UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Fault and side-channel resistant devices using non-linear robust codes and higher-order masking

Application to AES

Dissertation presented by
Georges-Henri LECLERCQ

for obtaining the Master's degree in
Electrical Engineering

Supervisor
François-Xavier STANDAERT

Readers
Jean-Didier LEGAT, Romain POUSSIER

Academic year 2015-2016

Contents

List of Abbreviations

ADV	**A**dditive **D**igest **V**alue
AES	**A**dvanced **E**ncryption **S**tandard
ARK	**A**dd **R**ound **K**ey
DFA	**D**ifferential **F**ault **A**nalysis
DPA	**D**ifferential **P**ower **A**nalysis
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DRP	**D**ual **R**ail **P**recharge
FA	**F**ault **A**ttack
GF	**G**alois **F**ield
GPQ	**G**enelle-**P**rouff-**Q**uisquater
HW	**H**amming **W**eight
ISW	**I**sshai **S**ahai **W**agner
KHL	**K**im-**H**ong-**L**im
LSB	**L**east **S**ignificant **B**yte
MDV	**M**ultiplicative **D**igest **V**alue
MIPS	**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
MSB	**M**ost **S**ignificant **B**yte
NIST	**N**ational **I**nstitute of **S**tandards and **T**echnology
NSA	**N**ational **S**ecurity **A**gency
PRNG	**P**seudo **R**andom **N**umber **G**enerator
RAM	**R**andom **A**ccess **M**emory
RP	**R**ivain-**P**rouff
RSA	**R**ivest-**S**hamir-**A**dleman
SB	**S**ub **B**ytes (also **S**-**B**ox)
SCA	**S**ide **C**hannel **A**nalysis
SPA	**S**imple **P**ower **A**nalysis
SR	**S**hift **R**ows
SRAM	**S**tatic **R**andom **A**ccess **M**emory
XOR	**E**xclusive-**OR**
ZT	**Z**ero **T**est

Chapter 1

Introduction

This master thesis focuses on securing embedded cryptographic devices against state-of-the-art physical attacks. Among them, side-channel attacks (SCA) and fault attacks (FA) are probably the most common ones. While their nature is different in the way they are put in practice, we can think of a combined approach to protect implementations of cryptographic algorithms against them. Surprisingly, very few (at least not to our knowledge) scientific literature deals with such a combination of defensive mechanisms. It appears that, even though those two types of attack are serious threats, secure chip vendors rely more on the creativity of their engineers than on proofs to ensure a complete protection of their circuits.

As cryptanalysis lies between mathematics, electricity and computer science, there are as many ways to attack -or protect- hardware and software implementations of cryptographic primitives. We will target here algorithmic countermeasures to these attacks and therefore focus more on software implementations, although we will try to keep things as generic as possible. Being more specific somewhat simplifies the attacker model and may lead to security holes. Interestingly though, this desire to see things from both hardware and software point-of-views led us to unexpected discoveries.

Computer science may be the primary aspect of this work, as it requires to dig into low-level architectures to implement software countermeasures (in C and Assembly codes). Specifically, Assembly code allows us to analyse the impact of any basic operation in terms of performance and security. This is an important point as we will try to measure the quality of our results in terms of both these aspects.

Mathematics, on the other hand, are at the heart of the cryptography and are necessary to understand the attacks that we want to protect from. They also justify the need for probabilistic countermeasures and are the most important tool when comparing security levels.

Last but not least, electrical aspects of the cryptographic devices is the reason all these attack exist. Often unconsidered, this is why real-world implementations are still vulnerable even when mathematically secure algorithms are used.

Throughout this work, we will study how to protect some particular devices running a specific algorithm : the Advanced Encryption Standard (AES). This is one of the most widely used ciphers and it is part of our daily lives, for instance in modern smart cards.

It is still planned to be used in the future as it fulfils modern security requirements as a cryptographic primitive. This justifies our goal of removing weaknesses that are part of its implementation in real-world devices.

As mentioned before, secure devices that are designed and manufactured probably deal with side-channel and fault attacks. However, very little is known about combining these protections, and the answer is not trivial, especially in terms of performance. Using solutions from the scientific literature, we will propose a combined scheme that achieves some defined level of security and analyse its cost. It will highlight some concerns that arise when combining different countermeasures. We will also show that the kind of countermeasures we selected to protect from fault attacks, which is often mentioned in articles and books, presents a critical weakness.

This master thesis is articulated as follows. Chapter 2 presents the theoretical background that is required to understand the rest of this thesis. It contains aspects of cryptography, attacks and countermeasures. Chapter 3 is the core of this work. It describes our countermeasures, and what we propose as a combined solution, and assesses its generic performance. It also shows a security hole in the fault countermeasure. Chapter 4 presents our results in terms of performance and relates some aspects of algorithmic design, while keeping the security order as a parameter. Chapter 5 illustrates how a real-world fault attack setup can be mounted from scratch and confirms the necessity of countermeasures for faults. It also validates the security hole that we expose in chapter 2.

Chapter 2

Theoretical background

In this chapter, we give a general overview of the most important concepts related to this master thesis. The objective is to bring the reader insights of modern cryptography. Specifically, the last sections of this chapter contain some of the last advances in the related topics.

2.1 Cryptography

Cryptography is the science of secure communications. It involves legitimate users that want to communicate in a secure fashion over an untrusted channel. This channel can be the target of ill-intentioned attackers. This is why people have worked on ways to prevent this attacker to succeed. The main security goals are **integrity**, **confidentiality** and **authenticity**.

- **Integrity** is the concern of having the intended payload, as transmitted, at the output of a communication system. We can identify it as being the *representational faithfulness* of data. Several characteristics of integer data can be pointed out :the data should be complete, timely, accurate and valid [1]. To ensure integrity, one can use *error-detecting codes* (hashes, checksums, ...) or *error-correcting codes* (redundant bits with minimum distance decoding, Turbo codes, LDPC, ...). Integrity is broken if an attacker can *modify* parts of a message.
- **Confidentiality** is the need to keep information private. In untrusted communication channels, this is probably the most important reason for using cryptography. No information (in the information-theoretical sense) should be leaked from a ciphered plaintext without knowledge of a secret. This secret is often related to a virtual *key* that only authorized people have. Confidentiality is broken if an attacker can *understand* or *guess information* in a message.
- **Authenticity** is the ability to validate the origin of a message. Messages can be signed by their original senders, then the receivers should be able to check the validity of it. Along with integrity, this notion can be related to *trust*. The most common way to prove the authenticity of a message is a *signature*. Authenticity is broken when an attacker successfully *impersonates* a legitimate user.

2.2 Symmetric-key cryptography

In symmetric-key cryptography, two or more users share a common *secret key* that is used to cipher and decipher messages. The most simple example of this is Vernam's One-time pad [2], where a binary message is XORed (exclusive-ored) with a binary mask of the same length (the key). A symmetric-key algorithm should at least provide *confidentiality* to its users. The most important drawback of these algorithms is the need for all the users to share this secret key before sending messages.

2.3 The Advanced Encryption Standard (AES)

The Advanced Encryption Standard is a symmetric-key block cipher (a block cipher operates on fixed-size blocks of data for encryption and decryption). It is the result of an international contest that aimed at obtaining the best symmetric cipher. It was won by Joan Daemen and Vincent Rijmen (two researchers from the KUL in Belgium). Their winning cipher is called "Rijndael" by concatenation of their names. It is the most popular block cipher in the world and is recommended by security authorities such as the NIST or the NSA.

The AES can use 128-, 192- and 256-bit keys and operates on 128-bit blocks. All those versions are reliable when used in their complete version but longer keys provide stronger security against some kinds of attacks. Usually, attackers target a round subkey (see below) and then immediately derive the master key. When dealing with AES-256, the master key derivation requires two subkeys, making attacks harder.

The AES is composed of rounds (iterations) during which the same operations are repeated. 10, 12 or 14 rounds are needed depending on the key size, corresponding to 128-, 192- and 256-bit keys. Figure 2.1 represents the algorithm behind the encryption of one block. All the rounds operate on the *AES state* matrix, which is a representation of the 16 bytes of the block that are being encrypted. The decryption process is the inverse of this process (excepted for the key expansion which is always done at the beginning or during the execution), including inverse operations. We describe below what lies behind these boxes. The algorithm is fully specified in the original AES paper [3].

2.3.1 Key expansion

The goal of this first step (also called *key schedule*) is to generate 128-bit *round keys* from the initial master key. *n+1* round keys are created, one for each of the *n* rounds plus one for the initial key addition (which is the raw master key). The generated round keys are transformations of the master key, but they are not independent : for AES-128,

[1] `http://developer.amd.com/resources/documentation-articles/articles-whitepapers/bulk-encryption-on-gpus/`

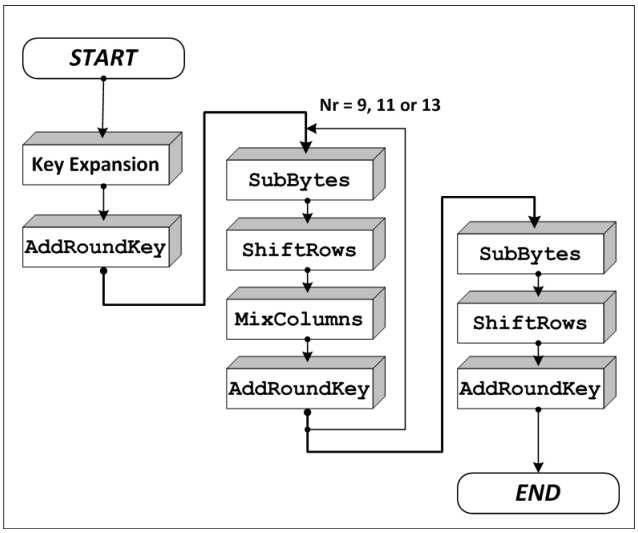

FIGURE 2.1: The encryption process of the AES [1]

a single round key reveals the entire master key, whereas 2 round keys are needed to recover the full AES-256 key.

2.3.2 AddRoundKey

This method simply XORs the state matrix with the corresponding round key. The state matrix is initially filled with the plaintext to be encrypted and then XORed with the initial master key. The subsequent calls use the generated round keys for each round.

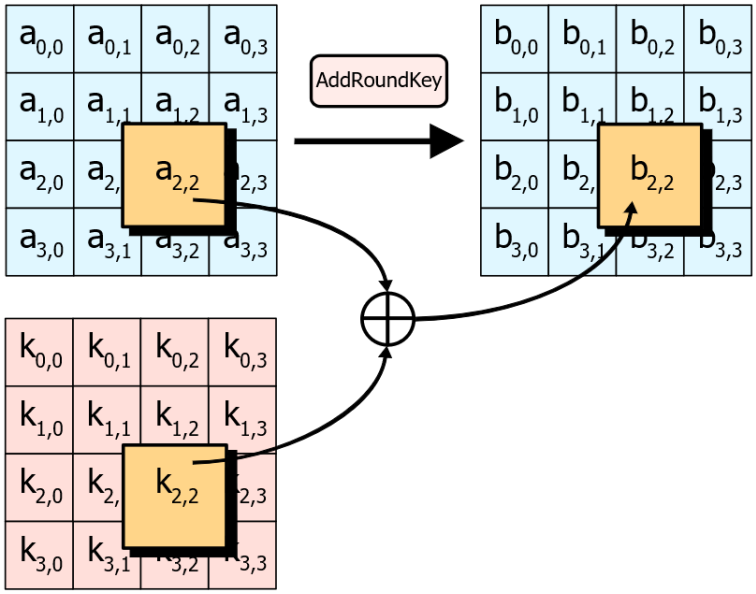

FIGURE 2.2: AddRoundKey transformation. Source : [4]

2.3.3 SubBytes

The SubBytes method applies on each state byte independently. It combines an inversion in *GF(2^8)* and an affine transformation. It is therefore a non-linear process. To speed it up , implementations use *Substitution Boxes* which are arrays that map each byte input to its mathematical result.

The S-Box maps all the elements in $GF(2^8)$ (the state bytes, with their MSB indexed to x_0 and their LSB mapped to x_7) with their multiplicative inverse : $GF(2^8) = GF(2)[x]/(x^8 + x^4 + x^3 + x + 1)$ (zero is mapped to zero). Then, an affine transformation is applied :

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

In this notation, the vector **x** is the multiplicative inverse of the original input.

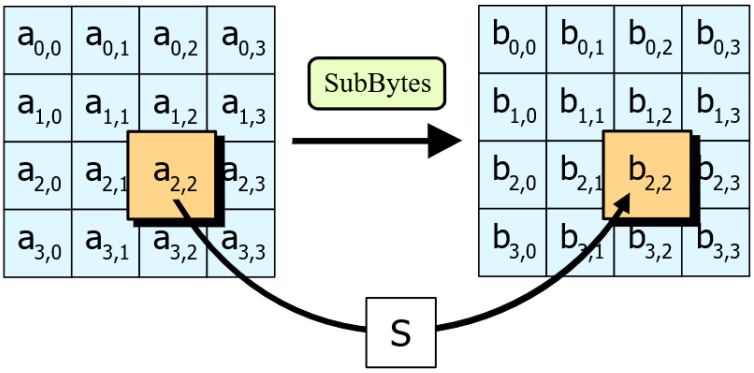

FIGURE 2.3: SubBytes transformation. Source : [4]

2.3.4 ShiftRows

This method operates on the rows of the state matrix. It cyclically shifts rows with different offsets. The offsets are 0, 1, 2 and 3 for the rows 1, 2, 3 and 4.

2.3.5 MixColumns

MixColumns considers the columns of the state as polynomials over *GF(2^8)* and multiplies them with a fixed polynomial *c(x)*, modulo $x^4 + 1$. c(x) is given by :

$$c(x) = 03x^3 + 01x^2 + 01x + 02$$

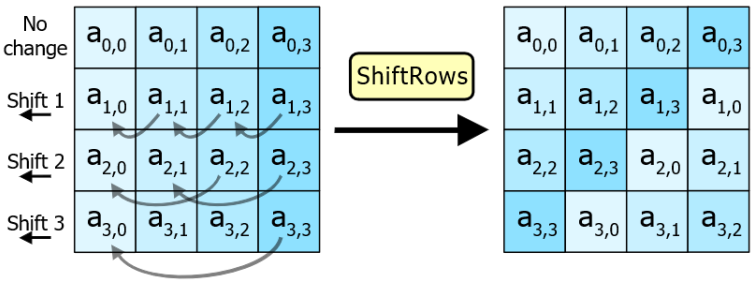

FIGURE 2.4: ShiftRows transformation. Source : [4]

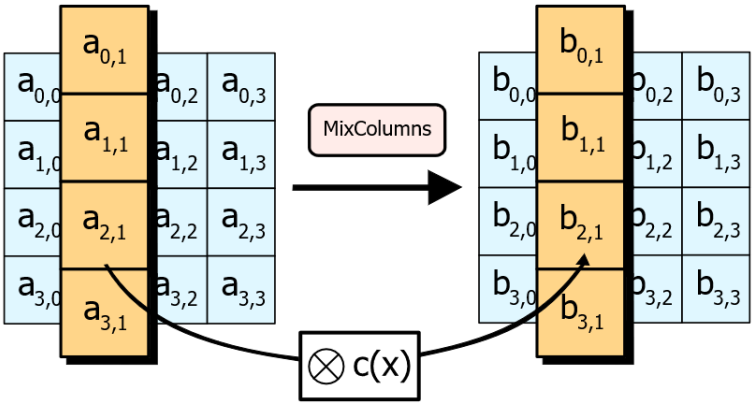

FIGURE 2.5: MixColumns transformation. Source : [4]

2.3.6 Mathematical notations

Here, we define notations that will be used throughout this chapter. We give the formalism that is common to all substitution ciphers and precise this model for the AES:

- The round function is denoted by

$$\rho(r) = \sigma_k[K^r] \circ \theta_r \circ \gamma_r$$

  where:

- r is the round number
- K^r is the r^{th} round key, with a total of R rounds (In AES-128, $R = 10$)
- γ_r is the substitution layer (SubBytes or S-Box in the AES)
- θ_r is the so-called diffusion layer, corresponding to ShiftRows and MixColumns in the AES. Diffusion means that there is data dependency between some state bytes at the input of ShiftRows and some bytes at its output.
- $\sigma_k[K^r]$ is the key addition (AddRoundKey in the AES) step. Formally,

$$\sigma[K](a) = b \Leftrightarrow b_j = a_j \oplus k_j, 1 \leq j \leq n$$

The last round omits the diffusion layer as it has no cryptographic significance. Moreover, a first key addition is performed before the first round. The whole Substitution-Permutation cipher encryption is given by :

$$E_k = \sigma_k[K^r] \circ \gamma_r\big(\bigcirc_{r=1}^{R-1} \sigma_k[K^r] \circ \theta_r \circ \gamma_r\big) \circ \sigma_k[K^0]$$

Where $\circ$ is the composition operator.

2.4 Side-channel attacks on cryptographic devices

Even though algorithms such as the AES are considered to be secure in their design, there is a difference between the theory of a cryptographic primitive and its implementation. In cryptanalysis, *side-channels* are a target of choice to break cryptographic systems. Those side-channels attacks focus on the information leaked by the environment that runs the cipher, such as radiation, power consumption, hardware leakage, timing analysis,... We explain those in which we are interested below. Some of the content of this section is inspired from the reference book "Power Analysis Attacks : Revealing the Secrets of Smart Cards" from Oswald, Mangard and Popp [5].

2.4.1 Power analysis attacks

Simple Power Analysis

In simple power analysis attacks, the attacker measures the current or power consumption of the device in order to determine key bits. The goal, as usual, is to recover the whole secret key.

A SPA only needs one or a few power traces. The objective is to find key-dependant consumption traces. Usually, the attacker knows the algorithm that he is attacking. He tries to establish a correlation between the power consumption and the corresponding operations (AND, XOR,logical operations such as a multiplication, jumps,...).

A simple example of SPA is the "square and multiply" attack applied on the RSA cipher. RSA makes conditional jumps depending only on the value of the key at some points. It will parse the binary representation of the key and make a multiplication only when the tested key bit is '1'. By identifying the power consumption of a multiplication on the device, one can directly guess the key by analysing timing and power consumption when performing the "square and multiply" operation.

Differential Power Analysis

In contrast to the above-mentioned SPA, DPA focuses on the data dependency of a large set of power traces at a given point in time. This is a more practical attack as it can be performed even on noisy traces.

The attacker performing a DPA proceeds in several steps:

- The attacker first selects an intermediate value of the cryptographic algorithm that he could exploit to reveal the key. Thus, this value should be a function of known data such as the plaintext, ciphertext and/or the key.
- The attacker proceeds to power measurements while running the algorithm with a fixed key and known plaintexts. The number of needed plaintexts depends on the complexity of the attack.
- The attacker computes all possible results for the intermediate value that he has chosen using the known relation between the known plaintext/ciphertext and the value. It will often depend on a (small) part of the key to make the attack easier (divide-and-conquer principle).
- The attacker modelizes and maps the intermediate results to power consumption traces. To achieve this, the attacker must choose a consumption model. The most popular ones are:
  - The Hamming Distance model : the consumption is related to the number of bit transitions
  - The Hamming Weight model : the consumption is related to the number of bits set to 1
  - The Zero Value model : the consumption is related to the number of bits set to 0

  Furthermore, if the attacker owns a device that he can characterise, he can perform a template attack. A template attack is a powerful method in which the attacker characterises a device that is similar to the target and collects relevant power trace points when using typical operations. He then runs the target device and tries to match the obtained power trace with his templates. Kocher *et al.* were among the first to discuss the application of such attacks to cryptographic devices [6].

  The drawback of template attacks is the need to own a copy of the device that can be programmed or modified to obtain the desired template traces.
- In the last step, the attacker tries to establish a statistical link (hypothesis test) between the simulated power traces and the measured ones. If he succeeds in doing so, he can reveal the key bits that he targeted.

The key points in this attack are to choose an adapted intermediate result and to reduce as much as possible the measurement noise (to make the statistical tests more powerful).

2.5 Fault Attacks

Fault attacks were first introduced in 1996 when Boneh *et al.* [7] found a new way to attack the RSA algorithm. It was first only applicable to public-key algorithms, but soon after, Biham and Shamir discovered similar attacks for DES and symmetric-key algorithms in [8]. This started a new challenge for security engineers and it is still an open problem nowadays.

Fault attacks are performed by perturbing the normal operation of a processor or cryptographic circuit in order to reveal information related to the secret key. This includes modification of the temperature, supply current, making clock glitches and radiating the circuit (e.g. with a laser). The goal is to alter either the process flow or the data. If the target is the data, the attacker then tries to analyse the output ciphertext with or without knowledge of the plaintext (*Differential Fault Analysis, DFA*). If the target is the process flow, this may lead to insecure implementations of cryptographic algorithms, that are weaker against other attacks. For instance, on attacker could target the round counter of AES, and set it to zero. The only operations that the algorithm will perform is XORing the plaintext with the initial master key and then output it as a ciphertext. If the plaintext is known, then the key is revealed.

2.5.1 Fault Models

When designing a cryptographic system, one should consider protection against DFA. This is not a trivial task as there is no way to protect a system against all errors. Furthermore, increasing resistance against faults often leads to overheads in chip space or/and computation time. For each kind of fault, there exists associated countermeasures, but the designer should assess the cost of the protection versus the security that it brings.

Specifically, there are several criteria that need to be thought about regarding the power of the attacker that we want to protect from :

- **Effect of the fault** : transient (only during the exposure with the attacking device), permanent or even destructive
- **Number of affected bits** : a single bit (very powerful), a whole byte/word or a random number of bits
- **Type of fault** : Stuck-at faults (the bit is forced to 0 or 1), bit flip, random fault
- **Multiplicity of the fault**: The number of separate zones that the attacker can target at the same time
- **Location of the fault** : The accuracy with which we drive the fault medium
- **Timing precision** : The control we have on the timing of the fault

The most powerful attacker is often considered as being able to generate multiple transient, bit-precise faults within a single clock cycle.

2.5.2 Differential Fault Attacks on AES

Let us now investigate how fault attacks can be used to efficiently recover a full 128-bit key from an AES unprotected implementation. Usually, the attacks target the last rounds of the encryption, so the errors do not spread more than needed (the spreading is due to the diffusion layer, MixColumns). We can identify mathematical relationships between the difference of outputs between the normal encryption and a faulty one. Often, several faulty ciphertexts are needed to recovery some or all the bytes of one round key. Moreover, in AES-128, it is trivial to recover the master key from a round key, as the key schedule process is a deterministic process that does not involve more secrets than the key itself.

The first practical DFA on the AES was described by Blömer and Seifert in [9]. The associated fault model requires the attacker to set a single bit to 0. The attack is repeated for each bit of the round key, 128 times. The plaintext needs to be chosen (0-only plaintext).

Two more attacks were presented by Giraud in [10]. They require several different faulty ciphertexts from the same plaintext, and require the fault to be set at a precise time and location. The first attack implies flipping one bit during the SubBytes step and needs about 50 ciphertexts to retrieve the full key. The second attack relaxes the fault model and requires inducing fault on bytes at several places, including the key schedule. About 250 faulty ciphertexts are needed and it is computation-intensive.

Piret and Quisquater's attack

As an example, we explain in details another attack, proposed by Piret and Quisquater in [11]. It is able to break AES-128 with only two faulty ciphertexts and requires to inject a byte fault in any of the state bytes between rounds 8 and 9 (which is practical). This attack can theoretically be applied to any Substitution-Permutation cipher.

A one-byte fault is induced on the state between the before-last diffusion layer θ_{R-2} and the last one, θ_{R-1} . Taking the output of the cipher, we can compute $255 * n = 2^{15}$ ($n = 8$ bytes for the AES) byte differences between the correct, known ciphertext and the faulty one. This is true because the last substitution and key addition do not change the number of errors (no diffusion).

From here, we will denote a correct/faulty ciphertext pair by (C,C*). We now consider 2 right ciphertext/faulty ciphertext pairs (C;C*) and (D;D*). We show below the steps of the attack :

1. Compute all the differences between θ_{R-1} and the output and store them in a list D
2. Consider a pair (C, C^*) from a plaintext P
3. Take a guess on the last round key K^R

4. Reverse the last round : compute the difference

$$\gamma_R^{-1} \circ \sigma[K^R](C) \oplus \gamma_R^{-1} \circ \sigma[K^R](C^*)$$

Look for its presence in the list D . If found, add the round key to the list L of candidates.

5. Choose a new plaintext P (with new (C, C^*)) and go back to step 2. This time round key guesses only go through the list L of possible candidates; if the difference computed at step 4 is not in D, remove the candidate from L. Repeat until there remains only one candidate in L.

The problem with this algorithm is that it needs an exhaustive search on the key K^R whose space size is 2^{128}, needing 2^{127} tries on average, which is way too much to be practical. Fortunately, Piret and Quisquater found a way to reduce considerably the set of keys to be tested by this algorithm.

We show below, directly from [11], this second algorithm :

1. Compute the 255n possible differences at the output of θ_{R-1}. Store them in a list L.
2. Consider 2 right ciphertext/faulty ciphertext pairs (C, C^*) and (D, D^*)
3. Consider the two left-most bytes of K^R:
   - For each of the 2^{16} candidates, compute [2] :

$$\gamma_R^{-1} \circ \sigma[\langle K_1^R, K_2^R \rangle](\langle C_1, C_2 \rangle) \oplus \gamma_R^{-1} \circ \sigma[\langle K_1^R, K_2^R \rangle](\langle C_1^*, C_2^* \rangle)$$

     and

$$\gamma_R^{-1} \circ \sigma[\langle K_1^R, K_2^R \rangle](\langle D_1, D_2 \rangle) \oplus \gamma_R^{-1} \circ \sigma[\langle K_1^R, K_2^R \rangle](\langle D_1^*, D_2^* \rangle)$$

   - Compare the results with the two left-most bytes of the 255n differences in list D. Make a list L of the $\langle K_1^R, K_2^R \rangle$ for which a match is found for both ciphertext pairs.
4. For each $K^\bullet \in L$, try to extend it by one byte:
   - Remove $K^\bullet$ from L.
   - For all 2^8 K_3^R , compute:

$$\gamma_R^{-1} \circ \sigma[\langle K_2^\bullet, K_3^R \rangle](\langle C_2, C_3 \rangle) \oplus \gamma_R^{-1} \circ \sigma[\langle K_2^\bullet, K_3^R \rangle](\langle C_2^*, C_3^* \rangle)$$

     and

$$\gamma_R^{-1} \circ \sigma[\langle K_2^\bullet, K_3^R \rangle](\langle D_2, D_3 \rangle) \oplus \gamma_R^{-1} \circ \sigma[\langle K_2^\bullet, K_3^R \rangle](\langle D_2^*, D_3^* \rangle)$$

[2] When the functions are applied to data of improper length(such as (C_1, C_2), consider the data to be right-appended with zeros)

- Compare the differences obtained with bytes 2 and 3 of the 255n differences in D. If a match is found (again for both ciphertext pairs), add the newly extended key $\langle K^{\bullet}, K_3^R \rangle$ to L.

5. Repeat step 4 until elements of L have a length of n bytes.
6. Apply now the first algorithm we gave using the same pairs $(C; C^*)$ and $(D; D^*)$, but consider only the candidates K_R in L (their number is much smaller than 2^{N_b} $=2^{128}$).

This huge reduction in complexity makes it possible to execute the whole attack in a few seconds. It is achieved by testing the bytes of the round key iteratively (same kind of complexity reduction as if we could test passwords letter-wise). We reduce the complexity of the bruteforce from $2^{8*16} = 2^{128}$ to $\mathcal{O}(2^8)$.

To summarize, we can expect that :

- A fault on state byte $a_{0,0}, a_{1,1}, a_{2,2}$, or $a_{3,3}$ will release information about round key bytes $K^R_{0,0}, K^R_{1,3}, K^R_{2,2}, K^R_{3,1}$.
- A fault on state byte $a_{0,1}, a_{1,2}, a_{2,3}$, or $a_{3,0}$ will release information about round key bytes $K^R_{0,1}, K^R_{1,0}, K^R_{2,3}, K^R_{3,2}$.
- A fault on state byte $a_{0,2}, a_{1,3}, a_{2,0}$, or $a_{3,1}$ will release information about round key bytes $K^R_{0,2}, K^R_{1,1}, K^R_{2,0}, K^R_{3,3}$.
- A fault on state byte $a_{0,3}, a_{1,0}, a_{2,1}$, or $a_{3,2}$ will release information about round key bytes $K^R_{0,3}, K^R_{1,2}, K^R_{2,1}, K^R_{3,0}$.

Figure 2.6 shows the actual propagation of an error located at $a_{0,0}$.

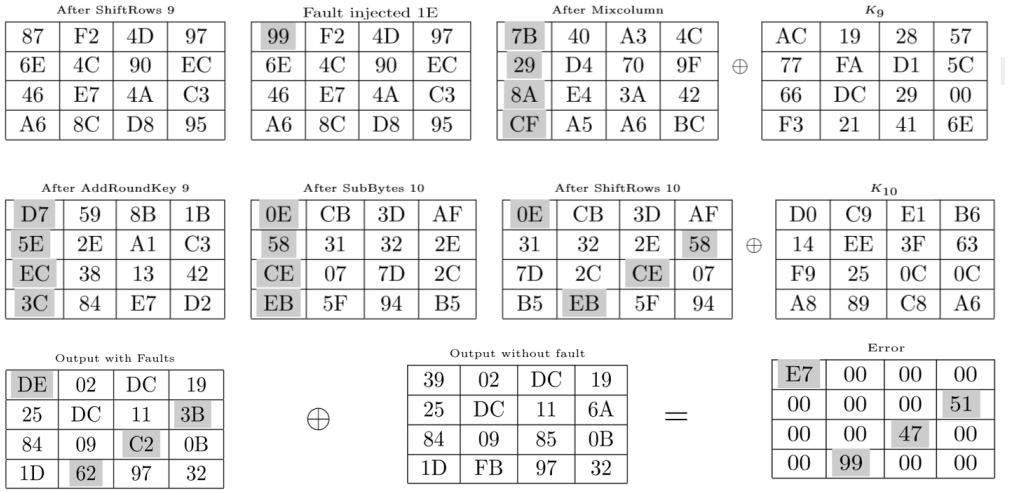

FIGURE 2.6: Propagation of an error at round 9. Source : [10]

This scheme needs 8 faulty/correct ciphertexts pairs to break the round key on average. Each fault induces faults on 4 bytes at the output but random location faults may lead to

the same fault location at the output. The attack goes even further, noticing that a fault induced between θ_{R-3} and θ_{R-2} propagates more and thus gives more information, reducing the number of needed pairs to 2.

This is one of the best DFA discovered so far. This technique has been slightly improved by Giraud and Thillard in [12], where only one pair of correct and faulty ciphertexts can be used, involving a bruteforce on some 2^{34} candidates afterwards. They also extend the attack to AES-192 and AES-256, only needing 2 and 3 pairs to break them. Recently, other theoretical attacks were presented by Tunstall *et al.* in [13], that only need one faulty ciphertext to break the entire key.

There are also powerful attacks against the key schedule when the state is DFA-protected [14], but they are less efficient than these ones.

2.6 Attack countermeasures

We now describe how symmetric-key block ciphers can be protected against two attacks of interest: Differential Fault Analysis and Differential Power Analysis. We will also speak about implementation countermeasures related to fault attacks such as instruction skips and bypassing comparisons with faults.

2.6.1 DFA Countermeasures

There are several ways to prevent attackers from using fault attacks. The most obvious way would be to prevent the attacker from inserting physical faults on the device. The chips are naturally protected with a plastic package which has first to be etched with mechanical or chemical means. This is a first protection against probing and laser attacks that is not so obvious to counter.

Next, there exists modified electronics circuits that are used to prevent the induction of a fault in a circuit. The most common way to do that is the *Dual Rail Implementation*, where any hardware bit is represented by itself *and* its opposite, in another wire. This logic can be extended to any *m-out-of-n* wires, where m wires carry a logical 1 and n wires carry a 0. It increases the complexity of an attack, needing the attacker to target bits and requiring to set precise values. Unfortunately, this method is very costly.

As this master thesis is focused on software implementations and attacks, we discuss below in more details coding-theoretical countermeasures instead of hardware ones.

Parity bits

The most obvious way to protect state or key bytes against faults is to use a parity bit scheme, i.e. extending each byte with a ninth bit, which is the binary sum of the other eight bits. It does not protect against all faults (all even-order bit faults are undetected)

but it has a low cost. The parity-bit error detection is equivalent to the use of a linear code.

The problem of the AES is the presence of a non-linear step, the S-Box, in its algorithm. Therefore, on cannot simply apply parity bits for this step. There are two solutions : the first one includes the parity bits in an extended S-Box (which increases the memory overhead) and the second one isolates the non-linear computation from the S-Box (inversion). The inversion is checked for faults by multiplying the input and outputs of this inversion, and verifying that the result is the neutral element. The other parts are still protected by parity bits. Some of these techniques are discussed in papers such as [15] , [16] or [17].

Operation-specific digests

To improve the fault coverage of error-detecting schemes, more complex digests than parity bits have been proposed; the most notorious ones come from the paper from Genelle *et al.* [18]. This paper suggest to cover the layers with two kinds of digests operating on the state bytes (denoted by x_i):

1. The linear steps can be protected with an additive digest of the form :

$$ADV(x) = \sum_{i=0}^{15} x_i$$

2. The non-linear inversion can be protected with a product-based digest,given by :

$$MDV(x) = \prod_{i=0,\ldots,15|x_i \neq 0} x_i$$

   In order to protect zero-valued inputs (which are not taken into account by the MDV), a Zero-Test is also performed :

$$ZT(x) = \sum_{i=0,\ldots,15|x_i 0} 2^i$$

All these digests are computed before and after each layer of each round, to check if a fault was induced on bytes. This method offers perfect resistance against 1-byte faults and can be extended to have a good statistical coverage of multiple-byte faults (with more complex digests). However, the performance overhead of this method is way bigger than the parity bits.

Another digest technique based on linear prediction was presented by Karpovsky *et al.* in [19]. We will present it in details in chapter 3 when we explain the protection scheme used for this master thesis.

Ring embedding

Another way to protect implementations against faults is to use a technique named *ring embedding*. One version of this technique was proposed by Medwed in [20]. The underlying principle is to extend the Galois Field used by the AES with another GF used by an error-detecting code. The plaintext and the key are assigned fixed signatures and the signature values after the encryption do not depend on the plaintext and key. This way, one can precompute the signature of a normal execution of the program and compare it with the output signature. This way, the state, key and program flow are protected against any one-byte, one-bit fault attack and instruction skips. Unfortunately, this solution is very costly. It needs 8-bit signatures with each byte of the state and the announced time overhead is lower-bounded by 100 %. The actual cycle overhead looks much higher with a 90 000 cycle count versus 3600 for an unprotected version of the AES.

Encrypt-Decrypt

With many symmetric-key ciphers, the process of encryption is really close to the decryption process. Thus, it is often possible to include the decryption module in the chip or code with a reasonable space overhead. One technique to prevent faults is to decrypt the output ciphertext and comparing it with the original plaintext before giving it to the user. If there is a mismatch, a fault is detected. The time overhead will again be doubled due to the decryption process.

Duplication - Voting Systems

Another well-known technique consists of duplicating (either in space for hardware circuits or in time for software computations) the encryption or any part of it. The two results are then compared and an error is detected if they do not match.

A higher-level protection scheme would involve more than two shares of computation performing the same operations. This enables error correction; if at least *m-out-of-n* shares are equal, then use this output as the correct one.

Fault protection schemes are often compared to a duplication solution, where we assume that the latter covers any single fault and has a space/time overhead of 100 %.

Reaction to faults

When a fault is detected, several decisions are possible :

1. Stop the computation and do not output the result. This solution will not leak information about the faulty ciphertext but will indicate the attacker that his attack was detected.

2. Use *Infective Computation* [21]: the encryption process is scrambled in such a way that any injected fault will spread in a way that makes the output unusable for DFA. Basically, several encoders are used and their intermediate results are mixed together. No comparison has to be done as in other protection mechanisms, so it is less vulnerable to attacks targeting comparisons. However, recent researches have shown that those countermeasures are still breakable [22].

3. Output random data. This way, the attacker will not know that his attack succeeded, as long as he does not know the original ciphertext. However, this assumption is not realistic and this needs a source of randomness, which is not always available. One possible improvement of this countermeasure would consist in replacing only the output bytes that would have been affected by the fault. This involves computing the diffusion pattern of AES and detecting where the fault took place. The attacker will have a faulty ciphertext that is useless to recover the key.

It can be important not to let know the attacker that his attack was countered. For instance, if the attacker has a 1% chance of success because of unknown parameters,but is able to repeat the same faults or refine them, it is crucial not to leak information about his success or failure. However, other side-channel analyses can reveal the countermeasures, such as timing analysis.

2.6.2 Execution-based countermeasures

The previous countermeasures were related to attacks targeting the cryptographic computations. Unfortunately, this is not the only weak point of the whole execution of a cipher. The attacker can aim at other targets such as:

- The round counter: in the AES and in many other block ciphers, the encryption process iterates by rounds. If the round counter value can be altered, the whole encryption can be broken. For instance, a 0-round encryption of AES process could (depending on the implementation) XOR the plaintext with the master key, immediately revealing that key to the attacker. Furthermore, many attacks (that are not even fault attacks) can be done on reduced versions of AES. For instance, a successful related-key attack can be performed on a 7-round version of AES-128 [23].

- Comparison instructions: many fault countermeasures need to perform a comparison between digests or specific values. The result of a comparison (*true* or *false*) is always stored in a specific register. One can attack this register in addition to its original attack to avoid being detected.

- The program counter : if the attacker is able to modify precisely the value of a register, the target of choice is the program counter. With some knowledge about

the implementation of the algorithm, he can skip critical instructions and output data directly related to the key.

- Skipping instructions: via fault attacks or other methods such as clock glitches, the attacker could skip other instructions that would ease an attack over that cipher.

2.6.3 DPA countermeasures

Power analysis is a very powerful tool to the attacker. The results of such an attack are immediate and the attack itself does not need days of processing. This is why a lot of effort has been deployed to counter simple and differential power analysis attacks. We only describe the countermeasures specific to DPA, as they generally protect from SPA as well.

The most natural way to prevent power analysis is to add noise to the power traces that can be collected on the device, making it harder to distinguish meaningful data from noise. It is called *hiding*. This countermeasure is therefore device-dependant and is almost unrelated to the specific cipher that is used. We make a distinction between hiding at the software and hardware level.

Software hiding

Several techniques can be applied to a software implementation to reduce the significance of a power trace :

- Instruction randomization: the order in which the instructions are processed can sometimes be changed without effect on the output of the cipher. This is a kind of obfuscation that is useful if the attacker does not have access to the implementation.
- Parallel activity : the more operations the chip is doing during the measurement window,the less the trace will be correlated with the targeted value.
- Choice of instructions : some instructions leak more information than others, so they need to be chosen carefully.
- Dummy computation : the insertion of dummy operations during the algorithm can generate "noise" in the measurements if the attacker is not aware of such operations.
- Key-dependant branching : the program should never branch its execution depending on the value of the key. Otherwise, determining the value of the key based on the power trace is straightforward.

Hardware/Cell level hiding

In addition to the software tricks we enumerated (some of them can be applied to hardware implementations), there are other ways to hide the power consumption of the cryptographic device :

- Dummy cycles: insert clock cycles that do not trigger logic cells during the execution
- Clock variations: change the clock frequency during execution or use multiple clock domains
- Filtering the power consumption: add capacitors to the power drain to smooth the power consumption that is observed by the attacker (but the attacker could bypass them)
- Noise engines : generate noise in parallel to the computation of the cipher
- Specific memory cells: although classical SRAM/DRAM leaks information because the consumption differs between 0 and 1's, other memory cells are tuned to avoid leakage. The most common logic cell style that is used is the *Dual Rail Precharge* (DRP) logic style. However, this kind of cells often consumes more power than classical memory cells.

Masking

Instead of trying to hide the power consumption of the cryptographic device, one could think of algorithmic schemes that do not leak information about the real values that are processed. In masked implementations, designers decide to apply a mask on each intermediate value used in the cryptographic process. Masks are applied to the input of a cipher and removed when the processing is finished.

There are two main types of masking: boolean and arithmetic. Boolean masks use the XOR operation to mask a value : $v_m = v \oplus m$, whereas arithmetic masking uses the addition or multiplication : $v_m = v + m (\text{mod } n)$ or $v_m = v \times m (\text{mod } n)$.

These two types of masking apply differently to the AES: boolean masking is suited to AES linear operations because $f(x \oplus m) = f(x) \oplus f(m)$. The linear operations that change a masked value can then be linked to an unmasked transformation of the value and the mask separately.

However, the S-Box step of the AES is non-linear and cannot be masked with a boolean operation. The non-linear operation involved is the multiplicative inverse ($f(x) = x^{-1}$). It is suited to multiplicative masking: $f(x \times m) = (x \times n)^{-1} = f(x) \times f(n)$.

These masking operations can be related to a secret-sharing scheme : when $v_m = v \oplus m$, the secret v_m is shared among two values, v and m. The secret can only be recovered if the two values are known. This secret sharing scheme can be extended to d masks,

thus requiring $d + 1$ values to determine the secret. This is the key point of *higher-order masking*. Chari *et al.* [24] were the first to propose such a protection scheme against DPA.

An important point of masking is that the intermediate values should stay masked all the time. No intermediate operation should try to unmask the value, otherwise this would lead to an information leakage that could break the protection scheme.

There are several state-of-the-art masking techniques based on these aforementioned practices. A good summary can be found in the article written by Grosso, Standaert and Faust [25].

Attacks on masking

With higher-order masking come higher-order DPA attacks. In fact, increasing the number of masks only increases the difficulty of an attacker to find information about the intermediate values. For instance, second-order DPA attacks rely on the joint leakage of two intermediate values. Likewise, a *d-th* order attack uses the combination of d intermediate values.

Therefore, the choice of the number of masks only depends on the power of the attacker that we want to protect from. This power is limited by the measurement noise in practice. Indeed, obtaining d intermediate values from a set of traces requires to compute higher-order sample moments. The complexity of these computations is expected to grow exponentially with respect to the masking order (at a fixed noise level).

Many people have tried to prove the security of higher-order masking schemes. Most of the time, these proofs rely on assumptions or hypotheses that are seldom verified in practice. Piret and Standaert [26] discuss the notion of *perfect security* in such schemes. Other papers such as [27], [28] and [29], among other, discuss the application of these security principles to practical masking schemes.

Combined attacks

In addition to the attacks listed before, other kinds of attacks also need to be mentioned: Combined attacks and Fault sensitivity analysis. In both cases, a combination of fault and side-channel attacks is used to lower the degree of security to a practical level where attacks can be attempted.

Fault sensivity analysis was first proposed by Li *et al.* in [30]. They propose to recover side-channel data when a fault attack is performed, even if the device is protected against faults. They find that there exist leakages that can be exploited to recover the key.

Combined attacks [31] , [32] aim at lowering the resistance of combined protection schemes to further apply known attacks and recover the key. However, these attacks target the key schedule and require repeatability of faults.

Chapter 3

Algorithmic design

3.1 Objectives and design criteria

The primary objective of this master thesis is to provide a combined solution to prevent both differential power and fault attacks on the same device, using algorithmic countermeasures. To achieve this, we gathered information about several protection mechanisms. We selected the fault protection scheme and the side-channel protection scheme separately according to design criteria and then tried to put the schemes together. The combination of such schemes is something that is often done in practice by manufacturers and designers, but there is a lack of scientific research regarding this combination. We did not find any paper dealing with this combination, although there are papers about attacks on DFA and SCA protected implementations ([32], [31]).

The goal of this chapter is to define the requirements of a device that could resist such attacks in addition to classical FA and DPA, while being cost-effective. We define such requirements in the following sections (3.1.1 and 3.1.2). Then, we propose to use existing schemes to protect against fault attacks in section 3.2 and against side-channel attacks in section 3.4. Finally, we propose new methods in section 3.5 and analyse their effectiveness in terms of security.

We define our security criteria based on what we will able to test with the current equipment in the laboratory. We also want to target low-cost setups as powerful attackers will have other means to break the system anyway. Nevertheless, we do not exclude more powerful attackers in our design, but we set a minimum level of security that has to be fulfilled in all the cases.

3.1.1 Fault model

First, we define the requirements in terms of fault detection. We are interested in a fault protection scheme that can be applied at the software level and can be embedded in low-end devices such as 8-bit microprocessors, for the sake of simplicity and feasibility.

The means of the attacker are the following :

- Inject one fault at a time [1] on a specific memory location (RAM or registers)
- The precision is at most one byte
- The effect of the fault can be random or possibly stuck-at
- The repeatability of the experiment is non-negligible
- The fault can be temporary or permanent
- The timing of the fault allows to target one instruction

These means are established from an estimation of what we are able to do in the labs (see chapter 5) and therefore represent an attacker with a set of tools that are not particularly designed for fault attacks.

3.1.2 Side-channel leakage model

The means of a side-channel attacker are somewhat harder to define than for fault attacks. One of the most important parameters for a successful side-channel attack is the measurement noise. It depends on how and where the measurement is made, and on the quality of the equipment. We exclude this parameter for the design of our solution, as it is too specific, and propose several levels of security that can be chosen according to the power of the real attacker.

Therefore, we limit ourselves to one parameter that is convenient to use with masking schemes: the number of 'probes' according to the *probing model*. It comes from the concept of *private circuits* introduced by Ishai *et al.* in [33]. The basic concept is the following: if an attacker can observe, simultaneously or not, d intermediate values in a computation (for example by the means of probes or via power analysis), then he can perform a d-th order DPA.

Similarly, a circuit (or program) is *d*-th order secure if the lowest degree of a successful side-channel attack is *d+1*.

It implies that any set of d values from the set of intermediate values processed by a cipher must be statistically independent to resist a d-th order DPA. This is achieved by sharing the computation into $d + 1$ independent values.

From a practical point-of-view, it becomes increasingly (at least exponentially) difficult for an attacker to perform a DPA when the probing security order increases [24], as he has to estimate higher-order moments. The requirement in terms of side-channel security is the need of a flexible scheme that allows to use any number of shares d. This parameter is linked to the real power of the attacker, because the measurement noise will prevent a succesful attack targeting a number of values d higher than some threshold.

[1] In the design, we do not exclude the possibility to inject several faults at the same time, for stronger attackers

3.2 Fault protection scheme

There exist many ways of protecting a cipher against faults (as explained in chapter 2.6.1). Comparisons between actual implementations of some of these countermeasures can be found in [34]. They point out that the solution of Karpovsky *et al.* [19] may deserve further analysis and improvements, because the hardware cost is close to the one of duplication, but with other benefits. We based our protection on this scheme for several reasons :

- It has a statistically better fault coverage than many other schemes and is convenient for random byte fault models
- The implementation overhead is reasonable compared to others, especially in software implementations
- It can benefit from the randomness brought by the masking scheme because the statistical error coverage assumes random inputs. Even a fixed-plaintext attack does not lower this fault detection probability.

This scheme is well-suited for both hardware and software implementations, although there is a time/security trade-off available in hardware that does not make sense in software (see below).

3.2.1 Protecting the state

The protection of the state is ensured by using a so-called "non-linear robust code". The code is based on a signature that is generated by a prediction at the beginning of the AES round followed by a compression of the state at the end of the same round. Non-linearity is added to extend the statistical fault coverage by making the probability data-dependant. There are other benefits of such a scheme that will be explained later.

Robustness (induced by the non-linearity) is a particular property of this code that provides equal error detection against all errors, assuming that the plaintext is randomly distributed (this assumption will be verified when we use the combined scheme, see section 3.5).

Mathematically, this property can be expressed as follows (from [19]):

Definition 1. *Let C be a (n, M)-code, where $M = |C|$:*

The code C is robust with respect to its error-masking probability iff the probability of missing an error e is less than one for all nonzero errors e:

$$Q(e) = \frac{|\{w|w \in C, w + e \in C\}|}{|C|} < 1, e \neq 0$$

Where $w, e \in GF(q^n)$

To obtain such a code, let us start with a simple binary linear (n,k)-code V (with $2k > n$)), a check matrix $H = [P|I]$ where I is a $(r \times r)$ identity matrix and and P is a $((n-k) \times k)$ matrix of rank $n - k = r$ over GF(2).

An error e is not detected iff $e \in V$. This code can be turned into a nonlinear robust (n, k)-code C_v such that the set of undetected errors in C_v is a $(k - r)$-dimensional subspace of V ($|E| = 2^{k-r}$ instead of 2^k for V). The proof of this theorem can be found in [35].

Table 3.1 compares the linear and robust nonlinear codes in terms of error detection:

	Robust nonlinear	linear
Undetected errors	2^{k-r}	2^k
Errors detected with a probability of 1	$2^{n-1} + 2^{k-1} - 2^{k-r}$	$2^n - 2^k$
Errors detected with a probability of $1 - 2^{-r+1}$	$2^{n-1} - 2^{k-1}$	0

TABLE 3.1: Comparison of the number of detected errors for robust and non robust schemes

The operation that is used to make the code nonlinear consists of cubing the signature associated to the code. Other operations (e.g. squaring) were considered in [19] but cubing was the most interesting one. We show below (figure 3.1) how to process this coding inside of an AES round. As shown in the schematic, several boxes (in orange) are added to the typical round execution.

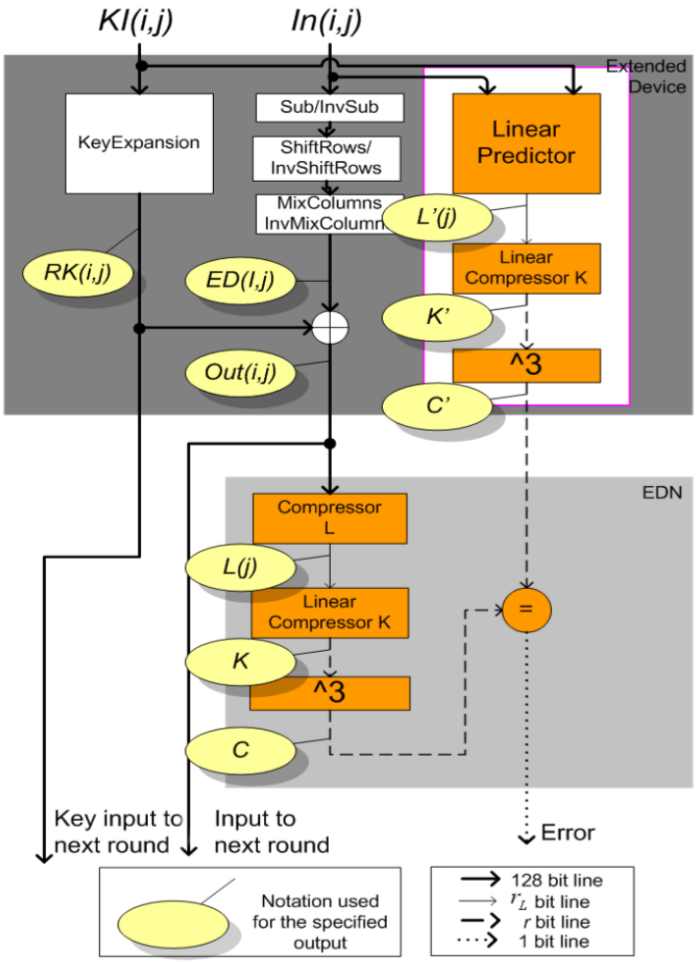

FIGURE 3.1: Fault protection scheme applied to the AES (from [35])

The compressor L

The compressor L is the core idea of this coding. It compresses the 128-bit state of the AES Out(i,j) in a 32-bit signature L(j). The linear coding at that stage is thus a (160,128)-code. The purpose of this signature is to reduce the number of operations needed to predict it from the input state (see 3.2.1). It is processed in the following way (figure 3.2 and algorithm 1):

$$\bigoplus \begin{array}{cccc} Out(0,0) & Out(0,1) & Out(0,2) & Out(0,3) \\ Out(1,0) & Out(1,1) & Out(1,2) & Out(1,3) \\ Out(2,0) & Out(2,1) & Out(2,2) & Out(2,3) \\ Out(3,0) & Out(3,1) & Out(3,2) & Out(3,3) \\ \hline L'(0) & L'(1) & L'(2) & L'(3) \end{array}$$

FIGURE 3.2: Compression of the AES state into a signature

Algorithm 1 State compression

Require: State matrix $s[4, 4]$
Ensure: L, the 32-bit linear signature of s

$L_0 \leftarrow s_{0,0} \oplus s_{0,1} \oplus s_{0,2} \oplus s_{0,3}$
$L_1 \leftarrow s_{1,0} \oplus s_{1,1} \oplus s_{1,2} \oplus s_{1,3}$
$L_2 \leftarrow s_{2,0} \oplus s_{2,1} \oplus s_{2,2} \oplus s_{2,3}$
$L_3 \leftarrow s_{3,0} \oplus s_{3,1} \oplus s_{3,2} \oplus s_{3,3}$
$L = L_0||L_1||L_2||L_3$

The linear predictor

The linear prediction consists of computing in advance the result of a compressed image of the state after the execution of one round. It takes the entire 128-bit state as an input and outputs a 32-bit signature. It is processed in a way that requires less computations than a duplication of the round. The performance of this prediction will be analysed in chapter 4.1.

As we want the prediction to be equal to the compression, we can express the output of the predictor in function of the output bytes :

$$L'(j) = \bigoplus_{i=0}^{3} Out(i,j) = \bigoplus_{i=0}^{3} (RK(i,j) \oplus ED(i,j)) \tag{3.1}$$

Where $RK(i,j)$ denotes the round key byte associated to its state byte and $ED(i,j)$ denotes the transformation of $In(i,j)$ through *SubBytes*, *ShiftRows* and *MixColumns*. It is possible to rewrite this equation with respect to the input $In(i,j)$ by considering the mathematical relationships explained in chapter 2.3.

Let us take the first compression byte:

$$\begin{aligned} L_{ED}(0) &= \{01\} \cdot \text{Sbox(In(0,0))} \oplus \{03\} \cdot \text{Sbox(In(1,1))} \oplus \text{Sbox(In(2,2))} \oplus \text{Sbox(In(3,3))} \oplus \\ &\quad \text{Sbox(In(0,0))} \oplus \{02\} \cdot \text{Sbox(In(1,1))} \oplus \{03\} \cdot \text{Sbox(In(2,2))} \oplus \text{Sbox(In(3,3))} \oplus \\ &\quad \text{Sbox(In(0,0))} \oplus \text{Sbox(In(1,1))} \oplus \{02\} \cdot \text{Sbox(In(2,2))} \oplus \{03\} \cdot \text{Sbox(In (3,3))} \oplus \\ &\quad \{03\} \cdot \text{Sbox(In(0,0))} \oplus \text{Sbox(In(1,1))(Sbox(In(2,2))} \oplus \{02\} \cdot \text{Sbox(In(3,3))} \\ &= \text{Sbox(In(0,0))} \oplus \text{Sbox(In(1,1))} \oplus \text{Sbox(In (2,2))} \oplus \text{Sbox(In(3,3))} \end{aligned} \tag{3.2}$$

Note that, although that this expression is inspired from [19] (p.5, first column), they forgot to include the *ShiftRows* transform in their expression, resulting in inconsistent equations (they correct it later in their paper without mentioning their mistake, which confuses the reader). As we compress the output state column-wise, we need to pick the corresponding inputs wisely, and the indexes are not the elements of the column anymore (that is, instead of performing *ShiftRows* after the Sbox access, we input the row-shifted version to the Sbox).

We can combine equations 3.1 and 3.2 to obtain :

$$L'(0) = \text{Sbox(In(0,0))} \oplus \text{Sbox(In(1,1))} \oplus \text{Sbox(In (2,2))} \oplus \text{Sbox(In(3,3))} \oplus \quad (3.3)$$
$$RK(0,0) \oplus RK(1,0) \oplus RK(2,0) \oplus RK(3,0) \quad (3.4)$$

This equation is far more simple than computing the whole state transformation as in a duplication system. It requires 8 XORs and 4 Sbox lookups per byte.

The equations for the other bytes as well as for the decryption are described in algorithm 2.

Algorithm 2 Signature prediction

Require: State matrix $s[4,4]$
Ensure: L, the 32-bit linear signature of s

$L_0 = \text{Sbox(In(0,0))} \oplus \text{Sbox(In(1,1))} \oplus \text{Sbox(In (2,2))} \oplus \text{Sbox(In(3,3))} \oplus$
$RK(0,0) \oplus RK(1,0) \oplus RK(2,0) \oplus RK(3,0)$
$L_1 = \text{Sbox(In(0,1))} \oplus \text{Sbox(In(1,2))} \oplus \text{Sbox(In (2,3))} \oplus \text{Sbox(In(3,1))} \oplus$
$RK(0,1) \oplus RK(1,1) \oplus RK(2,1) \oplus RK(3,1)$
$L_2 = \text{Sbox(In(0,2))} \oplus \text{Sbox(In(1,3))} \oplus \text{Sbox(In (2,0))} \oplus \text{Sbox(In(3,1))} \oplus$
$RK(0,2) \oplus RK(1,2) \oplus RK(2,2) \oplus RK(3,2)$
$L_3 = \text{Sbox(In(0,3))} \oplus \text{Sbox(In(1,0))} \oplus \text{Sbox(In (2,1))} \oplus \text{Sbox(In(3,2))} \oplus$
$RK(0,3) \oplus RK(1,3) \oplus RK(2,3) \oplus RK(3,3)$
$L = L_0 || L_1 || L_2 || L_3$

The linear compressor K

The role of this block is to reduce the size of the signatures L(j) to a smaller size, in order to compute the cube easier. However, this trades error coverage for speed. While it makes sense in hardware, software implementations compute the cube of a 32-bit signature as fast as a 28-bit signature (as suggested in the paper for a hardware implementation), therefore we choose to remove this block from the scheme and to keep 32-bit signatures.

The cubic operator

This block takes the whole signature as input and returns its cube in GF(2^8). It allows the code to become robust as it adds data dependency to the error detection probabilities.

The comparator

Finally, the comparator tests for the equality between the predicted signature and the one generated from the output of the round. If they are not equal, the result of the encryption/decryption must not be outputted. (Algorithm 3)

Algorithm 3 Signature comparison

Require: 32-bit signatures P and P'

if $P == P'$ **then**
    Continue
else
    Abort
end if

3.2.2 Protecting the key schedule

The protection scheme previously explained does not allow protecting the key schedule. Although there exist linear codes that allow protecting the key schedule (e.g. [18]), we choose not to use them as they are less efficient than the scheme we use. Furthermore, it would not be coherent to use those codes in addition to ours.

A lot of attacks target the key scheduling nowadays, especially for combined attacks ([30], [31]). The best countermeasure we found is to precompute the expanded key and to store it in ROM. It loses flexibility compared to a classical implementation from a systemic point of view. However, few systems require the key to change between executions in practice. In the case of embedded cryptographic chips such as smart cards, losing this feature for an improved security is a reasonable trade-off. However, it increases the performance of the system because it does not need to expand the key anymore (gain in time and memory).

Therefore, when a subkey is needed inside a round, it is simply loaded from the ROM to RAM. Loading the entire key to RAM should be avoided as it exposes the subkeys to attacks (different than for the key schedule in itself) during the entire execution.

3.2.3 Protecting the execution

The state and the key schedule are not the only targets that an attacker could think of. Specifically, the comparisons that are done :

- To check the equality of the signatures
- To count the number of rounds
- Any conditional jump that could potentially skip critical sections

Those should be protected from faults as well. Although these attacks do not belong to our fault model (it requires to target a single bit), simple instructions skips (e.g. with a clock glitch) could bypass our protection scheme. We do not implement all of them in our solution but point out that these should be taken care of. Some papers already describe software countermeasures, such as the article of Barenghi *et al.* [36] but they do not protect the comparisons. We suggest a simple method to protect from an attacker that could induce such faults.

Suppose that signatures are stored in two registers $r01$ and $r02$. The instructions below depict a comparison. It will be bypassed if the zero flag is set to 1 by a fault.

LISTING 3.1: Signature comparison

```
cp r01, r02 ; sets the Zero flag to 1 if r01==r02
brne <error>; branch if the Zero flag is cleared
```

Instead of doing only one comparison, we can add instructions that require the Zero flag to be set or cleared alternatively, requiring the attacker to change the value to 0 or 1 at each test. This is clearly not feasible for our attacker model.

LISTING 3.2: Safe signature comparison

```
cp r01, r02 ; sets the Zero flag to 1 if r01==r02
brne <error>; branch if the Zero flag is cleared
eor r01, r02 ; sets the Zero flag to 0 if r01==r02
breq <error>; branch if the Zero flag is set
```

Many solutions of this kind are possible but we cite this one as an illustration. It has a 100% time and program memory overhead for each instruction that must be protected. Practically, those critical comparisons do not represent a large part of the instructions so the global overhead can be kept quite low.

3.3 Security hole

When designing our countermeasure, it did not occur to us that the fault protection scheme that we chose is actually flawed. The flaw is more obvious in software implementations but the original hardware countermeasure also presents this weakness. The problem is pointed out in figure 3.3 and explained hereafter.

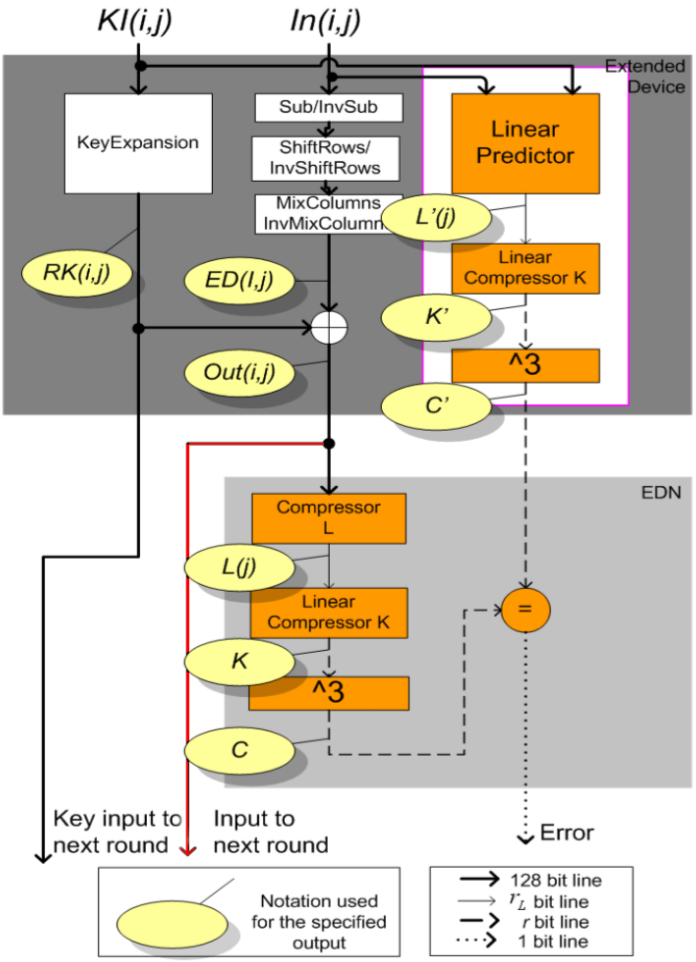

FIGURE 3.3: Fault protection scheme. The flaw is indicated in red

For software implementations, we have to compute sequentially the AES round and the prediction,with two copies of the state (otherwise one fault would propagate through the prediction and the round operations) resulting in the same signature. The idea is to make a copy of the output of a round to another memory zone when it is produced. This copy is then used for the prediction. If we look at the complete algorithm that we present further (algorithm 12), this copy must be done after step 15 (here done at step 16) and before the next step 10.

However, this still leaves some cycles unprotected, when we make a copy of the state. If we induce a fault in the (source) state there, it will propagate through the prediction and the real execution, producing the same signature for both computations. Even if we parallelize the procedures, there will always be some points in time where we can induce a fault like this. In fact, the problem lies in the copy of the state. If we were able to have an instantaneous copy of the state, the problem could be easily solved.

This security hole cannot be easily patched. To our knowledge, only a scheme that covers the entire execution in a whole could prevent that. Any error-detecting code that operates at the round level is open to this flaw. We could imagine unrolling the loops and make the prediction on all the rounds in one time, but our scheme cannot

do that because the predictor needs a complete input state, and produces a reduced output state after each round.

A possible countermeasure is the "ring embedding" solution that we presented in section 2.6.1, that produces a correcting code on the whole execution of the cipher. Also, duplication and encrypt-decrypt countermeasures do not have this weakness.

For hardware implementations, we could say that there is a single set of wires that connects the output of a round, the input to the next round and the input of the error-detection network. This way, if we change the value of a wire, we change it for both the error-detection network and for the following round, so the fault can be detected. We are sure that it could not be implemented like this without any registers, because the error-detection and the round function do not have the same timing constraints. If there are registers between rounds, then we can induce a fault at the input of the next round for both the round functions and the prediction. However, this requires to target specific registers in the circuit, which might be hard.

3.4 Side-channel protection scheme

We previously explained how to prevent side-channel attacks from a theoretical point of view. However, it is not trivial to apply d^{th}-order sharing to an AES implementation without leaking information about sensitive data on lower order moments. The starting point of the countermeasure we apply is the solution brought by Rivain and Prouff [29] and some other improvements that we discuss later. This is not the only d-th order secure masking scheme that exists (see [25]) and not the least costly one. But at the time of designing our protection scheme, we were not aware of those other solutions which are not that much more efficient anyway (same order of complexity). Those other approches of higher-order masking can be found in [37] (Genelle *et al.*), [38] (Kim *et al.*) or [39] (Roche *et al.*).

3.4.1 Masking the state

First, an initial sharing has to be applied to the state. The number of shares d is a design parameter and will be discussed later. Basically, the state masking is applied as follows (where rand(i) is a PRNG that outputs i bytes) :

Algorithm 4 Mask generation

for $i = 0$ to d **do**
  $s_i \leftarrow \text{rand}(16 \times 8)$
  $s_0 \leftarrow s_0 \oplus s_i$
end for

Algorithm 4 enables splitting the initial state into d statistically independent shares thanks to the generation of $d \times 16$ random bytes.

Masking through AddRoundKey, MixColumns and ShiftRows

The operations *AddRoundKey*, *MixColumns* and *ShiftRows* are all linear to the XOR operation. This is why we can treat each share independently for these operations.

To securely process the round key addition, a simple XOR operation between each share s_i and its own round key k_i^r share is performed.

$$\text{AddRoundKey}(s, k^r) = \bigoplus_{i=0}^{d} \text{AddRoundKey}(s_i, k_i^r) = \bigoplus_{i=0}^{d} (s_i \oplus k_i^r)$$

For the other operations, we can simply express them as follows :

$$\text{ShiftRows}(s) = \bigoplus_{i=0}^{d} \text{ShiftRows}(s_i)$$

$$\text{MixColumns}(s) = \bigoplus_{i=0}^{d} \text{MixColumns}(s_i)$$

The processing overhead of such a scheme grows linearly with the number of shares, as we have to apply the same operations as in a classical unprotected implementation, but with d shares.

Masking through SubBytes

Usually, the *SubBytes* step is performed by a lookup in a so-called S-box that maps all the 2^8 possible inputs to their transformation according to what was described in chapter 2.3.3. This operation has therefore a time complexity of $\mathcal{O}(1)$ and a space complexity of $\mathcal{O}(n)$ However, adding a random number to the non-linear *SubBytes* step makes it impossible to have a relationship such that we can treat the shares independently. It is required to treat all the shares as a whole (without XORing them) and to perform mask correction. The best we can do is to obtain a *SubBytes* step such that the input shares satisfy

$$\bigoplus_i x_i = x$$

and the output shares satisfy

$$\bigoplus_i y_i = S(x)$$

To achieve a secure masked S-box processing, we rely on the scheme proposed by Rivain and Prouff [29], further secured by Coron *et al.* [40] and enhanced by Grosso *et al.* [41].

The proposed methods computes the *SubBytes* step in a classical way, with multiplications and additions. The additions are trivial to protect as they are linear to the XOR operation. The multiplication problem can be seen as securing a AND gate with sharing.

In order to secure a multiplication with sharing, we first consider the Ishai-Sahai-Wagner (ISW) scheme [33].

Let a an b be two bits and let c denote AND(a,b) = ab. Let us assume that a and b have been respectively split into d+1 shares $(a_i)_{0\leq i\leq d}$ and $(b_i)_{0\leq i\leq d}$ such that $\bigoplus_i a_i = a$ and $\bigoplus_i b_i = b$. To compute the secure multiplication, the scheme goes as follows:

- For every $0 \leq i < j \leq d$, pick up a random bit $r_{i,j}$
- For every $0 \leq i < j \leq d$, compute $r_{j,i} = (r_{i,j} \oplus a_i b_j) \oplus a_j b_i$
- For every $0 \leq i \leq d$, compute $c_i = a_i b_i \oplus \bigoplus_{j\neq i} r_{i,j}$

Equation 3.5 computes a (d + 1)-tuple $(c_i)_{0\leq i\leq d}$ s.t. $\bigoplus_i c_i = c$:

$$\begin{aligned}
\bigoplus_i c_i &= \bigoplus_i (a_i b_i \oplus \bigoplus_{j\neq i} r_{i,j}) && (3.5)\\
&= \bigoplus_i (a_i b_i \oplus \bigoplus_{j>i} r_{i,j} \oplus \bigoplus_{j<i} (r_{j,i} \oplus a_i b_j \oplus a_j b_i)) && (3.6)\\
&= \bigoplus_i (a_i b_i \oplus \bigoplus_{j<i} (r_{j,i} \oplus a_i b_j \oplus a_j b_i)) && (3.7)\\
&= (\bigoplus_i a_i)(\bigoplus_i b_i) && (3.8)
\end{aligned}$$

The order operations, denoted with the brackets, is crucial. Algorithm 5 performs this secure multiplication.

We denote hereafter this algorithm as a 'Type III' multiplication. We also present two other types of secure multiplication algorithms, as stated by Grosso *et al.* in [41] . The goal is to achieve better performance by exploiting some linear properties of the *Frobenius endomorphism* [42].

Definition 2. *In a finite field $\mathbb{F}_{2^n}$, the squaring of an element $x \in \mathbb{F}_{2^n}$ is a Frobenius endomorphism. This endomorphism has an interesting linear property :*

$$\forall a, b \in \mathbb{F} \quad Frob_{\mathbb{F}_{p^n}}(a+b) = (a+b)^p = a^p + b^p = Frob_{\mathbb{F}}(a) + Frob_{\mathbb{F}}(b)$$

Where p is the characteristic of the finite field, i.e. 2 for Rijndael AES. Any exponentiation by a power $2^{\mathbb{N}}$ is also an endomorphism.

Definition 2 allows us to establish a new type of multiplication that we call 'Type I' multiplication, based on the endomorphism.

Algorithm 5 Type III Multiplication - d-th order secure multiplication over $\mathbb{F}_{2^n}$

Require: Shares a_i satisfying $\bigoplus_i a_i = a$, shares b_i satisfying $\bigoplus_i b_i = b$
Ensure: Shares c_i satisfying $\bigoplus_i c_i = ab$

for $i = 0$ to d **do**
  for $j = i + 1$ to d **do**
    $r_{i,j} \leftarrow \text{rand}(n)$
    $r_{j,i} \leftarrow (r_{i,j} \oplus a_i b_j) \oplus a_j b_i$
  end for
end for
for $i = 0$ to d **do**
  $c_i \leftarrow a_i b_i$
  for $j = 0$ to d, $j \neq i$ **do**
    $c_i \leftarrow c_i \oplus r_{i,j}$
  end for
end for

Algorithm 6 Type I Multiplication - Secure evaluation of a $\mathbb{F}_2$ linear function g

Require: Shares x_i satisfying $\bigoplus_i x_i = x$
Ensure: Shares y_i satisfying $\bigoplus_i y_i = g(x)$

for $i = 0$ to d **do**
  $y_i \leftarrow g(x_i)$
end for

In order to process any power of x in $\mathbb{F}_{2^n}$, and particularly x^{254}, which corresponds to the multiplicative inverse in GF(2^8), we add a third type of multiplication, first proposed by Coron *et al.* in [40], that is also d-th order secure (and corrects the security flaw of [29]). Algorithm 7 allows computing any multiplication of the type $x \times g(x)$, where $g(x)$ is a Frobenius endomorphism.

The most efficient way to combine those algorithm into a multiplicative inversion is given by Grosso *et al.* in [41]. Figure 3.4 and algorithm 8 show the most efficient combination of those algorithms.

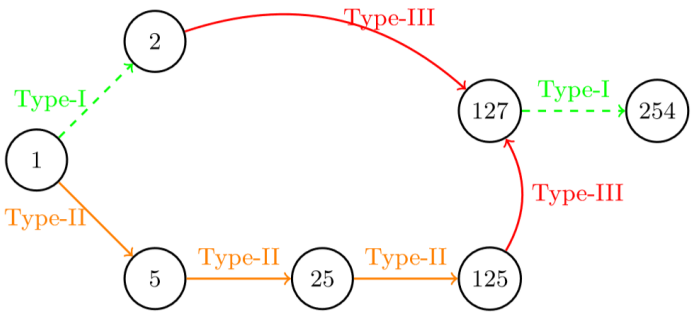

FIGURE 3.4: Multiplicative inverse computation (from [41])

From algorithm 8, we note that we can use look-up tables to tabulate the squaring and

Algorithm 7 Type II Multiplication - Secure evaluation of a product of $h(x) = x \times g(x)$

Require: Shares $(x_i)_i$ satisfying $\bigoplus_i x_i = x$
Ensure: Shares $(y_i)_i$ satisfying $\bigoplus_i y_i = h(x)$

for $i = 0$ to d **do**
  for $j = i + 1$ to d **do**
    $r_{i,j} \leftarrow \text{rand}(n)$
    $r_{i',j} \leftarrow \text{rand}(n)$
    $t \leftarrow r_{i,j}$
    $t \leftarrow t \oplus h(x_i \oplus r_{i',j})$
    $t \leftarrow t \oplus h(x_j \oplus r_{i',j})$
    $t \leftarrow t \oplus h((x_i \oplus r_{i',j}) \oplus x_j)$
    $t \leftarrow t \oplus h(r_{i',j})$
    $r_{j,i} \leftarrow t$
  end for
end for
for $i = 0$ to d **do**
  $y_i \leftarrow h(x_i)$
  for $j = 0$ to d, $j \neq i$ **do**
    $y_i \leftarrow y_i \oplus r_{i,j}$
  end for
end for

Algorithm 8 Secure evaluation of the multiplicative inverse (SecExp254)

Require: Shares x_i satisfying $\bigoplus_i x_i = x$
Ensure: Shares y_i satisfying $\bigoplus_i y_i = x^{-1}$

$y \leftarrow x^2$	Type-I
$x \leftarrow x \times x^4$	Type-II
$x \leftarrow x \times x^4$	Type-II
$x \leftarrow x \times x^4$	Type-II
$y \leftarrow y \times x$	Type-III
$y \leftarrow y^2$	Type-I

power 4 exponentiations as a time-memory trade-off. The sequence of multiplications of different types achieves the best performance [41] (compared to any other combination of these multiplications).

To complete the *SubBytes* transformation, we apply the affine function as described in chapter 2.3.3. This boils down to applying the affine function $Af(x)$ to each share followed by the addition of the constant **0x63**. This means that if the number of shares is even, there is no need to add this constant. Algorithm 9 describes the whole Sbox process.

Algorithm 9 SecSbox

Require: Shares (x_i) satisfying $\bigoplus_i x_i = x$
Ensure: Shares (y_i) satisfying $\bigoplus_i y_i = S(x)$

$(y_0, \cdots, y_d) \leftarrow$ SecExp254$(x_0, \cdots, x_d)$
for $i = 0$ to d **do**
  $y_i \leftarrow Af(y_i)$
end for
if $(d \bmod 2 = 1)$ **then**
  $y_0 \leftarrow y_0 \oplus$ 0x63
end if

3.4.2 Masking the key

As for the state, the key must also be masked in the same way as algorithm 4 but with the expanded key as input. This key masking must be done in a leak-free environment because manipulating the secret key leaks at the first order.

For our solution, we precompute expanded key shares and store them in ROM.

3.4.3 Masking the entire execution

The whole masking process is described by algorithm 10.

Algorithm 10 d-th order secure masked AES

Require: Plaintext p, expanded key shares k_i^r satisfying $\bigoplus_i k_i^r = k^r$
Ensure: Ciphertext x

$s_0 \leftarrow p$
*** State Masking ***
for $i = 0$ to d **do**
  $s_i \leftarrow \text{rand}(16 \times 8)$
  $s_0 \leftarrow s_0 \oplus s_i$
end for
for $i = 0$ to d **do**
  $s_i \leftarrow s_i \oplus k_i^0$
end for
*** All but last Round ***
for $r = 1$ to $N_r - 1$ **do**
  $(s_0, s_1, \cdots, s_d) \leftarrow \text{SecSbox}(s_0, s_1, \cdots, s_d)$
  for $i = 0$ to d **do**
    $s_i \leftarrow \text{MixColumns}(\text{ShiftRows}(s_i))$
    $s_i \leftarrow s_i \oplus k_i^r$
  end for
end for
*** Last Round ***
for $i = 0$ to d **do**
  $s_i \leftarrow s_i \oplus k_i^r$
end for
$(s_0, s_1, \cdots, s_d) \leftarrow \text{SecSbox}(s_0, s_1, \cdots, s_d)$
for $i = 0$ to d **do**
  $s_i \leftarrow \text{ShiftRows}(s_i))$
end for
for $i = 0$ to d **do**
  $s_i \leftarrow s_i \oplus k_i^{N_r}$
end for

3.5 Combined scheme

We described above how to protect AES implementations against fault attacks and side-channel attacks separately. We now investigate the combination of these protections and analyse how they can fit together. In 2011, Roche *et al.* [32] write one of the first papers arising questions about the combination of these protections. Without detailing how one should design a combined protection, they point out that implementations often unmask the state to perform a fault detection mechanism and then mount an attack based on that.

Lomné *et al.* [43] further compare new solutions to securely encrypt with both protections. They reject some solutions on behalf that comparisons can be bypassed with a second fault. While this is generally true, we proposed countermeasures to this kind of fault attacks in section 3.2.3, so their analysis does not reject the solutions we propose below.

The first solution just adds new blocks to the masking and fault protections. The second one is simpler and more efficient but requires slight modifications of the existing algorithms.

For our solutions, we highlight the non-linear step which is the non-trivial step when dealing with shares and error-detection codes.

3.5.1 First solution

For our first solution we propose the following (illustrated in figure 3.5) :

1. Compute a prediction of the error-detection code (32 bits per share) using a first set of random numbers r for the SubBytes step, along with a reduced set of other operations of the AES round $P[1..d]$) (see section 3.2.1) (
2. Proceed to the regular AES round with another set of random numbers r'
3. Compress each share into a 32-bit signature ($P'[1..d]$)
4. Recombine the signature shares into a 32-bit signature with a mask refreshing procedure to keep the unmasked state signature hidden.
5. Compare the newly masked 32-bit signatures

To recombine the signature shares without losing the security order, we proceed as follows (algorithm 11) :

Algorithm 11 d-th order secure signature processing

Require: d signature shares P_i satisfying $\bigoplus_i P_i = P$
Ensure: d-th order secure 32-bit signature $P = (\bigoplus_{i=0}^{d} P_i) \oplus (\bigoplus_{i=1}^{d} R_i)$

$P \leftarrow P_0$
for $i = 1$ to d **do**
  $R_i \leftarrow$ rand
  $P \leftarrow P \oplus R_i$
  $P \leftarrow P \oplus P_i$
end for

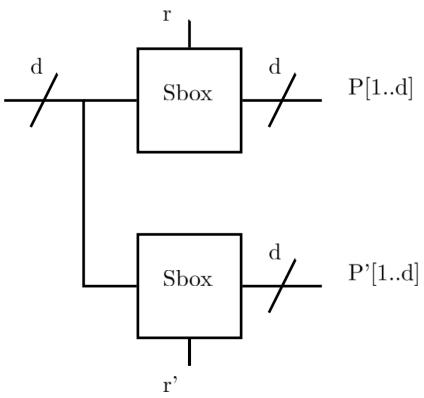

FIGURE 3.5: First combined scheme schematic

We made sure that this recombination with random numbers does break the security of the masking scheme. We computed the statistical moments of any order lower or equal to d and measured their information leakages. Any recombination of d intermediate values remained independent of the secret bytes (see section 3.6.2 for more detailed explanations).

3.5.2 Second solution

This second solution re-uses the random values used in the **SubBytes** step from the prediction to the actual computation (figure 3.6). It works as follows:

1. Compute a prediction of the error-detection code (32 bits per share) using a first set of random numbers r for the SubBytes step, along with a reduced set of other operations of the AES round (see section 3.2.1) ($P[1..d]$)
2. Proceed to the regular AES round with **the same set** of random numbers r

3. Compress each share into a 32-bit signature ($P'[1..d]$)
4. Compare each share of the 32-bit signatures ($P[i] == P'[i]$)

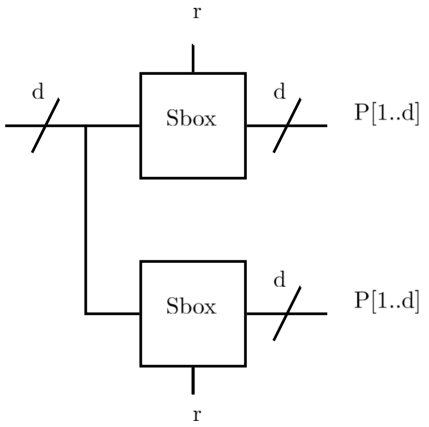

FIGURE 3.6: Second combined scheme schematic

3.6 Security and performance analysis

3.6.1 Security analysis

We proposed two solutions to combine coherently fault protection and side-channel protection. We first have a look on the fault protection metrics (which are common for the two solutions) and then compare the information leakages. Even though the two solutions aim a d-th order security, we will see that we can still discriminate the two schemes.

Fault protection

The round key schedule is fully protected. We expect the round keys not to be attacked, but if they were attacked during a round, they would be detected by the protection mechanism. If they are attacked before a round, ROM should be targeted, which is out of our fault model. We can still think of an attack on the application of the key share to a state share (during the short moment it travels through the RAM) but then it needs to be performed twice because the key share will be loaded for the prediction and real execution, so it requires 2 identical faults which is out of our fault model as well. If we

[1]These functions must be adapted according to the first or second solution.

Algorithm 12 d-th order secure masked and fault-protected AES

Require: Plaintext p, expanded key shares k_i^r satisfying $\bigoplus_i k_i^r = k^r$
Ensure: Ciphertext x

$s_0 \leftarrow p$
*** State Masking ***
for $i = 0$ to d **do**
    $s_i \leftarrow \text{rand}(16 \times 8)$
    $s_0 \leftarrow s_0 \oplus s_i$
end for
for $i = 0$ to d **do**
    $s_i \leftarrow s_i \oplus k_i^0$
    $s_i' \leftarrow s_i$
end for
*** All but last Round ***
for $r = 1$ to $N_r - 1$ **do**
    $P \leftarrow$Predict ^{1} (s')
    $(s_0, s_1, \cdots, s_d) \leftarrow \text{SecSbox}(s_0, s_1, \cdots, s_d)$
    for $i = 0$ to d **do**
        $s_i \leftarrow \text{MixColumns}(\text{ShiftRows}(s_i))$
        $s_i \leftarrow s_i \oplus k_i^r$
        $s_i' \leftarrow s_i$
    end for
    $P' \leftarrow \text{Compress}(s)$
    Compare$^1(P, P')$
end for
*** Last Round ***
for $i = 0$ to d **do**
    $s_i \leftarrow s_i \oplus k_i^r$
    $s_i' \leftarrow s_i$
end for
Predict$^1(s')$
$(s_0, s_1, \cdots, s_d) \leftarrow \text{SecSbox}(s_0, s_1, \cdots, s_d)$
for $i = 0$ to d **do**
    $s_i \leftarrow \text{ShiftRows}(s_i))$
end for
for $i = 0$ to d **do**
    $s_i \leftarrow s_i \oplus k_i^{N_r}$
end for
$P' \leftarrow \text{Compress}(s)$
Compare$^1(P, P')$

really want to protect this step against two faults, we can duplicate the load operation and compare the contents of the destination registers.

The execution-based countermeasures can be applied as described in section 3.2.3 and no further analysis will be made as it is more abstract.

The state protection can be summarized by using the previously mentioned table 3.1 replacing the values with the ones used in our design. The values that are used are: the size of the cubic signature $r = 32$; the size of the state $k = 128$; and the total (non)linear code length $n = 128 + 32 = 160$.

	Robust nonlinear
Number of undetected errors	$2^{k-r} = 2^{96}$
Number of errors detected with a probability of 1	$2^{n-1} + 2^{k-1} - 2^{k-r} \simeq 2^{159}$
Number of errors detected with a probability of $1 - 2^{-31}$	$2^{n-1} - 2^{k-1} \simeq 2^{159}$
Probability of undetected error[2]	2^{-64}

TABLE 3.2: Robust nonlinear fault detection in our scheme

We note that any 1-bit error is detected. This is valuable as 1-bit errors lead to very powerful fault attacks.

3.6.2 Side-channel protection

We considered so far our schemes to be d-th order secure according to the probing model (section 3.1.2). We desire to give more details about this resistance and to compare our two proposed combined schemes.

First, we want to make sure that our first solution does not break the d-order security, because we recombine the signature shares together. There is therefore a risk to weaken the protection as we unmask a compressed state (4 bytes of information instead of 16). Hopefully, we introduced a mask refreshing procedure to try to keep the masking valid. We show here that it works well, i.e. no d-tuple of noise-free traces can unmask the signatures.

To prove the soundness of our masking scheme, we compute the statistical moments up to the order d of a trace containing all the intermediates values of a mask refreshing scheme as in algorithm 3, except that we take 1-bit values for the experiment. We find the combination of operations that leaks the most information. To compute this leakage, we first generate all the d-tuples from the intermediate values based on a fixed secret. We then subtract the product of all the values of the tuple from the same tuple generated with a random secret. We then perform an hypothesis test on the mean of this experiment repeated n times. We also show that the $(d+1)$-order moment (that is, recombining $d+1$ well-chosen traces) contains information.

The well-chosen traces are obtained from a first run of the experiment among all the possible combinations of traces.

[2] Only true if plaintexts are random. They are random as far as we use sharing.

We show th results for $d = 2$, $n = 10000$, $H_0 : \mu = 0$, 99 % confidence interval (table 3.3) :

Order	Product of traces	Sample moment mean	p-value (99 %)
d=1	S_0	0.0046	0.0180
d=2	$(S_0 \oplus R_0) \times (S_0 \oplus R_0 \oplus S_1 \oplus R_1 \oplus S_2)$	0.0005	0.0158
d=3	$S_0 \times S_1 \times S_2$	0.1210	0.0105

TABLE 3.3: Leakages and hypothesis tests for the first combined scheme

This experiment can be run for any order d, the result is always the same : the null hypothesis is accepted for any order lower or equal to d and rejected for any order greater than d, meaning that there is no information in any moment lower or equal to d (assuming that the product of the values is related to that information). In other words, this mask refreshing scheme is compatible with a secure masking of order d.

We are also interested in the comparison of the leakages of our two solutions. Our second solution duplicates all the *SubBytes* transformation with the same random numbers and obtains the same signature shares. The first solution does not duplicate such traces until the final comparison. This is why we want to show the drawbacks of such a duplication when performing a $d + 1$-order DPA. We generated traces with a certain amount of noise for several orders of masking and compared the leakage of a order $d + 1$ attack comparing duplicated traces with unique traces (figure 3.7). The model for the leakage is a Hamming Weight leakage of a 1-bit masked secret with normally distributed noise.

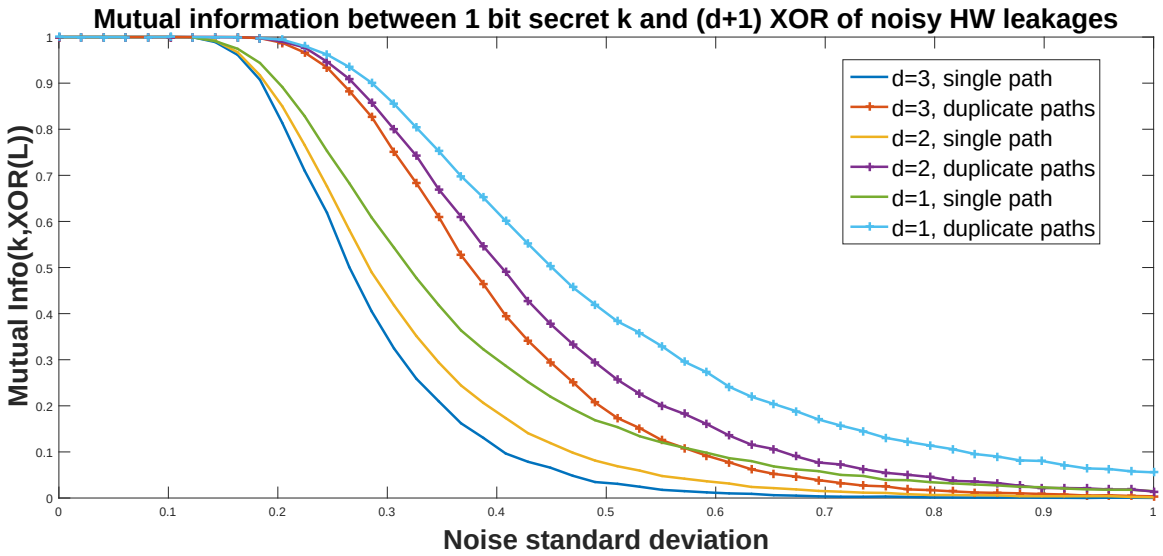

FIGURE 3.7: Leakages when performing (d+1)th order attacks for the first and second schemes

We see that the masking order has the expected impact (the mutual information decreases with an increasing slope). We also notice that having two duplicated traces shifts the curves to the right: we obtain the same leakage with an increased noise when

dealing with duplicated traces. It does not mean that, for instance, using a third order masking with duplicated traces is worse than using a first order masking with unique traces. The graph here represents only a one-bit masking in a simplified case (for the sake of simplicity and visibility), but more complex models would show a tighter difference between unique and duplicated traces of the same order, and a bigger difference between the orders of masking. The results are implementation-dependant and actual leakage traces can be impacted for other reasons (i.e. presence of a hardware cache that contains the same information as some on-chip memory).

3.6.3 Performance comparison

In this section, we want to highlight the difference in costs between the two solutions, without giving implementation results (see chapter 4.1 for numbers). This means we only mention what operations are needed and do not count how much cycles/memory it represents. This way, we stay as generic as possible and it may give meaning to hardware implementations as well.

First, we compare the computational costs of the two above-mentioned scheme and try to compare with unprotected implementations. Then, another comparison is made regarding the amount of randomness required.

Computational cost

- **For fault protection:** the key expansion step is removed, while functions computing the compression(XORs), the prediction (reduced duplication of the round), the cubic computation and the comparison of the signatures are added.
- **For side-channel protection:** the Sbox is no longer a simple lookup, it requires computing the *SubBytes* step securely, using the type I, II and III multiplications to obtain the multiplicative inverse plus an affine transform, plus an initial sharing (XORs). Furthermore, the round operations are performed for each share ($d + 1$ times).
- **For combined protection:** The fault protection functions are performed for each share. The first solution then requires to XOR $d + 1$ 32-bit signature with d random numbers and to cube two 32-bit signature (plus a comparison). The second solution requires to cube (d+1) 32-bit signatures and to compare each of them together.

Memory cost

- **For fault protection :** the expanded round key must be stored during the entire execution in ROM. Compared to some implementations where the expanded round key is computed on the fly, the memory cost increases by a factor 10/12/14 (for AES-128, -192 and -256) if we consider the ROM and RAM cost to be equivalent. For other implementations where the expanded key is computed in one

shot, the RAM requirement is just transposed to ROM. Also, the 4-byte prediction must be stored. Program memory should be roughly equivalent with the removal of the key expansion code and the addition of fault-protection functions.

- **For side-channel protection:** The expanded key shares(176 bytes, or at least the key shares, 16 bytes each) must be stored for the $d+1$ shares. The $d+1$ shares (16 bytes) must be stored in RAM. Furthermore, lookup tables for the SubBytes steps are required to mitigate the impact on the computational cost: a table for power 2 and 5 respectively (256 bytes each, in ROM or program memory). Program memory is also increased.
- **For combined protection:** the first solution requires to store only a 4-byte signature, plus the d random bytes that are used for the mask refreshing. The second solution requires to store $d+1$ signatures and the $\frac{7\times d\times(d+1)}{2}$ random bytes (per state byte) used during the *SubBytes* secure evaluation (see section 3.6.3) . There are a few extra lines of code required compared to the side-channel and fault protection.

Randomness cost

- **For fault protection:** no randomness is required, although the nonlinear code works better with random plaintexts. In case of fixed-plaintext attacks, masking and its randomness ensure the error-detection rate of the nonlinear code. Otherwise, the probability of missing an error is reduced to the performance of a non-robust linear code of the same size, namely 2^{-32} instead of 2^{-64}.
- **For side-channel protection:** $d \times 16$ bytes are required to create the shares. The expanded key shares are created offline so it is not taken into account. The **SubBytes** step requires a lot of random numbers: $d \times (d+1)$ per Type-II multiplication, $\frac{d\times(d+1)}{2}$ per type-III multiplication. 3 Type-II and 1 Type-III multiplications are called for each byte of the state. This gives in total $\frac{7\times d\times(d+1)}{2}$ random bytes per state byte and per round. This represents a huge amount of randomness that can impact very negatively the performance of a system if the random generation is not done in parallel. Improvements regarding the number of required random bytes can be found in other masking methods such as [38].
- **For combined protection:** the first solution doubles the mask refreshing randomness because it does not keep the random numbers fixed between the prediction and the real execution. This has a major impact as it doubles the $\frac{7\times d\times(d+1)}{2}$ bytes required per state byte and per round. On top of that, it needs $4 \times d$ random bytes per round to mask the signatures. The second solution, on the other hand, does not add anything.

3.6.4 Bottom line

We proposed here two schemes that achieve a very efficient fault coverage (for our fault model) and also achieves d-th order DPA security at the same time. The first solution

is more robust to DPA but costs much more in randomness. The second solution leaks a bit more information when computing signatures but it is not clear whether it has a significant impact on side-channel resistance, as it leaks a compressed state. On the other hand, the randomness cost of this second solution is significantly reduced.

Chapter 4

Implementation and results

We described in chapter 3 methods to protect the AES (and more generically substitution-permutation ciphers) against fault attacks and power analysis attacks. We are now interested in studying how these concepts translate to real implementations. We chose to test these protections on implementations of AES-128, and more particularly on the encryption process.

4.1 Implementations and performance

Several implementations have been made to measure the performance of our algorithms. We first proceeded to an implementation of our fault protection scheme in Assembly language. Unfortunately, the side-channel countermeasures were too heavy to be implemented at any order in Assembly language. While it is practically possible, we decided to switch to C language for practical reasons. There is a large performance overhead when it comes to producing Assembly code in C, as it is less optimized for the hardware we deal with.

All the implementations that are presented in this master thesis are fully provided in the appendices section. They are all designed to work on a ATMega 644P microcontroller. This chip is a low-cost device that fits particularly well on low-end smartcards and many cryptographic implementations fit on this device. Furthermore, several scientific papers compare the performance of their solutions on this chip; therefore, it will be easier to compare our performance with theirs.

Here are the specifications of the chip and the work environment that were used :

- 8-bit AVR Atmel microcontroller with MIPS architecture.
- Up to 1 cycle per instruction
- 20 MHz max clock frequency
- 64 Kb of Flash program memory
- 4 Kb SRAM , 32 general purpose registers

- Hardware 16-bit multiplication
- Programming : AVR Atmel Studio 5.1 with AVR-GCC compiler (O1 or O2 optimisation) or AVR Assembler

All the implementations were tested with a NIST test vector:

- Encryption key: 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
- Test vector: 6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a
- (Expected) cipher text: 3a d7 7b b4 0d 7a 36 60 a8 9e ca f3 24 66 ef 97

When required, the randomness was created by simply using a random seed as input to the program. To expand the randomness, a simple lookup in a S-box table is performed. While not being a good pseudo-random generator, this allows us to compare the performance of the programs without including the cost of generating randomness. We chose to do that because the random number generation can be as well done in parallel in secure chips. Furthermore, this leaves the choice to the people interested in developing their own PRNG to decide which impact on the performance it would have. We will discuss the time overhead due to random number generation separately. Further information on how good the randomness must be can be found in the article of Grosso, Standaert and Faust [25].

4.1.1 Assembly Fault-protected AES

We based our implementation on the Furious AVR AES implementation from 'Point-at-infinity'[1]. 32 general purpose registers were just enough to hold the principal values during the computation, and very few loads/stores in RAM were used. The full code is in appendix A.

We compare the performance of our code versus the original 'AVR AES Furious' implementation in terms of cycles and memory usage (table 4.1):

	Cycle count	Program memory (bytes)	Data space (bytes)
Furious	2738/3548[2]	588	528
Our implementation	8426	890	704

TABLE 4.1: Assembly code performance comparison

The data space increases because we store the precomputed round keys (176 bytes) among others, but we remove tables such as the round constant that are used for the key expansion. Since a copy of the state has to be made at the beginning of each round, memory costs further increases by 16 bytes.

[1] http://point-at-infinity.org/avraes/
[2] Without and with key expansion

Here is the detail of the cycle count for the functions within one round:

- Predict: 154 cycles
- GF cube: 126 cycles. A multiplication between two 32-bit numbers takes 36 cycles, that must be called twice, but there are also saves and loads.
- SubBytes+ ShiftRows: 76 cycles
- MixColumns: 155 cycles
- secAddRoundKey: 76 cycles
- Compress: 22 cycles
- EDN (Compress+ GFcube + other): 172 cycles
- State save/load: 40 cycles each

We immediately notice that the GF(2^{32}) multiplication is quite costly, as the GF cube function takes 2560 cycles per execution alone. This is why we suggest a simple time-performance trade-off. If we use only the linear prediction,the encryption takes 5046 cycles, which is closer to a Furious execution with key expansion (3548 cycles).

We also note that such a overhead would be drastically lowered if a 32-bit microcontroller was used. In this case, the multiplication of two 32-bit numbers would only take 2 or 3 cycles with the same kind of architecture as the one used here. We show in listing 4.1 how we computed the GF(2^{32}) multiplication (there are dedicated registers to hold one 32-bit signature [LP1 - LP4]).

LISTING 4.1: AVR 8-bit Assembly code to multiply two 32-bit numbers

```
;; GF multiplication
;; Pre:  First 32 bit signature  contained in [LP1 - LP4],
;;second signature contained in (LLSB) temp1, temp2, Yl,YH (MMSB)
;; Post : cubic signature contained in (LLSB) XL XH ZL(r30) ZH(r31)(MMSB)

modmult:

sts multregs, ST11 ; Backup r00 and r01
sts multregs+1, ST21
mul     LP4, temp1 ; multiply A LLSB with B LLSB
movw    XL, r0 ; store first result in XL:XH(result LLSB:LSB)
mul     LP3, temp2 ; multiply A LSB with B LSB
movw    r30, r0 ; store second result in r30:r31 (result MSB:MMSB)
mul     LP2, temp1 ; multiply A MSB with B LLSB
add     r30, r0 ; add without carry mult LSB to result MSB
adc     r31, r1 ; add mult MSB with previous carry to result MMSB
mul     LP4, YL ; multiply B MSB with A LLSB
add     r30, r0 ; add mult LSB to result MSB
adc     r31, r1 ; add mult MSB to result MMSB
mul     LP1, temp1 ; multiply B LLSB with A MMSB
```

```
add     r31, r0   ; add mult LSB to result MMSB
mul     LP2, temp2 ; multiply  A MSB with B LSB
add     r31, r0 ; add mult LSB to result MMSB
mul     LP3, YL ; multiply B MSB with A LSB
add     r31, r0 ; add mult LSB to result MMSB
mul     LP4, YH  ; multiply A LLSB with B MMSB
add     r31, r0 ; add mult LSB to result MMSB
eor     YH,YH ; B MMSB is not used anymore and we need a 0
mul     LP3, temp1 ; multiply a LSB with B LLSB
add     XH, r0 ; add mult LSB to result LSB
adc     r30, r1 ; add mult MSB to result MSB
adc     r31, YH ; add  carry  to result MMSB
mul     LP4, temp2 ; multiply B LSB with a LLSB
add     XH, r0 ; add mult LSB to result LSB
adc     r30, r1 ; add mult MSB to result MSB
adc     r31, YH; add  carry  to result MMSB
eor     r1, r1 ; MUST clear r1
lds ST21, multregs+1     ; restore states
lds ST11, multregs
ret
```

4.1.2 C SCA-protected AES

To measure the performance of the masking countermeasure, we used our fully-protected code and removed the fault protection related code. We also compare the performance of our code to the reference C implementation we used [3]. This reference implementation was not perfect and we made some adjustments to make it fully compatible with our countermeasures (multiplication algorithms, state representation). The produced code follows algorithm 10. It has been compiled with optimisation -O2 (and exceptionally -O1 for two shares as it is surprisingly faster).

The performance of the C code translated to Assembly is far worse than one could obtain in writing directly in Assembly. However, our goal is not to achieve the best performance in an implementation but to show a proof-of-concept and make measurements on the scalability of this architecture with higher-order masking. The reference implementation needs almost 15 000 cycles to encrypt a plaintext in AES-128, which is already 5 times greater than the Assembly version. Using our code without masking (i.e. only one share) takes around 45 000 cycles (3 times greater than the reference) because it is clearly not optimised for such use. We could increase the performance in terms of cycles by unrolling the code at the expense of memory usage. That would be effective especially at low orders of masking (for higher orders, the amount of computation required largely takes over the cost of making loops).

The cost of randomness was not taken into account. [25] suggests that counting 10 cycles per byte of randomness is a good estimate of the cost of generating randomness.

[3] https://github.com/kokke/tiny-AES128-C

As the number of required random numbers grows quadratically with the number of shares, it can have an impact on the performance of the schemes.

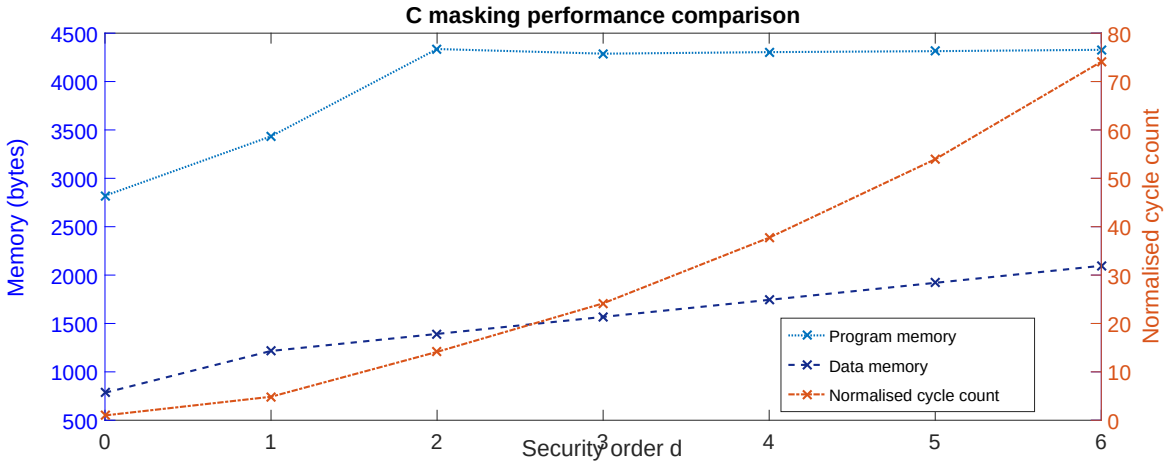

FIGURE 4.1: Performance comparison of C masked AES for different security orders

While the complexity of the higher-order masking is quadratic with respect to d, the total cost of the execution is less than quadratic (between linear and quadratic), due to the other operations that are executed besides the masking. In terms of normalised cycle counts x, it is approximately proportional to $10 \times x$ in this range of security orders. The number of cycles used as reference for the normalisation is 45 131 ($d = 0$).

For information, we give hereafter (table 4.2) the comparison between the cost in terms of cycles without taking randomness into account and the cost of the randomness assuming 10 cycles per random byte)and counting the number of random bytes required as explained in section 3.6.3:

Security Order	Original number of cycles	Cost of randomness
1	218 998	11 216
2	637 756	33 632
3	1 090 359	67 248
4	1 702 999	112 064
5	2 434 405	168 080
6	3 344 008	235 296

TABLE 4.2: Randomness cost for masked AES

4.1.3 C fully protected AES

For our combined scheme we chose to measure the performance of our first solution. The first code can be found in appendix B. We also implemented the second one (for which the code is not provided as it is roughly the same as the first). It showed that the performance of the two solutions are roughly the same (in terms of memory and

cycles), with a variation of around 1 %. Of course, those simulations do not take into account the cost of randomness, as mentioned before.

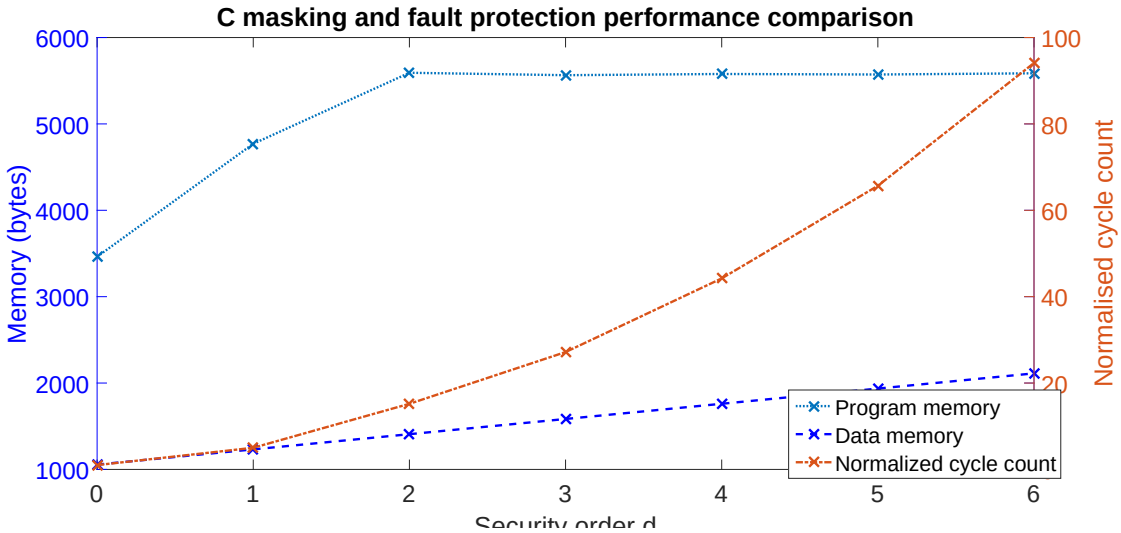

FIGURE 4.2: Performance comparison of C protected AES for different security orders

The number of cycles used as reference for the normalisation is 92 541 ($d = 0$).

Program memory increases by approximately 1 kilobyte, which is reasonable. The performance overhead induced by fault protection roughly doubles the computational cost. The combination method does not have a crucial impact but impacts more higher order levels of masking. The complexity tends more to quadratic complexity than the SCA-protected only implementation, but hopefully the two protection schemes do not make each other any worse (look at table 4.3 to convince yourself, the cost does not increase quadratically for this magnitude of orders).

Again, we give below the comparison between the cost in terms of cycles without taking randomness into account and the cost of the randomness, for the first (1) and second (2) solutions that we proposed, computing that cost in the same fashion as before (table 4.3):

Security Order	Original cycle count	Cost of randomness (1)	Cost of randomness (2)
1	463 891	22 404	11 216
2	1 405 768	67 208	33 632
3	2 514 740	134 412	67 248
4	4 093 337	224 016	112 064
5	6 077 320	336 020	168 080
6	8 708 580	470 424	235 296

TABLE 4.3: Randomness cost for the combined solutions

4.1.4 Comparison with other implementations and practical aspects

The most obvious question that can be raised about these results is the performance of this implementation. Is it fast enough to be used in embedded systems, and does it behave better than other protected implementations?

The first question is straightforward. We can just divide the number of cycles by the frequency of the CPU integrated to a smartcard, for instance. In this case, we take a reference value of 20 MHz frequency (corresponding to what we were able to work with, as well as a mean value of what can be found on the market). This leads to the results in table 4.4, when considering the second solution with random generation taken into account :

Security Order	Time required at 20 MHz (ms)
1	23,75
2	71,97
3	129,1
4	210,27
5	312,27
6	447,19

TABLE 4.4: Time needed for the protected encryption of a 128-bit block with the second solution

Generally, it is considered that users will accept a delay of a few hundred milliseconds for smartcard applications. However, this application would probably require more than just an encryption, but the other communications would not require heavy computations like in the encryption (see standards ISO 14443 and 9798). The time required for encrypting one block is therefore a relevant metric of the time needed for the application. Orders 1 to 3 are convenient while orders 4 to 6 are probably too long. However, the implementation we provided is not optimized for performance and several improvements can be made, such as loop unrolling. To have a significant gain on time, we could also apply the fault attack protection only where the practical attacks take place (e.g. protecting from the 7^{th} round would be enough). This would reduce the time required by around 25 %. Of course, another solution would be to use a higher-end processor that runs faster, or a 16-,32-bit processor that would significantly improve the performance.

Comparison with SCA-protected implementations

We already mentioned that there are no scientific papers that question the performance of combined schemes. Therefore, we cannot easily compare our results with other implementations. But we can try to match our results with SCA-protected implementations that were implemented and measured in [25]. Furthermore, we know that our fault protection countermeasure roughly doubles the time required when added to a SCA-protected implementation. It could not be the case with other masking schemes, but at least we can give a rough estimate of how those other masking schemes would

behave when improved with FA countermeasures. We basically doubled the numbers from [25] and compared them with our results.

The data that we could compare were either results without randomness and with rolled loops, or with randomness but with unrolled loops. As we only have results for rolled loops, this it what we chose to compare. We compared the following :

1. Our second solution, without randomness
2. Rivain-Prouff (RP), i.e. the same masking method as ours, but implemented differently (double cycle count from original source)
3. Kim-Hong-Lim (KHL) masking [38], that is close to Rivain-Prouff but improves it using subfields to improve multiplications and reduce the randomness cost (double cycle count from original source)
4. Genelle-Prouff-Quisquater (GPQ), which uses both boolean and arithmetic maskings, to speed up the S-Box masking [37] (double cycle count from original source)

The results of this comparison is shown in figure 4.3.

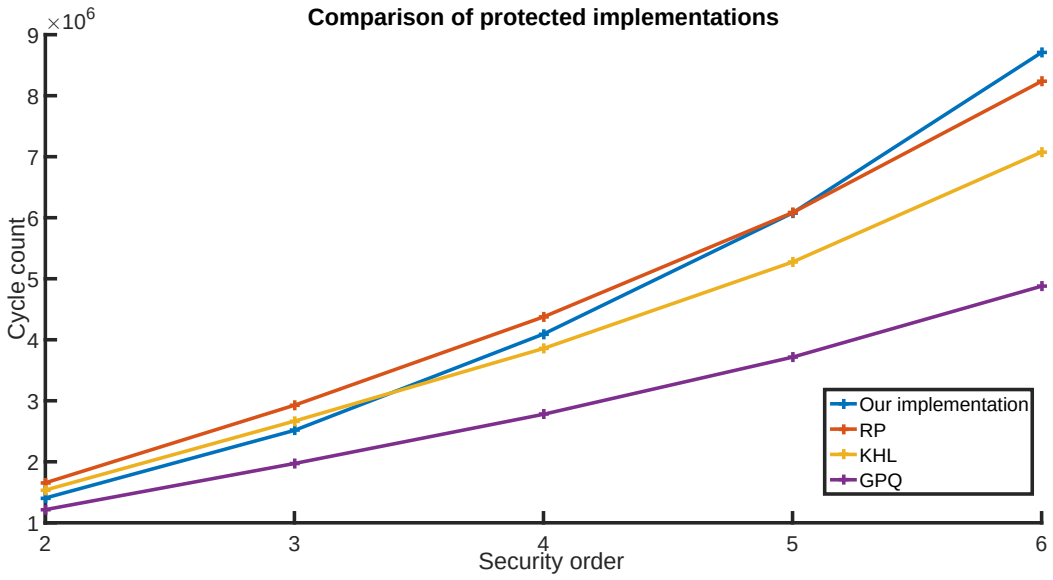

FIGURE 4.3: Performance comparison of different implementations of protected AES

We hopefully see that our implementation behaves as in Grosso *et al.* and that multiplying the cycles count by two is not completely accurate. Furthermore, the GPQ implementation has a significantly better performance, with a cycle count that is almost divided by two. Using this masking technique could make the higher order masking levels more interesting, dividing the timings in table 4.4 by two (if the FA protection behaves the same way).

4.2 Fault injection and fault resistance

After having measured the cost of those solutions, we are now interested in measuring the security performance of our countermeasures. We intentionally let the DPA security apart, as it is harder to evaluate as a whole. However, we trust the "proven" higher-order security scheme [44] and showed the impact of our combined schemes on information leakage (section 3.6.2). We are therefore interested in seeing how the higher-order masking could impact the fault protection. We leave apart the vulnerability that we exposed in chapter 3.3, but we recall that it indeed exists.

4.2.1 Fault model-based injection

To inject faults on our system, we chose to modify the program to change the contents of the memory at selected places in the code. This is a powerful fault model, but it allows us to show the weaknesses of our countermeasure.

As the statistical probability to successfully induce a fault is very low (2^{-64}) we did not run this tremendous amount of experiments to show that this probability is achieved. We selected faults that are known to work instead. However, the fault model that we described in section 3.1.1 does not allow to induce faults effectively (because of this very low probability). We therefore decide to extend the fault model to a more powerful one to show where and when faults can be induced without being detected.

As we said earlier, any fault on one byte of the extended state (state + signature) will be detected. However, if we consider a larger effect of the fault (several bytes from the state or mixed bytes of the state and the signature), we are then able to induce faults that are comprised in the statistical probability to induce a fault. A successful fault must satisfy the following condition : any even number of bit flips belonging to the same state column at the same bit of different column elements will not be detected.

A 'simple' example (it is simple to explain but harder to put in practice) consists of flipping a bit of two bytes of the same column of the state. The XOR will give the same result as if no fault was induced. The same can be done with two entire byte flips, but it is hard to explain how such things could be done with a single laser anyway.

Basically, we have to admit multiple-byte or multiple-bit faults in our model to mount a successful attack, unless we consider the security hole presented in section 3.3. Inserting a fault in these vulnerable spots leads to undetected errors (see also chapter 5.3).

The introduction of multiple shares does not make the attack more simple or more difficult, but shares should be manipulated with care : if the shares of the state travel through the same registers at different moments in time, we can induce faults on the same bytes of different shares without moving the laser. If we use our first solution, those faults will be XORed to generate the signature. If the faulty value is a bit(byte)-flip in the two different shares, that will not be detected. However, it is not clear whether two faults applied on random shares can be useful for an attack. If we use

our second solution, each share is protected by its own signature, so this attack has no effect. This somewhat makes the second solution more interesting.

4.3 Bottom line

This chapter allowed us to measure and compare the performance of our codes, in terms of execution time, ROM and RAM requirements. We were able to analyse the impact of the different components of our solutions. We were also able to make a comparison of the cycle counts of our combined solution versus other masking schemes and saw that the performance was as good as expected. There exist more efficient masking schemes nowadays that could further improve the performance of these implementations. We showed that, even at high order, the practical use of combined schemes is possible without being too slow.

For the fault resistance, we also exposed what kind of faults were not detected and verified it in practice.

Chapter 5

Real-world fault injection

After having described and compared algorithmic countermeasures and assessed their performance and security, we now present a real-word setup made from scratch, that successfully injects faults on unprotected AES implementations. We were able to generate faults as required for Piret and Quisquater fault attack (section 2.5.2). Moreover, we were able to induce faults in our fault-protected implementation based on the weakness we identified in section 3.3).

5.1 Setup description

All the material that was used for the setup belongs to the Crypto group [1] or the Welcome platform [2].

The necessary material includes :

- A depackaged ATMega644P chip
- A test board that allows programming and operating the chip using serial communications
- A laser emitter : The Alphanov PDM+ laser, controlled via serial communications and directed via optical fiber. Wavelength: 976 nm, max pulse power : 3340 mW.
- A XYZ table that allows moving the test board: 3 Newport Miniature Steel Linear Stages
- A controller for the XYZ table : the Newport ESP300 controller, controlled through a National Instruments GPIB-USB converter
- A computer running Matlab to operate the serial and GPIB communications.

This setup is valued a few thousands euros, but the results we obtained could be obtained with a much cheaper setup: the XYZ table is used for convenience and some simple optics can do the job for less than 100 dollars [45].

[1] http://www.uclouvain.be/crypto/
[2] https://sites.uclouvain.be/welcome/

Figure 5.1 shows the setup that we made. The optical fibre is placed perpendicularly to the depackaged chip. The table allows moving the chip to target a precise location on the target. The optical fibre is the blue wire and is connected to the laser emitter.

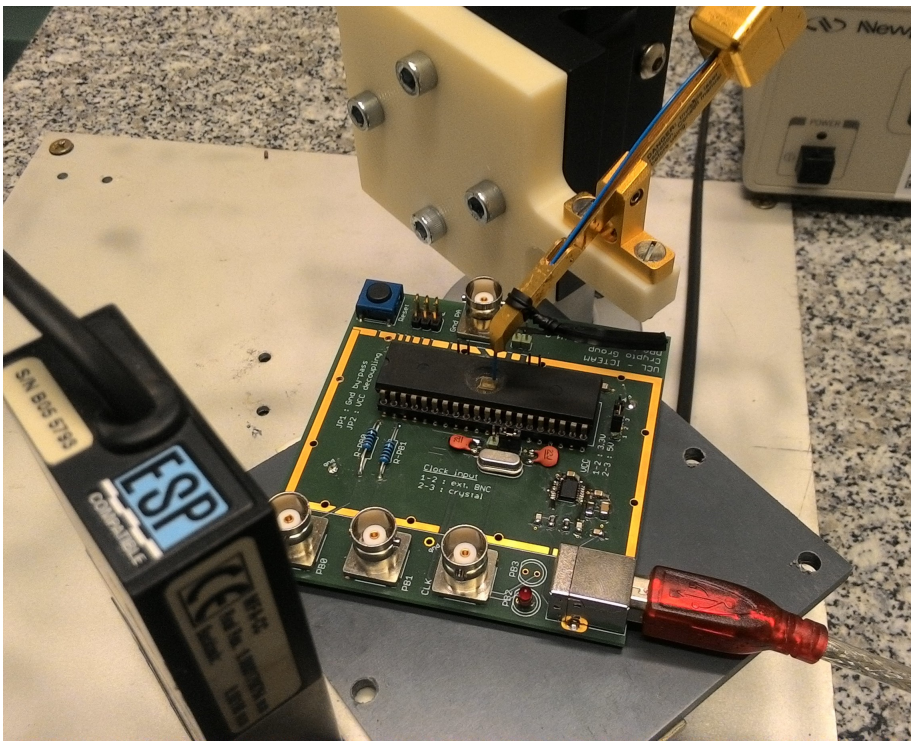

FIGURE 5.1: Our setup

5.2 Depackaging

The first necessary step to inject faults with a laser is the depackaging of the chip. First, the upper plastic layer is drilled (e.g. with a Dremel drill) until a thin layer of plastic is left (less than a millimetre). Then, the remaining package is carefully removed using acid. The goal is of course to allow the light from the laser source to penetrate the silicon layers of the chip, but also to make a visual inspection of the layout of the chip. In our case, the result of such a depackaging (taken from a digital microscope) is shown in figure 5.2.

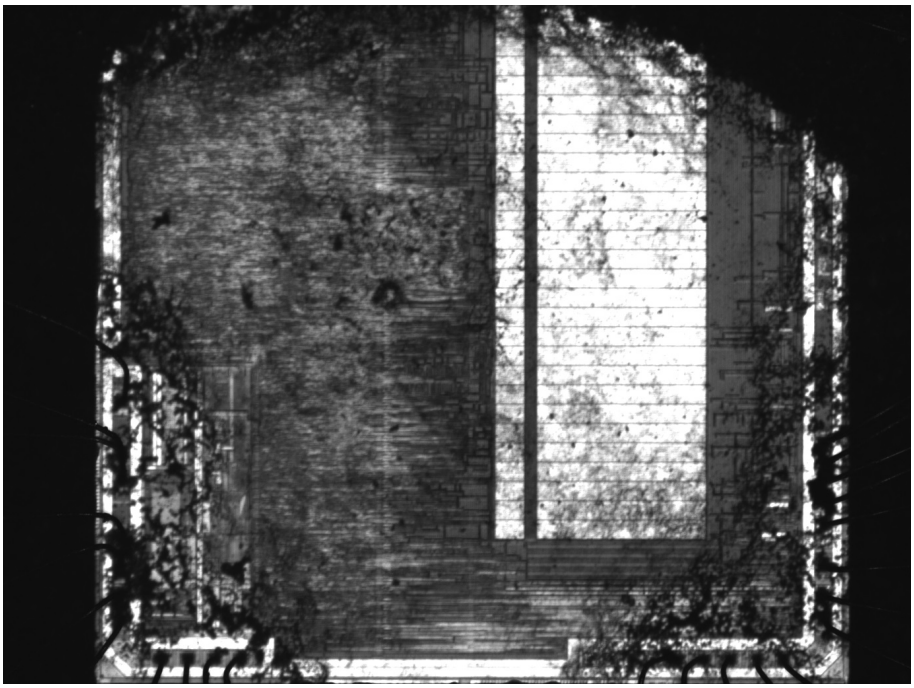

FIGURE 5.2: Microscope view of a depackaged AT644P

Unfortunately, the visual inspection did not lead to successful targeted attacks. Our primary objective was to find and target the SRAM to insert faults, however, we found another way to do so (see section 5.3).

5.3 Laser attack

This section aims at explaining how to obtain a successful laser attack with our setup (as described in section 5.1). At the time of writing this chapter, we found an article that proposed the almost the same methodology as ours to insert faults. The article called "Reproducible single-byte fault injection" [46] can bring other insights of practical fault insertion techniques.

First, a slightly modified version of an AES Assembly implementation is loaded into the chip. This version sends a trigger via serial communication to the computer when it reaches the ninth round of the cipher. For reliability purposes, the code does not loop. We enabled a watchdog timer that resets the chip periodically (every second in our case, but it could be much faster as the encryption takes only a few milliseconds) to avoid memory and program corruption by the laser. This way, the serial port outputs of the cipher yield cleaner results. Without faults, it simply outputs the correct, known ciphertext.

Second, the laser is activated in pulse mode. A short burst of a few hundred nanoseconds is enough. We chose to drive the laser at full power, i.e. 6,35 A, yielding a light output of around 3,3 W. Instead of modulating the power to tune the results, we decided to adjust the height of the laser, for practical reasons.

We originally wanted to target the SRAM to modify some values associated with the AES state. Unfortunately, exhaustive search on the surface of the chip did not give good results (serial communication going mad, for example, discouraged us to go on with that method). This could mean that the SRAM does not appear on the surface of the chip or that other effects are induced when targeting it.

Instead, we found zones that simply induce a fault during the computation of some values. By trial and error, we found several zones that worked for our attack. Figure 5.3 shows some of them. They are all close, or inside the shiny big zone on the chip.

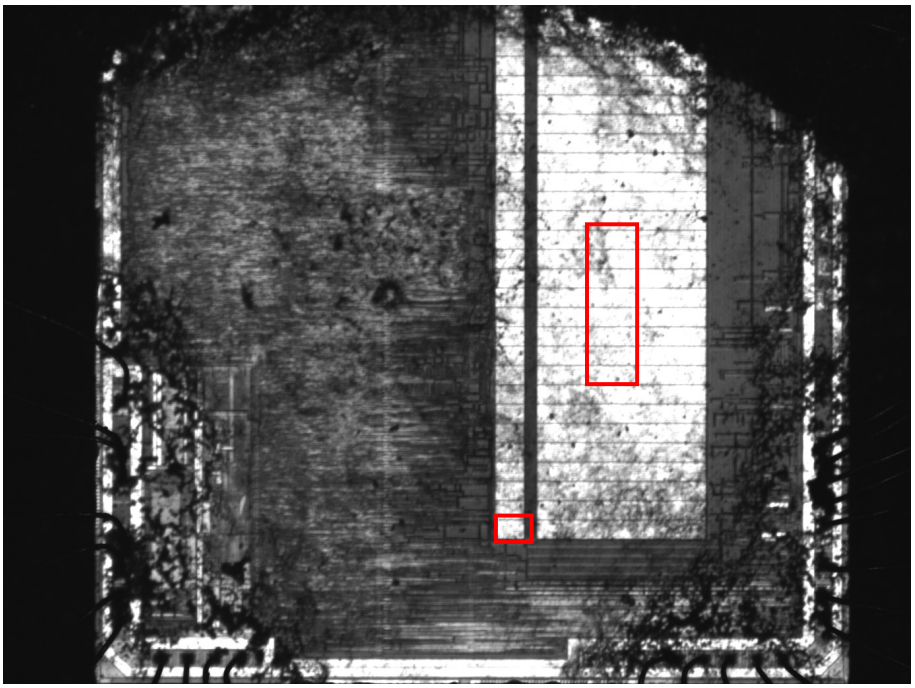

FIGURE 5.3: Microscope view of a depackaged AT644P

Once the laser is correctly positioned, adjustments need to be made until the ciphertext outputs contain 4 faulty bytes that match the diffusion pattern of AES. To do so, we first put the laser far from the chip and then approach it slowly until faults begin to appear.

This trick was necessary in our case, as the synchronisation between the chip and the laser was not perfect (latency of communicating via the computer and activating the laser). To make the attack more efficient, we repeated laser pulses many times during a time window around the ninth AES round. That leaded to undesired faults such as fully faulty ciphertexts or ciphertexts with only one fault, but also gave good results on the other hand. We discuss how to improve that in section 5.5. When the laser is correctly positioned in all the axes, we need on average a dozen ciphertexts to obtain one with 4 faulty bytes.

5.4 Results

To exploit the fault produced with the setup, we implemented in Matlab the Piret-Quisquater attack (which can be found in Appendix C) as described in chapter 2.5.2. We used the simple version where the fault is injected between the diffusion layers θ_{R-2} and θ_{R-1} to obtain 4 subkey bytes with 2 faulty ciphertexts. This is enough to prove that the attack worked as we know the secret key in advance.

For instance, with the following :

- Plaintext: 32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34
- Key: 2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
- Ciphertext: 39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
- Faulty ciphertext 1: 39 25 84 6A 02 DC 7B FB DC A3 85 97 18 6A 0B 32
- Faulty ciphertext 2: 39 25 84 80 02 DC 94 FB DC 31 85 97 75 6A 0B 32

We obtained the following subkey bytes: B6, 3F, 25, A8 , which correspond to subkey bytes 4, 7, 10, 13 of the last round subkey.

We guess that the faults were produced by perturbing the computation of a value rather than modifying the value of a byte in memory. Furthermore, as we cannot precisely define the origin of the fault (we can only tell the column in which the fault took place), it is rather hard to determine the effect of the laser on the values.

However, we tried to determine the effect of the laser with another experiment. We filled the whole SRAM with a constant value (corresponding to alternated '0' and '1' bits) and tried to see if there was any difference in the memory after a laser pulse by reading all the SRAM bytes. While our program gave some outputs, they were not constant enough (i.e. having different outputs without moving the laser) to infer that the SRAM was modified. We rather think that the laser modified the result of comparisons, resulting in false positives in the modification of the SRAM. We do not reject the possibility of modifying the SRAM with the laser, but we did not succeed in doing that.

This attack worked on the unprotected as well as protected implementations, due to the weakness of the security countermeasure exposed in chapter 3.3.

5.5 Possible improvements

Our setup is far from being perfect for several reasons. We list here a number of improvements that could increase the repeatability of the faults that we try to inject :

- Fixing the test board to a reference position such that it does not move. We taped the board with adhesive paper. This results in inconsistent reference positions between tests.
- Making the board perfectly plane to avoid deformation of the laser
- Using optics to obtain a calibrated laser spot. We do not have an idea of the size of the spot.
- Holding the optical fibre in a more robust way. It is currently bent with a special tool that does not ensure the perpendicularity with the board.
- Triggering the laser with the TTL input, directly from the test board, to have a very precise shot in time.
- Designing a new test board that has a USART connector, to enable in-circuit debug. This could allow us to observe the modifications that are made to the memory or to highlight the operations that are impacted by the laser.
- Making an automated test that scans the entire chip and finds a spot that produces useful errors. This has been done but not used because it requires some of the other improvements to work properly.

5.6 Bottom line

While being imperfect, our setup achieves its goal: successfully insert meaningful faults into a standard AES implementation. As we expected, it is also successful on a fault-protected implementation when it is correctly timed. It did not require particular skills with laboratory equipment to make this experiment a success, but it certainly took time to refine the method up to a successful one. The setup can be made more effective, or cheaper if one's goal is to make the cheapest attack possible.

Chapter 6

Conclusion

Throughout this thesis, we presented algorithmic countermeasures to side-channel and fault attacks and applied them to the Advanced Encryption Standard. We picked countermeasures that could fit together and produced a combined protection in a coherent manner. Efficiency (cost) and having a high degree of security were two major criteria for that selection. We studied their mechanisms and performance independently then proposed two solutions to combine them without losing security.

We implemented our theoretical solutions and were able to measure the cost of those solutions in terms of computations and memory usage. We could also verify the resistance and weaknesses of the solutions related to faults. This was used as a proof-of-concept of our combined schemes. We compared the performance of our implementation with other implementations.

We also created a fault attack setup from scratch, that provided good results in terms of repeatability and success rate. We showed how to improve it or how to reduce the costs.

A surprising finding was the security hole in the robust nonlinear code proposed by Karpovsky *et al.* in [19]. However, this does not discard our results in terms of combined solutions, but rather suggests that the fault protection scheme should be revised.

The objectives of this work, which consisted in analysing and implementing countermeasures and to combine them coherently were fulfilled.

6.1 Future work

This thesis leaves several questions open and also suggests improvements in what was realized.

First, it would be interesting to study combinations of masking with other fault countermeasures; particularly those that so not split the protection into rounds.

Another point of interest would be to quantify and compare the information leakage of fault countermeasures to side-channel analyses.

Regarding the experimental setup we created, many things could be improved. The setup could me more stable (fixed, plane board), more precise (improved optical beam), better synchronised and the trigger could also come from an automated power analysis (e.g. detecting the rounds with power analysis and then triggering the laser at the desired round)... The detection of a point of interest on the die could also be automated.

Finally, other methods could be tried to combine countermeasures. Among them, an interesting solution is the optimal tamper-resistant codes suggested by Gammel and Mangard in [47], where a linear code is used to protect from both probing and fault attacks. However, the implementation of such a solution should avoid generating those codes in an round-iterated fashion, and it is not clear whether such a thing is possible.

6.2 Acknowledgements

I would like to thank Pr. François-Xavier Standaert for his support and his guidance through my work; Romain Poussier and Santos Merino Del Pozo for their assistance in some aspects of my thesis; Pierre Gérard, Pascal Simon, David Spöte and Thomas Dieuzeide for their technical support with the laboratory equipment; Pr. Jean-Didier Legat for reading my thesis; and all the people that supported me through this year.

Appendix A

Code : Assembly fault-protected AES

...Memoire\simu\ProtectedAES\ProtectedAES\ProtectedAES.asm 1

```
;
; ProtectedAES.asm
;
;  Created: 19-11-15 15:10:18
;  Author: Georges-Henri Leclercq
; based on http://point-at-infinity.org/avraes/rijndaelfurious.asm.html
;

.include "m644Pdef.inc"

.def ST11=r0 ; Those should not be used because they are used for multiplication;
.def ST21=r1 ; When a mult is executed, r0 and r1 are saved to the RAM then
  restored
.def ST31=r2
.def ST41=r3
.def ST12=r4
.def ST22=r5
.def ST32=r6
.def ST42=r7
.def ST13=r8
.def ST23=r9
.def ST33=r10
.def ST43=r11
.def ST14=r12
.def ST24=r13
.def ST34=r14
.def ST44=r15
.def H1=r16
.def H2=r17
.def H3=r18
.def I=r21
.def LP1=r25
.def LP2=r24
.def LP3=r23
.def LP4=r22

;;can be changed in inner functions
.def temp1=r19
.def temp2=r20


.ORG 0x0000
; global interrupt disable
    cli
; initialize stack
    ldi     r31,HIGH(RAMEND)
    out     SPH,r31
    ldi     r31,LOW(RAMEND)
    out     SPL,r31


    rjmp    main

;****************** MAIN (START) ****************************
```

...Memoire\simu\ProtectedAES\ProtectedAES\ProtectedAES.asm 2

```
main:
; LOAD THE KEY FROM THE DB BELOW

    ldi ZH, high(key<<1)    ; load key into ST11-ST44
    ldi ZL, low(key<<1)
    lpm ST11, Z+
    lpm ST21, Z+
    lpm ST31, Z+
    lpm ST41, Z+
    lpm ST12, Z+
    lpm ST22, Z+
    lpm ST32, Z+
    lpm ST42, Z+
    lpm ST13, Z+
    lpm ST23, Z+
    lpm ST33, Z+
    lpm ST43, Z+
    lpm ST14, Z+
    lpm ST24, Z+
    lpm ST34, Z+
    lpm ST44, Z+


    clr r0
    sts keyoffset, r0
;START to load the plaintext from the inside set, see below
; ST (as key matrix) is filled in column-order by definition by Rijndael
    ldi ZH, high(text<<1)   ; load plaintext into ST11-ST44
    ldi ZL, low(text<<1)
    lpm ST11, Z+
    lpm ST21, Z+
    lpm ST31, Z+
    lpm ST41, Z+
    lpm ST12, Z+
    lpm ST22, Z+
    lpm ST32, Z+
    lpm ST42, Z+
    lpm ST13, Z+
    lpm ST23, Z+
    lpm ST33, Z+
    lpm ST43, Z+
    lpm ST14, Z+
    lpm ST24, Z+
    lpm ST34, Z+
    lpm ST44, Z+
;END


    ; now the registers ST11-ST44 contain the plaintext given below

    rcall   encrypt     ; encryption routine
    rjmp    main

text:
.db $6b,$c1,$be,$e2,$2e,$40,$9f,$96,$e9,$3d,$7e,$11,$73,$93,$17,$2a
key:
```

```
...Memoire\simu\ProtectedAES\ProtectedAES\ProtectedAES.asm                    3
.db $2b,$7e,$15,$16,$28,$ae,$d2,$a6,$ab,$f7,$15,$88,$09,$cf,$4f,$3c

; global variables
.DSEG
statestore:
.BYTE 16 ; duplicate state for prediction
ptrstore:
.BYTE 2 ; reserve 2 bytes to store pointers
signature:
.BYTE 4  ; place to store prediction of signature
multregs: ; to save R00 and R01
.BYTE 2
keyoffset: ; to remember ROM expanded key offset in secure AddRoundKey
.BYTE 1
.CSEG
;******************** MAIN (END) ********************************


;;; ***************************************************************************
;;;
;;; ENCRYPT
;;; This routine encrypts a 128 bit plaintext block (supplied in ST11-ST44),
;;; using an expanded key given in YH:YL. The resulting 128 bit ciphertext
;;; block is stored in ST11-ST44.
;;;
;;; Parameters:
;;;         YH:YL:  pointer to expanded key
;;;         ST11-ST44:  128 bit plaintext block
;;; Touched registers:
;;;     ST11-ST44,H1,H2,H3,I,ZH,ZL,YH,YL
;;; Clock cycles:   2739

encrypt:
    ldi I, 10 ;; load number of round iterations. Key is already expanded and set.
    rcall secAddRoundKey ; executed at each iteration including first pass.

encryp1:
    rcall savestate
    rcall predict
    rcall GFcube
    sts signature, XL ; Save signature (stored in ZH ZL XH XL) to data space
    sts signature+1, XH
    sts signature+2, ZL
    sts signature+3, ZH
    rcall loadstate
    ldi ZH, high(sbox<<1)   ; SubBytes + ShiftRows. ShiftRows is performed vy
      acessing another adress than intended in the S BOX
    mov ZL, ST11
    lpm ST11, Z
    mov ZL, ST12
    lpm ST12, Z
    mov ZL, ST13
    lpm ST13, Z
    mov ZL, ST14
    lpm ST14, Z
    mov H1, ST21
```

```
    mov ZL, ST22
    lpm ST21, Z
    mov ZL, ST23
    lpm ST22, Z
    mov ZL, ST24
    lpm ST23, Z
    mov ZL, H1
    lpm ST24, Z
    mov H1, ST31
    mov ZL, ST33
    lpm ST31, Z
    mov ZL, H1
    lpm ST33, Z
    mov H1, ST32
    mov ZL, ST34
    lpm ST32, Z
    mov ZL, H1
    lpm ST34, Z
    mov H1, ST44
    mov ZL, ST43
    lpm ST44, Z
    mov ZL, ST42
    lpm ST43, Z
    mov ZL, ST41
    lpm ST42, Z
    mov ZL, H1
    lpm ST41, Z

    dec I ;; if I ==0 breq will branch
    breq end
    rcall mixcolumns
    rcall secAddRoundKey ; executed at each iteration including first pass.

    rcall EDN; must also protect mixcolumns
    rjmp encryp1

end:
    rcall secAddRoundKey ; executed at each iteration including first pass.
    rcall EDN
    ret

    ;;; ************************************************************************
;;;
;;; SECADDROUNDKEY
;;;
;;; Modified version that uses the ROM expanded key and only loads the useful part ↵
  of it.
;;; Once the key is added to the state register, any modification targeting it     ↵
  would be detected by the predictor.
;;; the only weak point is the temporary register holding the key byte before      ↵
  XORing it with the state
;;; Parameters:
;;;         ST11-ST44:  128 bit state matrix
;;;         YH:YL:      pointer to ram location
;;; Touched registers:
;;;     ST11-ST41,H1,YH,YL
```

```
secAddRoundKey:

    ldi ZH, high(hardKey<<1)
    lds ZL, keyoffset
    lpm H1, Z+
    eor ST11, H1
    lpm H1, Z+
    eor ST21, H1
    lpm H1, Z+
    eor ST31, H1
    lpm H1, Z+
    eor ST41, H1
    lpm H1, Z+
    eor ST12, H1
    lpm H1, Z+
    eor ST22, H1
    lpm H1, Z+
    eor ST32, H1
    lpm H1, Z+
    eor ST42, H1
    lpm H1, Z+
    eor ST13, H1
    lpm H1, Z+
    eor ST23, H1
    lpm H1, Z+
    eor ST33, H1
    lpm H1, Z+
    eor ST43, H1
    lpm H1, Z+
    eor ST14, H1
    lpm H1, Z+
    eor ST24, H1
    lpm H1, Z+
    eor ST34, H1
    lpm H1, Z+
    eor ST44, H1
    sts keyoffset, ZL ; keyoffset/ZL quite vulnerable but hard to exploit
    ret

;;; ***************************************************************************
;;;
;;; MIXCOLUMNS
;;; This routine applies the MixColumns diffusion operator to the whole
;;; state matrix. The code is used for both encryption and decryption.
;;;
;;; Note: This routine is part of the encryption and decryption routines. You
;;; normally wont be interested in calling this routine directly.
;;;
;;; Parameters:
;;;         ST11-ST44:  128 bit state matrix
;;; Touched registers:
;;;     ST11-ST41,H1,H2,H3,ZH,ZL

mixcolumns:
    ldi ZH, high(xtime<<1)
```

```
    mov H1, ST11
    eor H1, ST21
    eor H1, ST31
    eor H1, ST41
    mov H2, ST11
    mov H3, ST11
    eor H3, ST21
    mov ZL, H3
    lpm H3, Z
    eor ST11, H3
    eor ST11, H1
    mov H3, ST21
    eor H3, ST31
    mov ZL, H3
    lpm H3, Z
    eor ST21, H3
    eor ST21, H1
    mov H3, ST31
    eor H3, ST41
    mov ZL, H3
    lpm H3, Z
    eor ST31, H3
    eor ST31, H1
    mov H3, ST41
    eor H3, H2
    mov ZL, H3
    lpm H3, Z
    eor ST41, H3
    eor ST41, H1

    mov H1, ST12
    eor H1, ST22
    eor H1, ST32
    eor H1, ST42
    mov H2, ST12
    mov H3, ST12
    eor H3, ST22
    mov ZL, H3
    lpm H3, Z
    eor ST12, H3
    eor ST12, H1
    mov H3, ST22
    eor H3, ST32
    mov ZL, H3
    lpm H3, Z
    eor ST22, H3
    eor ST22, H1
    mov H3, ST32
    eor H3, ST42
    mov ZL, H3
    lpm H3, Z
    eor ST32, H3
    eor ST32, H1
    mov H3, ST42
    eor H3, H2
    mov ZL, H3
```

```
    lpm H3, Z
    eor ST42, H3
    eor ST42, H1

    mov H1, ST13
    eor H1, ST23
    eor H1, ST33
    eor H1, ST43
    mov H2, ST13
    mov H3, ST13
    eor H3, ST23
    mov ZL, H3
    lpm H3, Z
    eor ST13, H3
    eor ST13, H1
    mov H3, ST23
    eor H3, ST33
    mov ZL, H3
    lpm H3, Z
    eor ST23, H3
    eor ST23, H1
    mov H3, ST33
    eor H3, ST43
    mov ZL, H3
    lpm H3, Z
    eor ST33, H3
    eor ST33, H1
    mov H3, ST43
    eor H3, H2
    mov ZL, H3
    lpm H3, Z
    eor ST43, H3
    eor ST43, H1

    mov H1, ST14
    eor H1, ST24
    eor H1, ST34
    eor H1, ST44
    mov H2, ST14
    mov H3, ST14
    eor H3, ST24
    mov ZL, H3
    lpm H3, Z
    eor ST14, H3
    eor ST14, H1
    mov H3, ST24
    eor H3, ST34
    mov ZL, H3
    lpm H3, Z
    eor ST24, H3
    eor ST24, H1
    mov H3, ST34
    eor H3, ST44
    mov ZL, H3
    lpm H3, Z
    eor ST34, H3
```

```
    eor ST34, H1
    mov H3, ST44
    eor H3, H2
    mov ZL, H3
    lpm H3, Z
    eor ST44, H3
    eor ST44, H1
    ret



;;; ***************************************************************************
;;; Linear predictor
;;; Preconditions : registers ST11-ST44 are filled with state bytes
;;; Postcondition : registers LP1-LP4 assigned with linear prediction (LP1 MMSB LP4
   LLSB)
predict:


ldi ZH, high(sbox<<1)
mov ZL, ST11 ;; to substitute a value with the s box, simply adress the s box with
  the value to be substitued
lpm temp1, Z
mov LP1, temp1 ; first time we just move the s box value to the predictor.Sbox
  works fine
mov ZL, ST22
lpm temp1, Z
eor LP1, temp1
mov ZL, ST33
lpm temp1, Z
eor LP1, temp1
mov ZL, ST44
lpm temp1, Z
eor LP1, temp1

mov ZL, ST12
lpm temp1, Z
mov LP2, temp1
mov ZL, ST23
lpm temp1, Z
eor LP2, temp1
mov ZL, ST34
lpm temp1, Z
eor LP2, temp1
mov ZL, ST41
lpm temp1, Z
eor LP2, temp1

mov ZL, ST13
lpm temp1, Z
mov LP3, temp1
mov ZL, ST24
lpm temp1, Z
eor LP3, temp1
mov ZL, ST31
lpm temp1, Z
```

```
eor LP3, temp1
mov ZL, ST42
lpm temp1, Z
eor LP3, temp1

mov ZL, ST14
lpm temp1, Z
mov LP4, temp1
mov ZL, ST21
lpm temp1, Z
eor LP4, temp1
mov ZL, ST32
lpm temp1, Z
eor LP4, temp1
mov ZL, ST43
lpm temp1, Z
eor LP4, temp1


ldi ZH, high(hardKey<<1)
lds ZL, keyoffset
lpm H1, Z+
eor LP1, H1
lpm H1, Z+
eor LP1, H1
lpm H1, Z+
eor LP1, H1
lpm H1, Z+
eor LP1, H1
lpm H1, Z+
eor LP2, H1
lpm H1, Z+
eor LP2, H1
lpm H1, Z+
eor LP2, H1
lpm H1, Z+
eor LP2, H1
lpm H1, Z+
eor LP3, H1
lpm H1, Z+
eor LP3, H1
lpm H1, Z+
eor LP3, H1
lpm H1, Z+
eor LP3, H1
lpm H1, Z+
eor LP4, H1
lpm H1, Z+
eor LP4, H1
lpm H1, Z+
eor LP4, H1
lpm H1, Z+
eor LP4, H1


ret
```

...Memoire\simu\ProtectedAES\ProtectedAES\ProtectedAES.asm 10

```
;;; ***************************************************************************
;;; Linear compressor
;;; Preconditions : registers ST11-ST44 are filled with state bytes
;;; Postcondition : registers LP1-LP4 assigned with compression (LP1 MMSB LP4 LLSB)
compress:
mov LP1, ST11
mov LP2, ST12
mov LP3, ST13
mov LP4, ST14

eor LP1, ST21
eor LP1, ST31
eor LP1, ST41

eor LP2, ST22
eor LP2, ST32
eor LP2, ST42

eor LP3, ST23
eor LP3, ST33
eor LP3, ST43

eor LP4, ST24
eor LP4, ST34
eor LP4, ST44
ret
;;;***************************************************************************
;; modular multiplication
;; Pre:  First 32 bit signature  contained in [LP1 - LP4], second signature ;;contained in (LLSB) temp1, temp2, Yl,YH (MMSB)
;; Post : cubic signature contained in (LLSB) XL XH ZL(r30) ZH(R31)(MMSB)
modmult:

sts multregs, ST11 ; Backup r00 and r01
sts multregs+1, ST21
mul LP4, temp1 ; multiply A LLSB with B LLSB
movw    XL, r0 ; store first result in XL:XH(result LLSB:LSB)
mul LP3, temp2 ; multiply A LSB with B LSB
movw    r30, r0 ; store second result in r30:r31 (result MSB:MMSB)
mul LP2, temp1 ; multiply A MSB with B LLSB
add r30, r0 ; add without carry mult LSB to result MSB
adc r31, r1 ; add mult MSB with previous carry to result MMSB
mul LP4, YL ; multiply B MSB with A LLSB
add r30, r0 ; add mult LSB to result MSB
adc r31, r1 ; add mult MSB to result MMSB
mul LP1, temp1 ; multiply B LLSB with A MMSB
add r31, r0  ; add mult LSB to result MMSB  (result is mod 32 so dont care about overflows, adding modulo 32)
mul LP2, temp2 ; multiply  A MSB with B LSB
add r31, r0 ; add mult LSB to result MMSB
mul LP3, YL ; multiply B MSB with A LSB
add r31, r0 ; add mult LSB to result MMSB
mul LP4, YH  ; multiply A LLSB with B MMSB
add r31, r0 ; add mult LSB to result MMSB
eor YH,YH ; create 0 value to use with adc . We can put B's MMSB because its not
```

```
  used anymore
mul LP3, temp1 ; multiply a LSB with B LLSB
add XH, r0 ; add mult LSB to result LSB
adc r30, r1 ; add mult MSB to result MSB
adc r31, YH ; add  carry  to result MMSB
mul LP4, temp2 ; multiply B LSB with a LLSB
add XH, r0 ; add mult LSB to result LSB
adc r30, r1 ; add mult MSB to result MSB
adc r31, YH; add  carry  to result MMSB
eor r1, r1 ; MUST clear r1
lds ST21, multregs+1    ; restore states
lds ST11, multregs
ret

;;;*****************************************************************************
;;; GF cubic operation on 32 bits (from 4 registers)

GFcube:
sts ptrstore, YL
sts ptrstore+1, YH
mov temp1, LP4; load B (LP) in temp regs
mov temp2, LP3
movw YL, LP2
rcall modmult ; perform mult between LP and LP
mov temp1, XL
mov temp2, XH
movw YL, ZL
rcall modmult ; perform mult between LP and (LP*LP)

lds YH, ptrstore+1  ; restore key pointer
lds YL, ptrstore

ret

;;;*****************************************************************************
;;;
;;; Error detection Network
;;; compute new signature based on new state and compare with stored signature.
;;; in case of error, output random bytes/share (here : clear state)
edn:
rcall compress
rcall GFcube
lds temp1, signature
lds temp2, signature+1
cp temp1, XL
brne infect
cp temp2, XH
brne infect
lds temp1, signature+2
lds temp2, signature+3
cp temp1, r30
brne infect
cp temp2, r31
brne infect
ret
```

```
infect:
clr ST11
clr ST12
clr ST13
clr ST14
clr ST21
clr ST22
clr ST23
clr ST31
clr ST32
clr ST33
clr ST34
clr ST41
clr ST42
clr ST43
clr ST44
ret

savestate:

ldi ZH, high(statestore)
ldi ZL, low(statestore)

st Z+, ST11
st Z+, ST12
st Z+, ST13
st Z+, ST14
st Z+, ST21
st Z+, ST22
st Z+, ST23
st Z+, ST24
st Z+, ST31
st Z+, ST32
st Z+, ST33
st Z+, ST34
st Z+, ST41
st Z+, ST42
st Z+, ST43
st Z,  ST44
ret

loadstate:

ldi ZH, high(statestore)
ldi ZL, low(statestore)

ld ST11, Z+
ld ST12, Z+
ld ST13, Z+
ld ST14, Z+
ld ST21, Z+
ld ST22, Z+
ld ST23, Z+
ld ST24, Z+
ld ST31, Z+
ld ST32, Z+
```

```
ld ST33, Z+
ld ST34, Z+
ld ST41, Z+
ld ST42, Z+
ld ST43, Z+
ld ST44, Z

ret



;;; ***************************************************************************
;;;
;;; S-BOX and "xtime" tables
;;; Rijndael consists of a non-linear step in its rounds (called "sbox step"),
;;; here implemented with two hard-coded lookup tables (the sbox itself and its
;;; inverse for decryption). To provide an implementation secure against power
;;; analysis attacks, the polynomial multiplication in the MixColumns operation
;;; is done via an auxiliary lookup table called xtime. See [1] for details.
;;;
;;; The three tables have to be aligned to a flash position with its lower
;;; address byte equal to $00. In assembler syntax: low(sbox<<1) == 0.
;;; To ensure the proper alignment of the sboxes, the assembler directive
;;; .ORG is used (below the sboxes are defined to begin at $800). Note, that
;;; any other address can be used as well, as long as the lower byte is equal
;;; to $00.
;;;
;;; The order of the sboxes is totally arbitrary. They even do not have to be
;;; allocated in adjacent memory areas.

.CSEG
.ORG $800

sbox:
.db $63,$7c,$77,$7b,$f2,$6b,$6f,$c5,$30,$01,$67,$2b,$fe,$d7,$ab,$76
.db $ca,$82,$c9,$7d,$fa,$59,$47,$f0,$ad,$d4,$a2,$af,$9c,$a4,$72,$c0
.db $b7,$fd,$93,$26,$36,$3f,$f7,$cc,$34,$a5,$e5,$f1,$71,$d8,$31,$15
.db $04,$c7,$23,$c3,$18,$96,$05,$9a,$07,$12,$80,$e2,$eb,$27,$b2,$75
.db $09,$83,$2c,$1a,$1b,$6e,$5a,$a0,$52,$3b,$d6,$b3,$29,$e3,$2f,$84
.db $53,$d1,$00,$ed,$20,$fc,$b1,$5b,$6a,$cb,$be,$39,$4a,$4c,$58,$cf
.db $d0,$ef,$aa,$fb,$43,$4d,$33,$85,$45,$f9,$02,$7f,$50,$3c,$9f,$a8
.db $51,$a3,$40,$8f,$92,$9d,$38,$f5,$bc,$b6,$da,$21,$10,$ff,$f3,$d2
.db $cd,$0c,$13,$ec,$5f,$97,$44,$17,$c4,$a7,$7e,$3d,$64,$5d,$19,$73
.db $60,$81,$4f,$dc,$22,$2a,$90,$88,$46,$ee,$b8,$14,$de,$5e,$0b,$db
.db $e0,$32,$3a,$0a,$49,$06,$24,$5c,$c2,$d3,$ac,$62,$91,$95,$e4,$79
.db $e7,$c8,$37,$6d,$8d,$d5,$4e,$a9,$6c,$56,$f4,$ea,$65,$7a,$ae,$08
.db $ba,$78,$25,$2e,$1c,$a6,$b4,$c6,$e8,$dd,$74,$1f,$4b,$bd,$8b,$8a
.db $70,$3e,$b5,$66,$48,$03,$f6,$0e,$61,$35,$57,$b9,$86,$c1,$1d,$9e
.db $e1,$f8,$98,$11,$69,$d9,$8e,$94,$9b,$1e,$87,$e9,$ce,$55,$28,$df
.db $8c,$a1,$89,$0d,$bf,$e6,$42,$68,$41,$99,$2d,$0f,$b0,$54,$bb,$16

xtime:
.db $00,$02,$04,$06,$08,$0a,$0c,$0e,$10,$12,$14,$16,$18,$1a,$1c,$1e
.db $20,$22,$24,$26,$28,$2a,$2c,$2e,$30,$32,$34,$36,$38,$3a,$3c,$3e
.db $40,$42,$44,$46,$48,$4a,$4c,$4e,$50,$52,$54,$56,$58,$5a,$5c,$5e
```

```
.db $60,$62,$64,$66,$68,$6a,$6c,$6e,$70,$72,$74,$76,$78,$7a,$7c,$7e
.db $80,$82,$84,$86,$88,$8a,$8c,$8e,$90,$92,$94,$96,$98,$9a,$9c,$9e
.db $a0,$a2,$a4,$a6,$a8,$aa,$ac,$ae,$b0,$b2,$b4,$b6,$b8,$ba,$bc,$be
.db $c0,$c2,$c4,$c6,$c8,$ca,$cc,$ce,$d0,$d2,$d4,$d6,$d8,$da,$dc,$de
.db $e0,$e2,$e4,$e6,$e8,$ea,$ec,$ee,$f0,$f2,$f4,$f6,$f8,$fa,$fc,$fe
.db $1b,$19,$1f,$1d,$13,$11,$17,$15,$0b,$09,$0f,$0d,$03,$01,$07,$05
.db $3b,$39,$3f,$3d,$33,$31,$37,$35,$2b,$29,$2f,$2d,$23,$21,$27,$25
.db $5b,$59,$5f,$5d,$53,$51,$57,$55,$4b,$49,$4f,$4d,$43,$41,$47,$45
.db $7b,$79,$7f,$7d,$73,$71,$77,$75,$6b,$69,$6f,$6d,$63,$61,$67,$65
.db $9b,$99,$9f,$9d,$93,$91,$97,$95,$8b,$89,$8f,$8d,$83,$81,$87,$85
.db $bb,$b9,$bf,$bd,$b3,$b1,$b7,$b5,$ab,$a9,$af,$ad,$a3,$a1,$a7,$a5
.db $db,$d9,$df,$dd,$d3,$d1,$d7,$d5,$cb,$c9,$cf,$cd,$c3,$c1,$c7,$c5
.db $fb,$f9,$ff,$fd,$f3,$f1,$f7,$f5,$eb,$e9,$ef,$ed,$e3,$e1,$e7,$e5

;Countermeasure against key schedule attacks
hardkey:

.db $2b,$7e,$15,$16,$28,$ae,$d2,$a6,$ab,$f7,$15,$88,$09,$cf,$4f,$3c
.db $a0,$fa,$fe,$17,$88,$54,$2c,$b1,$23,$a3,$39,$39,$2a,$6c,$76,$05
.db $f2,$c2,$95,$f2,$7a,$96,$b9,$43,$59,$35,$80,$7a,$73,$59,$f6,$7f
.db $3d,$80,$47,$7d,$47,$16,$fe,$3e,$1e,$23,$7e,$44,$6d,$7a,$88,$3b
.db $ef,$44,$a5,$41,$a8,$52,$5b,$7f,$b6,$71,$25,$3b,$db,$0b,$ad,$00
.db $d4,$d1,$c6,$f8,$7c,$83,$9d,$87,$ca,$f2,$b8,$bc,$11,$f9,$15,$bc
.db $6d,$88,$a3,$7a,$11,$0b,$3e,$fd,$db,$f9,$86,$41,$ca,$00,$93,$fd
.db $4e,$54,$f7,$0e,$5f,$5f,$c9,$f3,$84,$a6,$4f,$b2,$4e,$a6,$dc,$4f
.db $ea,$d2,$73,$21,$b5,$8d,$ba,$d2,$31,$2b,$f5,$60,$7f,$8d,$29,$2f
.db $ac,$77,$66,$f3,$19,$fa,$dc,$21,$28,$d1,$29,$41,$57,$5c,$00,$6e
.db $d0,$14,$f9,$a8,$c9,$ee,$25,$89,$e1,$3f,$0c,$c8,$b6,$63,$0c,$a6
```

Appendix B

Code : C DPA and FA-protected AES

D:\Documents\ELEC22MS\Memoire\simu\C_AES_DPA\C_AES_DPA\aes.c 1

```
/*

Author : Georges-Henri Leclercq,
based on Tiny AES : https://github.com/kokke/tiny-AES128-C

The implementation is verified against the test vectors in:
  National Institute of Standards and Technology Special Publication 800-38A 2001 ↪
    ED

ECB-AES128
----------

  plain-text:
    6bc1bee22e409f96e93d7e117393172a
    ae2d8a571e03ac9c9eb76fac45af8e51
    30c81c46a35ce411e5fbc1191a0a52ef
    f69f2445df4f9b17ad2b417be66c3710

  key:
    2b7e151628aed2a6abf7158809cf4f3c

  resulting cipher
    3ad77bb40d7a3660a89ecaf32466ef97
    f5d3d58503b9699de785895a96fdbaaf
    43b1cd7f598ece23881b00e3ed030688
    7b0c785e27e8ad3f8223207104725dd4

*/


/*****************************************************************************/
/* Includes:                                                                 */
/*****************************************************************************/
#include <stdint.h>
#include "aes.h"

/*****************************************************************************/
/* Defines:                                                                  */
/*****************************************************************************/
// The number of columns comprising a state in AES. This is a constant in AES. ↪
  Value=4
#define Nb 4
// The number of 32 bit words in a key.
#define Nk 4
// Key length in bytes [128 bit]
#define KEYLEN 16
// The number of rounds in AES Cipher.
#define Nr 10
// The number of shares used for DPA protecion
#define Ns 2

// jcallan@github points out that declaring Multiply as a function
// reduces code size considerably with the Keil ARM compiler.
// See this link for more information: https://github.com/kokke/tiny-AES128-C/ ↪
  pull/3
#ifndef MULTIPLY_AS_A_FUNCTION
```

```
  #define MULTIPLY_AS_A_FUNCTION 1
#endif


/*****************************************************************************/
/* Private variables:                                                        */
/*****************************************************************************/
// state - array holding the intermediate results during encryption.
typedef uint8_t state_t[4][4]; // State type
/* Important note about the state : the C representation is
S= {{(0,0),(0,1),(0,2),(0,3)};
    {(1,0),(1,1),(1,2),(1,3)};
    {(2,0),(2,1),(2,2),(2,3)};
    {(3,0),(3,1),(3,2),(3,3)}}
And So is the paper representation. HOWEVER:
This code fills the state and the key line by line,
whereas the papers fill it column by column;
This code then considers the State as being a transposed matrix
Meaning that a column is a line and vice versa.
Therefore, mixColumns And ShiftRows will operate on C columns and C rows
  respectively.
Crazy, isn't it?

The best part is, the key originally does not follow the same rule, as it is
  represented inline.
The corresponding key byte to a state is given by state[i,j] <=> key[4*i + j]

I modified my proposed solutions for DFA in the same fashion to ensure full
  compatibility
Meaning I inverted every row and column access.

*/
typedef state_t nstate_t[Ns];
typedef uint8_t LP_t[4];  // Linear predictor array type


static state_t nstate[Ns];
static state_t nstate_copy[Ns];
// The Linear predictor values
static uint8_t LPIndex[16]={0,3,2,1,1,0,3,2,2,1,0,3,3,2,1,0};
static uint8_t LPIN[4], LPOUT[4];
static uint32_t signatureIN, signatureOUT;
static uint8_t PredictionMask[Ns][4];
// Random pool initially filled with the seed then filled with PRNG
static uint8_t RandPool[16];
// RandCount is used in Functions RandomMasks and SecExp254 to determine whhen to
  regenerate randomNess
static uint8_t RandCount=0;
// Used for secure multiplications; easier and more efficient? to deal with
  preallocated stuff. Otherwise, belong to SecExp254
static state_t z[Ns];

#if defined(KEYEXPAND) && KEYEXPAND
    // The array that stores the round keys.
    static uint8_t RoundKey[176];
    static const uint8_t* Key;
```

```
D:\Documents\ELEC22MS\Memoire\simu\C_AES_DPA\C_AES_DPA\aes.c                3
#else
// Securing key schedule operations by hardcoding the roundkey
    static const uint8_t nRoundKey[Ns][176] = {{
      //0     1    2     3     4    5     6     7     8    9     A     B    C
          D     E     F
      0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09,
         0xcf, 0x4f, 0x3c,
      0xa0, 0xfa, 0xfe, 0x17, 0x88, 0x54, 0x2c, 0xb1, 0x23, 0xa3, 0x39, 0x39, 0x2a,
         0x6c, 0x76, 0x05,
      0xf2, 0xc2, 0x95, 0xf2, 0x7a, 0x96, 0xb9, 0x43, 0x59, 0x35, 0x80, 0x7a, 0x73,
         0x59, 0xf6, 0x7f,
      0x3d, 0x80, 0x47, 0x7d, 0x47, 0x16, 0xfe, 0x3e, 0x1e, 0x23, 0x7e, 0x44, 0x6d,
         0x7a, 0x88, 0x3b,
      0xef, 0x44, 0xa5, 0x41, 0xa8, 0x52, 0x5b, 0x7f, 0xb6, 0x71, 0x25, 0x3b, 0xdb,
         0x0b, 0xad, 0x00,
      0xd4, 0xd1, 0xc6, 0xf8, 0x7c, 0x83, 0x9d, 0x87, 0xca, 0xf2, 0xb8, 0xbc, 0x11,
         0xf9, 0x15, 0xbc,
      0x6d, 0x88, 0xa3, 0x7a, 0x11, 0x0b, 0x3e, 0xfd, 0xdb, 0xf9, 0x86, 0x41, 0xca,
         0x00, 0x93, 0xfd,
      0x4e, 0x54, 0xf7, 0x0e, 0x5f, 0x5f, 0xc9, 0xf3, 0x84, 0xa6, 0x4f, 0xb2, 0x4e,
         0xa6, 0xdc, 0x4f,
      0xea, 0xd2, 0x73, 0x21, 0xb5, 0x8d, 0xba, 0xd2, 0x31, 0x2b, 0xf5, 0x60, 0x7f,
         0x8d, 0x29, 0x2f,
      0xac, 0x77, 0x66, 0xf3, 0x19, 0xfa, 0xdc, 0x21, 0x28, 0xd1, 0x29, 0x41, 0x57,
         0x5c, 0x00, 0x6e,
      0xd0, 0x14, 0xf9, 0xa8, 0xc9, 0xee, 0x25, 0x89, 0xe1, 0x3f, 0x0c, 0xc8, 0xb6,
         0x63, 0x0c, 0xa6 } ,

       {
      //0     1    2     3     4    5     6     7     8    9     A     B    C
          D     E     F
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00 }


      };
```

D:\Documents\ELEC22MS\Memoire\simu\C_AES_DPA\C_AES_DPA\aes.c 4

```
#endif
// The Key input to the AES Program


// The lookup-tables are marked const so they can be placed in read-only storage
  instead of RAM
// The numbers below can be computed dynamically trading ROM for RAM -
// This can be useful in (embedded) bootloader applications, where ROM is often
  limited.
static const uint8_t sbox[256] =   {
  //0     1    2      3     4    5     6     7      8    9     A      B    C     D
        E     F
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe,
    0xd7, 0xab, 0x76,
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c,
    0xa4, 0x72, 0xc0,
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71,
    0xd8, 0x31, 0x15,
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb,
    0x27, 0xb2, 0x75,
  0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29,
    0xe3, 0x2f, 0x84,
  0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a,
    0x4c, 0x58, 0xcf,
  0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50,
    0x3c, 0x9f, 0xa8,
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10,
    0xff, 0xf3, 0xd2,
  0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64,
    0x5d, 0x19, 0x73,
  0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde,
    0x5e, 0x0b, 0xdb,
  0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91,
    0x95, 0xe4, 0x79,
  0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65,
    0x7a, 0xae, 0x08,
  0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b,
    0xbd, 0x8b, 0x8a,
  0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86,
    0xc1, 0x1d, 0x9e,
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce,
    0x55, 0x28, 0xdf,
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0,
    0x54, 0xbb, 0x16 };

#if defined(KEYEXPAND) && KEYEXPAND
// The round constant word array, Rcon[i], contains the values given by
// x to th e power (i-1) being powers of x (x is denoted as {02}) in the field GF
  (2^8)
// Note that i starts at 1, not 0).
static const uint8_t Rcon[255] = {
  0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,
    0xab, 0x4d, 0x9a,
  0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,
    0xc5, 0x91, 0x39,
```

D:\Documents\ELEC22MS\Memoire\simu\C_AES_DPA\C_AES_DPA\aes.c 5

```
  0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,
    0x83, 0x1d, 0x3a,
  0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b,
    0x36, 0x6c, 0xd8,
  0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
    0x7d, 0xfa, 0xef,
  0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
    0x33, 0x66, 0xcc,
  0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20,
    0x40, 0x80, 0x1b,
  0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,
    0x6a, 0xd4, 0xb3,
  0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,
    0x25, 0x4a, 0x94,
  0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
    0x08, 0x10, 0x20,
  0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63,
    0xc6, 0x97, 0x35,
  0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
    0x61, 0xc2, 0x9f,
  0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d,
    0x01, 0x02, 0x04,
  0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f,
    0x5e, 0xbc, 0x63,
  0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72,
    0xe4, 0xd3, 0xbd,
  0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74,
    0xe8, 0xcb  };
 #endif

 /// Square and 4th power LUT

static const uint8_t squareLUT[256] = {
 0x00, 0x01, 0x04, 0x05, 0x10, 0x11, 0x14, 0x15, 0x40, 0x41, 0x44, 0x45, 0x50,
   0x51, 0x54, 0x55,
 0x1B, 0x1A, 0x1F, 0x1E, 0x0B, 0x0A, 0x0F, 0x0E, 0x5B, 0x5A, 0x5F, 0x5E, 0x4B,
   0x4A, 0x4F, 0x4E,
 0x6C, 0x6D, 0x68, 0x69, 0x7C, 0x7D, 0x78, 0x79, 0x2C, 0x2D, 0x28, 0x29, 0x3C,
   0x3D, 0x38, 0x39,
 0x77, 0x76, 0x73, 0x72, 0x67, 0x66, 0x63, 0x62, 0x37, 0x36, 0x33, 0x32, 0x27,
   0x26, 0x23, 0x22,
 0xAB, 0xAA, 0xAF, 0xAE, 0xBB, 0xBA, 0xBF, 0xBE, 0xEB, 0xEA, 0xEF, 0xEE, 0xFB,
   0xFA, 0xFF, 0xFE,
 0xB0, 0xB1, 0xB4, 0xB5, 0xA0, 0xA1, 0xA4, 0xA5, 0xF0, 0xF1, 0xF4, 0xF5, 0xE0,
   0xE1, 0xE4, 0xE5,
 0xC7, 0xC6, 0xC3, 0xC2, 0xD7, 0xD6, 0xD3, 0xD2, 0x87, 0x86, 0x83, 0x82, 0x97,
   0x96, 0x93, 0x92,
 0xDC, 0xDD, 0xD8, 0xD9, 0xCC, 0xCD, 0xC8, 0xC9, 0x9C, 0x9D, 0x98, 0x99, 0x8C,
   0x8D, 0x88, 0x89,
 0x9A, 0x9B, 0x9E, 0x9F, 0x8A, 0x8B, 0x8E, 0x8F, 0xDA, 0xDB, 0xDE, 0xDF, 0xCA,
   0xCB, 0xCE, 0xCF,
 0x81, 0x80, 0x85, 0x84, 0x91, 0x90, 0x95, 0x94, 0xC1, 0xC0, 0xC5, 0xC4, 0xD1,
   0xD0, 0xD5, 0xD4,
 0xF6, 0xF7, 0xF2, 0xF3, 0xE6, 0xE7, 0xE2, 0xE3, 0xB6, 0xB7, 0xB2, 0xB3, 0xA6,
   0xA7, 0xA2, 0xA3,
 0xED, 0xEC, 0xE9, 0xE8, 0xFD, 0xFC, 0xF9, 0xF8, 0xAD, 0xAC, 0xA9, 0xA8, 0xBD,
```

```
    0xBC, 0xB9, 0xB8,
  0x31, 0x30, 0x35, 0x34, 0x21, 0x20, 0x25, 0x24, 0x71, 0x70, 0x75, 0x74, 0x61,
    0x60, 0x65, 0x64,
  0x2A, 0x2B, 0x2E, 0x2F, 0x3A, 0x3B, 0x3E, 0x3F, 0x6A, 0x6B, 0x6E, 0x6F, 0x7A,
    0x7B, 0x7E, 0x7F,
  0x5D, 0x5C, 0x59, 0x58, 0x4D, 0x4C, 0x49, 0x48, 0x1D, 0x1C, 0x19, 0x18, 0x0D,
    0x0C, 0x09, 0x08,
  0x46, 0x47, 0x42, 0x43, 0x56, 0x57, 0x52, 0x53, 0x06, 0x07, 0x02, 0x03, 0x16,
    0x17, 0x12, 0x13 };

  static const uint8_t power5LUT[256] = {
  0x00, 0x01, 0x20, 0x33, 0x6C, 0x72, 0x3A, 0x36, 0x2F, 0x8D, 0xC2, 0x72, 0x01,
    0xBC, 0x9A, 0x35,
  0x97, 0xD8, 0x10, 0x4D, 0x33, 0x63, 0xC2, 0x80, 0x20, 0xCC, 0x6A, 0x94, 0xC6,
    0x35, 0xFA, 0x1B,
  0x7D, 0xCB, 0x5E, 0xFA, 0x36, 0x9F, 0x63, 0xD8, 0x3A, 0x2F, 0xD4, 0xD3, 0x33,
    0x39, 0xAB, 0xB3,
  0x6C, 0x94, 0xE8, 0x02, 0xEF, 0x08, 0x1D, 0xE8, 0xB3, 0xE8, 0xFA, 0xB3, 0x72,
    0x36, 0x4D, 0x1B,
  0x39, 0xCB, 0x08, 0xE8, 0x35, 0xD8, 0x72, 0x8D, 0x9A, 0xCB, 0x66, 0x25, 0xD4,
    0x9A, 0x5E, 0x02,
  0x01, 0xBD, 0x97, 0x39, 0xC5, 0x66, 0x25, 0x94, 0x3A, 0x25, 0x61, 0x6C, 0xBC,
    0xBC, 0x91, 0x83,
  0x2F, 0x6A, 0x1D, 0x4A, 0x04, 0x5E, 0x40, 0x08, 0xE4, 0x02, 0x1B, 0xEF, 0x8D,
    0x74, 0x04, 0xEF,
  0x91, 0x9A, 0x04, 0x1D, 0x72, 0x66, 0x91, 0x97, 0xC2, 0x6A, 0x9A, 0x20, 0x63,
    0xD4, 0x4D, 0xE8,
  0x61, 0x25, 0x08, 0x5E, 0x1B, 0x40, 0x04, 0x4D, 0xFA, 0x1D, 0x5E, 0xAB, 0xC2,
    0x3A, 0x10, 0xFA,
  0xC6, 0xCC, 0x08, 0x10, 0x74, 0x61, 0xCC, 0xCB, 0xC5, 0x6C, 0xC6, 0x7D, 0x35,
    0x83, 0x40, 0xE4,
  0x20, 0xD3, 0x4A, 0xAB, 0x7D, 0x91, 0x61, 0x9F, 0xD3, 0x83, 0x74, 0x36, 0xCC,
    0x83, 0x1D, 0x40,
  0x01, 0xBC, 0xCC, 0x63, 0x94, 0x36, 0x2F, 0x9F, 0x6A, 0x74, 0x6A, 0x66, 0xBD,
    0xBC, 0xCB, 0xD8,
  0x97, 0x20, 0xEF, 0x4A, 0x8D, 0x25, 0x83, 0x39, 0x80, 0x94, 0x35, 0x33, 0xD8,
    0xD3, 0x1B, 0x02,
  0x9F, 0x66, 0x40, 0xAB, 0x4D, 0xAB, 0xE4, 0x10, 0x10, 0x4A, 0x02, 0x4A, 0x80,
    0xC5, 0xE4, 0xB3,
  0xBD, 0xBD, 0xC6, 0xD4, 0x80, 0x9F, 0x8D, 0x80, 0xC2, 0x61, 0x74, 0xC5, 0xBD,
    0x01, 0x7D, 0xD3,
  0x33, 0x7D, 0xEF, 0xB3, 0xC6, 0x97, 0x6C, 0x2F, 0xD4, 0x39, 0xC5, 0x3A, 0x63,
    0x91, 0x04, 0xE4 };

/*****************************************************************************/
/* Private functions:                                                        */
/*****************************************************************************/

static void BlockCopy(uint8_t* output, uint8_t* input)
{
  uint8_t i;
  for (i=0;i<KEYLEN;++i)
  {
        output[i] = input[i];
  }
}
```

D:\Documents\ELEC22MS\Memoire\simu\C_AES_DPA\C_AES_DPA\aes.c 7

```
static void BlockXOR(uint8_t* output, uint8_t* input) // Modified to match papers notations when matrices are used. Works.
{
  uint8_t i;
  for (i=0;i<KEYLEN;++i)
  {
        output[i] ^= input[i];
  }
}


static uint8_t getSBoxValue(uint8_t num)
{
  return sbox[num];
}

// This function produces Nb(Nr+1) round keys. The round keys are used in each round to decrypt the states.
#if defined(KEYEXPAND) && KEYEXPAND
static void KeyExpansion(void)
{
  uint32_t i, j, k;
  uint8_t tempa[4]; // Used for the column/row operations

  // The first round key is the key itself.
  for(i = 0; i < Nk; ++i)
  {
    RoundKey[(i * 4) + 0] = Key[(i * 4) + 0];
    RoundKey[(i * 4) + 1] = Key[(i * 4) + 1];
    RoundKey[(i * 4) + 2] = Key[(i * 4) + 2];
    RoundKey[(i * 4) + 3] = Key[(i * 4) + 3];
  }

  // All other round keys are found from the previous round keys.
  for(; (i < (Nb * (Nr + 1))); ++i)
  {
    for(j = 0; j < 4; ++j)
    {
      tempa[j]=RoundKey[(i-1) * 4 + j];
    }
    if (i % Nk == 0)
    {
      // This function rotates the 4 bytes in a word to the left once.
      // [a0,a1,a2,a3] becomes [a1,a2,a3,a0]

      // Function RotWord()
      {
        k = tempa[0];
        tempa[0] = tempa[1];
        tempa[1] = tempa[2];
        tempa[2] = tempa[3];
        tempa[3] = k;
      }

      // SubWord() is a function that takes a four-byte input word and
```

D:\Documents\ELEC22MS\Memoire\simu\C_AES_DPA\C_AES_DPA\aes.c 8

```
      // applies the S-box to each of the four bytes to produce an output word.

      // Function Subword()
      {
        tempa[0] = getSBoxValue(tempa[0]);
        tempa[1] = getSBoxValue(tempa[1]);
        tempa[2] = getSBoxValue(tempa[2]);
        tempa[3] = getSBoxValue(tempa[3]);
      }

      tempa[0] =  tempa[0] ^ Rcon[i/Nk];
    }
    else if (Nk > 6 && i % Nk == 4)
    {
      // Function Subword()
      {
        tempa[0] = getSBoxValue(tempa[0]);
        tempa[1] = getSBoxValue(tempa[1]);
        tempa[2] = getSBoxValue(tempa[2]);
        tempa[3] = getSBoxValue(tempa[3]);
      }
    }
    RoundKey[i * 4 + 0] = RoundKey[(i - Nk) * 4 + 0] ^ tempa[0];
    RoundKey[i * 4 + 1] = RoundKey[(i - Nk) * 4 + 1] ^ tempa[1];
    RoundKey[i * 4 + 2] = RoundKey[(i - Nk) * 4 + 2] ^ tempa[2];
    RoundKey[i * 4 + 3] = RoundKey[(i - Nk) * 4 + 3] ^ tempa[3];
  }
}

#endif
// This function adds the round key to state.
// The round key is added to the state by an XOR function.
static void AddRoundKey(state_t *state,uint8_t round, uint8_t shareNumber)
{
  uint8_t i,j;

  for(i=0;i<4;++i)
  {

    for(j = 0; j < 4; ++j)
    {
      (*state)[i][j] ^= nRoundKey[shareNumber][round * Nb * 4 + i * Nb + j]; //
        Inverted j and i to have the same convention than papers

    }
  }
}


// The SubBytes Function Substitutes the values in the
// state matrix with values in an S-box.
static void SubBytes(state_t *state)
{
  uint8_t i, j;
  for(i = 0; i < 4; ++i)
  {
```

D:\Documents\ELEC22MS\Memoire\simu\C_AES_DPA\C_AES_DPA\aes.c 9

```
    for(j = 0; j < 4; ++j)
    {
      (*state)[j][i] = getSBoxValue((*state)[j][i]);
    }
  }
}


// The ShiftRows() function shifts the rows in the state to the left.
// Each row is shifted with different offset.
// Offset = Row number. So the first row is not shifted.
static void ShiftRows(state_t *state)
{
  uint8_t temp;

  // Rotate second row 1 columns to left
  temp           = (*state)[0][1];
  (*state)[0][1] = (*state)[1][1];
  (*state)[1][1] = (*state)[2][1];
  (*state)[2][1] = (*state)[3][1];
  (*state)[3][1] = temp;

  // Rotate third row 2 columns to left
  temp           = (*state)[0][2];
  (*state)[0][2] = (*state)[2][2];
  (*state)[2][2] = temp;

  temp       = (*state)[1][2];
  (*state)[1][2] = (*state)[3][2];
  (*state)[3][2] = temp;

  // Rotate fourth row 3 columns to left
  temp       = (*state)[0][3];
  (*state)[0][3] = (*state)[3][3];
  (*state)[3][3] = (*state)[2][3];
  (*state)[2][3] = (*state)[1][3];
  (*state)[1][3] = temp;
}

static uint8_t xtime(uint8_t x)
{
  return ((x<<1) ^ (((x>>7) & 1) * 0x1b));
}

// MixColumns function mixes the columns of the state matrix
static void MixColumns(state_t *state)
{
  uint8_t i;
  uint8_t Tmp,Tm,t;
  for(i = 0; i < 4; ++i)
  {
    t   = (*state)[i][0];
    Tmp = (*state)[i][0] ^ (*state)[i][1] ^ (*state)[i][2] ^ (*state)[i][3] ;
    Tm  = (*state)[i][0] ^ (*state)[i][1] ; Tm = xtime(Tm);  (*state)[i][0] ^= Tm ^
       Tmp ;
```

D:\Documents\ELEC22MS\Memoire\simu\C_AES_DPA\C_AES_DPA\aes.c 10

```
    Tm  = (*state)[i][1] ^ (*state)[i][2] ; Tm = xtime(Tm);  (*state)[i][1] ^= Tm ^
       Tmp ;
    Tm  = (*state)[i][2] ^ (*state)[i][3] ; Tm = xtime(Tm);  (*state)[i][2] ^= Tm ^
       Tmp ;
    Tm  = (*state)[i][3] ^ t ;        Tm = xtime(Tm);  (*state)[i][3] ^= Tm ^ Tmp ;
  }
}

// Multiply is used to multiply numbers in the field GF(2^8)

#if MULTIPLY_AS_A_FUNCTION

static uint8_t Multiply(uint8_t a, uint8_t b) { // performance better than initial
  Multiply, and CORRECT
    uint8_t p = 0; // the product of the multiplication
    while (b) {
            if (b & 1) // if b is odd, then add the corresponding a to p (final
              product = sum of all a's corresponding to odd b's)
                p ^= a; // since we're in GF(2^m), addition is an XOR

            if (a & 0x80) // GF modulo: if a >= 128, then it will overflow when
              shifted left, so reduce
                a = (a << 1) ^ 0x1b; // XOR with the primitive polynomial x^8 + x^4
                   + x^3 + x + 1 -- you can change it but it must be irreducible
            else
                a <<= 1; // equivalent to a*2
            b >>= 1; // equivalent to b // 2
    }
    return p;
}

#else

#define Multiply(x, y)                                \
      (  ((y & 1) * x) ^                              \
      ((y>>1 & 1) * xtime(x)) ^                       \
      ((y>>2 & 1) * xtime(xtime(x))) ^                \
      ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^         \
      ((y>>4 & 1) * xtime(xtime(xtime(xtime(x))))))   \


#endif



// DFA protection related functions

static void GFCube(LP_t* LP, uint32_t *signature){
    uint32_t LP32= *((uint32_t *)&LP[0]); // transform array in 32bit word
    *signature= (uint32_t) (LP32*LP32*LP32);

}


static void Infect(void){
    uint8_t i;
```

```
    uint8_t nullArray[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    for(i=1;i<Ns;i++){
        BlockCopy((uint8_t *)&nstate[i], nullArray);
    }
}


// DPA related functions

// Warning: as it is, this random function is not secure and is used only for test purposes. The knowledge of this
// function to an attacker  would break the randomness
static void MyRand(void){
    SubBytes((state_t *)RandPool);
}


static void SecMult(nstate_t a, nstate_t b, nstate_t c, uint8_t index){ // TYPE III transform
    uint8_t i,j;
    uint8_t r[Ns][Ns];
    for (i=0;i<Ns;i++){
        for (j=i+1;j<Ns;j++){
            if(RandCount==16){
                MyRand();
                RandCount=0;
            }
            r[i][j]=RandPool[RandCount++];
            r[j][i]= r[i][j] ^(Multiply(((uint8_t *)&a[i])[index],((uint8_t *)&b[j])[index]) ^(Multiply(((uint8_t *)&a[j])[index], ((uint8_t *)&b[i])[index])));
        }
    }

    for (i=0;i<Ns;i++){
        ((uint8_t *) &c[i])[index]= Multiply(((uint8_t *)&a[i])[index],((uint8_t *)&b[i])[index]); // conventional GF product.
        for (j=0;j<Ns;j++){
            if (j!=i){
                ((uint8_t *) &c[i])[index]^=  r[i][j];  // masking
            }
        }
    }
}

static void SecEval(nstate_t a,nstate_t b, uint8_t index,const uint8_t *LUT){ //  TYPE II transform
    uint8_t r[Ns][Ns], rp[Ns][Ns];
    uint8_t t,i,j;
    for (i=0;i<Ns;i++){
        for (j=i+1;j<Ns;j++){
            if(RandCount>14 ){
                MyRand();
                RandCount=0;
            }
```

D:\Documents\ELEC22MS\Memoire\simu\C_AES_DPA\C_AES_DPA\aes.c 12

```
            r[i][j]=RandPool[RandCount++];
            rp[i][j]=RandPool[RandCount++];
            t=r[i][j];
            t^= LUT[((uint8_t *)&a[i])[index] ^ rp[i][j]];
            t^= LUT[((uint8_t *)&a[j])[index] ^ rp[i][j]];
            t^= LUT[(((uint8_t *)&a[i])[index] ^ rp[i][j]) ^ ((uint8_t *)&a[j])
              [index]];
            t^= LUT[rp[i][j]];
            r[j][i]= t;

        }
    }

        for (i=0;i<Ns;i++){
            ((uint8_t *) &b[i])[index]= LUT[((uint8_t *)&a[i])[index]];
            for (j=0;j<Ns;j++){
                if (j!=i){
                    ((uint8_t *) &b[i])[index]^=  r[i][j];  // masking
                }
            }
        }

}

static void SecExp254(uint8_t index){
    uint8_t i;
    SecEval(nstate, z, index, power5LUT);  //z<- x^5
    SecEval(z, z, index, power5LUT);  //z<- x^25
    SecEval(z, z, index, power5LUT);  //z<- x^125
    for (i=0;i<Ns; i++){
        ((uint8_t *)nstate[i])[index]=squareLUT[((uint8_t *)nstate[i])[index]];  //
           y <- x^2
    }
    SecMult(nstate,z,nstate,index); // y <- x^125 * x^2 = x^127
    for (i=0;i<Ns; i++){
        ((uint8_t *)nstate[i])[index]=squareLUT[((uint8_t *)nstate[i])[index]];  //
           y <- x^254
    }

}


static void SecSBox(uint8_t index){
    SecExp254(index);
    uint8_t i,j,y;
    // affine transform. See codes for S Box generation for details
    for (i=0;i<Ns;i++){
        y =((uint8_t *) nstate[i])[index]; // the byte on which we apply affine
        for(j=0; j<4; j++) {
            y = (y << 1) | (y >> 7);
            ((uint8_t *) nstate[i])[index] ^= y;
        }
        ((uint8_t *) nstate[i])[index] ^= 0x63;
    }
    #if ( Ns%2 == 0) // optimisation is possible here, just preventing first share
      to be XORED in the first place
```

D:\Documents\ELEC22MS\Memoire\simu\C_AES_DPA\C_AES_DPA\aes.c 13

```
        ((uint8_t *) nstate[0])[index] ^= 0x63;
    #endif
}



static void SecurePredict(uint8_t round) {

    uint8_t i,index,temp[Ns];

    for (i=0; i<4*Ns;i++){
        if(RandCount==16){
            MyRand();
            RandCount=0;
        }
        ((uint8_t *)PredictionMask)[i]=RandPool[RandCount++]; // store the random
          masks that are used
    }
    LPIN[0]=0;
    LPIN[1]=0;
    LPIN[2]=0;
    LPIN[3]=0;
    for(index=0;index<KEYLEN;index++){

        for (i=0; i<Ns;i++){
            temp[i]=((uint8_t *)nstate[i])[index]; // Save state because a call to
              SecSbox modifies the state.
        }
        SecSBox(index);
        for (i=0; i<Ns;i++){


            LPIN[LPIndex[index]]^= ((uint8_t *)nstate[i])[index]; // perform
              prediction first mix (mixed SubBytes).
            ((uint8_t *)nstate[i])[index]=temp[i];   // restore state
            // Xor with keys and a random number to avoid revealing information
              afterwards

        }
    }
    for (i=0; i<Ns;i++){
        LPIN[0] ^= nRoundKey[i][round * Nb * 4    ] ^nRoundKey[i][round * Nb * 4 +
              + 1] ^nRoundKey[i][round * Nb * 4 +        + 2] ^nRoundKey[i][round *
           Nb * 4 +        + 3] ^PredictionMask[i][0];
        LPIN[1] ^= nRoundKey[i][round * Nb * 4 + Nb   ] ^nRoundKey[i][round * Nb *
          4 +  Nb   + 1] ^nRoundKey[i][round * Nb * 4 +    Nb + 2] ^nRoundKey[i]
          [round * Nb * 4 +    Nb    + 3] ^PredictionMask[i][1];
        LPIN[2] ^= nRoundKey[i][round * Nb * 4 + 2*Nb] ^nRoundKey[i][round * Nb * 4
           +  2*Nb + 1] ^nRoundKey[i][round * Nb * 4 + 2 * Nb + 2] ^nRoundKey[i]
          [round * Nb * 4 + 2 * Nb + 3] ^ PredictionMask[i][2];
        LPIN[3] ^= nRoundKey[i][round * Nb * 4 + 3*Nb] ^nRoundKey[i][round * Nb * 4
           + 3* Nb + 1] ^nRoundKey[i][round * Nb * 4 + 3 * Nb + 2] ^nRoundKey[i]
          [round * Nb * 4 + 3 * Nb + 3] ^PredictionMask[i][3];
    }
    GFCube(&LPIN,&signatureIN);
```

```
}
static void SecureCompress(void) {
    uint8_t i;
    LPOUT[0]=0;
    LPOUT[1]=0;
    LPOUT[2]=0;
    LPOUT[3]=0;
    for(i=0;i<Ns;i++){
        LPOUT[0] ^= nstate[i][0][0] ^ nstate[i][0][1] ^nstate[i][0][2] ^nstate[i]
          [0][3] ^PredictionMask[i][0] ; // XOR a "column" together
        LPOUT[1] ^= nstate[i][1][0] ^ nstate[i][1][1] ^nstate[i][1][2] ^nstate[i]
          [1][3] ^PredictionMask[i][1]  ;
        LPOUT[2] ^= nstate[i][2][0] ^ nstate[i][2][1] ^nstate[i][2][2] ^nstate[i]
          [2][3] ^PredictionMask[i][2] ;
        LPOUT[3] ^= nstate[i][3][0] ^ nstate[i][3][1] ^nstate[i][3][2] ^nstate[i]
          [3][3] ^PredictionMask[i][3] ;
    }
    GFCube(&LPOUT,&signatureOUT);

}

static void SecureEDN(void){
    SecureCompress();
    int i;
    for (i=0;i<4;i++){
        if(LPIN[i]!=LPOUT[i])
            Infect();
    }
}


// Cipher is the main function that encrypts the PlainText.
static void Cipher(uint8_t *seed)
{
  uint8_t round = 0;
  uint8_t i,j;

  // Initialize random pool
  BlockCopy(RandPool, seed);
  // Create the shares : Xor initial state with all random shares and store them as
     shares, so the original plaintext is preserved when we xor all the shares
  for (i=1;i<Ns;i++){
      MyRand();
      BlockCopy((uint8_t *)&nstate[i], RandPool);
      BlockXOR((uint8_t *)&nstate[0], RandPool);

  }
  // Add the First round key to the state before starting the rounds.

  for(i=0;i<Ns;i++){
     AddRoundKey(&nstate[i],0,i);
  }
  // There will be Nr rounds.
  // The first Nr-1 rounds are identical.
```

D:\Documents\ELEC22MS\Memoire\simu\C_AES_DPA\C_AES_DPA\aes.c 15

```
  // These Nr-1 rounds are executed in the loop below.
  for(round = 1; round < Nr; ++round)
  {
            for (i=0;i<Ns;i++){
            BlockCopy((uint8_t *)&nstate_copy[i],(uint8_t *)&nstate[i]);
        }

        SecurePredict(round);
        // SubBytes on all the shares simultaneously.
        for (j=0; j<KEYLEN; j++){
        SecSBox(j);
        }
    for(i=0;i<Ns;i++){
        ShiftRows(&nstate[i]);
        MixColumns(&nstate[i]);
        AddRoundKey(&nstate[i],round,i);
    }
    SecureEDN();
  }

  // The last round is given below.
  // The MixColumns function is not here in the last round.
        for (i=0;i<Ns;i++){
            BlockCopy((uint8_t *)&nstate_copy[i],(uint8_t *)&nstate[i]);
        }
      SecurePredict(Nr);

  for (j=0; j<KEYLEN; j++){
      SecSBox(j);
  }
  for(i=0;i<Ns;i++){
      ShiftRows(&nstate[i]);
      AddRoundKey(&nstate[i],Nr,i);
  }
 SecureEDN();

  // Xor all the shares to initial state
  for (i=1;i<Ns;i++){
        BlockXOR((uint8_t *) &nstate[0], (uint8_t *) &nstate[i]);
  }
}



/*****************************************************************************/
/* Public functions:                                                         */
/*****************************************************************************/


void AES128_ECB_encrypt(uint8_t* input, const uint8_t* key, uint8_t* output,
  uint8_t* seed) // out = array of 16 (=KEYLEN) uint8
{

  BlockCopy((uint8_t *)&nstate[0],input);

  #if defined(KEYEXPAND) && KEYEXPAND
```

D:\Documents\ELEC22MS\Memoire\simu\C_AES_DPA\C_AES_DPA\aes.c 16

```
    Key = key;
    KeyExpansion();
  #endif

  // The next function call encrypts the PlainText with the Key using AES
    algorithm.
  Cipher(seed);
  BlockCopy(output, (uint8_t *) &nstate[0]);
}
```

Appendix C

Code : Matlab Piret-Quisquater Fault attack

4/05/16 17:08 D:\Documen...\recover_key_byte_fault_new.m 1 of 1

```
function subkey = recover_key_byte_fault_new(ciphertext, diff_list, inv_s_box,xtime)

 good_tuples=cell(numel(diff_list),256); % good tuples (in matrix form) sorted by↙
error
for l= 1:numel(diff_list)
    [j,i]=find(diff_list{l}>0); % Output difference indexes= (1,4) (2,3) (3,2) (4,1).↙
Reversed because of column based fill of the state
    for e=1:255
        % Generate the set of differences after MC : (1,4) = 2e, (2,4) = e,
        % (3,4) = e, (4,4) = 3e. Attack from left side
        diff_set=[xtime(e+1), e, e, bitxor(e,xtime(e+1),'uint8')];
        key_set=cell(4,1);
        for n=1:4% Build a tuple
            for k=0:255% Go byte by byte.
                %attack from right side
                ciphertext_good=ciphertext;
                ciphertext_fault=bitxor(ciphertext,diff_list{l},'uint8');
                ciphertext_good(i(n),j(n))=bitxor(ciphertext_good(i(n),j(n)),↙
k,'uint8'); % begin with only first difference
                ciphertext_good= inv_shift_rows(ciphertext_good);
                ciphertext_good= sub_bytes(ciphertext_good, inv_s_box);
                ciphertext_fault(i(n),j(n))=bitxor(ciphertext_fault(i(n),j(n)),↙
k,'uint8');
                ciphertext_fault= inv_shift_rows(ciphertext_fault);
                ciphertext_fault= sub_bytes(ciphertext_fault, inv_s_box);
                % Now compare good and faulty states, they must satisfy diff_set
                error=bitxor(ciphertext_good,ciphertext_fault);
                if error(n,j(1))== diff_set(n)
                    %continue: save key byte to tuple and test next byte
                    key_set{n}=[key_set{n},k];
                end
            end
        end
        if (isempty(key_set{1}) ||isempty(key_set{2}) || isempty(key_set{3}) ||↙
isempty(key_set{4}))
            %Do nothing
        else
            good_tuples{l,e+1}=cartprod(key_set{1},key_set{2},key_set{3},key_set{4});
        end
    end
end
% intersect good tuples
A=[256 256 256 256]; % Dummy values
B=[257 257 257 257];% Dummy values
for i=1:256
    if(~isempty(good_tuples{1,i}))
        A=[A;good_tuples{1,i}];
    end
    if(~isempty(good_tuples{2,i}))
        B=[B;good_tuples{2,i}];
    end
end
subkey=intersect(A,B,'rows');
dec2hex(subkey)
```

Bibliography

[1] J. E. Boritz, "Is practitioners' views on core concepts of information integrity", *International Journal of Accounting Information Systems*, vol. 6, no. 4, pp. 260–279, 2005.

[2] *Vernam's one time pad scheme*, `http://www.mils.com/uploads/media/TEC-OTP-04e-h.pdf`, Accessed: 2015-12-05.

[3] J. Daemen and V. Rijmen, "The rijndael block cipher: Aes proposal", in *First Candidate Conference (AES1)*, 1999, pp. 343–348.

[4] *Wikipedia, the advanced encryption standard*, `https://en.wikipedia.org/wiki/Advanced_Encryption_Standard`, Accessed: 2015-12-05.

[5] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards*. Springer Science & Business Media, 2008, vol. 31.

[6] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis", in *Advances in Cryptology—CRYPTO'99*, Springer, 1999, pp. 388–397.

[7] D. Boneh, R. DeMillo, and R. Lipton, "Cryptanalysis in the presence of hardware faults", *Personal communications*, 1996.

[8] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems", in *Advances in Cryptology CRYPTO'97*, Springer, 1997, pp. 513–525.

[9] J. Blömer and J.-P. Seifert, "Fault based cryptanalysis of the advanced encryption standard (aes)", in *Financial Cryptography*, Springer, 2003, pp. 162–181.

[10] C. Giraud, "Dfa on aes", in *Advanced Encryption Standard–AES*, Springer, 2005, pp. 27–41.

[11] G. Piret and J.-J. Quisquater, "A differential fault attack technique against spn structures, with application to the aes and khazad", in *Cryptographic Hardware and Embedded Systems-CHES 2003*, Springer, 2003, pp. 77–88.

[12] C. Giraud and A. Thillard, "Piret and quisquater's dfa on aes revisited.", *IACR Cryptology ePrint Archive*, vol. 2010, p. 440, 2010.

[13] M. Tunstall, D. Mukhopadhyay, and S. Ali, "Differential fault analysis of the advanced encryption standard using a single fault", in *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, Springer, 2011, pp. 224–233.

[14] C. H. Kim and J.-J. Quisquater, "New differential fault analysis on aes key schedule: Two faults are enough", in *Smart Card Research and Advanced Applications*, Springer, 2008, pp. 48–60.

[15] M. M. Kermani and A. Reyhani-Masoleh, "Parity-based fault detection architecture of s-box for advanced encryption standard", in *Defect and Fault Tolerance in VLSI Systems, 2006. DFT'06. 21st IEEE International Symposium on*, IEEE, 2006, pp. 572–580.

[16] M. Mozaffari-Kermani and A. Reyhani-Masoleh, "A lightweight concurrent fault detection scheme for the aes s-boxes using normal basis", in *Cryptographic Hardware and Embedded Systems–CHES 2008*, Springer, 2008, pp. 113–129.

[17] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, "A parity code based fault detection for an implementation of the advanced encryption standard", in *Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings. 17th IEEE International Symposium on*, IEEE, 2002, pp. 51–59.

[18] L. Genelle, C. Giraud, and E. Prouff, "Securing aes implementation against fault attacks", in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on*, IEEE, 2009, pp. 51–62.

[19] M. Karpovsky, K. J. Kulikowski, and A. Taubin, "Robust protection against fault-injection attacks on smart cards implementing the advanced encryption standard", in *Dependable Systems and Networks, 2004 International Conference on*, IEEE, 2004, pp. 93–101.

[20] M. Medwed and J.-M. Schmidt, "A continuous fault countermeasure for aes providing a constant error detection rate", in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*, IEEE, 2010, pp. 66–71.

[21] B. Gierlichs, J.-M. Schmidt, and M. Tunstall, "Infective computation and dummy rounds: Fault protection for block ciphers without check-before-output", in *Progress in Cryptology–LATINCRYPT 2012*, Springer, 2012, pp. 305–321.

[22] A. Battistello and C. Giraud, "Fault analysis of infective aes computations", in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, IEEE, 2013, pp. 101–107.

[23] N. Ferguson, J. Kelsey, S. Lucks, B. Schneier, M. Stay, D. Wagner, and D. Whiting, "Improved cryptanalysis of rijndael", in *Fast software encryption*, Springer, 2001, pp. 213–230.

[24] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards sound approaches to counteract power-analysis attacks", in *Advances in Cryptology—CRYPTO'99*, Springer, 1999, pp. 398–412.

[25] V. Grosso, F.-X. Standaert, and S. Faust, "Masking vs. multiparty computation: How large is the gap for aes?", *Journal of Cryptographic Engineering*, vol. 4, no. 1, pp. 47–57, 2014.

[26] G. Piret and F.-X. Standaert, "Security analysis of higher-order boolean masking schemes for block ciphers (with conditions of perfect masking)", *Information Security, IET*, vol. 2, no. 1, pp. 1–11, 2008.

[27] E. Prouff, M. Rivain, and T. Roche, "On the practical security of a leakage resilient masking scheme", in *Topics in Cryptology–CT-RSA 2014*, Springer, 2014, pp. 169–182.

[28] J. Blömer, J. Guajardo, and V. Krummel, "Provably secure masking of aes", in *Selected Areas in Cryptography*, Springer, 2005, pp. 69–83.

[29] M. Rivain and E. Prouff, "Provably secure higher-order masking of aes", in *Cryptographic Hardware and Embedded Systems, CHES 2010*, Springer, 2010, pp. 413–427.

[30] Y. Li, K. Sakiyama, S. Gomisawa, T. Fukunaga, J. Takahashi, and K. Ohta, "Fault sensitivity analysis", in *Cryptographic Hardware and Embedded Systems, CHES 2010*, Springer, 2010, pp. 320–334.

[31] F. Dassance and A. Venelli, "Combined attacks on the aes key schedule.", *IACR Cryptology ePrint Archive*, vol. 2012, p. 98, 2012.

[32] T. Roche, V. Lomné, and K. Khalfallah, "Combined fault and side-channel attack on protected implementations of aes", in *Smart Card Research and Advanced Applications*, Springer, 2011, pp. 65–83.

[33] Y. Ishai, M. Prabhakaran, A. Sahai, and D. Wagner, "Private circuits ii: Keeping secrets in tamperable circuits", in *Advances in Cryptology-EUROCRYPT 2006*, Springer, 2006, pp. 308–327.

[34] T. G. Malkin, F.-X. Standaert, and M. Yung, "A comparative cost/security analysis of fault attack countermeasures", in *Fault Diagnosis and Tolerance in Cryptography*, Springer, 2006, pp. 159–172.

[35] K. J. Kulikowski, M. G. Karpovsky, and A. Taubin, "Robust codes and robust, fault-tolerant architectures of the advanced encryption standard", *Journal of systems Architecture*, vol. 53, no. 2, pp. 139–149, 2007.

[36] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, "Countermeasures against fault attacks on software implemented aes: Effectiveness and cost", in *Proceedings of the 5th Workshop on Embedded Systems Security*, ACM, 2010, p. 7.

[37] L. Genelle, E. Prouff, and M. Quisquater, "Thwarting higher-order side channel analysis with additive and multiplicative maskings", in *Cryptographic Hardware and Embedded Systems–CHES 2011*, Springer, 2011, pp. 240–255.

[38] H. Kim, S. Hong, and J. Lim, "A fast and provably secure higher-order masking of aes s-box", in *Cryptographic Hardware and Embedded Systems–CHES 2011*, Springer, 2011, pp. 95–107.

[39] T. Roche and E. Prouff, "Higher-order glitch free implementation of the aes using secure multi-party computation protocols", *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 111–127, 2012.

[40] J.-S. Coron, E. Prouff, M. Rivain, and T. Roche, "Higher-order side channel security and mask refreshing", in *Fast Software Encryption*, Springer, 2013, pp. 410–424.

[41] V. Grosso, E. Prouff, and F.-X. Standaert, "Efficient masked s-boxes processing–a step forward–", in *Progress in Cryptology–AFRICACRYPT 2014*, Springer, 2014, pp. 251–266.

[42] *Wikipedia,frobenius endomorphism*, `https://en.wikipedia.org/wiki/Frobenius_endomorphism`, Accessed: 2016-03-13.

[43] V. Lomné, T. Roche, and A. Thillard, "On the need of randomness in fault attack countermeasures-application to aes", in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2012 Workshop on*, IEEE, 2012, pp. 85–94.

[44] J.-S. Coron, E. Prouff, M. Rivain, and T. Roche, "Higher-order side channel security and mask refreshing", in *Fast Software Encryption*, Springer, 2014, pp. 410–424.

[45] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks", in *Cryptographic Hardware and Embedded Systems-CHES 2002*, Springer, 2003, pp. 2–12.

[46] J.-M. Dutertre, A.-P. Mirbaha, D. Naccache, and A. Tria, "Reproducible single-byte laser fault injection", in *6th Conference on Ph. D. Research in Microelectronics & Electronics, PRIME 2010*, 2010.

[47] B. M. Gammel and S. Mangard, "On the duality of probing and fault attacks", *Journal of Electronic Testing*, vol. 26, no. 4, pp. 483–493, 2010.

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