AES

2.5.1 Introduction

Leaving the AES unprotected against a side-channel adversary (SCA) $\mathcal{A}$ that has access to the power traces of encryptions on a micro-processor does not hold the secrecy. In fact we will see in Chapter 4 that only a small number of traces is needed in order to recover the correct key.

PAAs are feasible because there are correlations between a model of the data (e.g. the Hamming weight of a byte of the state during the computations, or the mean values of the leakages at given points, ...) and the power consumption of the chip. The secret information that we want to secure affects the power consumption of the chip. The main idea is then to remove this correlation by splitting the sensitive data into several shares that are mutually independent from each other. To secure the physical implementation of a cryptographic algorithm, we split every *sensitive data* into shares. The masking at the d^{th} order of the sensitive data x is

$$x = x_0 \perp x_1 \perp ... \perp x_{d-1} \perp x_d \tag{2.5}$$

where every $(x_i)_{0 \leq i \leq d}$ is called a *share* and where $\perp$ is a group operation. There are three main types of masking: the *Boolean masking* that is used to mask Boolean operations such as the `xor` ($\oplus$) or the shift ($<<$), the *arithmetic masking* that is used to mask arithmetic operations such as the addition or the substraction in $GF(2^n)$ and the *multiplicative masking* that is used to hide the multiplication in $GF(2^n)$. We will here only be interested in two types of maskings, each one with its own group operation. In the case of *additive masking*, the group operation is the exclusive-or, denoted $\oplus$ and in the case of *multiplicative masking*, the group operation is the field

multiplication, denoted $\odot$. Every share is random and x_0, the *secret*, is computed such that the above equality is correct:

$$x_0 = x \perp x_1 \perp ... \perp x_{d-1} \perp x_d \tag{2.6}$$

In a masked implementation of the AES, the computations are not directly performed on the key k and the plaintext p but they are performed on the shares $\{k_i\}_{i=0}^{d}$ and $\{p_i\}_{i=0}^{d}$ instead. The equation 2.5 is still valid and we then have that $\perp_{i=0}^{d} k_i = k$ and $\perp_{i=0}^{d} p_i = p$. Masking a block cipher at the order d must ensure two properties:

- *Completeness*: at the end of the computation, the recombination of all the ciphertext shares must yield to the correct ciphertext. It means that, starting from the shares of a plaintext $p_0, p_1, ..., p_d$ and of a key $k_0, k_1, ..., k_d$, the resulting ciphertext shares $c_0, c_1, ..., c_d$ should be such that $\perp_{i=0}^{d} c_i = c = \mathsf{Enc}(\perp_{i=0}^{d} k_i, \perp_{i=0}^{d} p_i)$.
- *d^{th} order SCA security*: any tuple of d or less intermediate variables should leak no information about the sensitive variables.

A difficulty in masking the AES is masking the non-linear part in $GF(2^8)$ of the algorithm, which is the S-box. In [24], Rivain and Prouff came with a really elegant solution on how to mask the AES. But there was an undetected theoretical security flaw hiding in one of their algorithm. This flaw is difficult to exploit in practice but in [7], Coron *et al.* explained it and proposed a solution to fix it. In [12], Genelle *et al.* proposed a completely different solution, using both additive and multiplicative sharing to hide the AES. All those solutions will be explored later.

As explained in [1], if we assume an additive white Gaussian noise ν on the chip, $\nu \sim \mathcal{N}(0, \sigma_\nu)$, then the data complexity of a side-channel attack against a masked implementation at order d is proportional to $(\sigma_\nu^2)^d$. This roughly means that the difficulty to attack a masked implementation increases exponentially with the masking order d.

2.5.2 The Ishai-Sahai-Wagner Scheme

In [15], Ishai *et al.* proposed a solution to secure an hardware implementation of any cryptographic algorithm at any order d. They did their work for circuit only containing `not` and `and` gates (since with only those two gates, any fully combinatorial logic can be implemented). Securing a `not` gate at any order d is the same as securing one `not` since: $\overline{x} = \overline{x_0} \oplus x_1 \oplus ... \oplus x_d$ where $\overline{\cdot}$ denotes the bitwise `not` operation.

The real challenge came from securing the `and` gate. Let a and b be binary digits (i.e. $\in GF(2)$), that are secret and split into $d+1$ shares $\{a_i\}_{i=0}^{d}$ and $\{b_i\}_{i=0}^{d}$ such that $\bigoplus_{i=0}^{d} a_i = a$ and $\bigoplus_{i=0}^{d} b_i = b$. To securely compute $\texttt{and}(a, b)$ over $GF(2)$, Ishai *et al.* proposed the following algorithm:

1. For every $0 \leq i < j \leq d$, select a random bit $r_{i,j} \xleftarrow{\$} \{0, 1\}$
2. For every $0 \leq i < j \leq d$, compute $r_{j,i} = (r_{i,j} \oplus a_i b_j) \oplus a_j b_i$
3. For every $0 \leq i \leq d$, compute $c_i = a_i b_i \oplus (\bigoplus_{j=0, j\neq i}^{d} r_{i,j})$

At the end of this algorithm, the `and` operation has been securely computed:

$$\texttt{and}(a, b) = ab = c = \bigoplus_{i=0}^{d} c_i = \left(\bigoplus_{i=0}^{d} a_i\right)\left(\bigoplus_{i=0}^{d} b_i\right)$$

This algorithm has been used by Rivain and Prouff as a basis for their secure multiplication algorithm over $\mathbb{F}_{2^n}$, that works for any n at any order d.

2.5.3 Rivain and Prouff masking scheme

In 2010, Rivain and Prouff came with a new idea to compute the masked computation of the AES in [24]. The masking can be separated in two groups: the linear and the non-linear operations. Masking a linear operation is trivial: masking $g(x)$ for g linear in $\mathbb{F}_{2^n}$ and $x = x_0 \oplus x_1 \oplus ... \oplus x_d$ consists of masking each share x_i separately:

$$\mathsf{Masking}\{g(x)\} = \mathsf{Masking}\{g(x_0)\} \oplus \mathsf{Masking}\{g(x_1)\} \oplus ...\mathsf{Masking}\{g(x_d)\}$$

Masking the square over $\mathbb{F}_{2^8}$

Since the squaring is a linear operation on any field of characteristic 2, we have that

$$x^2 = x_0^2 \oplus x_1^2 \oplus ... \oplus x_d^2$$

It follows that to securely compute a squaring operation, one can securely compute the squaring of each share separately.

Masking the field multiplication

To securely compute the field multiplication over any field of characteristic 2, Rivain and Prouff created the Algorithm 2, inspired by the ISW algorithm.

Algorithm 2 SecMult

Inputs: shares a_i and b_i such that $\bigoplus_{i=0}^{d} a_i = a$ and $\bigoplus_{i=0}^{d} b_i = b$
Outputs: shares c_i such that $c = \bigoplus_{i=0}^{d} c_i = a \odot b$

1: **for** $i = 0$ **to** d **do**
2: **for** $j = 1 + i$ **to** d **do**
3: $r_{i,j} \xleftarrow{\$} \mathbb{F}_{2^8}$
4: $r_{j,i} \leftarrow (r_{i,j} \oplus a_i \odot b_j) \oplus a_j \odot b_i$
5: **for** $i = 0$ **to** d **do**
6: $c_i \leftarrow a_i \odot b_i \oplus (\bigoplus_{j=0, j\neq i}^{d} r_{i,j})$
return $(c_0, c_1, \cdots, c_d)$

In a practical implementation, the basic field multiplication $a \odot b$ can be implemented as three look-ups into the two tables Log and AntiLog. But this operation must be implemented with a special care since it must be constant-time to avoid any type of timing attack. A secure implementation of the field multiplication is given in the appendix of [18] and is given in the algorithm 3. It uses the extended AntiLog table (denoted $\overline{alog}$, which is the AntiLog table extended at 255):

$$\overline{alog}(x) = \begin{cases} alog(x) & \text{if } 0 \leq x < 255 \\ alog(0) & \text{if } x = 255 \end{cases}$$

Masking the exponentiation to the 254^{th} power over $\mathbb{F}_{2^8}$

The exponentiation to the 254^{th} power in $\mathbb{F}_{2^8}$ is securely done by SecExp254 given in the Algorithm 4. We use this exponentiation because for any $x \in GF(2^8)$, we have that $x^{256} = x$ leading to $x^{255} = 1$ and $x^{254} = x^{-1}$.

The RefreshMasks routine is given in the Algorithm 5.

Algorithm 3 SPA-secure Galois Field Multiplication GFMult

Inputs: $a, b \in \mathbb{F}_{2^8}$, two tables log and $\overline{alog}$ based on the irreducible polynomial p over $GF(2^8)$ such that $\mathbb{F}_{2^8} = \mathbb{F}[x]/p(x)$
Output: $c = a \odot b \in \mathbb{F}_{2^8}$

$t = log(a)$
$s = (t + log(b)) \bmod 2^8$
$r = \overline{alog}((s < t) + s)$
return $(a\&\&b) \cdot r$

Algorithm 4 SecExp254

Inputs: shares x_i such that $\bigoplus_{i=0}^{d} x_i = x$
Outputs: shares y_i such that $y = \bigoplus_{i=0}^{d} y_i = x^{254}$

for $i = 0$ **to** d **do** $z_i \leftarrow x_i^2$
$(z_0, z_1, \cdots, z_d) \leftarrow \mathsf{RefreshMasks}(z_0, z_1, \cdots, z_d)$
$(y_0, y_1, \cdots, y_d) \leftarrow \mathsf{SecMult}((z_0, z_1, \cdots, z_d), (x_0, x_1, \cdots, x_d))$
for $i = 0$ **to** d **do** $w_i \leftarrow y_i^4$
$(w_0, w_1, \cdots, w_d) \leftarrow \mathsf{RefreshMasks}(w_0, w_1, \cdots, w_d)$
$(y_0, y_1, \cdots, y_d) \leftarrow \mathsf{SecMult}((y_0, y_1, \cdots, y_d), (w_0, w_1, \cdots, w_d))$
for $i = 0$ **to** d **do** $y_i \leftarrow y_i^{16}$
$(y_0, y_1, \cdots, y_d) \leftarrow \mathsf{SecMult}((y_0, y_1, \cdots, y_d), (w_0, w_1, \cdots, w_d))$
$(y_0, y_1, \cdots, y_d) \leftarrow \mathsf{SecMult}((y_0, y_1, \cdots, y_d), (z_0, z_1, \cdots, z_d))$
return $(y_0, y_1, \cdots, y_d)$

Algorithm 5 RefreshMasks

Inputs: shares x_i such that $\bigoplus_{i=0}^{d} x_i = x$
Outputs: shares x_i such that $\bigoplus_{i=0}^{d} x_i = x$

for $i = 1$ **to** d **do**
    $t \xleftarrow{\$} \mathbb{F}_{2^8}$
    $x_0 \leftarrow x_0 \oplus t$
    $x_i \leftarrow x_i \oplus t$
return $(x_0, x_1, \cdots, x_d)$

Masking the AES S-box

The computation of the SubBytes operation is the application of the S-box to every byte of the state. This can be viewed as an affine operation on x^{-1} in $\mathbb{F}_{2^8}$ where x^{-1} can be computed as the exponentiation to the 254^{th} power in $\mathbb{F}_{2^8}$. The affine transformation A that we apply has an additive part that is equal to 0x63 and a multiplicative part M. We denote the inverse function in $\mathbb{F}_{2^8}$ as f.

$$\text{S-box}(x) = Af(x) = A(x^{-1}) = M(x^{-1}) \oplus \texttt{0x63} = M(x^{254}) \oplus \texttt{0x63}$$

The multiplicative transformation M is defined as:

$$M : \mathbb{F}_{2^8} \mapsto \mathbb{F}_{2^8} : x \mapsto M(x) = (x << 1 | x >> 7) \oplus (x << 2 | x >> 6) \oplus (x << 3 | x >> 5) \oplus (x << 4 | x >> 4)$$

where $<<$ (resp. $>>$) represents the logical shift left (resp. the logical shift right) operation and $|$ represents the bitwise `or` operation. This operation can easily be tabulated.

Masking the affine transformation A is straightforward since it is a linear operation. To mask the exponentiation to the 254^{th} power, we use the algorithm SecExp254 defined above. If d is even, $d+1$ is odd and $Af(x) = Mf(x_0) \oplus \texttt{0x63} \oplus Mf(x_1) \oplus \texttt{0x63} \oplus \cdots \oplus Mf(x_d) \oplus \texttt{0x63} = Mf(x_0) \oplus Mf(x_1) \oplus \cdots \oplus Mf(x_d) \oplus \texttt{0x63} = Mf(x) \oplus \texttt{0x63}$ which is correct. On the other hand, if d is odd then $d+1$ is even, $Af(x) = Mf(x)$ and the additive part is missing. In this case we add it to the first share x_0. The whole algorithm is summarized here in the Algorithm 6.

Algorithm 6 SecSbox by Rivain and Prouff

Inputs: shares x_i such that $\bigoplus_{i=0}^{d} x_i = x$
Outputs: shares y_i such that $\bigoplus_{i=0}^{d} y_i = y = \text{S-box}(x)$

1: $(y_0, y_1, \cdots, y_d) \leftarrow \text{SecExp254}(x_0, x_1, \cdots, x_d)$
2: **for** $i = 0$ **to** d **do** $y_i \leftarrow M(x_i)$
3: **if** $d \bmod 2 == 1$ **then** $y_0 \leftarrow y_0 \oplus$ 0x63
return $(y_0, y_1, \cdots, y_d)$

2.5.4 Coron et al. masking scheme

In [7], Coron *et al.* found a flaw in the Rivain-Prouff SecSbox scheme, especially in SecExp254. They show that even if the Algorithms 2 and 5 are both SCA-secure to the d^{th} order when considered separately, their sequential application as done in the Algorithm 4 is not. They showed that there is a flaw of order $\lceil d/2 \rceil + 1$ in the scheme. To fix this, they changed the lines 2-3 and 5-6 of the Algorithm 4. Those lines were doing the refreshing of the masks and the multiplication in $\mathbb{F}_{2^8}$ of a number $x \in \mathbb{F}_{2^8}$ and a linear function $g(x)$, such as the square or the fourth power. They proposed two secure implementations of the multiplication of x by a $\mathbb{F}_2$-linear function $g(x)$. We will be interested in the second one that is using look-up tables to compute the squaring and the fourth power in $\mathbb{F}_{2^8}$ for time efficiency.

We define two functions SecPow3 and SecPow5 defined as:

$$\text{SecPow3} : \mathbb{F}_{2^8} \mapsto \mathbb{F}_{2^8} : x \mapsto x \odot x^2 = x^3$$

$$\text{SecPow5} : \mathbb{F}_{2^8} \mapsto \mathbb{F}_{2^8} : x \mapsto x \odot x^4 = x^5$$

Those functions are shown in the Algorithm 7 where $n = 3$ and $n = 5$ respectively.

The new SecExp254 algorithm is given in the Algorithm 8. It can be used in the Algorithm 6 to implement a new version of the SecSbox algorithm secure at order d.

Algorithm 7 SecPown

Inputs: shares a_i such that $\bigoplus_{i=0}^{d} a_i = a$ and a look-up table h such that $h(a) = a^n \in \mathbb{F}_{2^8}$
Outputs: shares c_i such that $\bigoplus_{i=0}^{d} c_i = c = a^n$ in $\mathbb{F}_{2^8}$.

1: **for** $i = 0$ **to** d **do**
2: **for** $j = i + 1$ **to** d **do**
3: $r_{i,j} \xleftarrow{\$} \mathbb{F}_{2^8}$
4: $r'_{i,j} \xleftarrow{\$} \mathbb{F}_{2^8}$
5: $r_{j,i} \leftarrow r_{i,j} \oplus h(a_i \oplus r'_{i,j}) \oplus h(a_j \oplus r'_{i,j}) \oplus h((a_i \oplus r'_{i,j}) \oplus a_j) \oplus h(r'_{i,j})$
6: **for** $i = 0$ **to** d **do**
7: $c_i \leftarrow h(a_i) \oplus (\bigoplus_{j=0, j\neq i}^{d} r_{i,j})$
return $(c_0, c_1, \cdots, c_d)$

Remark: The operations have to be operated from left to right, with the brackets first to ensure security.

Algorithm 8 SecExp254 by Coron *et al.*

Inputs: shares x_i such that $\bigoplus_{i=0}^{d} x_i = x$, two look-up tables h_2 and h_4 with the squares and the fourth power in $\mathbb{F}_{2^8}$ respectively
Outputs: shares y_i such that $y = \bigoplus_{i=0}^{d} y_i = x^{254}$

1: **for** $i = 0$ **to** d **do** $z_i \leftarrow x_i^2$
2: $(y_0, y_1, \cdots, y_d) \leftarrow \mathsf{SecPow3}(x_0, x_1, \cdots, x_d)$
3: **for** $i = 0$ **to** d **do** $w_i \leftarrow y_i^4$
4: $(y_0, y_1, \cdots, y_d) \leftarrow \mathsf{SecPow5}(y_0, y_1, \cdots, y_d)$
5: **for** $i = 0$ **to** d **do** $y_i \leftarrow y_i^{16}$
6: $(y_0, y_1, \cdots, y_d) \leftarrow \mathsf{SecMult}((y_0, y_1, \cdots, y_d), (w_0, w_1, \cdots, w_d))$
7: $(y_0, y_1, \cdots, y_d) \leftarrow \mathsf{SecMult}((y_0, y_1, \cdots, y_d), (z_0, z_1, \cdots, z_d))$
return $(y_0, y_1, \cdots, y_d)$

2.5.5 Rivain and Prouff scheme with a quadratic refresh masks procedure

The flaw in the RP scheme comes from the successive application of the refresh masks and secure multiplication procedures. Even if this flaw exists, it is in practice very hard to exploit to attack the scheme. A first way to correct that flaw is to change the way the algorithm works, as done in the Coron scheme described in Section 2.5.4. Another way to correct the flaw is to use another refresh masks procedure that is non-linear with the order. Indeed, the flaw in the scheme is induced because of the linearity of the number of random bytes used with the order. For $d = 1$, there is one generated random byte, for $d = d^*$, there is d^* random bytes generated. The correction that we use in this problem is to use the ISW refresh masks procedure described in [15] that is quadratic with the masking order d and whose implementation is given in the Algorithm 9.

Algorithm 9 ISW RefreshMasks

Inputs: shares x_i such that $\bigoplus_{i=0}^{d} x_i = x$
Outputs: shares x_i such that $\bigoplus_{i=0}^{d} x_i = x$

1: **for** $i = 0$ **to** d **do**
2: **for** $j = i + 1$ **to** d **do** $r_{i,j} \xleftarrow{\$} \mathbb{F}_{2^8}$
3: **for** $i = 0$ **to** d **do** $x_i \leftarrow x_i \oplus \bigoplus_{j=0, j\neq i}^{d} r_{i,j}$
return $(x_0, x_1, \cdots, x_d)$

In this implementation of the refresh masks, the number of random bytes used with respect to d is quadratic. There is

$$\sum_{i=0}^{d-1} d - i \approx \frac{d^2 + d}{2}$$

random bytes needed for the order d. The rest of the procedure is exactly the same than the classical RP scheme. For $d = 1$, the RP refresh mask procedure is the same as the ISW one, which means that the two schemes will show difference only for $d > 1$.

2.5.6 Genelle et al. masking scheme

In [12], Genelle *et al.* proposed a different way to compute the SecSbox algorithm at the order d. Actually, they proposed a way to secure any function of the form $\lambda' \circ \gamma \circ \lambda(x \oplus k)$ but we will only be interested in the AES here, that is: $\lambda(x) = x$, $\gamma(x) = x^{-1}$ and $\lambda'(x) =$ MixColumns(ShiftRows($M(x) \oplus$ 0x63)). The main idea is to switch from additive sharing to multiplicative sharing when doing the inversion in $GF(2^8)$ (i.e. the exponentiation to the power 254) in order to only have linear operations with respect to the shares. Indeed, while most of the S-box is linear w.r.t. additive shares, the inversion operation in $GF(2^8)$ is linear w.r.t. multiplicative shares. To switch between the types of sharing, two new functions are introduced: AMtoMM and MMtoAM.

The first one, AMtoMM takes a set $(x_0, x_1, \cdots, x_d)$ of additives shares such that $x = x_0 \oplus x_1 \oplus \cdots \oplus x_d$ and changes it into a set $(z_0, z_1, \cdots, z_d)$ of multiplicative shares such that $z_0 \odot z_1^{-1} \odot \cdots \odot z_d^{-1} = x$. This function is shown in the Algorithm 10.

The second function, MMtoAM takes as input a set of multiplicative shares $(z_0, z_1, \cdots, z_d)$ such that $z = z_0 \odot z_1^{-1} \odot \cdots \odot z_d^{-1}$ and transforms it into a set of additive shares $(x_0, x_1, \cdots, x_d)$ such that $x_0 \oplus x_1 \oplus \cdots \oplus x_d = z$. This function is shown in the algorithm 11.

The scheme uses the computation of Dirac function $\delta_0(x) = \begin{cases} 1 \text{ if } x = 0 \\ 0 \text{ if } x \neq 0 \end{cases}$ to map the elements of $GF(2^8)$ to $GF(2^8)^\times$ (i.e. the set of the invertible elements of $GF(2^8)$ which is

Algorithm 10 Secure AMtoMM

Inputs: additives shares $(x_0, x_1, \cdots, x_d)$ such that $\bigoplus_{i=0}^{d} x_i = x$
Outputs: multiplicative shares $(z_0, z_1, \cdots, z_d)$ such that $z_0 \odot \bigodot_{i=1}^{d} z_i^{-1} = x$

$z_0 \leftarrow x_0$
for $i = 1$ **to** d **do**
  $z_i \xleftarrow{\$} \mathbb{F}_{2^8}^{\times}$
  $z_0 \leftarrow z_0 \odot z_i$
  for $j = 1$ **to** $d - i$ **do**
    $r \xleftarrow{\$} \mathbb{F}_{2^8}$
    $x_j \leftarrow z_i \odot x_j$
    $x_j \leftarrow x_j \oplus r$
    $z_0 \leftarrow z_0 \oplus x_j$
    $x_j \leftarrow r$
  $x_{d-i+1} \leftarrow z_i \odot x_{d-i+1}$
  $z_0 \leftarrow z_0 \oplus x_{d-i+1}$
return $(z_0, z_1, \cdots, z_d)$

Algorithm 11 Secure MMtoAM

Inputs: multiplicative shares $(z_0, z_1, \cdots, z_d)$ such that $z_0 \odot \bigodot_{i=1}^{d} z_i^{-1} = z$
Outputs: additives shares $(x_0, x_1, \cdots, x_d)$ such that $\bigoplus_{i=0}^{d} x_i = z$

$x_0 \leftarrow z_0$
$(z_1, z_2, \cdots, z_d) \leftarrow (z_1^{-1}, z_2^{-1}, \cdots z_d^{-1})$
for $i = 1$ **to** d **do**
  $x_i \xleftarrow{\$} \mathbb{F}_{2^8}$
  $x_0 \leftarrow x_0 \oplus x_i$
  $x_0 \leftarrow x_0 \odot z_i$
  for $j = 1$ **to** i **do**
    $x_j \leftarrow x_j \odot z_i$
    $r \xleftarrow{\$} \mathbb{F}_{2^8}$
    $x_j \leftarrow x_j \oplus r$
    $x_0 \leftarrow x_0 \oplus x_j$
    $x_j \leftarrow r$
return $(x_0, x_1, \cdots, xs_d)$

$GF(2^8)\backslash\{0\}$). It replaces the zero values by one, since 0 has no inverse in $GF(2^8)$. It is needed for the multiplicative shares.

Algorithm 12 SecDirac

Inputs: $(d+1)$ (8×8)-bit matrices of additives shares $(\mathbf{s}^0, \mathbf{s}^1, \cdots, \mathbf{s}^d)$ such that $\bigoplus_{i=0}^{d} \mathbf{s}^i = \mathbf{s}$ whose lines are denoted s_i
Outputs: additives shares $(\Delta^0, \Delta^1, \cdots, \Delta^d)$ such that $\bigoplus_{i=0}^{d} \Delta^i = \Delta = \delta_0(s_0)||\delta_0(s_1)||\cdots||\delta_0(s_7)$

$\mathbf{s}^0 \leftarrow \overline{\mathbf{s}}^0$
for $i = 0$ **to** d **do**
    $\mathbf{t}^i \leftarrow (\mathbf{s}^i)^\intercal$
$(\Delta^0, \Delta^1, \cdots, \Delta^d) \leftarrow (t_0^0, t_0^1, \cdots, t_0^d)$
for $i = 1$ **to** 7 **do**
    $(\Delta^0, \Delta^1, \cdots, \Delta^d) \leftarrow \mathsf{SecAnd}\left((\Delta^0, \Delta^1, \cdots, \Delta^d), (t_0^0, t_0^1, \cdots, t_0^d)\right)$
return $(\Delta^0, \Delta^1, \cdots, \Delta^d)$

In [11], they proposed an implementation of SecDirac based on matrices of bits. This implementation is given in the algorithm 12. It works on $(\ell \times \ell)$-bits matrices. In our case, we set $\ell = 8$ since we are working on a 8-bit MCU. It means that we have to apply the algorithm $2(d+1)$ times to cover all the shares, i.e. twice per share, once for the bytes 1 to 8 and once for the bytes 9 to 16. This operation needs the computation of transpositions of matrices of bits and the computation of secure bitwise `and`, this is the function SecAnd that is given in the Algorithm 13. This function is implemented using the ISW scheme, in a similar way than SecMult.

Algorithm 13 SecAnd

Inputs: shares a_i and b_i such that $\bigoplus_{i=0}^{d} a_i = a$ and $\bigoplus_{i=0}^{d} b_i = b$
Outputs: shares c_i such that $c = \bigoplus_{i=0}^{d} c_i = ab$

for $i = 0$ **to** d **do**
    for $j = 1 + i$ **to** d **do**
        $r_{i,j} \xleftarrow{\$} \mathbb{F}_{2^8}$
        $r_{j,i} \leftarrow (r_{i,j} \oplus a_i b_j) \oplus a_j b_i$
for $i = 0$ **to** d **do**
    $c_i \leftarrow a_i b_i \oplus (\bigoplus_{j=0, j\neq i}^{d} r_{i,j})$
return $(c_0, c_1, \cdots, c_d)$

This implementation returns a set of $d+1$ binary values $\{\Delta^i\}_{i=0}^{d}$ that are xored to the values of the state to avoid having a zero value in the multiplicative part of the algorithm. The complete routine is described in Algorithm14, where the function $M(\cdot)$ is the affine transformation of the AES S-box, already described in Section 2.5.3.

The inverse in $\mathbb{F}_{2^8}$ can be computed using the Log/AntiLog tables, in a similar way than the field multiplication (see Algorithm 15). The easiest and fastest way to compute it is to use a look-up table containing all the inverses in $GF(2^8)$.

The different operations on single bits instead of 8-bit registers used in the scheme along with the transpositions of matrices of bits are not efficient in a 8-bit architecture. This induces a large overhead but in practice, this overhead is attenuated by the fact than once the Dirac values are computed, the inverse part of the S-box becomes nearly free since it is reduces to a look-up in a table.

Algorithm 14 SecSbox from Genelle *et al.*

Inputs: additives shares $(x_0, x_1, \cdots, x_d)$ such that $\bigoplus_{i=0}^{d} x_i = x$, the Dirac values $(\Delta_j^0, \Delta_j^1, \cdots, \Delta_j^d)$ where j is the position of the word x in the state $\mathbf{s}$ and Δ_j is the j^{th} bit of Δ

Outputs: additive shares $(x_0, x_1, \cdots, x_d)$ such that $\bigoplus_{i=0}^{d} x_i = \mathsf{S\text{-}box}(x)$

1: $(\Delta_j^0, \Delta_j^1, \cdots, \Delta_j^d) \leftarrow \mathsf{SecDirac}(\mathbf{s}^0, \mathbf{s}^1, \cdots, \mathbf{s}^d)$
2: $(x_0, x_1, \cdots, x_d) \leftarrow (x_0, x_1, \cdots, x_d) \oplus (\Delta_j^0, \Delta_j^1, \cdots, \Delta_j^d)$
3: $(z_0, z_1, \cdots, z_d) \leftarrow \mathsf{AMtoMM}(x_0, x_1, \cdots, x_d)$
4: $(z_0, z_1, \cdots, z_d) \leftarrow (z_0^{-1}, z_1^{-1}, \cdots, z_d^{-1})$
5: $(x_0, x_1, \cdots, x_d) \leftarrow \mathsf{MMtoAM}(z_0, z_1, \cdots, z_d)$
6: **for** $i = 0$ **to** d **do**
7: $\quad x_i \leftarrow x_i \oplus \Delta_j^i$
8: $\quad x_i \leftarrow M(x_i) \oplus \mathtt{0x63}$
return $(x_0, x_1, \cdots, x_d)$

Algorithm 15 SPA-secure Galois Field Inversion GFinv

Input: $a \in \mathbb{F}_{2^8}$, two tables log and $\overline{alog}$ based on the irreducible polynomial p over $GF(2^8)$ such that $\mathbb{F}_{2^8} = \mathbb{F}[x]/p(x)$
Output: $b = a^{-1} \in \mathbb{F}_{2^8}$

1: $t = log(a)$
2: $r = \overline{alog}(255 - t)$
return $(a \texttt{ != } 0) \cdot r$

2.5.7 Masking the whole AES cipher

As recalled in the section 2.1.4, all the rounds of the AES are composed of 4 stages: AddRoundKey, SubBytes, ShiftRows and MixColumns, except for the last one which does not have the MixColumns operation. The subkeys used by the AddRoundKey operation are priorly computed by the KeyExpension algorithm.

To mask an AES computation at the order d, the initial state $\mathbf{s}_0$ holding the plaintext p must be splitten in $d+1$ states $\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_d$. The states $(\mathbf{s}_i)_{i=1}^{d}$ are randomized and the state $\mathbf{s}_0$ is chosen such that the addition of all the shares leads to the correct initial state.

1. For each $1 \leq i \leq d$, $\mathbf{s}_i \xleftarrow{\$} \{0,1\}^{128}$
2. Compute $\mathbf{s}_0 = \mathbf{s} \oplus \bigoplus_{i=1}^{d} \mathbf{s}_i$

The masker key $\mathbf{k}$ must also be splitted into shares in the same fashion:

1. For each $1 \leq i \leq d$, $\mathbf{k}_i \xleftarrow{\$} \{0,1\}^{128}$
2. Compute $\mathbf{k}_0 = \mathbf{k} \oplus \bigoplus_{i=1}^{d} \mathbf{k}_i$

Masking the KeyExpansion algorithm

The RotWord operation is linear in any field of characteristic 2, so the RotWord operation is applied on each share. The SubWord operation is non-linear and must use the SecSbox algorithm implemented by Rivain-Prouff, Coron *et al.* or Genelle *et al.* Finally, Rcon must be added to one share. The whole d^{th}-order secure AES-128 key expansion is described in Algorithm 16.

Algorithm 16 SecKeyExpansion

Inputs: shares $\mathbf{k}_i$ such that $\bigoplus_{i=0}^{d} \mathbf{k}_i = \mathbf{k}$
Outputs: shares $(\mathbf{k}_i^1, \mathbf{k}_i^2, \cdots, \mathbf{k}_i^{10})_{i=0}^{d}$ such that $\bigoplus_{i=0}^{d} \mathbf{k}_i^r = \mathbf{k}^r$

1: **for** $j = 1$ **to** 4 **do**
2: **for** $i = 0$ **to** d **do** $(\mathbf{w}_i)_{*,j} \leftarrow (\mathbf{k}_i)_{*,j}$
3: **for** $j = 5$ **to** 44 **do**
4: **for** $i = 0$ **to** d **do** $t_i \leftarrow (\mathbf{w}_i)_{*,j-1}$
5: **if** $j \bmod 4 == 0$ **then**
6: **for** $l = 1$ **to** 4 **do** $((t_0)_l, (t_1)_l, \cdots, (t_d)_l) \leftarrow \mathsf{SecSbox}((t_0)_l, (t_1)_l, \cdots, (t_d)_l)$
7: **for** $i = 0$ **to** d **do** $t_i \leftarrow \mathsf{RotWord}(t_i)$
8: $t_0 \leftarrow t_0 \oplus \mathsf{Rcon}_{j/4}$
9: **for** $i = 0$ **to** d **do** $(\mathbf{w}_i)_{*,j} \leftarrow (\mathbf{w}_i)_{*,j-4} \oplus t_i$
10: **for** $r = 1$ **to** 10 **do**
11: **for** $i = 0$ **to** d **do** $\mathbf{k}_i^r \leftarrow \Big((\mathbf{w}_i)_{*,4r+1} \quad (\mathbf{w}_i)_{*,4r+2} \quad (\mathbf{w}_i)_{*,4r+3} \quad (\mathbf{w}_i)_{*,4r+4}\Big)$
return $(\mathbf{k}_i^1, \mathbf{k}_i^2, \cdots, \mathbf{k}_i^{10})_{i=0}^{d}$

Masking the SubBytes layer

To mask the SubBytes transformation, we apply the SecSbox algorithm to every $d+1$-tuples $((\mathbf{s}_0)_{l,j}, (\mathbf{s}_1)_{l,j}, \cdots, (\mathbf{s}_d)_{l,j})$ for every row-coordinate l and every column-coordinate j such that $1 \leq l, j \leq 4$. The SecSbox operation can be implemented by any scheme presented earlier: Rivain-Prouff, Coron *et al.* or Genelle *et al.*.

Masking the ShiftRows layer

The ShiftRows operation is linear in $\mathbb{F}_{2^8}$, so we apply it to each share separately:

$$\mathsf{ShiftRows}(\mathbf{s}) = \bigoplus_{i=0}^{d} \mathsf{ShiftRows}(\mathbf{s}_i)$$

Masking the MixColumns layer

The MixColumns operation is also linear in $\mathbb{F}_{2^8}$, so we apply it to each share separately:

$$\mathsf{MixColumns}(\mathbf{s}) = \bigoplus_{i=0}^{d} \mathsf{MixColumns}(\mathbf{s}_i)$$

Masking the AddRoundKey layer

The masked AddRoundKey layer at round r consists in adding to each each states $(\mathbf{s}_i)_{i=0}^{d}$ the shared round key $\mathbf{k}_i^r$ generated by the SecKeyExpansion algorithm given in the algorithm 16.

$$\mathbf{s} \oplus \mathbf{k}^r = (\mathbf{s}_0 \oplus \mathbf{k}_0^r) \oplus (\mathbf{s}_1 \oplus \mathbf{k}_1^r) \oplus \cdots \oplus (\mathbf{s}_d \oplus \mathbf{k}_d^r)$$

Masking the whole AES

The masking of the whole AES is summarized in the algorithm 17.

Algorithm 17 Complete masked AES

Inputs: a plaintext p, a master key k
Output: a ciphertext c

1: $\mathbf{s}_0 \leftarrow p$
2: $\mathbf{k}_0 \leftarrow k$
3: **for** $i = 1$ **to** d **do**
4: $\quad \mathbf{s}_i \xleftarrow{\$} \{0,1\}^{128}$
5: $\quad \mathbf{s}_0 \leftarrow \mathbf{s}_0 \oplus \mathbf{s}_i$
6: $\quad \mathbf{k}_i \xleftarrow{\$} \{0,1\}^{128}$
7: $\quad \mathbf{k}_0 \leftarrow \mathbf{k}_0 \oplus \mathbf{k}_i$
8: $\begin{pmatrix} \mathbf{k}_0^1 & \mathbf{k}_0^2 & \cdots & \mathbf{k}_0^{N_r} \\ \mathbf{k}_1^1 & \mathbf{k}_1^2 & \cdots & \mathbf{k}_1^{N_r} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{k}_d^1 & \mathbf{k}_d^2 & \cdots & \mathbf{k}_d^{N_r} \end{pmatrix} \leftarrow \mathsf{SecKeyExpansion}(\mathbf{k}_0, \mathbf{k}_1, \ldots, \mathbf{k}_d)$
9: **for** $i = 0$ **to** d **do** $\mathbf{s}_i^1 \leftarrow \mathbf{s}_i \oplus \mathbf{k}_i$
10: **for** $r = 1$ **to** $N_r - 1$ **do**
11: $\quad$ **for** $l = 1, j = 1$ **to** 4 **do**
12: $\quad\quad ((\mathbf{s}_0)_{l,j}, (\mathbf{s}_1)_{l,j}, \cdots, (\mathbf{s}_d)_{l,j}) \leftarrow \mathsf{SecSbox}((\mathbf{s}_0^r)_{l,j}, (\mathbf{s}_1^r)_{l,j}, \cdots, (\mathbf{s}_d^r)_{l,j})$
13: $\quad$ **for** $i = 0$ **to** d **do**
14: $\quad\quad \mathbf{s}_i \leftarrow \mathsf{ShiftRows}(\mathbf{s}_i)$
15: $\quad\quad \mathbf{s}_i \leftarrow \mathsf{MixColumns}(\mathbf{s}_i)$
16: $\quad\quad \mathbf{s}_i^{r+1} \leftarrow \mathbf{s}_i \oplus \mathbf{k}_i^r$
17: **for** $l = 1, j = 1$ **to** 4 **do**
18: $\quad ((\mathbf{s}_0)_{l,j}, (\mathbf{s}_1)_{l,j}, \cdots, (\mathbf{s}_d)_{l,j}) \leftarrow \mathsf{SecSbox}((\mathbf{s}_0^{N_r})_{l,j}, (\mathbf{s}_1^{N_r})_{l,j}, \cdots, (\mathbf{s}_d^{N_r})_{l,j})$
19: **for** $i = 0$ **to** d **do**
20: $\quad \mathbf{s}_i \leftarrow \mathsf{ShiftRows}(\mathbf{s}_i)$
21: $\quad \mathbf{s}_i \leftarrow \mathbf{s}_i \oplus \mathbf{k}_i^{N_r}$
22: $c \leftarrow \bigoplus_{i=0}^{d} \mathbf{s}_i$
return c

Remark: There is no need to keep the different $\mathbf{s}_i^r$ in memory during the whole execution

2.6 Leakage detection methods

2.6.1 Introduction

We have seen that masking is a sound countermeasure to thwart PAAs against AES implementations. To ensure that these masking schemes are properly working, methods called *leakage detection methods* can be used against power traces recorded from the targeted design to evaluate whether sensitive data are leaking or not. We will present here two of those methods: the CRI's *fixed-vs-random t-test* and the *ρ-test*. There is an analogy between the DPA intuition and the t-test as well as between the CPA and the ρ-test, since they use the same distinguishers.

2.6.2 Welch's t-test as a detection leakage method

In the field of SCA, a well know and widely used leakage detection test is the CRI's *fixed-vs-random t-test* which is a Welch's t-test that is used between two classes of traces. It is an empirical test that checks whether the traces of the two different classes have different mean values for a given fixed key.

Let $\mathbf{T}$ be a set of n traces $\mathbf{T}_{i,*}$ of m samples each put row-wise in a matrix. Let $T_{i,j}$ denote the j^{th} sample point of the i^{th} trace in the set, $1 \leq i \leq n, 1 \leq j \leq m$. Let also $\mathbf{T}_{*,j}$ be the j^{th} column of $\mathbf{T}$.

$$\mathbf{T} = \begin{pmatrix} \mathbf{T}_{1,*} \\ \mathbf{T}_{2,*} \\ \vdots \\ \mathbf{T}_{n,*} \end{pmatrix} = \begin{pmatrix} T_{1,1} & T_{1,2} & \cdots & T_{1,m} \\ T_{2,1} & T_{2,2} & \cdots & T_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ T_{n,1} & T_{n,2} & \cdots & T_{n,m} \end{pmatrix} = \begin{pmatrix} \mathbf{T}_{*,1} & \mathbf{T}_{*,2} & \cdots & \mathbf{T}_{*,m} \end{pmatrix}$$

The Welch's t-test between two sets of traces $\mathbf{T}^a$ and $\mathbf{T}^b$ is computed as a vector $\Delta = (\Delta_1, \Delta_2, \cdots, \Delta_m)$ of all the different tests computed at each time sample j of the traces.

$$\Delta_j = \frac{\mu_j^a - \mu_j^b}{\sqrt{\frac{\sigma^{a2}_j}{n_a} + \frac{\sigma^{b2}_j}{n_b}}}$$

where μ_j^x and σ^{x2}_j are the mean value and the variance of the column vector $\mathbf{T}^x_{*,j}$ and where n_x is the number of traces in the set $\mathbf{T}^x$. In the set $\mathbf{T}^a$, we put traces of the encryptions of a fixed plaintext p with a fixed key k. In the set $\mathbf{T}^b$ we put traces of the encryptions of random plaintexts p_i with the same fixed key k.

We say that there are leakage at points where $|\Delta_j| \geq 5$. In [1], Balasch *et al.* explain how to find this threshold. The Welch's t-test between two classes determines whether the samples in the two classes belong to the same population, or more precisely if the two classes have the same mean. The null hypothesis $\mathcal{H}_0$ in that case is that the sample of both sets were drawn from population with the same mean. The other hypothesis $\mathcal{H}_1$ is that the samples were drawn from distributions with different means. In the context of masking, $\mathcal{H}_0$ corresponds to an efficient masking and $\mathcal{H}_1$ corresponds to a non-efficient since a good masking implementation should have a constant mean for every sensitive data over large populations. For large sample sizes, this threshold of ± 5 is nearly equivalent to a correct rejection of $\mathcal{H}_0$ with a probability of 99.999%. As explained in [10, 26], the fixed-vs-random t-test is an efficient way to perform leakage detection only but it is also unable to detect certain kind of leakages, e.g. if the two sets have an identical mean. To complete this test, the ρ-test is a good complement.

2.6.3 ρ-tests

In [10], Durvaux and Standaert introduced a new leakage detection method based on the CPA distinguisher.

Starting from a set $\mathbf{T}$ of n traces of m samples each, we begin by cutting this set of traces in K subsets $\mathcal{L}^{(j)}$, $1 \leq j \leq K$ of approximatively equal sizes in order to do a k-fold cross validation[4]. We then define the profiling sets as $\mathcal{L}_p^{(j)} = \bigcup_{i=1, i\neq j}^{K} \mathcal{L}^{(j)}$ and the test sets as $\mathcal{L}_t^{(j)} = \mathcal{L} \backslash \mathcal{L}_p^{(j)} = \mathcal{L}^{(j)}$. The number of traces in $\mathcal{L}_p^{(j)}$ is denoted n_p and the number of traces in $\mathcal{L}_t^{(j)}$ is denoted n_t, such that $n = n_p + n_t$. The ρ-test focuses a specific byte p_f of the plaintext, i.e. one of the 16, and is defined as follows:

$$\hat{r}^{(j)}(\tau) = \hat{\rho}\left(L_X^{(j)}(\tau), \hat{\mathsf{model}}_\tau^{(j)}(X)\right)$$

where $\hat{\rho}$ is the estimated Pearson's correlation coefficient, $L_X^{(j)}(\tau)$ is the set of the leakage traces in $\mathcal{L}_t^{(j)}$ at the timestep τ, X is the vector containing the corresponding focused plaintext bytes of the traces in L_X and $\hat{\mathsf{model}}_\tau^{(j)}(X)$ corresponds to the vector X passed through the model learned from the data in $\mathcal{L}_p^{(j)}$ for a given timestep τ.

The K cross-validation results can then be averaged in order to get a single unbiased correlation trace:

$$\hat{r}(\tau) = \frac{1}{K}\sum_{j=1}^{K} \hat{r}^{(j)}(\tau)$$

We then normalize the Fischer's z-transformation of $\hat{r}$ by the standard deviation $\frac{1}{\sqrt{n_t - 3}}$ to get results that can be interpreted as a normal distribution $\hat{\mathfrak{t}} \sim \mathcal{N}(0,1)$:

$$\hat{\mathfrak{t}}(\tau) = \frac{1}{2}\ln\left(\frac{1+\hat{r}(\tau)}{1-\hat{r}(\tau)}\right)\sqrt{n_t - 3}$$

Then, we know that according to our model, there is leakage at every time step τ where $\hat{\mathfrak{t}}(\tau) \geq 5$ or $\hat{\mathfrak{t}}(\tau) \leq -5$. According to [10], those thresholds are chosen similar to the ones used for the t-test, for similar reasons.

Different models can be used and the strength of the results of the test will be highly dependent on the pertinence of the model. We will here present two different models: the means and the Hamming weights.

Profiled ρ-test based on the means

We can use the mean values of the power traces for every timestep and for every possible value (from 0 to 255) of the focused plaintext byte p_f to construct our model. We use the profiling sets to obtain the matrices

$$\boldsymbol{\mu}^{(j)} = \begin{pmatrix} \mu_0^{(j)}(t_1) & \mu_0^{(j)}(t_2) & \cdots & \mu_0^{(j)}(t_m) \\ \mu_1^{(j)}(t_1) & \mu_1^{(j)}(t_2) & \cdots & \mu_1^{(j)}(t_m) \\ \vdots & \vdots & \ddots & \vdots \\ \mu_{255}^{(j)}(t_1) & \mu_{255}^{(j)}(t_2) & \cdots & \mu_{255}^{(j)}(t_m) \end{pmatrix}$$

for every $1 \leq j \leq K$ where $\mu_x^{(j)}(\tau)$ is the mean value at timestep τ of the traces in $\mathcal{L}_p^{(j)}$ corresponding to the encryption of a focused plaintext byte x.

Using this model, if $X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n_t} \end{pmatrix}$, then $\hat{\mathsf{model}}_\tau^{(j)}(X) = \begin{pmatrix} \mu_{x_1}^{(j)}(\tau) \\ \mu_{x_2}^{(j)}(\tau) \\ \vdots \\ \mu_{x_{n_t}}^{(j)}(\tau) \end{pmatrix}$.

[4] a validation method where each of the K combinations of $K-1$ sets are used to learn a model and the last one is used to validate it, i.e. compute the error of the model

ρ-test based on the Hamming weights

Another approach is the one using the Hamming weight at the output of the S-box. This model does not need a profiling step but requires the knowledge of the encryption key k, or at least the key byte k_f corresponding to the focused plaintext byte p_f. We construct the model, that is independent of the power consumption and of the time, for every value of the focused plaintext byte p_f (from 0 to 255) as:

$$\mathbf{h} = \begin{pmatrix} h_0 \\ h_1 \\ \vdots \\ h_{255} \end{pmatrix}$$

where $h_x = H(\mathsf{S\text{-}box}(x \oplus k_f))$.

Using this model, if $X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n_t} \end{pmatrix}$ then $\hat{\mathsf{model}}_{\tau}^{(j)}(X) = \begin{pmatrix} h_{x_1} \\ h_{x_2} \\ \vdots \\ h_{x_{n_t}} \end{pmatrix}$.

Chapter 3

ChipWhisperer-Lite

3.1 Introduction

In this chapter we present the hardware part of this thesis. The board on which we are working is called the ChipWhisperer-Lite. This board is developed and sold by NewAE Technology Inc, to help understand power analysis concepts. All the component aspects, the place and route, the power planning, the tape-out and the fabrication were already done. Moreover, it is low cost and came with some documentation. A first part of this thesis was to make this ChipWhisperer-Lite board plug-and-play for the students of the LELEC2760 course[1] and controllable from MATLAB so that they could run some cryptanalytic attacks to break an AES key. To do so, some functionalities have also been developed to give the students the possibility to insert faults or receive both the plaintext and the trace of an encryption. We begin this chapter by describing what is the ChipWhisperer-Lite, we also explain what has been done to make it operational for students and we give some implementation details.

3.2 What is the ChipWhisperer-Lite ?

The ChipWhisperer-Lite (CW-L) is a board developed by NewAE Technology Inc.[2], a Canadian company founded by Colin O'Flynn and Hilary Taylor. A picture of the board is available in FIGURE 3.1. This chip has been designed to contain everything needed for the security analysis of an 8-bit target embedded micro-controller: the `ATxMEGA128d4`. Both side-channel power analysis and fault attacks can be operated.

The CW-L contains two micro-controllers (the target and the SAM3U), a FPGA (Xilinx Spartan 6), a 10-bit ADC, a LNA, a micro-USB port and many other components. The CW-L board is composed of two parts that are connected by default but can be separated:

- the main board containing the FPGA, the ADC, the SAM3U micro-controller and the LNA.
- the CW303 target containing the target micro-controller.

Here is a brief overview of the functions of the different components:

- The SAM3U MCU is used to program the FPGA and to handle the communication between the FPGA and the computer through the USB port.
- The FPGA is the central piece of the board. It generates the clock, and is in charge of the communication with the target. It emulates a user of the target. It also starts acquiring data from the ADC when receiving a signal from the `ATxMEGA128d4`.

[1] `http://perso.uclouvain.be/fstandae/teaching.html`
[2] `https://newae.com/`

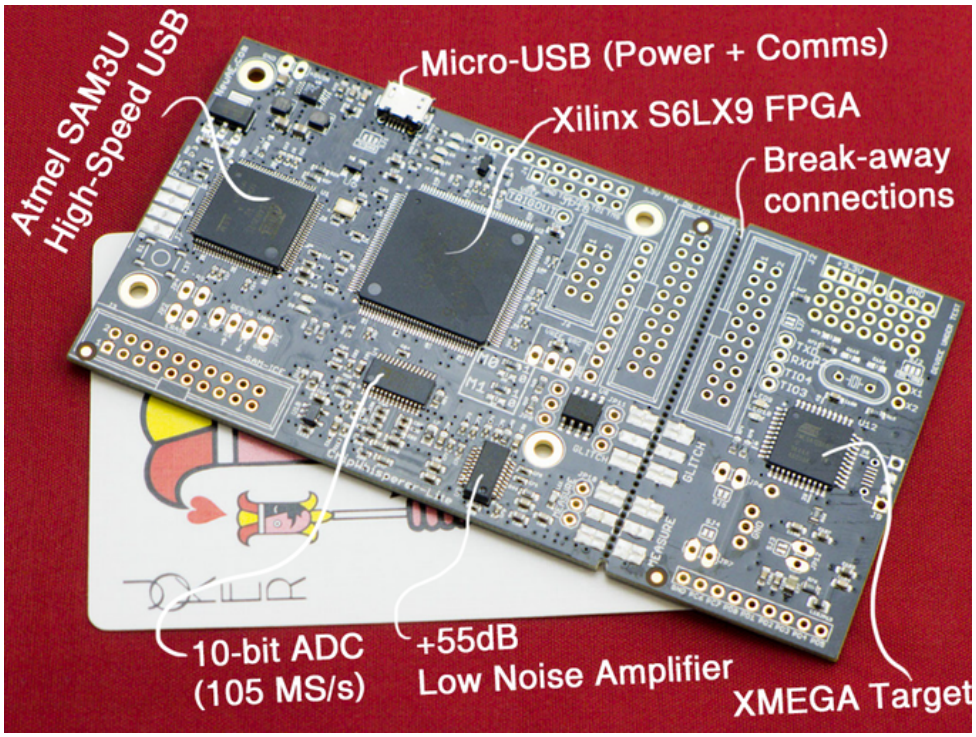


FIGURE 3.1 – The ChipWhisperer-Lite board (image from `https://www.kickstarter.com/projects/coflynn/chipwhisperer-lite-a-new-era-of-hardware-security`)

- The target MCU runs the encryption algorithm (the AES in this case). It sends a trigger to the FPGA to start recording its power consumption. Its default clock frequency is at 7.37MHz.

- The ADC samples the power consumption of the target on a small resistor of 49Ω. The ADC clock is 4 times faster than the target clock, which corresponds to 29.48MHz.

3.3 Make the system operational for students

The first goal was to make the whole system as easy to use as possible. The students should be able to plug the ChipWhisperer-Lite, install the driver and directly use it from MATLAB to launch attacks on implementations of their choice. By default, the CW-L is given with lots of scripts written in Python2, allowing to record power traces and to attack them, but not to dive in the implementation on the target. Thanks to Pierre Gérard, we got rid of all the Python interface necessary to use the board when we received it. He developed a dynamic link library (dll) containing all the functions that were needed by MATLAB to use the board using a computer running Windows. The main functions contained in this dll are as follows :

- $[s] \leftarrow \mathsf{MLWrite\_Serial}(\mathrm{ID}, \mathrm{Data})$
  Sends given data to the UART of the target processor and returns the status of the operation.

- $[s, d, Nb] \leftarrow \mathsf{MLRead\_Serial}(\mathrm{ID}, \mathrm{Timeout})$
  Reads serial data sent by the target processor, its size and the status of the operation. If the Timeout is reached, the function returns and Nb contains 0.

- $[s] \leftarrow$ MLArm_Acq(ID)
  Arms the acquisition of ADC data, and returns the status of the operation.
- $[s, l] \leftarrow$ MLRead_Adc_Data(ID)
  Gets the leakage trace back from the ADC and the status of the operation.
- MLWriteMaxSamples(ID, NSamples)
  Specifies the number of samples of the leakage trace. Maximum value is 24400.
- MLWriteOffset(ID, Offset)
  Specifies the offset in number of samples between the trigger received by the FPGA and the start of the acquisition.
- MLWriteGain(ID, Gain)
  Specifies the Gain applied to the leakage trace by the LNA.
- MLWritePhase(ID, Phase)
  Specifies the phase between the clock of the ADC and the power measurements, with a value between -255 and 255.
- $[s] \leftarrow$ MLDisableGlitch(ID)
  Disables the glitch functionality and returns the status of the operation.

It also contains functions to initialize the system, to program the FPGA and the target `ATxMEGA128d4`. These functions are encapsulated in the following MATLAB function:

- [ID] $\leftarrow$ init_function(DLL, HexFile)

that generates a handler for a given hex file and a link to the dll.

These functions are sufficient to obtain encryption results and power traces, but are not very user-friendly. We built three higher level functions with a specific purpose to improve this aspect:

- $[c, l] \leftarrow$ measure_AES(MyCw, p, k)
  Performs an encryption of the plaintext p with the key k. Returns a ciphertext c and a leakage trace l. Used for non-masked implementations of the AES.
- $[c, l] \leftarrow$ measure_masked_AES(MyCw, p, k, s)
  Performs an encryption of the plaintext p with the key k, with randomness generated based on a seed s. If s is not specified, it is randomly generated by MATLAB. Returns a ciphertext c and a leakage trace l.
- $[fc, l] \leftarrow$ glitch_AES(MyCw, $p, k, x, g_{show}, w, o, N$)
  Performs a faulty encryption of the plaintext p with the key k, from the N glitches at location x of width w with an offset o. The g_{show} parameter is a boolean value that is used to plot the results when enabled. Returns a faulty ciphertext fc and a leakage trace l. This function only needs a plaintext, a key and a glitch location, the other parameters have default values.

3.4 Simple Serial protocol

The Simple Serial protocol is the communication protocol used to communicate on the serial bus by the computer and the target chip. It is typically used to send the parameters of the AES to the target and to get the ciphertext back. It works as follows:

- The user sends k$KEY

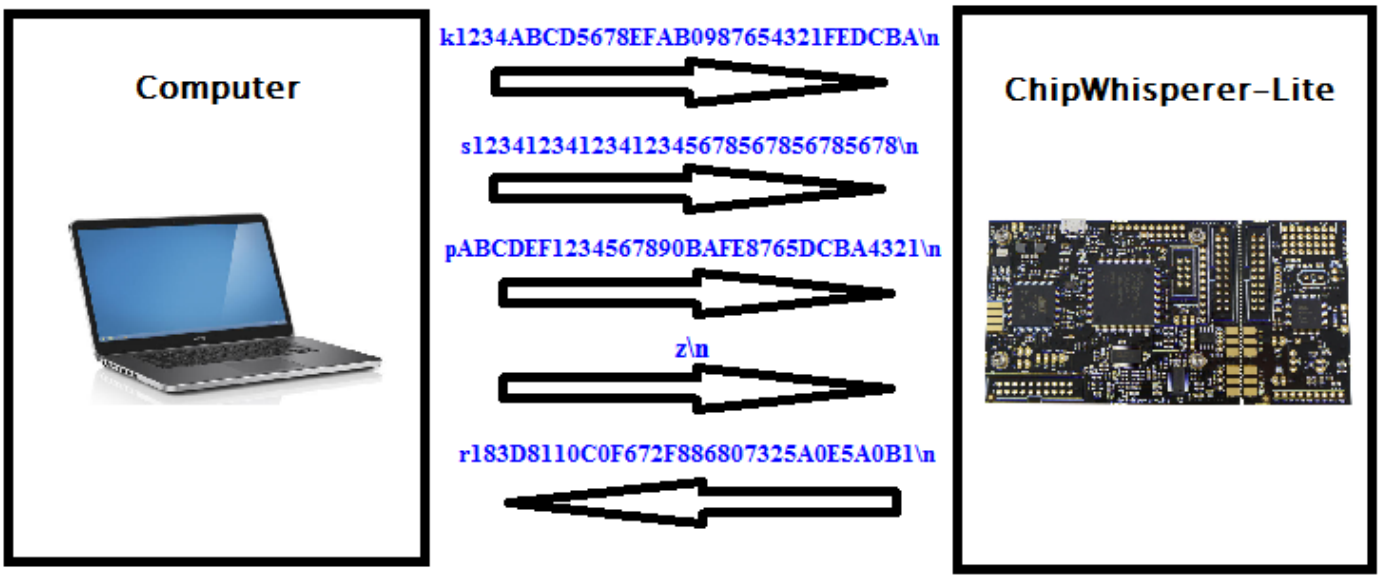


FIGURE 3.2 – Simple-serial AES protocol

Address	Content
0x3000→0x300F	Key
0x3010→0x301F	Plaintext
0x3020→0x302F	Seed
0x3030→0x303F	Ciphertext
0x3500→0x351F	ASCII buffer, containing an ASCII version of the plaintext/key/seed
0x3520→0x352F	Hexadecimal translation of the ASCII buffer
0x3550→0x356F	ASCII translation of the hexadecimal ciphertext, before the putch

TABLE 3.1 – Memory addresses used

- (optionally) The user sends s$SEED
- The user sends p$PLAINTEXT
- The user sends z
- The user receives from the target r$CIPHERTEXT.

where $KEY, $SEED, $PLAINTEXT and $CIPHERTEXT respectively are the 128-bit key, seed, plaintext and ciphertext written in a hexadecimal basis. An example of the protocol is given in FIGURE 3.2. The different parameters can be sent in any order.

3.5 ASM implementation and addresses used for the SimpleSerial Protocol

This section presents a detailed view of the implementation of how the chip handles the data transfer. To keep the granularity of the code, the handling of this communication is completely separated from the encryption itself. FIGURE 3.3 shows the full finite state machine of our implementation.

At the start of the micro-controller, there is a first initialization of the UART bus and of the trigger. Shortly after, a default plaintext and a default key are written in the memory at the

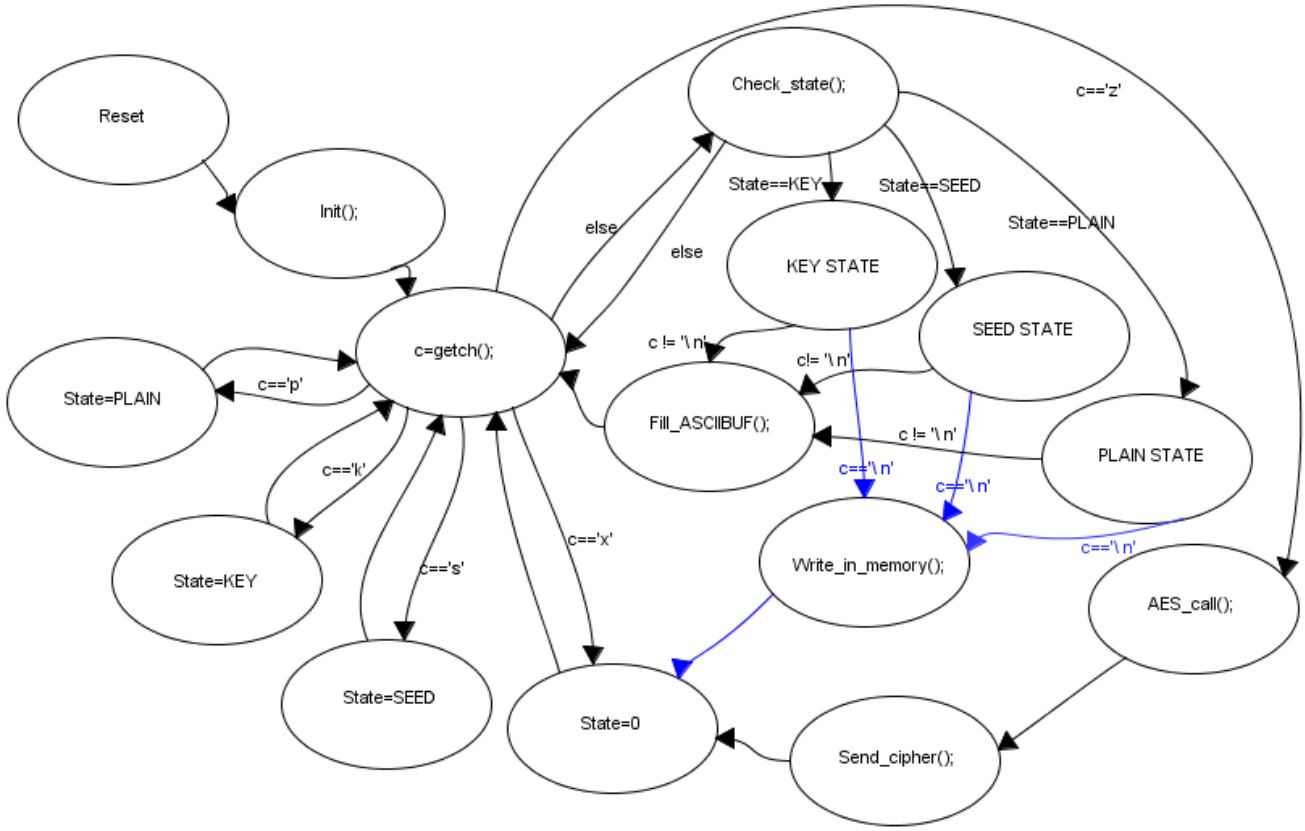


FIGURE 3.3 – Finite State Machine of the Simple-serial AES protocol

Register	Function
r0→16	Used only by the AES implementation
r17→18	Used by `hex_decode` and `hex_print`
r19	Not used
r20→21	Temporary values
r22	Current state
r23	Position in key/plaintext/seed when receiving on the bus
r24	Last handled character (last `getch` or next to `putch`)
r25	Used by `putch` to check the availability of the bus
r26→31	Memory pointers

TABLE 3.2 – Registers used

addresses given in TABLE 3.1. The chip then enters the FSM, and executes a call to `getch`. This is a blocking call, so it waits until receiving a character on the bus.

The data that arrives to the target comes from the FPGA and is encoded in ASCII. As mentioned in the previous section, the expected first characters are either a 'k', a 'p' or a 's' followed by 32 ASCII characters and a '\n', a 'z' or a 'x'. Receiving a 'x' means that the MCU must reset the FSM and wait on `getch`. Receiving a 'z' means that the MCU must start an encryption with the values contained at the memory addresses specified in TABLE 3.1. There is a default value for the key, the plaintext and the seed that will be used if a 'z' character is sent before all the other parameters. Receiving any of the three other characters starts a routine to replace the values contained in the memory.

This routine works as follows: the MCU jumps to the corresponding state and starts filling the ASCII buffer with the next 32 characters. These characters are encoded in ASCII but are supposed to be used as hexadecimal values by the AES, so they need to be translated. When receiving a '\n', this translation is performed by `hex_decode` whose output is then written at 0x3520→0x352F and transfered to the correct address. The previously contained key/plaintext/seed are erased by the arriving values.

At the end of the encryption, the hexadecimal ciphertext is written at 0x3030→0x303F. First, its translation in ASCII is performed by `hex_print`, whose output is written at 0x3550→0x356F. The ASCII characters are then fed to the `putch`, that sends them back to the bus, after adding a 'r' in front and a '\n' at the end.

The detailed use of all the registers can be found in TABLE 3.2.

Chapter 4

Power Analysis Attacks against an unmasked AES

4.1 Introduction

This chapter will first show the leakages that exist in an unprotected implementation of the AES. The basic attacks described in Section 2.4 will be applied to a real implementation of the AES on the 8-bit MCU target of the ChipWhisperer-Lite. The AES code running on the CPU is the *AES furious* written by B. Poettering which is interfaced by the Simple-Serial protocol. The attack model described here is *chosen-plaintext attacks*, which limits their use to restricted situations.

4.2 How much does it leak ?

FIGURE 4.1 presents an overview of the power trace and the first order leakages of the full AES execution. The trace is divided in three main parts: the key-scheduling, the loading of the plaintext from the memory and the 10 AES rounds. As expected, during the key scheduling (2500 first time samples), no influence of the input plaintext is observed by the t-test nor the ρ-test.

Since the loading of the data and the computations done by the AES depend at almost every cycle on the plaintext, the null hypothesis can not be rejected at most of the time samples. That means that there is a difference of means between the two sets used by the t-test (fixed and random plaintexts), which implies that mounting an attack on these points should be possible given the right model.

The profiled ρ-test is more specific, and based on a model of the mean leakage at each time sample for each plaintext value. In this case, it targets the influence of the value of the first byte of the plaintext. We can observe four different groups of time samples where there is a detection of leakage. These groups correspond to the loading of the data, the AddRoundKey, the S-box and ShiftRows together, and the MixColumns of the first round.

The results given by the ρ-test using the Hamming weight model are similar to the ones given by the profiled ρ-test. The main difference is that it does not exhibit leakage during the loading of the data and the AddRoundKey. This follows our expectations, since it targets the values at the output of the S-box, and these values are only handled during the actual S-box and during the MixColumns. It can be noted that the values returned by this test are lower than the ones from the other test.

At this point, the profiled ρ-test seems better than the Hamming weight ρ-test. It shows more leakage points, with higher values.

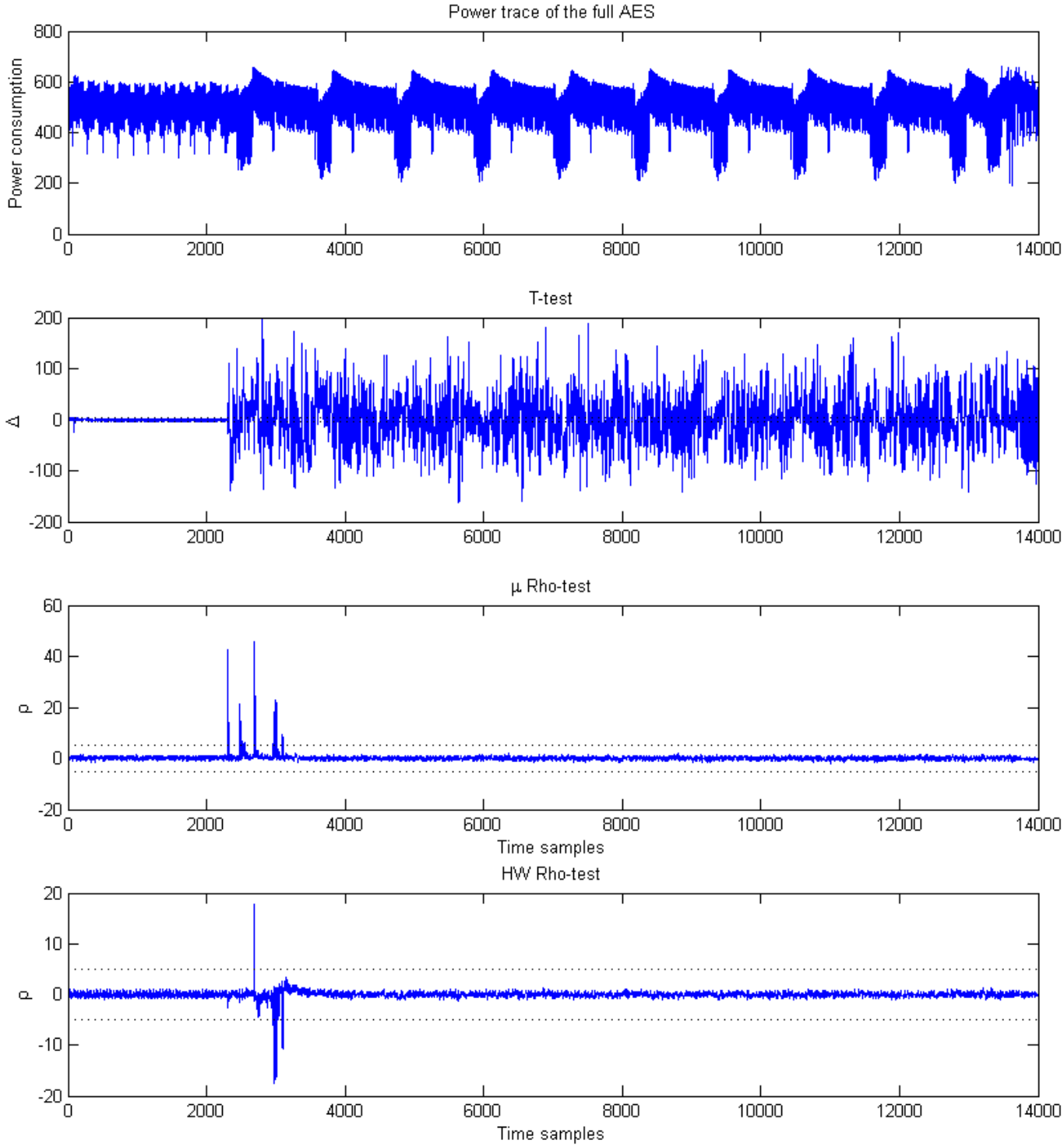


FIGURE 4.1 – AES-furious first order leakage

4.3 Single-bit Differential Power Analysis

In single-bit DPA, the attacker wants to attack a single bit in each S-box of the first round of the AES. Let us say that the attack will be on the most significant bit of each S-box. In practice, any bit can be attacked on any round. The attack on the first round is faster since there is no key scheduling involved.

The attacker $\mathcal{A}$ records the N_t traces $(T_1, T_2, ..., T_{N_t})$ of the N_t encryptions of random plaintexts $(p^1, p^2, ..., p^{N_t})$ with a fixed key k^*.

We define I_j^l as the output of the j^{th} S-box of the first round when encrypting the l^{th} plaintext $(1 \leq j \leq 16,\ 1 \leq l \leq N_t)$ with the key byte x:

$$I_j^l(x) = \mathsf{S\text{-}box}(p_j^l \oplus x)$$

where p_j^l is the j^{th} byte of the l^{th} plaintext.

The selection function g returns the MSB of its input byte and is then defined as follows:

$$g : \{0,1\}^8 \mapsto \{0,1\} : g(x) = \begin{cases} 1 & \text{if } 128 \leq x \leq 255 \\ 0 & \text{if } 0 \leq x < 128 \end{cases} = x >> 7$$

Then, $\mathcal{A}$ can fill the sets $S_0^{j,x}$ and $S_1^{j,x}$ for every key byte hypothesis x. Those sets are expressed as:

$$S_b^{j,x} = \{l \mid g(I_j^l(x)) = b\} = \{l \mid g(\mathsf{S\text{-}box}(p_j^l \oplus x)) = b\}$$

$\mathcal{A}$ then computes the first-order differential traces $\Delta_1^j(x)$ as the difference of the means of the traces of each set $S_0^{j,x}$ and $S_1^{j,x}$:

$$\Delta_1^j(x) = \left| \langle T_l \rangle_{l \in S_0^{j,x}} - \langle T_l \rangle_{l \in S_1^{j,x}} \right|$$

$$\Leftrightarrow \Delta_1^j(x) = \left| \frac{1}{\left|\left|S_0^{j,x}\right|\right|} \sum_{l \in S_0^{j,x}} T_l - \frac{1}{\left|\left|S_1^{j,x}\right|\right|} \sum_{l \in S_1^{j,x}} T_l \right|$$

The correct key hypothesis for the j^{th} byte x^* is the one that has the greatest value in the differential traces $\Delta_1^j(x)$.

$$x^* = \max_x \ \Delta_1^j(x)$$

If the number of traces N_t is large enough, x^* is the correct j^{th} byte of k^*. To recover the whole key, $\mathcal{A}$ can repeat this whole process by changing the value of $j \in [1, 16]$.

The whole attack is given in Algorithm 18 as a pseudocode.

Results

FIGURE 4.2 shows a DPA attack on the MSB of the 16 key bytes. The attack was performed with 5000 traces, on the first round of the AES. Every correct key byte can easily be distinguished from the others. FIGURE 4.3 shows the evolution of the difference of means depending of the number of traces used to estimate it. We can observe that for some key bytes, less than 200 traces are necessary but for some others at least 500 traces are needed to correctly distinguish the byte. On average, the value of the difference of means becomes quite stable after 1000 traces. It is interesting to note that the DPA works only when focusing the right part of the power trace. This attack was also performed on the whole power trace, but did not succeed in recovering the key. FIGURE 4.4 shows the difference of means calculated by the DPA attacking the first byte at each time sample of the trace. The correct key is in red, and a peak during the AES encryption can clearly be observed. However, the points after the AES encryption, which correspond to the simple-serial protocol, exhibits much higher peaks. If the attacker is not careful when selecting the part of the trace to attack, he might mistake one of these peaks for the correct key.

Algorithm 18 Single-bit DPA

Inputs: a set of N_t traces $(T_1, T_2, \cdots, T_{N_t})$
Output: the master key k

1: **for** $l = 1$ **to** N_t **do** $T_l \leftarrow$ measure_AES(p_j^l, k^*)
2: **for** $j = 1$ **to** 16 **do**
3: **for** $x = 0$ **to** 255 **do**
4: $S_0^{j,x} \leftarrow \varnothing$
5: $S_1^{j,x} \leftarrow \varnothing$
6: **for** $l = 1$ **to** N_t **do**
7: **if** S-box$(p_j^l \oplus x) \geq 128$ **then**
8: $S_1^{j,x} \leftarrow S_1^{j,x} \cup \{l\}$
9: **else**
10: $S_0^{j,x} \leftarrow S_0^{j,x} \cup \{l\}$
11: $\Delta_1^j(x) \leftarrow \left| \frac{1}{||S_0^{j,x}||} \sum_{l \in S_0^{j,x}} T_l - \frac{1}{||S_1^{j,x}||} \sum_{l \in S_1^{j,x}} T_l \right|$
12: $x^* \leftarrow \max_x \Delta_1^j(x)$
13: $k_j \leftarrow x^*$

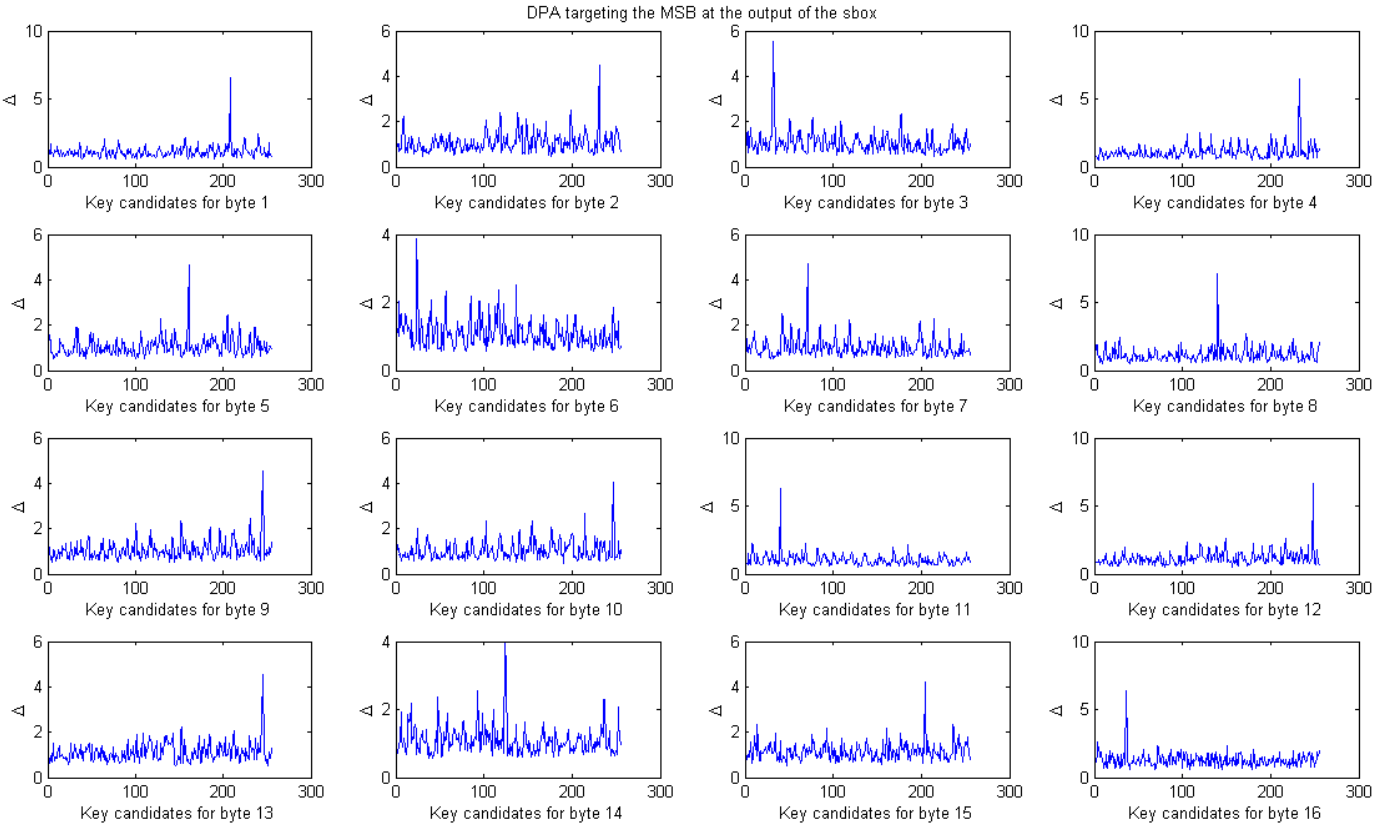


FIGURE 4.2 – DPA attack on the MSB of the 16 key bytes

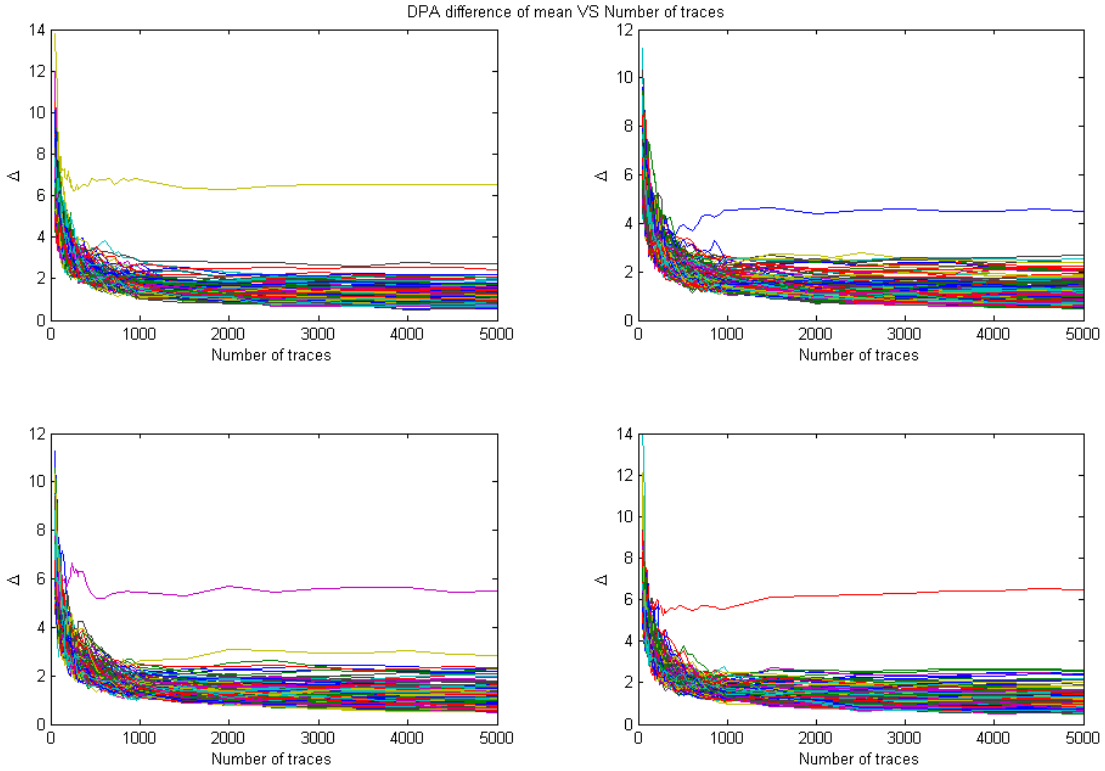


FIGURE 4.3 – DPA difference of means VS Number of traces attacking the MSB of 4 key bytes

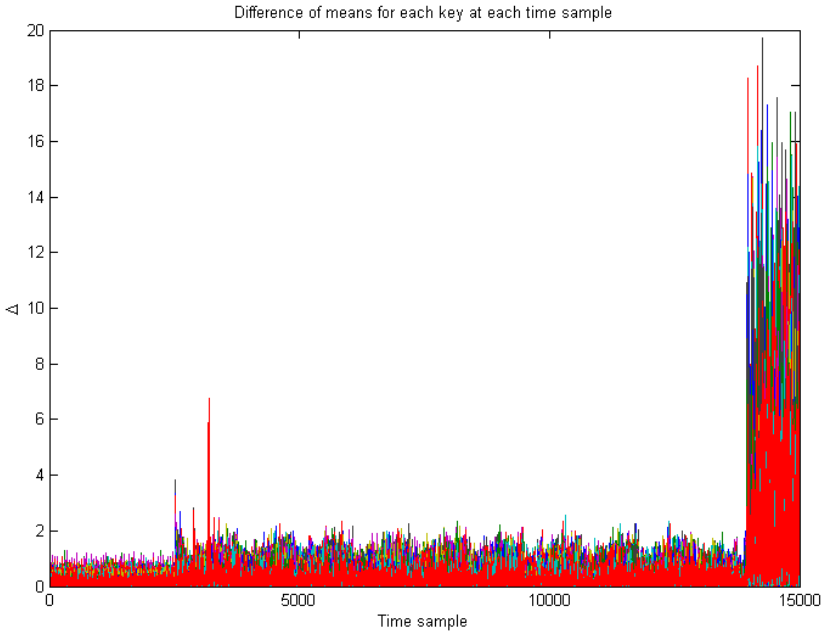


FIGURE 4.4 – DPA difference of means for each key at each time sample. In red the correct key.

4.4 Correlation Power Analysis

First-order CPA is another way to recover the key in an unprotected implementation of the AES. In order to perform a first-order CPA, an adversary $\mathcal{A}$ encrypts N_t random plaintexts $(p^1, p^2, ..., p^{N_t})$ with the secret fixed key k^* and records the N_t corresponding traces $(T_1, T_2, ..., T_{N_t})$ of size m. To recover the whole key k^*, $\mathcal{A}$ must attack each of the 16 S-boxes to recover each of the 16 key bytes k_j^* such that $k^* = k_1^*||k_2^*||...||k_{16}^*$.

In order to recover the j^{th} byte k_j^* of the secret key, the adversary $\mathcal{A}$ computes for all $1 \leq \ell \leq N_t$ and for all $0 < x \leq 255$ the intermediate value

$$y_{\ell,x} = \mathsf{S\text{-}box}(p_\ell^j, x)$$

Let also $z_{\ell,x} = H(y_{\ell,x})$ be the Hamming weight of $y_{\ell,x}$. The best key candidate x^* is the one that is maximizing the absolute value of the correlation between $T_\ell(t^*)$ and $z_{\ell,x}$ at the timestep t^*, that is called a point of interest (POI) of the trace. We use an estimation to compute the value of this correlation:

$$\hat{\rho}_{Tz}(t,x) = \frac{N_t \sum_{\ell=1}^{N_t} T_\ell(t) z_{\ell,x} - \sum_{\ell=1}^{N_t} T_\ell(t) \sum_{\ell=1}^{N_t} z_{\ell,x}}{\sqrt{(N_t \sum_{\ell=1}^{N_t} (T_\ell(t))^2 - (\sum_{\ell=1}^{N_t} T_\ell(t))^2)(N_t \sum_{\ell=1}^{N_t} z_{\ell,x}^2 - (\sum_{\ell=1}^{N_t} z_{\ell,x})^2)}}$$

The POI and the best key candidate are then given by:

$$(t^*, x^*) = \arg\max_{(t,x)} |\hat{\rho}_{Tz}(t,x)|$$

The retrieved key byte is given by x^*: $k_j = x^*$. The whole algorithm is given in Algorithm 19 as a pseudocode.

Algorithm 19 First-order CPA

Inputs: a set of N_t traces $(T_1, T_2, \cdots, T_{N_t})$

Outputs: a vector POI containing all the point of interests of the different S-boxes and the master key k

```
1: for l = 1 to N_t do T_l ← measure_AES(p_j^l, k*)
2: for j = 1 to 16 do
3:     for x = 0 to 255 do
4:         for l = 1 to N_t do
5:             y_{ℓ,x} ← S-box(p_j^ℓ ⊕ x)
6:             z_{ℓ,x} ← H(y_{ℓ,x})
7:         ρ̂_{Tz}(t,x) ← corr(T,z)
8:     (t*,x*) ← arg max_{(t,x)} |ρ̂_{Tz}(t,x)|
9:     k_j ← x*
10:    POI_j ← t*
```

Remark: In the MATLAB implementation, a lot of computation resources and time can be saved by working with matrix instead of loops.

Results

The CPA works and correctly recovers the correct key with a high probability when the number of traces N_t is high enough. FIGURE 4.5 shows the evolution of the correlation value for each candidate for the first four key bytes. This graph shows that after a 100 traces, the correct byte already stands out from the possible values. The correlation values still fluctuate until around

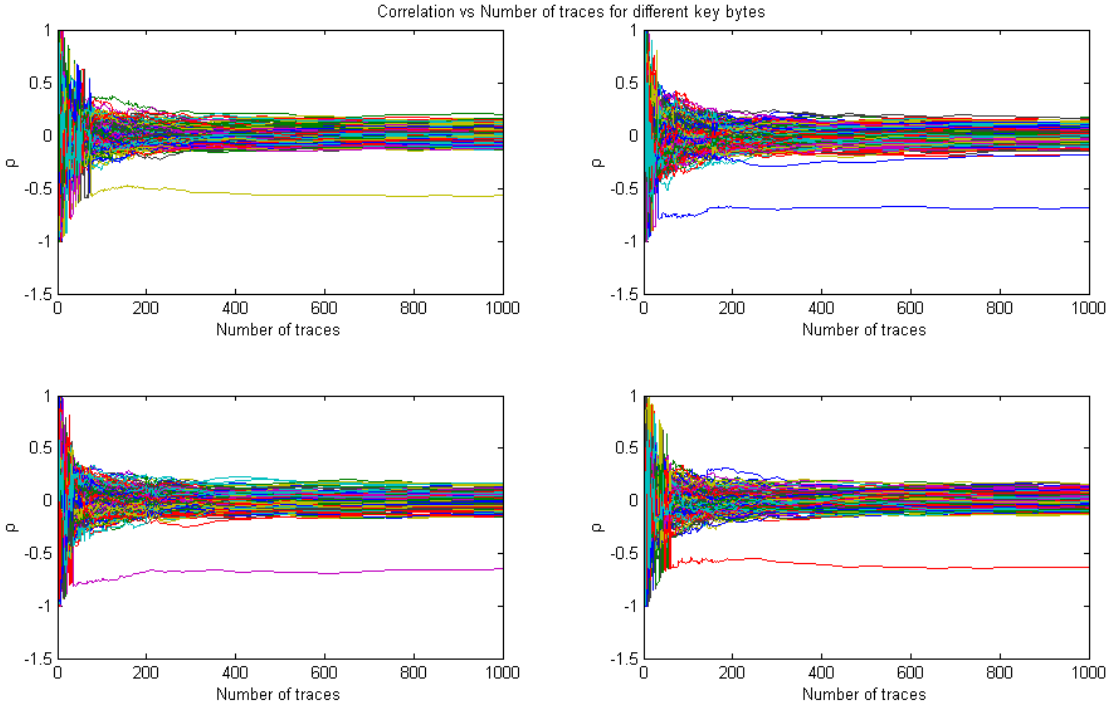


FIGURE 4.5 – Correlation value VS Number of traces for the first four key bytes

600 traces but becomes stable afterwards. FIGURE 4.6 confirms this, by showing the correlation with 100 traces for all the 16 key bytes. The correct key clearly stands out for every attacked S-box. The points of interest used for the computation of this correlation on the power trace are marked in red in FIGURE 4.7.

4.5 Univariate Template Attack

The univariate template attack is one of the most used attack to retrieve the secret key in a device if the attacker can obtain a similar one. Its main advantage is that once the learning phase is completed, the attacks can be performed successfully with only a few traces. The attacker also needs to be able to record power traces from the device that he actually wants to attacks. The similar device is used to build the profiling models.

The pseudo-code for an univariate template attack against the j^{th} byte of the key is given in Algorithm 20. This algorithm can be used to recover all the bytes of the secret key. The points of interests of the different bytes can be defined as the points with the maximum of correlation from a CPA, or taken from a leakage detection test.

Results

The template attack allows the recovery of the whole key easily, with previous knowledge of the different POIs of the device. In practice, we used the POIs computed by the CPA. FIGURE 4.8 shows that the correct value of each key byte stands out. The experiment is performed with a set of 5000 traces, of which 100 are used to perform the actual attack and 4900 are used for the profiling. Theoretically, we would like to use as many traces as possible for the actual attack for a maximized precision. In practice, multiplying too many times by small values makes $\mathbb{P}_k$ tends to 0. To counter this problem, we take the sum of the logarithms of the different $\mathbb{P}_{k,q}$, since

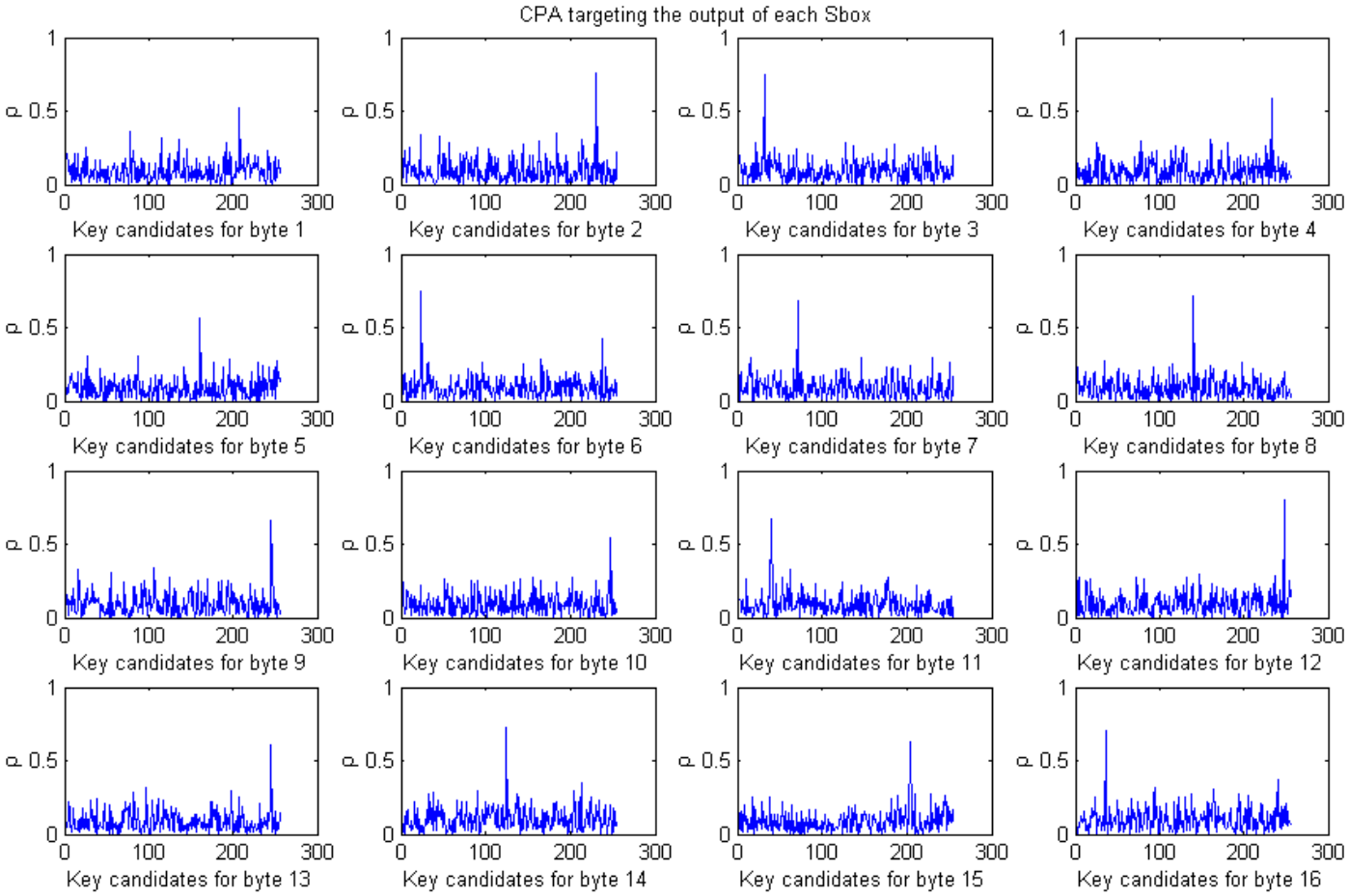


FIGURE 4.6 – Correlation for each key candidate for 100 power traces

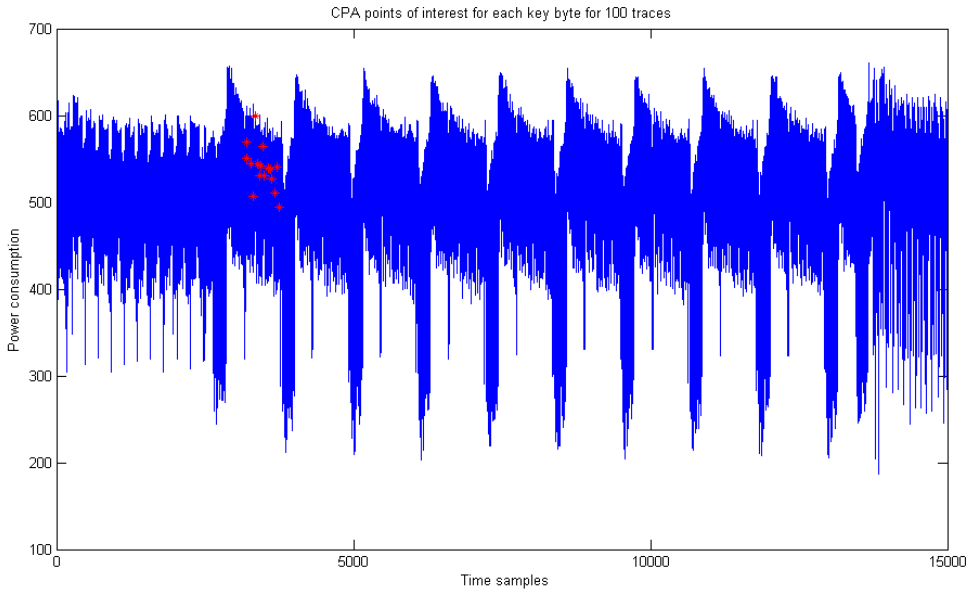


FIGURE 4.7 – Points of interest in the power trace based on FIGURE 4.6

Algorithm 20 Univariate Template Attack against the j^{th} byte of the key

Inputs: a set **k** and **p** containing each N_p keys/plaintexts, the corresponding set of N_p traces $(T_1^p, T_2^p, \cdots, T_{N_p}^p)$, a point of interest POI of the j^{th} byte, a set $\mathcal{P}$ containing Q random plaintexts and the corresponding traces $(T_1^o, T_2^o, \cdots, T_Q^o)$ of their encryption with the secret key k^* that we want to recover

Outputs: the j^{th} byte k_j^* of the master key k^*

$(L_0, L_1, \cdots, L_{255}) \leftarrow \varnothing$
for $i = 1$ **to** N_p **do** ▷ Learning phase
  $x \leftarrow \mathsf{S\text{-}box}(p_j^i \oplus k_j^i)$
  $L_x \leftarrow L_x \cup T_i^p(POI)$
for $i = 0$ **to** 255 **do**
  $\mu_i \leftarrow \mathsf{mean}(L_i)$
  $\sigma_i \leftarrow \mathsf{std}(L_i)$
for $k = 0$ **to** 255 **do** ▷ Online phase
  for $q = 1$ **to** Q **do**
    $x = \mathsf{S\text{-}box}(k \oplus \mathcal{P}_q)$
    $\mathbb{P}_{k,q} \leftarrow \dfrac{\mathcal{N}(T_q^o|\sigma_x, \mu_x)}{\sum_{i=0}^{255} \mathcal{N}(T_q^o|\sigma_i, \mu_i)}$
  $\mathbb{P}_k \leftarrow \prod_{q=1}^{Q} \mathbb{P}_{k,q}$
$k_j^* \leftarrow \arg\max_k \mathbb{P}_k$
return k_j^*

taking $k_j^* \leftarrow \arg\max_k \mathbb{P}_k$ for $\mathbb{P}_k \leftarrow \prod_{q=1}^{Q} \mathbb{P}_{k,q}$ or $\mathbb{P}_k \leftarrow \sum_{q=1}^{Q} \log \mathbb{P}_{k,q}$ is the same as the logarithm is a monotonically increasing function.

FIGURE 4.9 shows the evolution of the logarithm of the likelihood regarding the number of traces used to perform the attack. Based on these graphs, after only 30 to 40 traces, the correct key can already be guessed with quite high probability. Since the likelihood is logarithmic, the small observable differences are actually of orders of magnitude.

The critical part of this attack is the profiling part, because a wrongly built profile will never achieve successful attacks. It needs at least one sample for each intermediate value, but ideally much more to get rid of the noise. For this specific chip, we found out that good profiles could be built with at least 3000 traces. To improve the results, we used in practice a profile based on 4900 traces.

4.6 Conclusion

As seen in this chapter, a fast and unprotected implementation of the AES on the `ATxmega128d4` is not SCA-resilient. The key can be recovered easily using the basic attacks presented above, without even requiring great computational power. This is partly due to (*i*) the internal configuration of the chip which can leak information through the power consumption of given operations or registers content and (*ii*) the low level of noise in the chip.

This chapter shows the need that exists in embedded security to implement countermeasures against power analysis attacks. The next chapter will present some implementations of software masking and the following chapter will show the attacks that exist against those masking schemes.

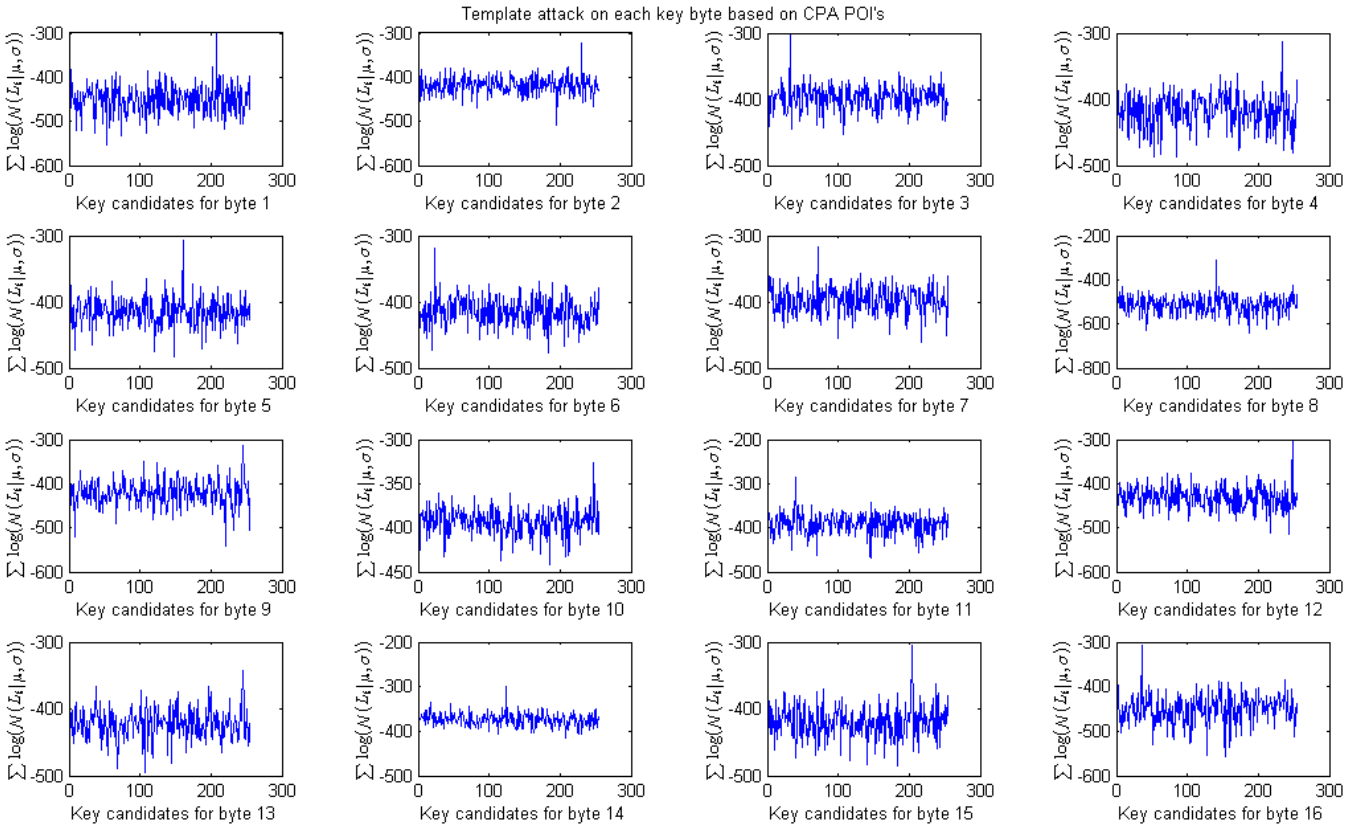


FIGURE 4.8 – Template attack for the 16 key bytes

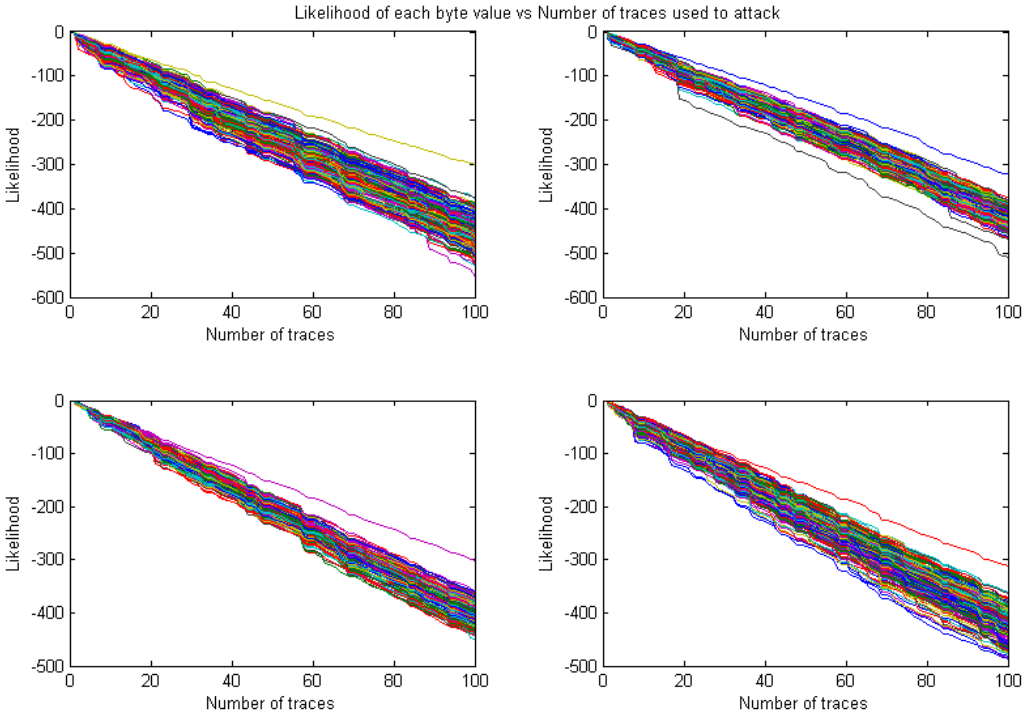


FIGURE 4.9 – Evolution of the likelihood VS Number of traces for the first 4 bytes of the key

Chapter 5

Implementations of the masking schemes

5.1 Introduction

In this chapter we will present technical details about the implementations of the different masking schemes on the `ATxMEGA128d4`. This is partly motivated by the 8-bit architecture of the processor, which limits the flexibility of the operations. We will first be interested in the implementation of some basic building blocks in the code such as the randomness generation and the secure field multiplication in $GF(2^8)$. The different functions that are implemented must be secure at the order d. A major prerequisite for this is to be constant-time no matter the data that is treated. Then, the architectures of the different schemes are also presented. Due to our architecture of 8 bits, we will implement all the schemes in a serial fashion, which means treating the shares one at a time. In this chapter, every register from the micro-controller and routine in assembly will be written in `typewriter font`.

5.2 Our pseudorandom generator

Random numbers are sequences of statistically independent and unbiased random bits. In a finite state machine as the ChipWhisperer-Lite, no true randomness is achievable. Then, instead of dealing with pure randomness, we use pseudorandomness. Pseudorandom numbers generators are numbers generators that apply a deterministic function f to their input called the *seed* s of size m to output a deterministic pseudorandom sequence $y = f(s)$ of size l such that $l >> m$. Moreover, a pseudorandom generator should respect some properties:

- It should be *fast*
- It should be *surjective*
- It should be *well-balanced* (i.e. unbiased from a uniform distribution): $\mathbb{P}[f(s) = y] \approx 2^{-m}$ $\forall s$.

In our case, we chose $l = n \cdot m$ and we construct the function $f : \{0,1\}^m \mapsto \{0,1\}^l$ by using recursively n times a function $g : \{0,1\}^m \mapsto \{0,1\}^m$ such that $x_t = g(x_{t-1})$ where $x_0 = s$, $|x_i| = m$ such that

$$y = g(s)||g(g(s))||\cdots||\underbrace{g(\cdots g(}_{n\ times}\ s\ \underbrace{)\cdots)}_{n\ times}$$

We chose to use $m = 128$ and $n = 96$ leading to $l = 12288$. We define our function g as three last rounds of the AES using the seed as first subkey and the three following subkeys as the three first subkeys generated by the key expansion of the seed.

Let us define the AES round for the round 1 to 9 by $\psi_k(x)$ and for the last round by $\psi'_k(x)$ for a subkey k and a state x:

$$\psi_k(x) = \mathsf{MixColumns}(\mathsf{ShiftRows}(\mathsf{S\text{-}box}(k \oplus x)))$$

$$\psi'_k(x) = \mathsf{ShiftRows}(\mathsf{S\text{-}box}(k \oplus x))$$

The function g is then defined as

$$g(s) = k_3 \oplus \psi'_{k_2}(\psi_{k_1}(\psi_s(0^{128})))$$

where $(k_1, k_2, k_3) \leftarrow \mathsf{KeyExpansion}(s)$ and where 0^{128} is the initial state containing 128 zero binary values. The functions (S-box, AddRoundKey, ShiftRows, MixColumns, KeyExpansion) used in our PRNG are the ones from the Rijndael Furious implementation.

To generate the table of pseudorandom values as described above, we use the function InitRdm that takes the seed as input, generates the key expansion of the seed and loops on the g function to generate all the pseudorandom bytes. To get a random byte from this table, we use the function `GenRdm`. We associate the memory register `X` to the generation of the random values. The `GenRdm` function is then reduced to a simple look-up in the table of random. Its implementation is given here:

```
1   GenRdm:
2   ; in : XL, XH
3   ; out : rdm
4   cpi XH, 0x2F
5   brlo PC+5
6   cpi XL, 0xFF
7   brlo PC+3
8   ldi XL, 0x00
9   ldi XH, 0x2A
10  ld rdm, X+
11  ret
```

Each time a random byte is needed, the random byte at the index `X` is returned and the pointer in the table (the value of the `X` register) in incremented. The implementation consists in checking whether we arrived to the end of the table. The table stands from the address 0x2A00 to the address 0x2FFF in the memory. If we are not at the end of the table, then we return a random byte and post-increment the value of the pointer `X`, else we go back to the beginning of the table. It means that if the masking scheme needs more random bytes than the size of the table, it can reuse the pseudo-random table from the beginning, this is called *recycling*. As explained in [1], recycling randomness is not an issue in software implementations of masking schemes written in assembly for $d = 1$ and $d = 2$, which is our case.

5.3 How to get an efficient and constant-time field multiplication ?

The field multiplication is used everywhere in every masking scheme that we present here. We want it to be constant-time in order to avoid any type of timing attack. It is implemented using the Log and AntiLog tables as explained in Algorithm 3. The two inputs a and $b \in \mathbb{F}_{2^8}$ are stored in the registers `io1` and `io2`. The output $a \odot b \in \mathbb{F}_{2^8}$ is stored in `io1`. Our ASM implementation is given hereafter. This implementation is constant-time and SPA-secure according to [18].

```
GFMult:
; inputs: io1, io2
; output: io1
push ZL
push ZH
ldi ZH, high(Logtable<<1)
mov ZL, io2
lpm temp2, Z
mov ZL, io1
lpm ZL, Z
ldi ZH, high(ALogtable<<1)
add ZL, temp2
cpi io1, 0
brne PC+3
ldi io1, 0
rjmp PC+3
ldi io1, 0x10 ; wait one cycle
ldi io1, 0xFF
cpi io2, 0
brne PC+3
andi io1, 0
rjmp PC+3
andi io1, 0xFF ; wait one cycle
andi io1, 0xFF
cp ZL, temp2
brlo PC+3
ldi temp2, 0
rjmp PC+3
ldi temp2, 10 ; wait one cycle
ldi temp2, 1
add ZL, temp2
lpm temp2, Z
and io1, temp2
pop ZH
pop ZL
ret
```

There are three lines in the code where that are waiting instructions to delay the execution. It allows to have the same number of cycles no matter the inputs of the algorithm. The same instructions are executed regardless of the inputs a and b, even if $a = 0$ or $b = 0$ directly induces $a \odot b = 0$, to keep the power consumption constant. The whole operation needs 46 cycles to be computed, including the `call` and `ret` instructions. To improve the speed, both the Log and Antilog tables are aligned with an address ending by 0x00. That means that we only need to load one byte in the `ZL` register to get access to the elements of the tables.

5.4 Implementations of the masking schemes

5.4.1 General procedure

All our masked schemes are in fact different ways of computing the AES S-boxes securely. The rest of the algorithm stays the same for every scheme and is described here. We will explain the data path from the computer query of encryption until the response to the computer with the ciphertext.

1. First there is the Simple Serial interface as explained in the Chapter 3. At the end of this protocol, the key is stored in 0x3000, the plaintext is at 0x3010 and the seed is at 0x3020.

2. Then, there is the `InitRdm` procedure call which will create the table containing the random bytes from 0x2A00 to 0x2FFF as explained in Section 5.2.

3. After that, there is the `InitStates` procedure call that first will generate d random states $\mathbf{s}^1, \mathbf{s}^2, \cdots, \mathbf{s}^d$ for a masking at the d^{th} order by generating $16d$ random bytes. Then it will compute each byte of $\mathbf{s}^0$ by xoring the real plaintext stored at 0x3010 with the d masks:

$$(s_0^0, \cdots, s_{15}^0) \leftarrow (s_0, \cdots, s_{15}) \oplus (s_0^1, \cdots, s_{15}^1) \oplus \cdots \oplus (s_0^d, \cdots, s_{15}^d)$$

   The masked states ($\mathbf{s}^0$, $\mathbf{s}^1$, $\cdots$, $\mathbf{s}^d$) are stored in the DRAM memory at 0x3100, 0x3110, etc.

4. Then, there is the `InitKeys` procedure call that will do the same thing as `InitStates` for the key k stored at 0x3000. The key shares ($\mathbf{k}^0$, $\mathbf{k}^1$, $\cdots$, $\mathbf{k}^d$) are stored in the DRAM memory at 0x2500, 0x2600, $\cdots$

5. Finally the `SecKeyExp` procedure is called and will expand the $d+1$ key shares in the memory. The first key share $\mathbf{k}^0$ which is located at 0x2500 will be extended from 0x2510 to 0x25AF, the second key share $\mathbf{k}^1$ which is located at 0x2600 will be extended from 0x2610 to 0x26AF, etc. All the key expansions are done using the SecKeyExp algorithm showed is section 2.5.7. The S-box that is used in the secure key expansion is the one of the algorithm that is implemented, the Rcon table is hardcoded in memory.

6. The `trigger_high` procedure is called to start the recording of the trace by the ADC. It simply is a writing in an I/O register of the micro-controller that is caught by the FPGA to launch the data acquisition.

7. The masked AES is called.

8. At the end of the AES, the ciphertext is written at 0x3030 and the Simple Serial protocol returns the value of the ciphertext on the serial bus.

As showing all the written assembly codes in this chapter is impossible due to their size, we chose to show some interesting routines for $d = 1$ in each implementation.

5.4.2 Rivain-Prouff masking scheme

The main points of the masked AES implemented by the Rivain-Prouff method from [24] as recalled in section 2.5.3 are explained here:

1. It begins with a classical linear layer which is the AddRoundKey

2. The SubByte layer is implemented using the `SecSbox` routine on each byte which call the `SecExp254` routine and apply the affine transformation.

3. The ShiftRows and MixColumns linear layers are applied on each share separately.

The Rivain-Prouff "corrected" masking of section 2.5.5 is exactly the same except that the refresh masking procedure is replaced by the refresh mask of ISW. We give here the implementation of a `SecMult` procedure for $d = 1$:

```
1 SecMult:
2     ; inputs : a0, a1 in r0,r1 and b0,b1 in r3,r4
3     ; outputs : c0,c1 in r0,r1
4     ;
5     push r6
6     push r7
7     call GenRdm
8     mov r6, rdm
9     mov io1, r0
```

```
10      mov io2, r4
11      call GFMult
12      mov r7, io1
13      eor r7, r6
14      mov io1, r3
15      mov io2, r1
16      call GFMult
17      eor r7, io1
18      mov io1, r0
19      mov io2, r3
20      call GFMult
21      mov r0, io1
22      eor r0, r6
23      mov io1, r1
24      mov io2, r4
25      call GFMult
26      mov r1, io1
27      eor r1, r7
28      pop r7
29      pop r6
30      ret
```

5.4.3 Coron masking scheme

The Coron masking scheme of section 2.5.4 is built the same way than the scheme presented above, except for some parts of the exponentiation to the power 254 in $\mathbb{F}_{2^8}$. It uses `SecPow3` and `SecPow5` that include the refresh mask, instead of the two first `SecMult` routines. We give here the `SecPow3` routine for $d = 1$:

```
1  SecPow3:
2       ;; In : r0,r1 : a = r0 xor r1
3       ;; Out: r0, r1  : a^3 = r0 xor r1
4       push r3 push r4 push r5  push ZL push ZH
5       call GenRdm
6       mov r3, rdm
7       call GenRdm
8       mov r4, rdm
9       ldi ZH, high(GFThird<<1)
10      mov r5,r3
11      mov ZL, r0
12      eor ZL, r4
13      lpm io1, Z
14      eor r5, io1
15      mov ZL, r1
16      eor ZL, r4
17      lpm io1, Z
18      eor r5, io1
19      mov ZL, r0
20      eor ZL, r4
21      eor ZL, r1
22      lpm io1, Z
23      eor r5, io1
24      mov ZL, r4
25      lpm io1, Z
26      eor r5, io1
27      mov ZL, r0
28      lpm r0, Z
29      eor r0, r3
30      mov ZL, r1
31      lpm r1, Z
32      eor r1, r5
```

```
33      pop ZH pop ZL pop r5 pop r4 pop r3
34      ret
```

`GFThird` is a table containing the cubes in $\mathbb{F}_{2^8}$ directly hardcoded in the memory at an address with the lower byte equal to 0x00. `SecPow5` is exactly the same function, except that we use the `GFFifth` table instead.

5.4.4 Genelle masking scheme

The Genelle masking scheme of section 2.5.6 is also the same for the points 1 and 3 of RP. The second points varies from the previous implementation since the computation of the SubBytes is done differently. Indeed, while in the other schemes the computation of the SubBytes is done for one S-box at a time, this scheme computes once per round all the secure Dirac values needed to map the masked states from $\mathbb{F}_{2^8}$ to $\mathbb{F}_{2^8}^{\times}$. This step is time consuming but once it has been done, all the non-linear part of the S-box is reduced to a look-up of $x^{-1} \in \mathbb{F}_{2^8}$. This induces a huge speed improvement. For this speed improvement to be possible, we need the functions `AMtoMM` and `MMtoAM` to be fast, which is the case compared to the number of cycles needed for a `SecExp254` in the scheme of Coron or RP. We give here the `AMtoMM` and `MMtoAM` routines for $d = 1$:

```
1       AMtoMM:
2       ; inputs: x0, x1 in r0, r1 such that x = x0 xor x1
3       ; outputs; z0, z1 in r0, r1 such that GFMult(z0, z1^-1) = x
4       push r3 push r4
5       mov r3, r0
6       mov r4, r1
7       call GenRdm
8       cpi rdm, 0
9       brne PC+3
10      inc rdm
11      rjmp PC+3
12      inc rdm ;waiting cycle
13      dec rdm ;waiting cycle
14      mov r1, rdm
15      mov io1, r0
16      mov io2, r1
17      call GFMult
18      mov r0, io1
19      mov io2, r4
20      mov io1, r1
21      call GFMult
22      mov r4, io1
23      eor r0, r4
24      pop r4 pop r3
25      ret
```

In this code, the first call to `GenRdm` is special since it should draw a random value in $\mathbb{F}_{2^8}^{\times}$. It means that the value can not be 0 so a conditional branch is used to increment the value of the random byte from 0 to 1 if it is the case. To keep the execution time constant, we need to add operations doing nothing in the other case. This induces a little bias in our random generator since $\mathbb{P}[\texttt{rdm} = 1] = 2/256$ while all the other values have a probability $\mathbb{P}[\texttt{rdm} = x | x \neq 1] = 1/256$.

```
1       MMtoAM:
2       ; inputs: z0, z1 in r0, r1 such that GFMult(z0, z1^-1) = z
3       ; outputs; x0, x1 in r0, r1 such that x0 xor x1 = z
4       push r3 push r4 push ZL push ZH
5       ldi ZH, high(GFinv<<1)
```

```
6       mov ZL, r1
7       lpm r1, Z
8       pop ZH pop ZL
9       mov r3, r0
10      call GenRdm
11      mov r4, rdm
12      eor r3, r4
13      mov io1, r1
14      mov io2, r3
15      call GFMult
16      mov r3, io1
17      mov io1, r4
18      mov io2 ,r1
19      call GFMult
20      mov r4, io1
21      call GenRdm
22      eor r4, rdm
23      eor r3 ,r4
24      mov r4 ,rdm
25      mov r0, r3
26      mov r1, r4
27      pop r4  pop r3
28      ret
```

`GFinv` is a table containing the inverses in $\mathbb{F}_{2^8}$ directly hardcoded in the memory at an address with the lower byte equal to 0x00.

5.5 Comparison of the performances

In this section we show the performances of the different masking schemes.

- *(All implementations)*

Routines	$(d = 1)$ # Cycles	$(d = 1)$ # random bytes	$(d = 2)$ # Cycles	$(d = 2)$ # random bytes
`InitRdm`	78985	0	78985	0
`InitStates`	432	16	737	32
`InitKeys`	432	16	736	32
`SecKeyExp`	5387	40	7587	80
`trigger_high`	12	0	12	0
AddRoundKey	359	0	536	0
ShiftRows	134	0	193	0
MixColumns	328	0	492	0

- *(Rivain-Prouff)*

Routines	$(d = 1)$ # Cycles	$(d = 1)$ # random bytes	$(d = 2)$ # Cycles	$(d = 2)$ # random bytes
Complete scheme	198578	1232	390241	2560
`SecSbox`	1169	6	2374	16
		(Corrected version)		
Complete scheme	-	-	403078	2880
`SecSbox`	-	-	2406	18

- *(Coron)*

	$(d = 1)$		$(d = 2)$	
Routines	# Cycles	# random bytes	# Cycles	# random bytes
Complete scheme	153138	1232	295878	2880
`SecSbox`	885	6	1736	18

- *(Genelle)*

	$(d = 1)$		$(d = 2)$	
Routines	# Cycles	# random bytes	# Cycles	# random bytes
Complete scheme	103501	620	195223	1700
Complete S-box[1]	~586	~4	~1126	~11
`SecDirac`	2946	14	4680	42
`AMtoMM`	139	1	325	4
`MMtoAM`	163	2	365	5

5.6 Conclusion

The results that we obtain in term of cycles and random numbers seem coherent as they follow the tendencies we were expecting. Indeed, Grosso *et al.* already showed in [13] that the scheme of Genelle *et al.* from [12] was faster than the one from Rivain and Prouff from [24]. The scheme of Coron *et al.* from [7] is also faster than the one of RP but needs more random words and is still slower than the one from Genelle *et al.* Actually, there is always a trade-off between the complexity of the scheme, the number of cycles and the number of random words that are needed for the computation. As all of those results are from our own implementations in assembly on a 8-bit micro-controller, they have to be taken with care, as there might be possible improvements. It appears that the scheme from Genelle *et al.* is way faster than the others while requiring less random bytes. Since efficiency is not the only figure of merit to be taken into account when studying those implementations, we will evaluate their security in the next chapter.

[1] Estimated by dividing a full SubBytes operation by 16

Chapter 6

Power Analysis Attacks against masked AES implementations

6.1 Introduction

This chapter is the core of the second part of this thesis, which was about secure computing of the AES on the `ATxmega128d4` of the ChipWhisperer-Lite board. In this chapter, we will present statistic security evaluations using leakage detection tests and evolved attacks to break the masked schemes presented in Section 2.5 along with the obtained results.

The leakage detection methods are used to detect leakages of a given order which corresponds to the number of points that are simultaneously tested together to detect leakage. Typically, we say that a scheme masked at order d (i.e. there is $d+1$ shares) is broken if there are (exploitable) leakages of information when doing a leakage detection test at order $d' < d$, i.e. there is a tuple of $d'+1$ intermediate values that is correlated with any sensitive data. The scheme then becomes d'-secure instead of d-secure.

6.2 First order leakage evaluations

The first step to analyze the security of our implementations is to test their first-order security. The first order leakage detection methods analyses one point of the trace at a time to compute the test. Therefore, we would like to see no information leakage from the different masked implementation in theory. As explained in Section 2.6, we perform Welch's t-tests and ρ-tests to detect the points where there is information leakage. Various techniques have been developed to detect the points of interest beyond those points and discard the others, as explained in [10]. When we run leakage detection tests against an implementation, we also check the difference between the results when the seed is fixed and when the seed is random. When the seed is fixed, the pseudo-random numbers generator always outputs the same bytes. That means that if we encrypt the same plaintext with the same key many times, we will have the same intermediate values every time.

To make a comparison between the different results that we obtained, we decided to always focus on the first S-box of the first round of the AES encryption. We The following sections present the case study of the Coron masking scheme to analyse the different types of first-order leakage.

6.2.1 Coron masking scheme with two shares

A trace of the first S-box for the Coron masking scheme with two shares is given in Figure 6.1.

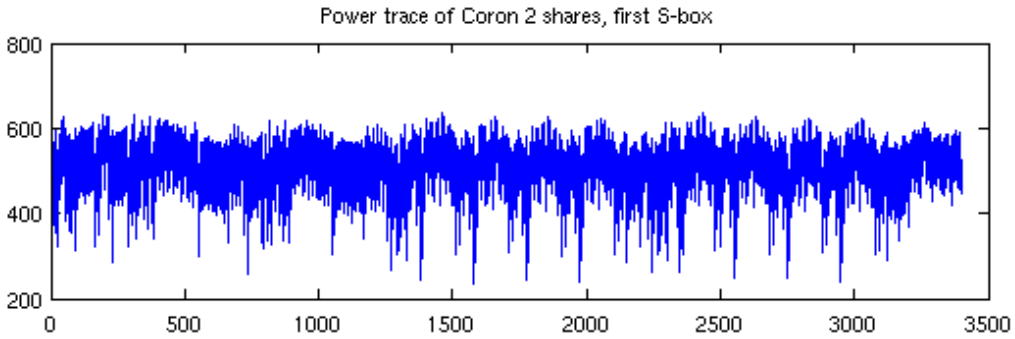


FIGURE 6.1 – A power trace of the first S-box of an AES encryption masked by Coron with $d = 1$

Fixed seed

Let us take a look on the results when we use the same seed every time. We expect the scheme to be broken in that case since the security of the scheme is not ensured if the random data used to generate the masks, refresh it and do the secure computations are not good enough. In fact, if we disable the seed, the security of our implementation becomes of order 0. It means that this implementation becomes as unsecure as the unprotected version of the AES.

Let us first analyse the fixed-vs-random Welch's t-test, between a class of power traces encrypting random plaintexts with a fixed key and a class of power traces encrypting a fixed plaintext with the same key, with the seed fixed in both cases. This gives the results in FIGURE 6.2a. As expected, the values of t-test easily pass the threshold of $\pm$ 5. This means that the null hypothesis $\mathcal{H}_0$ as defined in Section 2.6 is rejected and that there are significant differences between the mean values of the points of the traces in the two different classes. Intuitively, the power traces of the first class (fixed plaintexts) should remain the same for every encryption at the exception of the noise. The second class should contain timesteps with high variations that depend on the plaintexts since the key and the seed are fixed. These differences form the peaks that can be observed.

The second test that we will analyse is the profiled ρ-test using the means model given in FIGURE 6.2b. This test will check whether there is a correlation between the value of the first byte of the plaintext and the mean value of the power traces at a given point of the trace, for every points of the trace. As expected, the threshold of ± 5 is also completely broken. It means that there is also a lot of information leakage. Intuitively, the ρ-test shows leakage at timesteps where the recorded power depends on the value of the target input byte.

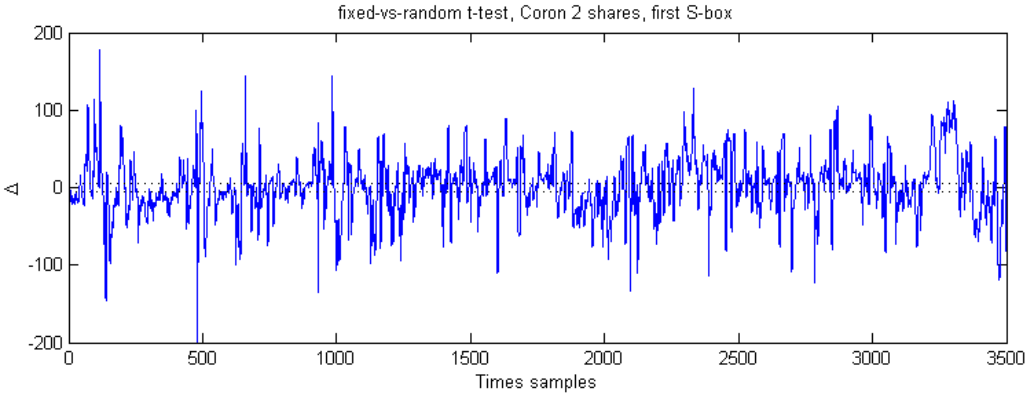
The third and last leakage test that we use is the ρ-test using the hamming weight model and is given is FIGURE 6.2c. This one does not passes the threshold. The values of the test stay between -5 and +5 at every timestep. The model that is assumed to be followed by the data to pass this test is not respected for masked implementations. This test does not work for any of the masked schemes that we have implemented, and therefore we will not show it anymore.

We see that the results are as expected when the seed is fixed. Indeed, when the PRNG is disabled, every test shows information leakage, and the security order of such implementation is 0, the same as for an unprotected implementation.

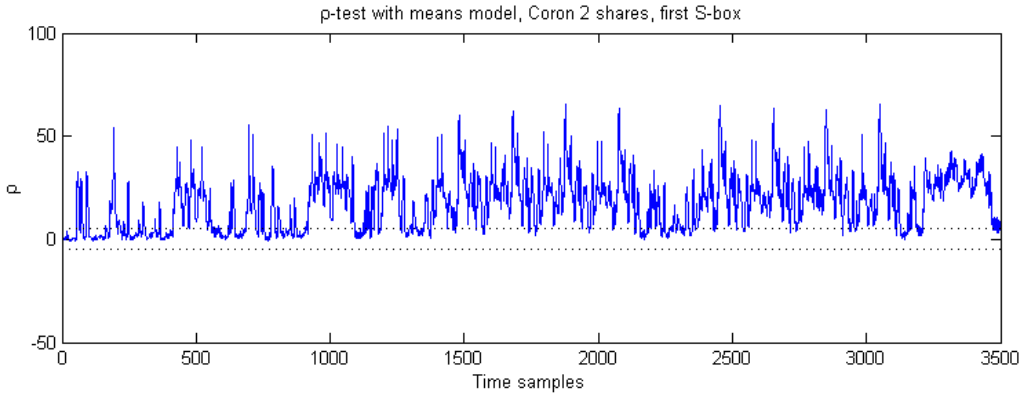
Random seeds

Let us now look at the results with a seed changing at each encryption. Each seed is generated using the MATLAB `randi([0 255],1, 16)` function.

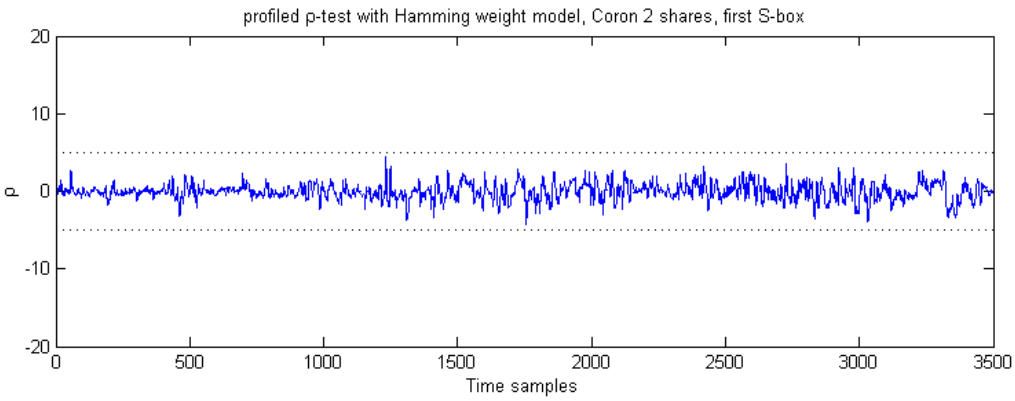
In [1], Balasch *et al.* recall that in practice, for software masked implementations, the types of leakage can be of two types, that depend on the device under test. Those leakage functions



(a)



(b)

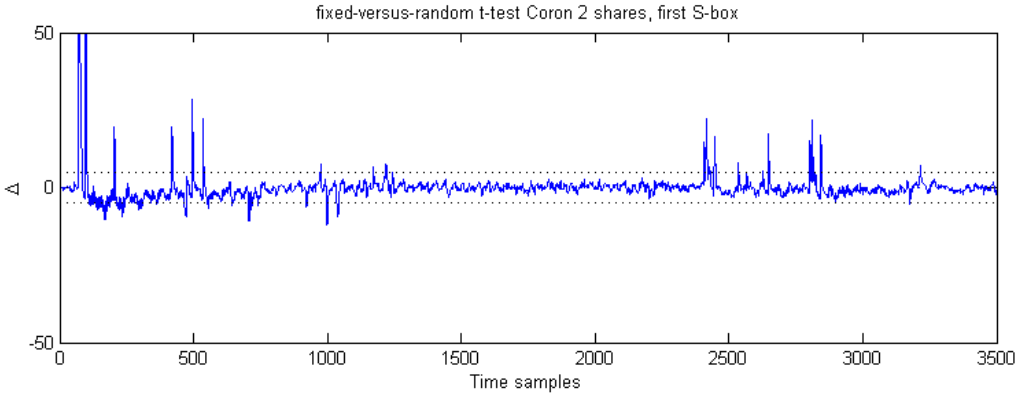


(c)

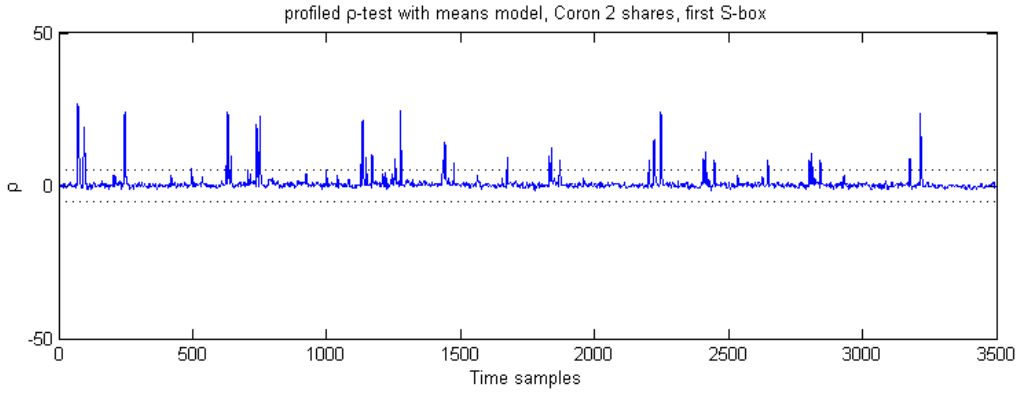
FIGURE 6.2 – Three detection leakage methods on the first S-box of Coron, $d = 1$ for a fixed seed

can be *value-based* or *transition-based*. Value-based leakage happens when the leakage of the device is related to the value of a register or a memory value. Transition-based leakage happens when the leakage value depends on the transition from one value to a new value in a register or in the memory. Overwriting a data provokes the actual information leak. When $d = 1$, there is only one mask applied to each sensitive data. Let us take as example the case where there is the value x_1 of the first share in a register and as the software runs, it needs to write the value x_2 of the second share in the same register, $x = x_1 \oplus x_2$ being a sensitive variable. If the leakage function is transition-based (generalizing the Hamming distance model) the leakage will

be proportional to $x_1 \oplus x_2 = x$. In this case, the chip should leak information on this sensitive data at first order. This is not acceptable since the definition of security at order d states that a tuple of $d+1$ internal variable is needed to get such dependence, when we can see that in fact d is enough. This is called *distance recombination* (or *shares recombination*).



(a)



(b)

FIGURE 6.3 – Two detection leakage methods on the first S-box of Coron, $d = 1$ for random seeds

The fixed-vs-random Welch's t-test is shown in FIGURE 6.3a. The test passes the +5 boundary which means that there is information leakage. The profiled ρ-test is shown in FIGURE 6.3b. The values of the test also passes the +5 threshold.

Those results allow us to say that the leakage function of our device is mostly transition-based. Indeed, if it was value based, no first order leakage should be observed for a 2 shares implementation. This shows a need to have implementations of higher order to counteract this type of first-order leakage.

Theses two graphs also allow us to observe the limitations of Welch's t-test compared to a profiled ρ-test. It only targets a difference of means between fixed and random plaintexts when the ρ-test takes advantage of a model, and can be more precise.

We will now take a look at the implementation of the Coron masking scheme with $d = 2$, i.e. with 3 shares.

6.2.2 Coron masking scheme with three shares

A trace of the first S-box for the Coron masking scheme with three shares is given in FIGURE 6.4. We have seen in the previous subsection that an implementation with $d = 1$ is not going to be secure against first order attacks. Now, we want to know if we can find the same problems with

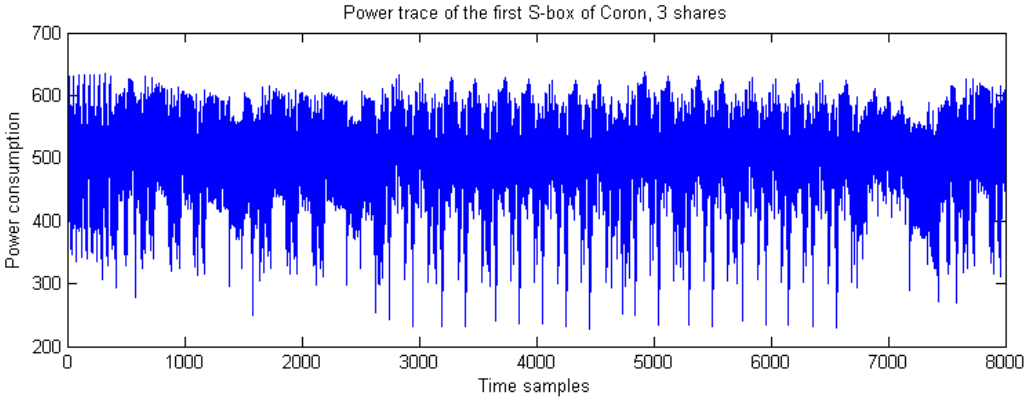


FIGURE 6.4 – A power trace of the first S-box of an AES encryption masked by Coron with $d = 2$

an implementation with 3 shares. We expect this scheme to be fully secure against first order leakage when the seed is random, and to leak when the seed is fixed.

Fixed seed

When the seed is fixed, the leakage should exhibit the same behavior as with $d = 1$ and this should also be the case for every order d. Cutting off the randomness is equivalent to destroying the security of the chip. The results of the tests are given in FIGURE 6.5a and 6.5b.

In both tests, the values are higher than the threshold for the same reasons than for $d = 1$.

Random seeds

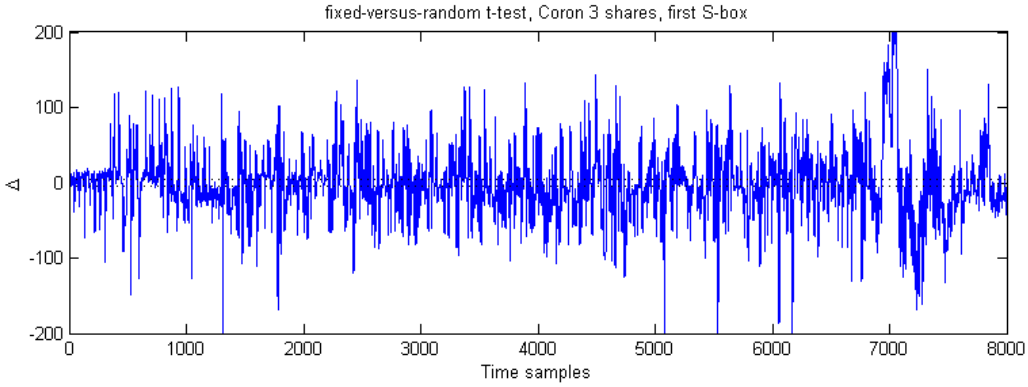
With new random seeds at each encryption, no first order leakage should appear and no time sample in the tests should have a value greater than the threshold of 5. Results are given in FIGURE 6.6a and 6.6b. As we can see from both the fixed-vs-random Welch's t-test and the profiled ρ-test, no value is greater than the threshold for the implementation with $d = 2$. This is an expected result that confirms that our implementation does not leak at first order anymore.

6.3 Power Analysis Attacks against the masked implementations

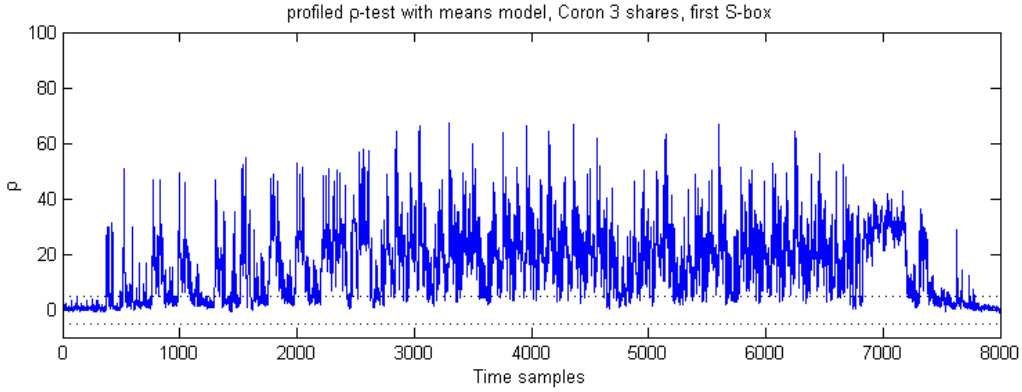
In this section, we attack the different masked schemes that we implemented. First, we focus on those who have shown first order leakage using a simple univariate template attack that should be sufficient to recover the key byte. Higher order template attacks are necessary to recover information from implementations secure at first order.

6.3.1 Univariate template attack against first-order leaking implementations

The univariate template attack is already explained in Section 2.4.6 and used against a masked implementation in Section 4.5. This attack seems to be the most efficient to break the key of our unsecured masked scheme since it focuses a single POI. This POI can be retrieved by maximizing the leakages tests Δ and $\hat{\mathfrak{r}}$: $\mathsf{POI}_t = \arg\max_\tau \Delta(\tau)$ and $\mathsf{POI}_r = \arg\max_\tau \hat{\mathfrak{r}}(\tau)$. Generally, those two POIs are not the same, since they do not represent the same leakage. In practice, attacking any of the high peak of the two tests allows to recover the key. Attacking the fixed seed schemes using this attack is easy and fast. We will take the example of attacking the Coron masking scheme with $d = 2$ when the seed is fixed. A power trace of the first S-box of the Coron masked scheme with $d = 2$ and the seed fixed is given in FIGURE 6.7 with in green all the points where



(a)



(b)

FIGURE 6.5 – Two detection leakage methods on the first S-box of Coron, $d = 2$ for a fixed seed

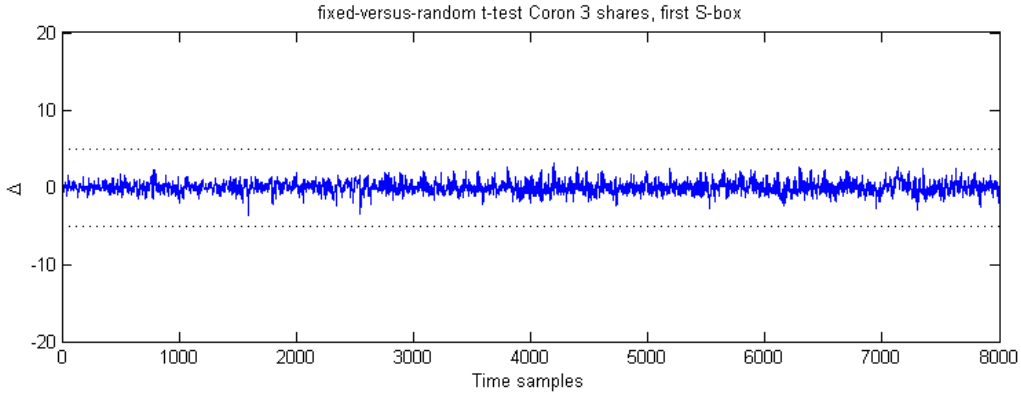
$|\hat{\mathfrak{r}}(\tau)| \geq 50$ and in red all the points where $|\Delta(\tau)| \geq 150$. Theoretically, any points with a value greater than 5 can be a POI but in this case, almost every point on the trace fulfils that condition. To keep things clear, we decided to show only points that passes the threshold by a big margin. By taking any of those points as the POI of an univariate template attack, we can easily recover the first key byte of the correct key. The attack was performed with sets of 5000 traces, with a learning set of 98 % of the traces, i.e. $N_p = 4900$ and an online test set of 2%, i.e. $Q = 100$. As expected, the schemes with a fixed seed are as not secure at first order.

We use exactly the same attack and the same procedure to recover the key byte from an implementation with $d = 1$ and with random seeds due to the shares recombination. The trace and value of the leakages are given in FIGURE 6.8. The correct key byte is also always recovered for the same parameters as above. The major leakage points that can be observed in both leakage tests are at the very beginning of the S-box, when both shares are loaded into registers. This is not surprising, since the combination of these two shares forms the sensitive variable. The other highlighted points of the ρ-test correspond to other similar movements between registers and memory.

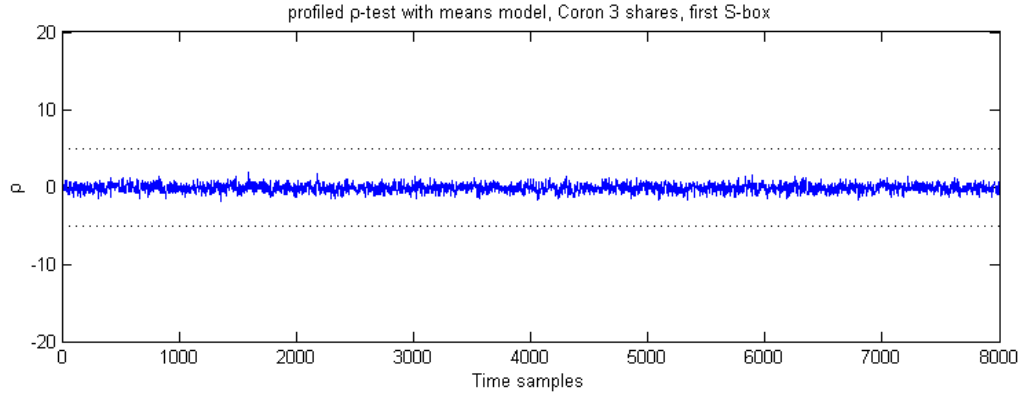
6.3.2 Bivariate template attack against a first-order secure implementation

Section 6.2.2 showed that implementations with three shares do not leak at first order. This implies that no univariate attack should reveal sensitive information. We explain in this section how a second order attack could successfully recover the key of these implementations.

This attack is based on the generalisation of the concept of share recombination showed



(a)



(b)

FIGURE 6.6 – Two detection leakage methods on the first S-box of Coron, $d = 2$ for random seeds

in Section 6.3.1. Indeed, since a point in time exists with information on $x_1 \oplus x_2$ for an implementation with 2 shares, the same point also exists for an implementation with 3 shares. This point does not contain information on the three shares when considered alone, but does when combined with information on x_3. This means that schemes with 3 shares can be attacked with a pair of points. The next question that arises is how many power traces do we need to build a precise enough profile to mount the attack?

Many research papers ([6, 9, 23, 25]) showed that the data complexity of SCA against masked implementations increases exponentially with the number of shares. Assuming Gaussian noise, this data complexity is proportional to $(\sigma_n^2)^d$, where σ_n^2 is the noise variance. In [1], Balasch *et al.* show that in practice, for transition based leakage models such as ours, this data complexity actually tends to $\lceil \frac{d}{2} \rceil$, because of share recombination problems.

In our case, the fact that it is exponential is a problem. Indeed, a first order template attack needs 3000 traces to compute a good profile (see Section 4.5). That means that a good profile for a pair of points requires around 9 million traces. The CW-L is not designed to record such a big number of traces, so we did not perform the actual attack.

Trying to find the right pair of point is also a very interesting topic, but we do not address it fully in this paper. [1, 10] present methodologies to efficiently find these pairs of points. Another, but rather intuitive approach is to 'translate' the leaking point from the 2 shares to the 3 shares implementation and to combine it with a time sample using the third share. To achieve this translation, access to the actual implementation of the different schemes is required, which makes it impossible to generalize. Section 2.4.6 recalls how to perform the actual attack when the model

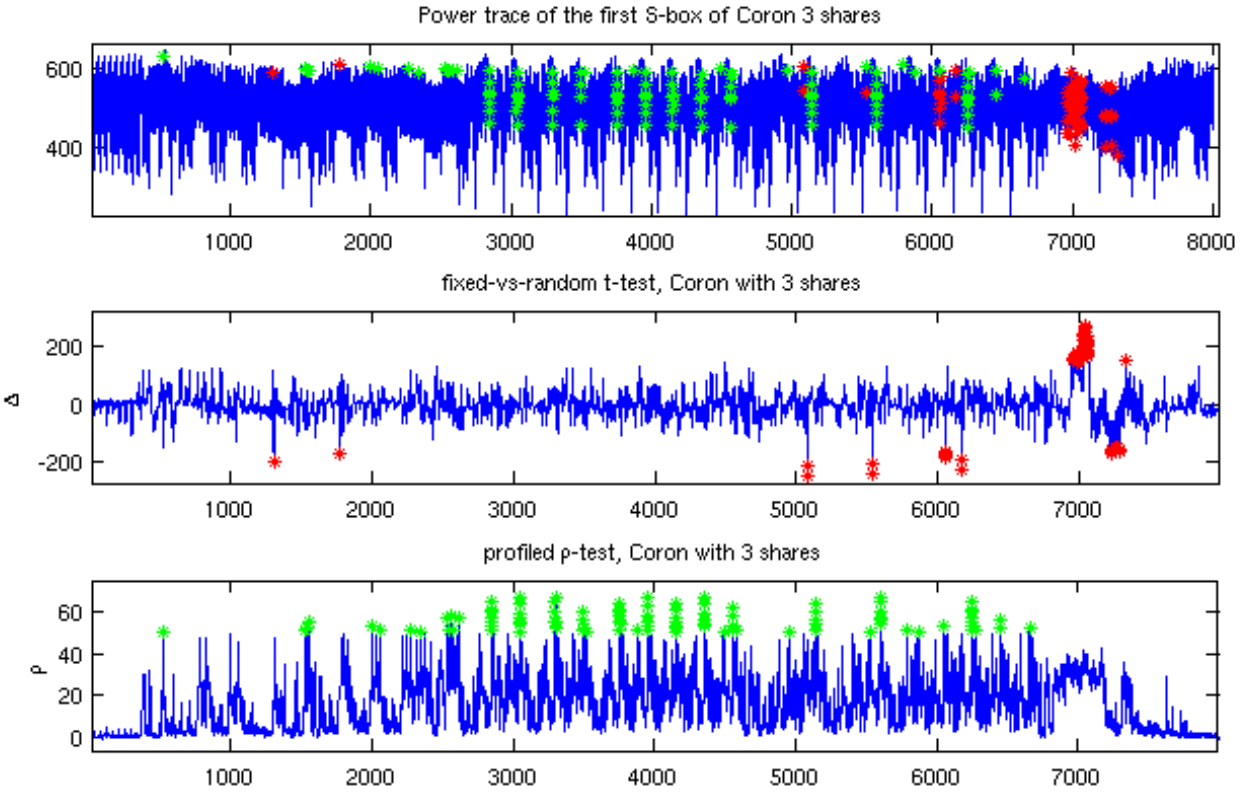


FIGURE 6.7 – Trace of the first S-box of the Coron masking scheme with $d = 2$ and a fixed seed. The values in green are the ones where $|\hat{\mathfrak{t}}| \geq 50$ and the values in red are the values where $|\Delta| \geq 150$

is built for a chosen pair of points.

6.4 Results of the leakage tests against the other masked implementations

The same first-order leakage detection tests have been applied on the different masking schemes that we have implemented, for both a fixed seed or random seeds. The results are in the annex B for clarity and are commented here. All the leakage detections tests are based on a set of 5000 traces.

Remark: In the following results, the dashed horizontal lines in the FIGURES correspond to the $|\hat{\mathfrak{t}}| \geq 5$ and $|\Delta| \geq 5$ thresholds.

- *Rivain-Prouff masking scheme with 2 shares.* Results are given in FIGURE B.1. Those results act exactly as the results of the Coron masking scheme with 2 shares.
- *Rivain-Prouff masking scheme with 3 shares.* Results are given in FIGURE B.2. No first order leakage can be observed. The values of both the t-test and the profiled ρ-test are between the boundaries at every timestep.
- *Rivain-Prouff masking scheme corrected with 3 shares.* Results are given in FIGURE B.3. The t-test shows some strange behavior. Indeed, there are three peaks that pass the

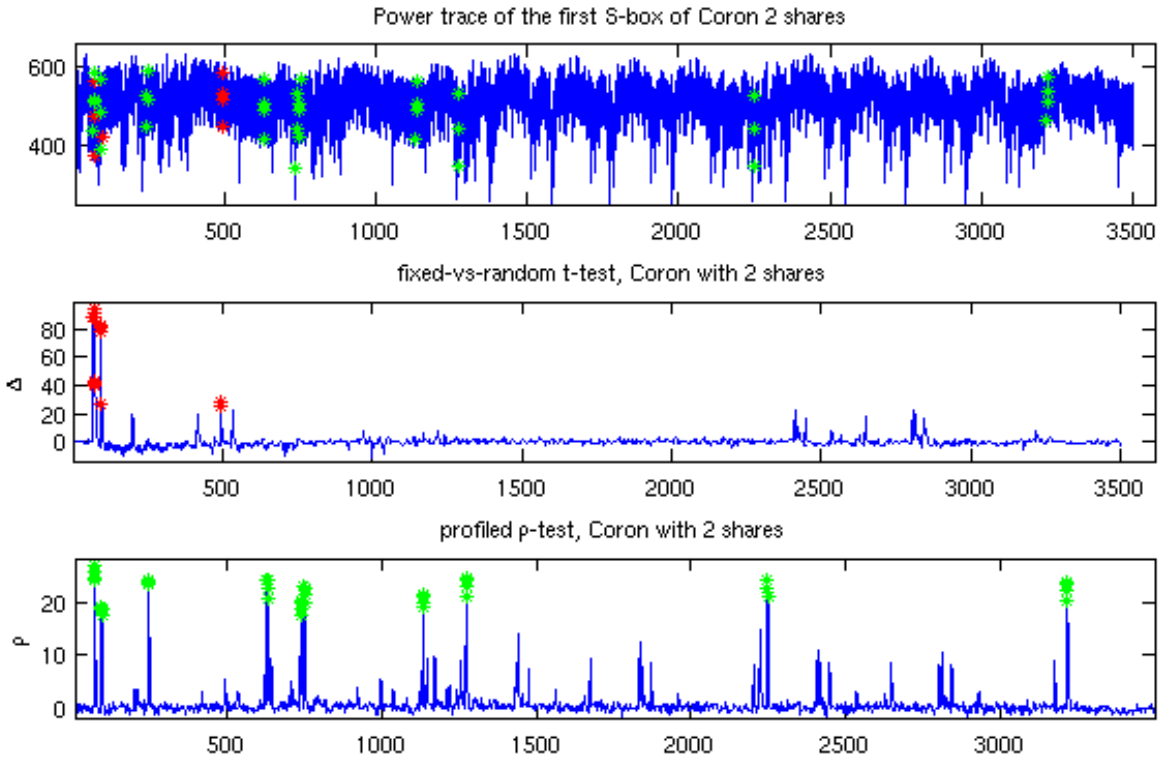


FIGURE 6.8 – Trace of the first S-box of the Coron masking scheme with $d = 1$ and random seeds. The values in green are the ones where $|\hat{\mathrm{t}}| \geq 15$ and the values in red are the values where $|\Delta| \geq 25$

threshold. Those peaks corresponds to the first `GenRdm` call of the second, third and fourth `SecMult` occurring during the first S-box. We tried to attack the scheme at those points with a first-order template attack, but we got no relevant result. Moreover, the profiled ρ-test does not perceive any leakage which comforts us thinking that the peaks are in fact ghost-peaks. This strange because the classic Rivain-Prouff masking scheme with three shares (the flawed one) does not show this behavior. The only difference between these two schemes is a supplementary random byte generated in the corrected version. From these results, we conclude that those peaks do not carry information and should not be taken into account.

- *Genelle masking scheme with 2 shares.* Results are given in FIGURE B.4. In Genelle masking scheme, the S-box is divided in two parts : the `SecDirac` and the inversion in $\mathbb{F}_{2^8}$ with multiplicative shares. `SecDirac` is composed of three parts : the bit matrix transposition (from time sample ~1800 to ~7800), two `SecAnd`'s (from ~7800 to ~9500 and from ~9800 to ~11500) and the xor of the computed Dirac shares with the state (from ~11600 to ~13500). The last part of the trace contains the first three inversions in $\mathbb{F}_{2^8}$. The first part of the trace is the AddRoundKey between the different shares of the plaintext and the key.

  We can clearly see that when the seed is fixed, the t-test shows that the first two matrix transpositions are leaking a lot, but that the following two do not. Actually, that can be explained because the information on the plaintext is in the first share, and the second one is a random that does not change. The rest of the trace leaks as expected. The ρ-test only

captures leakage at time samples when the first byte is processed. This corresponds to a part of the first matrix transposition, the first `SecAnd`, some time samples during the xor between the first share and the Dirac matrix, and during multiplicative shares processing.

When the seed is random, the t-test still captures leakage during the two SecAnd and the inversion in $\mathbb{F}_{2^8}$. The ρ-test captures leakage at similar time samples.

We performed a univariate template attack against both random and fixed seed power traces, and could retrieve the key bytes with the leaking points in the two leakage detection tests. This shows that share recombination is also a problem for Genelle masking scheme.

- *Genelle masking scheme with 3 shares.* Results are given in FIGURE B.5. The S-box is divided in the same way as the previous scheme, with the matrix transposition from time sample ~2000 to ~12000, the two `SecAnd` from ~12000 to ~18000, and the three first instances of multiplicative shares processing between ~21000 and the end of the power trace.

  When the seed is fixed, the same time samples as for the 2 shares implementation are leaking in the t-test and in the ρ-test.

  When the seed is random, the ρ-test does not leak, but we can see three small peaks during the first multiplicative processing in the t-test. We tried to attack these peaks with an univariate template attack, but did not succeed in retrieving the correct key. This comforts us thinking that they are not exploitable (at least with 5000 traces), and that the scheme is first-order secure.

- *Coron masking scheme with 2 and 3 shares.* Finally, for an easier comparison between the different implementations, we joined the same graphs for Coron with 2 shares and with 3 shares in FIGURE B.6 and B.7.

6.5 Conclusion

In this chapter, we provided the tools to analyse the security level of masked AES implementations, and applied them to several state-of-the-art masking schemes. We firstly tried to look at their behavior with the randomness cut off, and we showed that the security of such implementations depends heavily on that aspect. We showed that supposedly first order secure masking schemes are actually easily broken by univariate template attacks because of the shares recombination problem. We also showed that this problem is due to the transition-based leakage models of current MCU's, and that it can be generalized to a degradation of the theoretical d-security level to a practical one of $\left\lceil \frac{d}{2} \right\rceil$. The current solution proposed by [1] is to increase the theoretical security level until the practical desired one is reached. Finally, we explained how to perform a bivariate template attack on a 3 share masking scheme, that can be generalized at higher orders.

Chapter 7

Conclusion

The goal of this master thesis was to answer the following question: "What can an untrusted party with dishonest intentions do to recover a secret key embedded in a device?" We showed that the answer of this question heavily depends on the implementation that is targeted.

All the work produced for this thesis is specifically designed for an 8-bit off-the-shelf MCU not intended for cryptographic purposes. The idea was to use a system as common as possible, and to see how efficient it could be from a security point of view. The ChipWhisperer-Lite fits this description, with low cost and average components. This 8-bit architecture comes with several constraints in terms of clock speed, memory and data size.

We showed that leaving the AES unprotected on such a MCU does not guarantee any kind of security against power analysis attacks. Indeed, it leaks at first order and can be easily broken by different basic attacks using low computational resources. This explains the need for today's cryptographers to create secure masking schemes in order to preserve the privacy of the data.

In practice, those implementations provide a security boost but induce a huge overhead in terms of execution time and resources. We chose to implement some state-of-the-art masking schemes and to compare them in terms of performances and resources needed. The masking scheme from Genelle *et al.* appears to be faster than the three others. This result is confirmed by the literature but must be taken with care because valid only in the context of an 8-bit MCU.

From a security point of view, the different implementations present the same characteristics. First order security is only achievable with at least three shares and enough randomness. This is a direct consequence of the shares recombination problem that occurs because the leakage model of MCUs are mostly transition-based. This can be generalized to a loss of security from a theoretically d-secure implementation to a practical $\lceil \frac{d}{2} \rceil$ security level.

This has led to a lazy engineering trend, which consists in increasing the security order of the implementations to get a system that is in practice secure at the right order. The problem of such an approach is that the execution time and required randomness of this type of scheme grows exponentially with the security order. Good security becomes very expensive, which limits the practical use for such implementations.

This thesis only presented very specific types of masking at limited order, mostly based on operations in $\mathbb{F}_{2^8}$. To go further, one could try to implement either these masking schemes at higher order, to find schemes based on different properties or to use different platforms, that are less limiting. A lot of the newest trends in masking come for hardware designs, with either dedicated ASICs or FPGAs where the limitations are very different. New set-ups might include the capability to record a huge number of traces, parallelism of instructions or better process and recording speed.

Embedded security is a trending topic in the cryptographic world, that has real implications in both the academic and the industrial worlds with a large set of applications, and full of potential for future improvements.

Bibliography

[1] Balasch Josep, Gierlichs Benedikt, Grosso Vincent, Reparaz Oscar and Standaert François-Xavier. *On the Cost of Lazy Engineering for Masked Software Implementations* (extended version), in the proceedings of CARDIS 2014, Lecture Notes in Computer Science, vol 8968, Springer, pp 64-81, Paris, France, November 2014.

[2] Barthe Gilles, Belaïd Sonia, Dupressoir François, Fouque Pierre-Alain, Grégoire Benjamin and Strub Pierre-Yves. *Verified Proofs of Higher-Order Masking* in IACR-EUROCRYPT-2015, 2015.

[3] Bol David. *Power and energy consumption of digital circuits*, Chap. 1 in PhD, 2008.

[4] Brier Eric, Clavier Christophe and Olivier Francis. *Correlation Power Analysis with a Leakage Model* in Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop., pp 16-29, 2004.

[5] Carlet Claude, Goubin Louis, Prouff Emmanuel, Quisquater Michaël and Rivain Matthieu. *Higher-order masking schemes for s-boxes* in: Canteaut, A. (ed.) FSE 2012. LNCS, vol. 7549, pp. 366–384. Springer, Heidelberg, 2012.

[6] Chari Suresh, Jutla Charanjit S., Rao Josyula R., and Rohatgi Pankaj. *Towards Sound Approaches to Counteract Power-Analysis Attacks* in M. J. Wiener, editor, CRYPTO, volume 1666 of LNCS, pages 398–412. Springer, 1999.

[7] Coron Jean-Sébastien, Prouff Emmanuel, Rivain Matthieu and Roche Thomas. *Higher-Order Side Channel Security and Mask Refreshing*, 2014.

[8] Daemen Joan and Rijmen Vincent. *The Advanced Encryption Standard*, in FIPS-197, 2001.

[9] Duc Alexandre, Dziembowski Stefan and Faust Sebastian. *Unifying leakage models: from probing attacks to noisy leakage* in Cryptology ePrint Archive, Report 2014/079, 2014.

[10] Durvaux François and Standaert François-Xavier. *From Improved Leakage Detection to the Detection of Points of Interests in the Leakage Traces* in EUROCRYPT2016, Part I, pp.240-262, 2016.

[11] Genelle Laurie, Prouff Emmanuel and Quisquater Michaël. *Montgomery's Trick and Fast Implementation of Masked AES* in AFRICACRYPT 2011, pp. 153-169, 2011.

[12] Genelle Laurie, Prouff Emmanuel and Quisquater Michaël. *Thwarting Higher-Order Side Channel Analysis with Additive and Multiplicative Maskings* in CHES 2011, pp. 240-255, 2011.

[13] Grosso Vincent, Standaert François-Xavier and Faust Sebastian. *Masking vs. Multiparty Computation: How Large is the Gap for AES?* (extended version) in the Journal of Cryptographic Engineering, vol 4, num 1, pp 47-57, Springer, April 2014.

[14] Goodwill Gilbert, Jun Benjamin, Jaffe Josh and Rohatgi Pankaj. *A testing methodology for side-channel resistance validation* in NIST Non-Invasive Attack Testing Workshop, 2011.

[15] Ishai Yuval, Sahai Amit and Wagner David. *Private Circuits: Securing Hardware against Probing Attacks.* in D. Boneh (editor) Advances in Cryptology – CRYPTO 2003, volume 2729 of Lecture Notes in Computer Science, pages 463–481. Springer, 2003.

[16] Joye Marc, Paillier Pascal and Schoenmakers Berry. *On Second-Order Differential Power Analysis* in CHES 2005, Lecture Notes in Computer Science 3659, Springer-Verlag, pp. 293-208, 2005.

[17] Katz Jonathan and Lindell Yehuda. *Introduction to Modern Cryptography : Second Edition*, CRC Press, 2015.

[18] Kim HeeSeok, Hong Seokhie and Lim Jongin. *A Fast and Provably Secure Higher-Order Masking of AES S-Box* in: Preneel B., Takagi T. (eds) Cryptographic Hardware and Embedded Systems – CHES 2011. CHES 2011. Lecture Notes in Computer Science, vol 6917. Springer, Berlin, Heidelberg, 2011.

[19] Kocher Paul, Jaffe Joshua and Jun Benjamin. *Differential Power Analysis* in Advances in Cryptology - Proceedings of Crypto '99, Lecture Notes in Computer Science, Vol. 1666, Springer-Verlag, pp. 388-397,1999.

[20] Mangard Stefan, Popp Thomas and Gammel Berndt M. *Side-Channel Leakage of Masked CMOS Gates* in Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 351–365. Springer, Heidelberg, 2005.

[21] Mangard Stefan, Oswald Elisabeth and Popp Thomas. *Power analysis attacks - revealing the secrets of smart cards*, Springer, 2007.

[22] Mangard Stefan, Oswald Elisabeth and Standaert François-Xavier. *One for All - All for One: Unifying Standard DPA Attacks* in IET Information Security, vol 5, issue 2, pp 100-110, June 2011.

[23] Prouff Emmanuel and Rivain Matthieu. *Masking against Side-Channel Attacks: A Formal Security Proof* in: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 142–159. Springer, Heidelberg, 2013.

[24] Rivain Matthieu and Prouff Emmanuel. *Provably Secure Higher-Order Masking of AES*, in: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 413–427. Springer, Heidelberg, 2010.

[25] Standaert François-Xavier, Veyrat-Charvillon Nicolas, Oswald Elisabeth, Gierlichs Benedkit, Medwed Marcel, Kasper Markus and Mangard Stefan. *The world is not enough: another look on second-order DPA* in: Abe, M. (eds) ASIACRYPT, Lecture Notes in Computer Science, vol. 6477, pp. 112–129. Springer, Berlin, 2010.

[26] Standaert François-Xavier. *How (not) to Use Welch's T-test in Side-Channel Security Evaluations* (not published yet), 2017.

[27] Zhang Hailong. *How to Effectively Decrease the Resource Requirement in Template Attack?* in M. Yoshida and K. Mouri (Eds.): IWSEC 2014, LNCS 8639, pp. 119–133, Switzerland, 2014.

Appendices

Appendix A

Operations in $\mathbb{F}_{2^8}$

This appendix is largely inspired from [8].

A.1 Notations

We use three equivalent notations in this paper to represent a *byte*, which is a group of 8 bits.

- The *binary notation*, 0b10010110 or 10010110_2
- The *hexadecimal notation*, 0x96 or {96}
- The *polynomial notation*, $x^7 + x^4 + x^2 + x$

Operations on bytes are defined as the operations on the *finite field* $\mathbb{F}_{2^8}$, also called the *Galois field* $GF(2^8)$. In the notation $GF(a^b)$ or $\mathbb{F}_{a^b}$, a is called the *characteristic* and b the *order* of the field.

The field $GF(a^b)$ contains a^b different elements of the form $\sum_{i=0}^{b-1} \alpha_i x^i$ where $\alpha_i \in [0, a-1]$. As an example, the 16 elements of $GF(2^4)$ are:

$0, 1, x, x{+}1, x^2, x^2{+}1, x^2{+}x, x^2{+}x{+}1, x^3, x^3{+}1, x^3{+}x, x^3{+}x{+}1, x^3{+}x^2, x^3{+}x^2{+}1, x^3{+}x^2{+}x, x^3{+}x^2{+}x{+}1$

A.2 Addition

The addition in $GF(2^8)$ is performed modulo 2, which is equivalent to the exclusive or operation, called the `xor` and denoted by $\oplus$.

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Example :

- 0b11100011 $\oplus$ 0b00011111 = 0b11111100
- 0xE3 $\oplus$ 0x1F = 0xFC
- $(x^7 + x^6 + x^5 + x + 1) \oplus (x^4 + x^3 + x^2 + x + 1) = x^7 + x^6 + x^5 + x^4 + x^3 + x^2$

A.3 Multiplication

Multiplication in $GF(2^8)$ (that we denote $\odot$) corresponds to a multiplication of two polynomials modulo an *irreducible polynomial*. We will be interested in the polynomial specific to AES. This polynomial is denoted $m(x)$ and is

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

or 0x011B. It maps any element of higher order than 7 to elements of $GF(2^8)$.

Example : 0x57$\odot$ 0x83 = 0xC1 because

$$(x^6 + x^4 + x^2 + x + 1) \odot (x^7 + x + 1)$$

$$= x^{13} + x^{11} + x^9 + x^8 + \underbrace{x^7 + x^7}_{=0} + x^6 + x^5 + x^4 + x^3 + \underbrace{x^2 + x^2}_{=0} + \underbrace{x + x}_{=0} + 1 \pmod{x^8 + x^4 + x^3 + x + 1}$$

$$= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \pmod{x^8 + x^4 + x^3 + x + 1}$$

$$= ((x^5 + x^3) \odot (x^8 + x^4 + x^3 + x + 1)) + (x^7 + x^6 + 1) = x^7 + x^6 + 1$$

The multiplication by x of a polynomial $b(x) \in \mathbb{F}_{2^8}$ corresponds to a shift to the left if the *Most Significant Bit* (MSB) of $b(x)$ is zero (i.e. if $b_7 = 0$) or to a shift followed by a xor operation with 0x1B if the MSB is one.

Appendix B

Results

In this annex, we present the different results we get from the leakage detection methods at first order, for each implementation and at each order d in the following fashion:

1. The power trace of the first S-box of the masked implementation
2. The Welch's t-test for random seeds
3. The profiled ρ-test for random seeds
4. The Welch's t-test for a fixed seed
5. The profiled ρ-test for a fixed seed

Here is the list of the masked implementations available in this appendix:

- The results of Rivain-Prouff with $d = 1$ are given in Figure B.1
- The results of Rivain-Prouff with $d = 2$ are given in Figure B.2
- The results of Rivain-Prouff (corrected with the ISW refresh mask) with $d = 2$ are given in Figure B.3
- The results of Genelle with $d = 1$ are given in Figure B.4
- The results of Genelle with $d = 2$ are given in Figure B.5
- The results of Coron with $d = 1$ are given in Figure B.6
- The results of Coron with $d = 2$ are given in Figure B.7

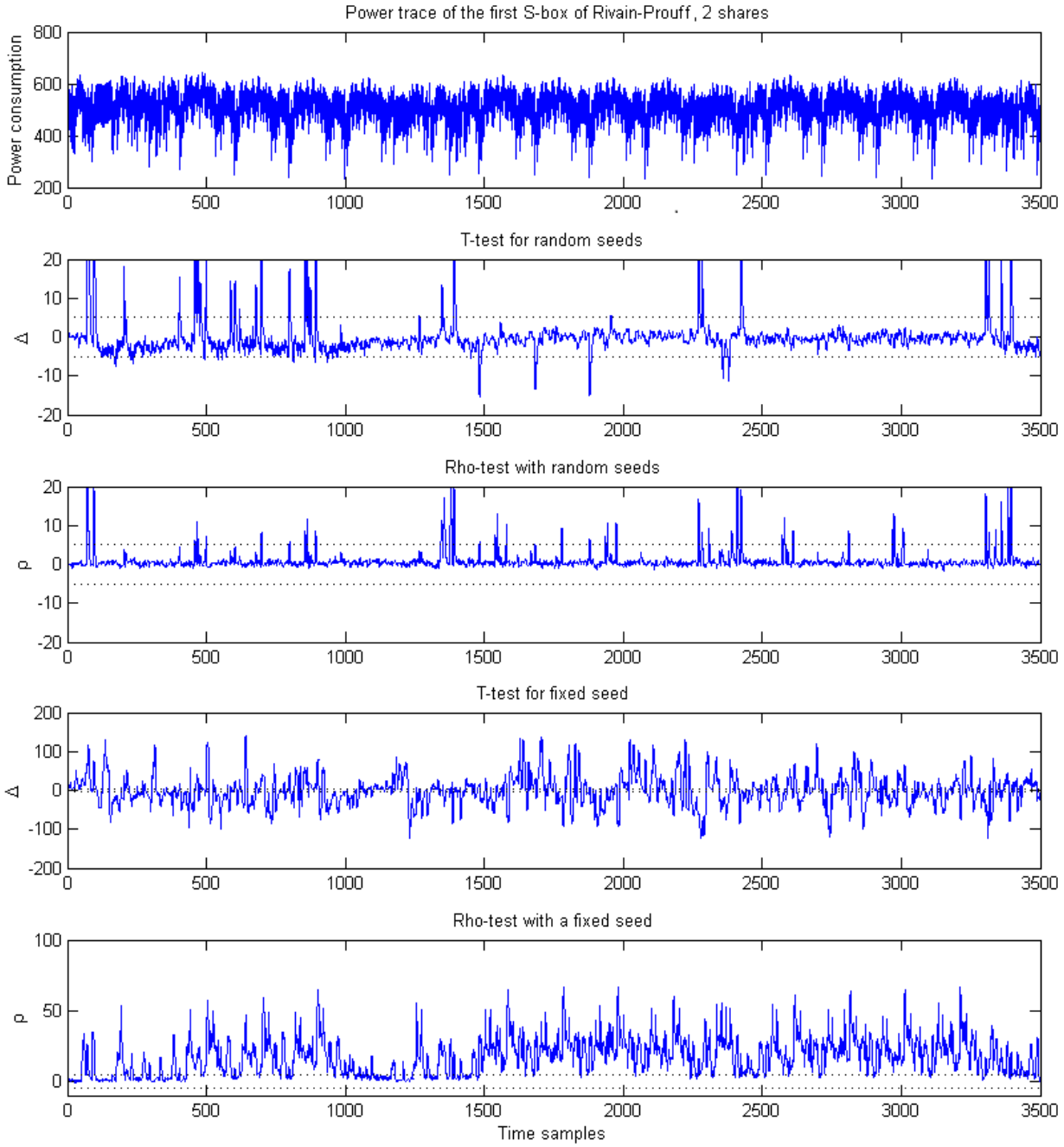


FIGURE B.1 – First order leakage evaluation for Rivain-Prouff implementation with 2 shares

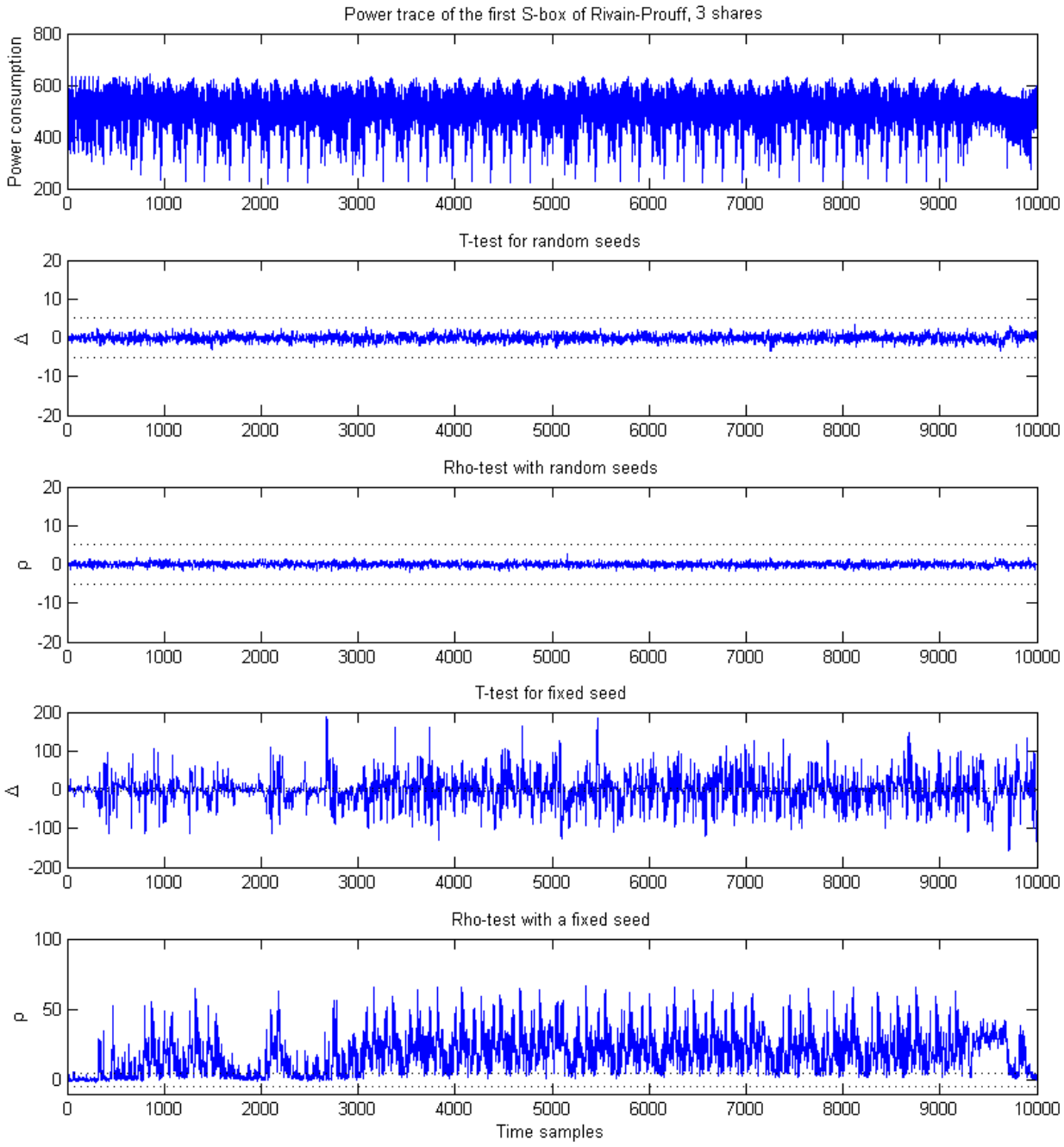


FIGURE B.2 – First order leakage evaluation for Rivain-Prouff implementation with 3 shares

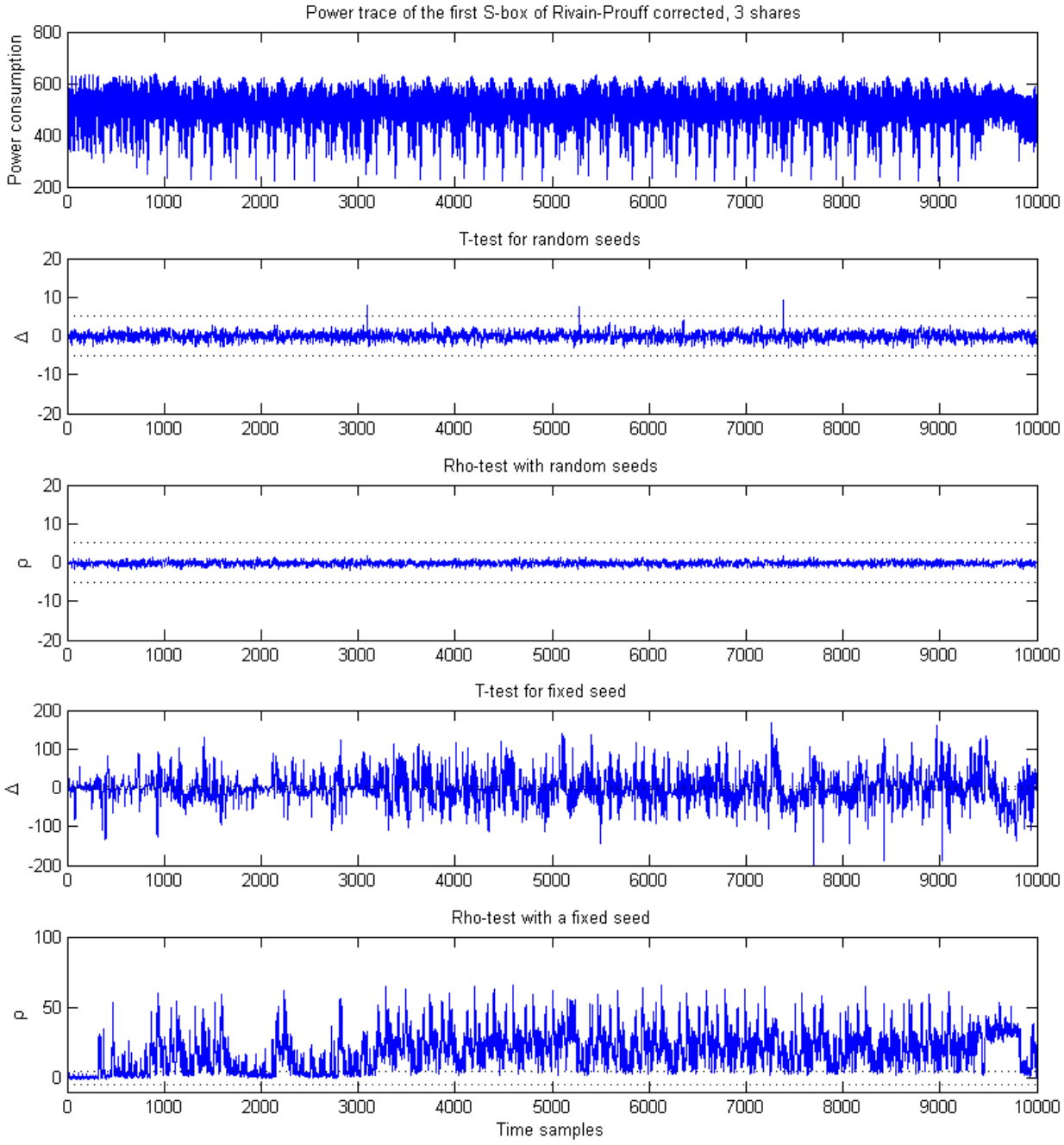


FIGURE B.3 – First order leakage evaluation for Rivain-Prouff implementation with 3 shares and the refresh mask of ISW

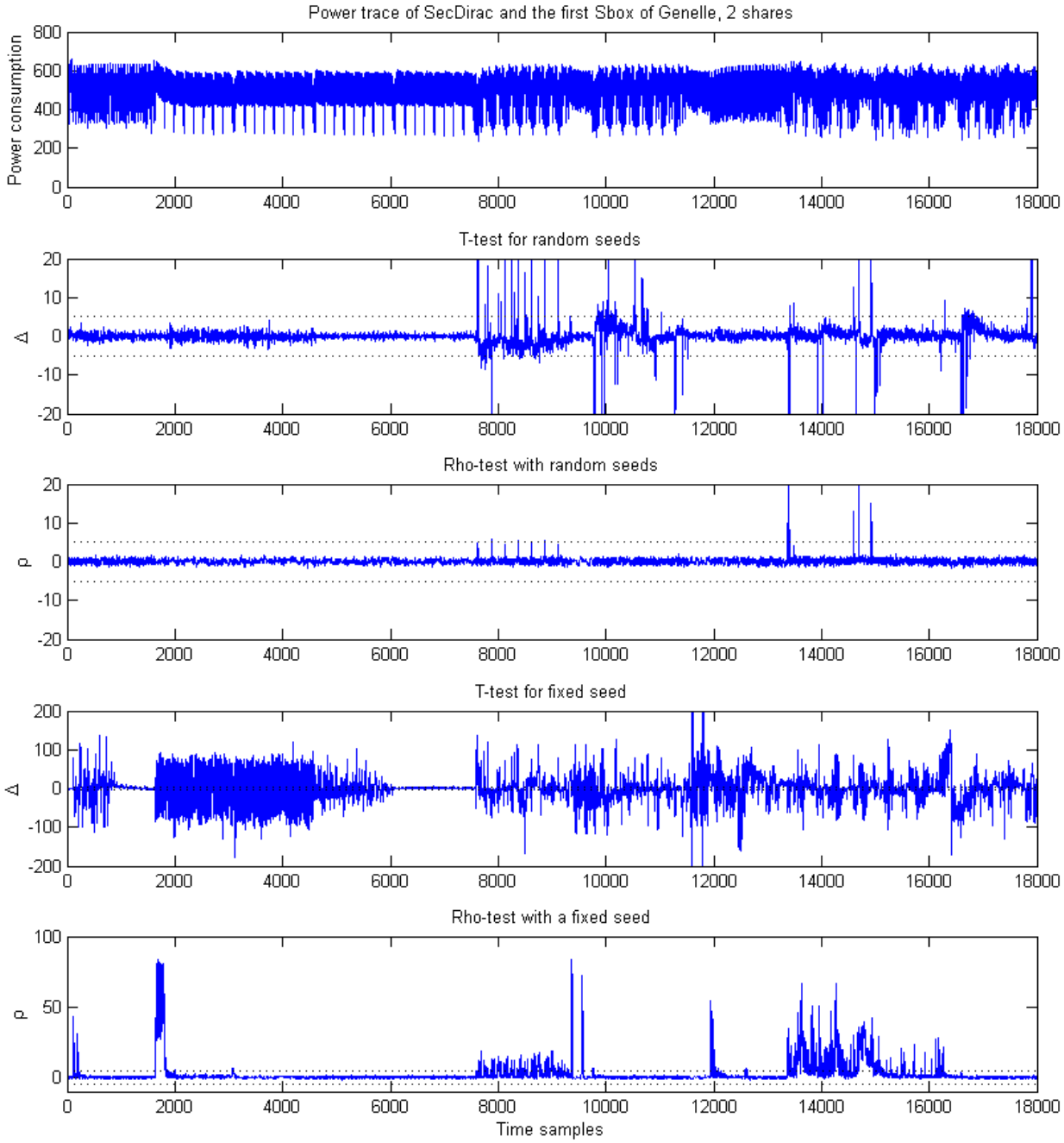


FIGURE B.4 – First order leakage evaluation for Genelle implementation with 2 shares

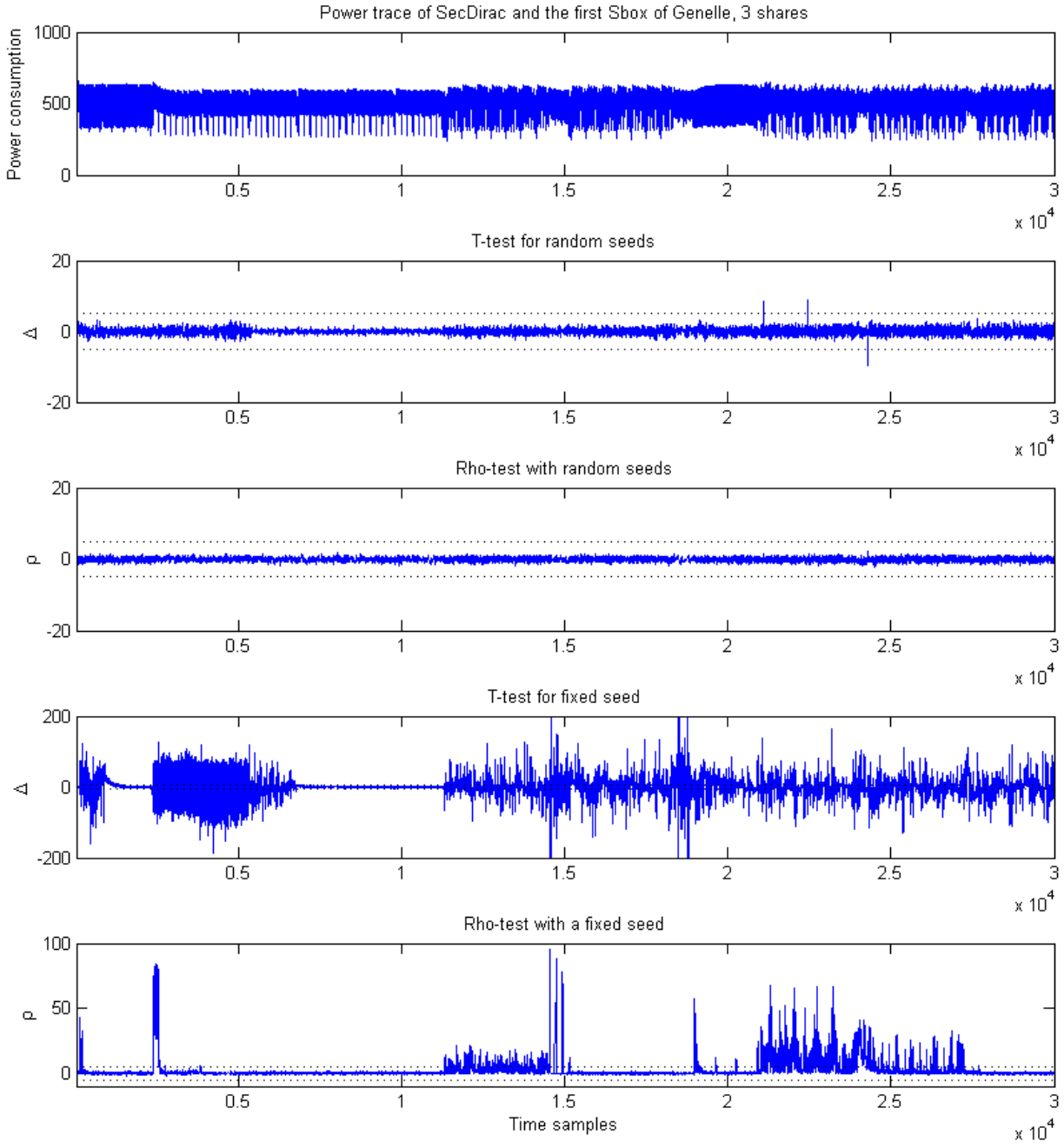


FIGURE B.5 – First order leakage evaluation for Genelle implementation with 3 shares

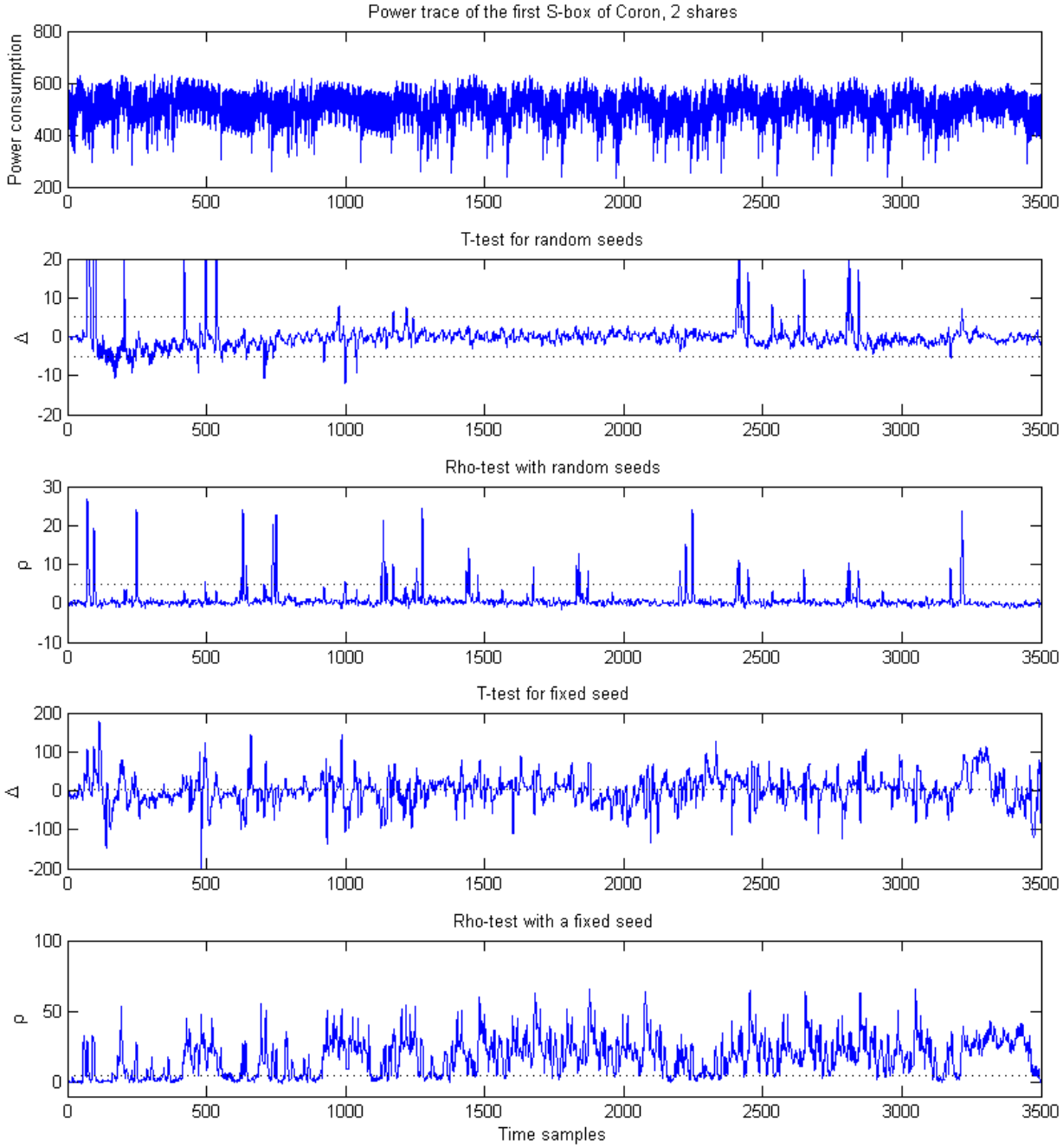


FIGURE B.6 – First order leakage evaluation for Coron implementation with 2 shares

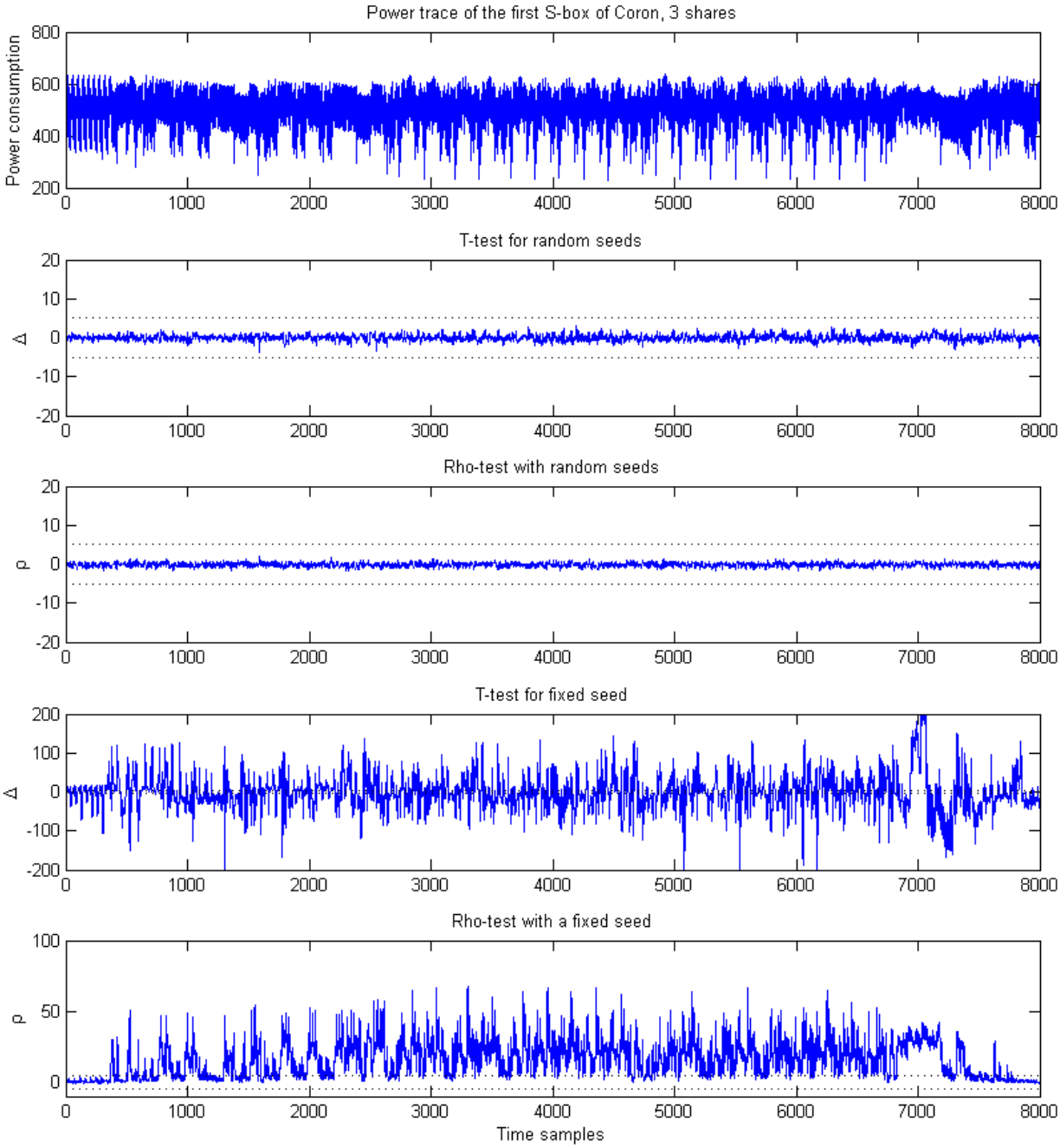


FIGURE B.7 – First order leakage evaluation for Coron implementation with 3 shares

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**