UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Hardware-Trojans: attacks implementations against (un)-protected block cipher designs

Dissertation presented by
Charles MOMIN

for obtaining the Master's degree in
Electrical Engineering

Supervisor(s)
François-Xavier STANDAERT

Reader(s)
David BOL, Oliver BRONCHAIN

Academic year 2017-2018

Abstract

Thanks to Snowden revelations, a societal awareness about security issues has emerged simultaneously with the exponential number of connected electronic devices (i.e. IoT). This need for security is generally answered by designers based on well known mathematical descriptions of cryptographic algorithms. However, their practical hardware implementations may be exposed to security failures considering physical attacks. As an example, due to the globalization of ICs devices design, the threat of untrusted manufacturers involved in the fabrication process has emerged. This ultimate physical attack is known as hardware Trojan and can expose ICs to huge security failures.

In this work, the practical of hardware Trojan exploiting side-channel/physical behavior of electronic devices is explored. First, it presents in details a parametric Trojan implementation based on timing violation and inducing faults in a pipelined AES block cipher architecture. Based on it, it describes secret key recovery methods using classical differential fault analysis and discuss the trade off encountered using the proposed strategy. Second, it presents in details a side-channel Trojan implementation, targeting protected AES and MOE block cipher implementation using the Trojan resilient framework proposed at CCS 2016 by Dziembowski et. al. Again, a key recovery strategy is described and discussions about the different trade off encountered and potential ameliorations are provided.

Acknowledgement

First of all, I would like to express my gratitude to my advisor Professor Standaert for the proposed topic, his constant availability, his valuable comments and suggestions during our various interviews, as well as giving me the freedom to choose my work directions and to focus on my interests.

A special thanks to Olivier for the global support provided, both in my research, during the implementation on FPGA and during my writing. His experience, availability, advices and the discussions shared during coffee breaks were a great help.

Thank you to Professor Bol for answering my few questions and accepting to be a member of my jury.

Thanks to Brigitte Dupont for the provision of the various tools used and her sympathy.

Finally, thank you to my friends and family for their help, support and encouragement.

Contents

Nomenclature

AES Advanced Encryption Standard

ATPG Automatic Test Pattern Generation

CMP Chemical-Mechanical Planarization

DFA Differential Fault analysis

DOS Denial of Service

DPA Differential Power Analysis

EDA Electronic Design Automation

FIB Focus Ion Beam

LSB Least Significant Bit

LUT Look-Up Tables

MI Mutual Information

MPC Multi-Party Computation

MSB Most Significant Bit

NIST National Institute of Standard and Technologies

PRNG Pseudo Random Number Generator

PRP Pseudo Random Permutation

SEM Scanning Electron Microscope

SPN Substitution-Permutation Network

VLSI Very-Large-Scale Integration

Chapter 1

Introduction

Motivations

In the current society, the use of electronic devices is in constant expansion. The growth of connected devices usage, such as smart-phones, PCs or any other connected devices indicates that the connectivity tends to expand to everyday life objects: cars, lightning systems, communication systems, household devices, ... This revolution is more known as the Internet of Things (IoT). According to a 2011 Cisco's study [15], the amount of connected device is bigger than the world population since 2008. Moreover, it estimates that this will grow to approximately 50 billion by 2020. The Ericsson company, for its part, estimates that this number would be 25 billion [14].

Jointly, the need for security is growing. Indeed, numerous fields of applications require data protection at some points. Obvious examples are the requirement of encryption and signature schemes in order to have secure communication channels in payment systems, online account management or military systems. On the other hand, nowadays society is under the pressure of a constant need for performance growth. In these purposes, one may considered to implement security algorithms in hardware to boost performances. Unfortunately, despite the fact that mathematical descriptions of the algorithms are supposed secure, the actual implementation can be exposed to security failures if not done properly. For example, an attacker may take advantage of chips side-channel leakages, such as power traces [1],[27], Electromagnetic waves (EM) [24] or faults appearing at some steps [34], in order to obtain sensitive information.

Problem statement

Nowadays economic tends to drive Integrated Circuits (ICs) design and manufacturing into a globalized process (as seen in Fig.1.1). First,transistor feature size and time to market are continuously shrinking. Second, there is a growing demand for low power and high performance ICs. Because of the high technology price required, it becomes difficult for many companies to use a unique design house for design and fabrication. For example, the approximated minimal cost needed to build a fabrication plant of 300mm wafers in 65nm technology is about 4 billions dollars [2]. Based on that, it is in practice increasingly difficult to verify that chips are correctly built regarding their specifications. In particular, one may assume that members of the ICs manufacturing flow may not be trusted. Considering that, design may be altered at some points by malicious actors, for instance under government pressure. This introduces a specific threat

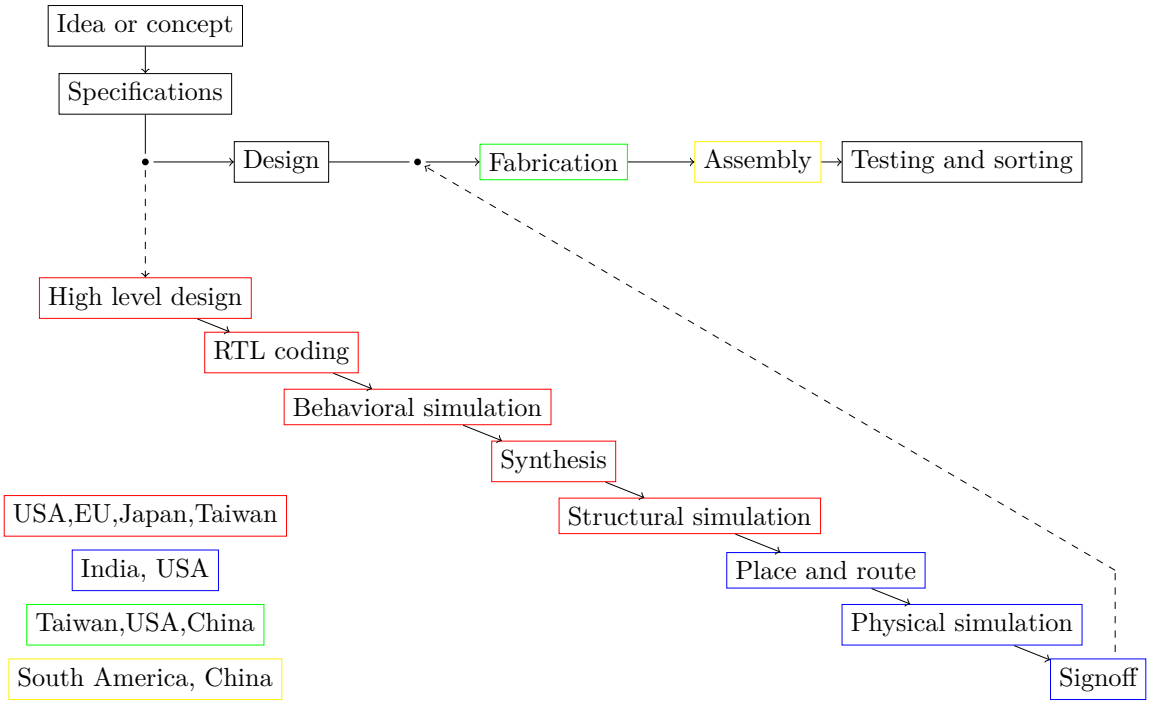

Figure 1.1: ICs design business organization

known as hardware Trojan.

According to [5], a hardware Trojan is a malicious, undesired, intentional modification of an electronic circuit or design, resulting in the undesired behaviour of an electronic device when in operation. If done properly, these modifications may lead in fact to stealthy modifications of the system specifications. Several taxonomies have been proposed [9],[35] in order to characterize their different attributes . Without generality loss, Trojans are designed to trigger under conditions known by the attacker called the trigger conditions. These are usually chosen by the latter during the Trojan design phase in order to avoid any involuntary activation of the Trojan, which may lead to its detection. Here, involuntary means that the Trojan is activated without a specific activation mechanism used by the attacker. A common distinction between Trojan is made according to their trigger conditions:

- **Digitally triggered**: Trojans of this type are activated if they reach a specific internal state, thanks to individual or sequential particular digital inputs (known as cheat code). A typical example is an activation that occurs when a precise input data is provided to the infected device. This type also includes Trojans that are activated once the device is used a certain amount of time (known as time-bomb)

- **Physically triggered**: Trojans of this type are activated thanks to specific running environment conditions. A typical example is an activation that is based on particular temperature, pressure or location. This type also includes Trojans that are activated thanks to external trigger signals, using for example an embedded antenna.

Once triggered, the Trojan proceeds to its malicious purposes called the payload. Numerous types of payload can be found in the literature. In [23], authors designed a malicious processor allowing attackers to get root privileges without checking credentials or creating log entries. In [25], authors designed a Trojan inducing physical side-channel leakages to convey secret information. In [31], authors designed a Trojan that degrades network services inside a corporate network once triggered.

Nowadays, correct functionalities regarding chips specifications is ensured by tests and verifications strategies[17]. Unfortunately, an extra functionality may not be detected by such mechanisms. In fact, this is a real challenge: given the amount of different trigger possibilities, a Trojan can easily be hidden in all system with a consistent logic. Unfortunately, there is no miracle methods and detection mechanisms have their strengths and weaknesses in detecting different types of design alterations.

Numerous proposed Trojans designs require additional logic to properly operate [23],[20], [31]. To avoid detection, it is desirable for attacker to implement efficient Trojans with the smallest impact on targeted design. The optimal objective is obviously to design ones requiring no additional logic at all while still allowing to induce malicious behavior. In this purpose, low level Trojans designs are proposed in the literature [16], [13]. One may wonder about the practical side of the insertion of such Trojans. In another hand, some protection mechanisms have been developed in order to protect computation based on untrusted chips [12]. However, as shown in this work, the security assured by the countermeasure is broken by slightly deviating from the considered threat menace in a practical way.

Contributions

The first contribution brought by this work is a Trojan implementation allowing key recovery for the Advanced Encryption Standard (AES) block cipher, while requiring no additional logic to properly operate. Mixing ideas of [16] and [13], the Trojan is triggered by using a clock frequency that exceed a specific threshold, which leads to a violation of setup timing constraints. The key is then recovered using a Differential Fault Analysis (DFA).

The second contribution is a Trojan implementation allowing payload delivery, while the targetted chip is under a Trojan protection mechanism proposed in [12]. More particularly, this work focus on block cipher implementations. Based on testing amplification using secure 3-party protocol, majority vote and random testing, the countermeasure bounds the probability of successful attack using digitally triggered Trojan. This works shows that slightly deviate from this threat model by considering a specific type of side-channel Trojan makes room to powerful attacks.

This work begins with useful background required to understand the contributions. It follows with the description of the formal framework used. Then both Trojans are described in details, following a to a top-down approach. A threat model is described, followed by the targetted design architecture description. Then a high level description of Trojans and design choices are given as well as actual implementation results.

Chapter 2

Background

This section aims to remember the basics required in order to properly understand the contributions. The discussed concepts are the symmetric-key encryption, the block cipher design and more particularly the AES, the DFA against AES and finally the setup timing constraint in ICs design. If a reader is comfortable with these, free to him to skip the following parts.

2.1 Symmetric-key encryption

Symmetric key encryption algorithms allow to set up secure channel between different actors. These use a shared secret key k required to secure the data that are flowing through the channel. The security is done using an encryption algorithm E_k and a decryption algorithm D_k. A typical example is shown in Fig.2.1. The latter represents two actors, Alice and Bob, that want to communicate between each other in a secure way, while their communications are observed by a third actor Eve. The communication happens as follows:

- Alice and Bob agree on a shared secret key k that will be used during the time of the communication.
- Alice (Bob) wants to send a message to Bob (Alice). She (he) first encrypts her (its) message, known as the plaintext p, using k with E_k. From this encryption, she (he) obtains the encrypted message known as the ciphertext $c = E_k(p)$. Thanks to cryptographic properties, it is assumed to be difficult for Eve to retrieve p from c if she does not know k.
- Alice (Bob) sends c over the channel to Bob.
- Bob (Alice) receives c, decrypts it using k and D_k to obtain the plaintext encrypted by Alice (Bob) $p' = D_k(c) = p$.

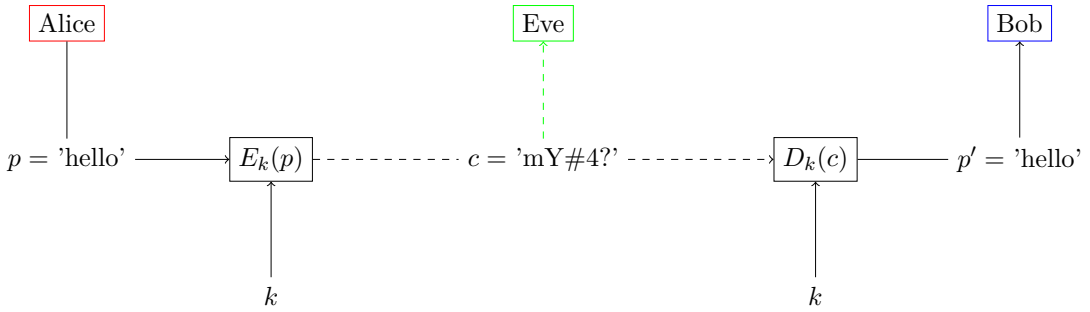

Figure 2.1: Use of block cipher in secure channel communication

2.2 Block cipher

A block cipher is [32] a deterministic algorithm transforming a plaintext block of fixed size n into a ciphertext block of similar size. The transformation is made by repeatedly applying an invertible transformation ρ, called the round transformation, which is specified by a symmetric key k. Denoting the plaintext p_0 and the ciphertext p_R, it holds:

$$\begin{aligned} p_{r+1} &= \rho_{k_r}(p_r) \\ r &= 1, 2, ..., R \end{aligned}$$

where the k_r values are called the sub-keys and are generated by a key scheduling algorithm.

More formally, block ciphers are Pseudorandom Permutations (PRP) implementations [22]. A PRP is a one-to-one keyed function F that cannot be distinguished from a random permutation with uniform distributions. The function F is denoted as

$$F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* : (k, x) \to F(k, x) = F_k(x)$$

The main idea in such an encryption process is that the key k defines which permutation is used. Considering a size of n bits, the maximum amount of permutations from $\{0,1\}^* \times \{0,1\}^*$ is given by $(2^n)!$. However, taking into account a key size of n bits, there are at most 2^n different permutations. The key size is called the computational security parameter and must be set according to targeted security level (typical current size $n = 128$). Additionally, F is said to be efficient if there are deterministic polynomial-time algorithms to compute $F_k(x)$ and $F_k^{-1}(x)$ given k, x.

Since describing a full n bits permutation requires $n \cdot 2^n$ bits, it is nowadays impossible to achieve. Instead, block ciphers are built based on constructions that use smaller permutations. However, since PRP need to look random, concatenating results of permutations of size n' ($n' < n$) does not work. Indeed, for one bit that toggles in the plaintext, there are at most n' bits changing in the ciphertext. To encounter the problem, block ciphers are built using the following algorithm known as the *confusion-diffusion paradigm*:

- Compute $F(x) = f_1(x_1)||f_2(x_2)||...$ where $f_i(x_i)$ is a permutation over $n' < n$ bits known as S-box and $|x_i| = n'$. This is the confusion operation.
- Reorder the bits of $F(x)$. This is the diffusion operation.
- Repeat these steps several times.

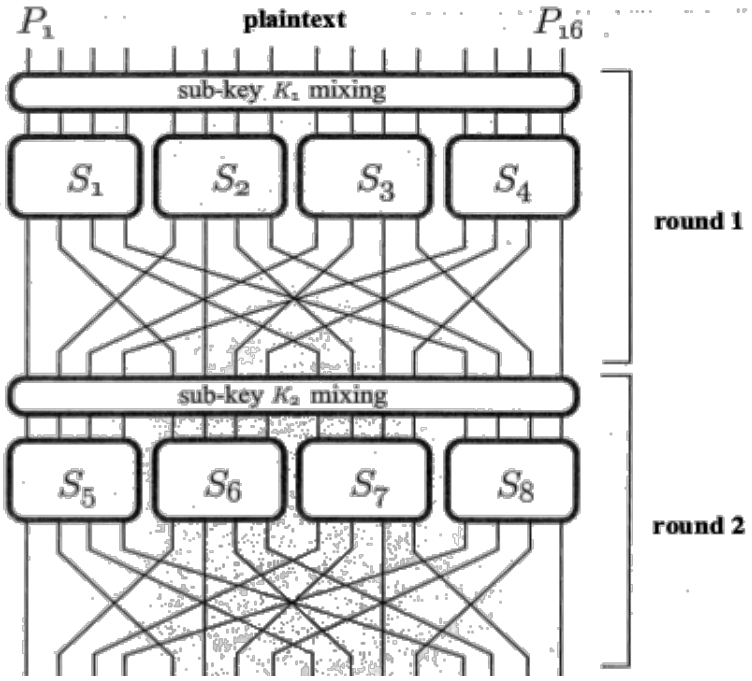

Figure 2.2: Substitution-Permutation Network [22]

A Substitution-Permutation Network (SPN) is an application of the *confusion-diffusion paradigm* (shown in Fig.2.2) In practice fixed S-boxes are used. The key dependency is achieved by combining sub-keys k_r with the S-boxes inputs. These S-boxes must be invertible in order to be able to decrypt the produced ciphertext. Moreover, the reordering combined with the S-boxes must be designed such as each change at the input must result in a large change at the output (known as the avalanche effect).

2.2.1 Advanced Encryption Standard

The Advanced Encryption Standard (also known as Rijndael) [10],[11] is a block cipher adopted by the National Institute of Standard and Technologies (NIST) in 2001. It operates on 128 bits long plaintexts with keys size of 128, 192 or 256 bits. This work focus on the version using 128 bits of key size. The encryption process (presented in Fig.2.3) involves 10 successive rounds performing 4 different operations defined at the byte level. More formally, the bytes are representing elements of the Galois Field $GF(2^8)$. As specified before, the operations follow the *confusion-diffusion* paradigm.

`AddRoundKey` Operation needed to involve key dependency. It consists in a linear addition over $GF2^8$, between the input S and the subkey of the round k_r. The `AddRoundKey` output of the r^{th} round is denoted $S_{r,AK}$.

$$S = (S_{i,j}) \in \left(GF(2^8)\right)^{4\times 4} \quad \rightarrow \quad S_{r,AK} \in \left(GF(2^8)\right)^{4\times 4}$$

$$S = \begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix} \rightarrow \begin{bmatrix} S_{0,0} \oplus k_{r,0,0} & S_{0,1} \oplus k_{r,0,1} & S_{0,2} \oplus k_{r,0,2} & S_{0,3} \oplus k_{r,0,3} \\ S_{1,0} \oplus k_{r,1,0} & S_{1,1} \oplus k_{r,1,1} & S_{1,2} \oplus k_{r,1,2} & S_{1,3} \oplus k_{r,1,3} \\ S_{2,0} \oplus k_{r,2,0} & S_{2,1} \oplus k_{r,2,1} & S_{2,2} \oplus k_{r,2,2} & S_{2,3} \oplus k_{r,2,3} \\ S_{3,0} \oplus k_{r,3,0} & S_{3,1} \oplus k_{r,3,1} & S_{3,2} \oplus k_{r,3,2} & S_{3,3} \oplus k_{r,3,3} \end{bmatrix} = S_{r,AK}$$

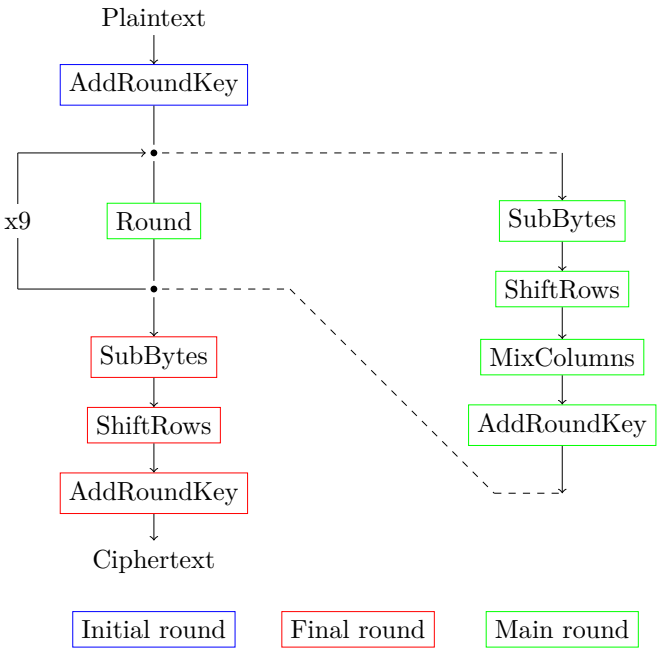

Figure 2.3: AES encryption process

`SubBytes` Operation involved in the confusion task. It consists in a non-linear substitution of each element of the input S using the Rijndael S-box $SubBytes(x)$. The `SubBytes` output of the r^{th} round is denoted $S_{r,SB}$.

$$\begin{aligned} S = (S_{i,j}) \in \left(GF(2^8)\right)^{4\times 4} &\rightarrow S_{r,SB} = (S_{r,SB,i,j}) \in \left(GF(2^8)\right)^{4\times 4} \\ \forall i,j : S_{r,SB,i,j} &= SubBytes(S_{i,j}) \end{aligned}$$

The $SubBytes(x)$ substitution is constructed by applying two successive transformations to the byte x.

1. Taking the multiplicative inverse of x in $GF(2^8)$ denoted as x^{-1}. The byte $'00'$ is mapped to itself.

2. Applying an affine transformation over $GF(2^8)$ defined by:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0^{-1} \\ x_1^{-1} \\ x_2^{-1} \\ x_3^{-1} \\ x_4^{-1} \\ x_5^{-1} \\ x_6^{-1} \\ x_7^{-1} \end{bmatrix}$$

   where $y = SubBytes(x)$ and x_0^{-1}, y_0 are the Least Significant Bit (LSB) of the bytes x^{-1} and y.

`ShiftRows` Operation involved in the diffusion task. It consists in a cyclic shift of the input rows $S_{i,*}$ depending on the row position i. From the smallest to the largest value of i, the rows

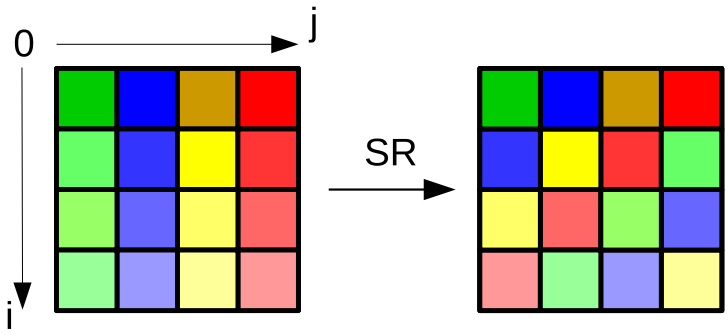

Figure 2.4: Illustration of the `ShiftRows` operation in AES

are shifted over 0,1,2 or 3 bytes to the left (cfr Fig.2.4). The `ShiftRows` output of the r^{th} round is denoted $S_{r,SR}$.

$$S = (S_{i,j}) \in \left(GF(2^8)\right)^{4\times 4} \quad \rightarrow \quad S_{r,SR} \in \left(GF(2^8)\right)^{4\times 4}$$

$$S = \begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix} \Rightarrow \begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,1} & S_{1,2} & S_{1,3} & S_{1,1} \\ S_{2,2} & S_{2,3} & S_{2,0} & S_{2,1} \\ S_{3,3} & S_{3,0} & S_{3,1} & S_{3,2} \end{bmatrix} = S_{r,SR}$$

`MixColumns` Operation involved in the diffusion task. Consists in a multiplication over GF(2^8) modulo $x^4 + 1$ between the input columns $S_{*,j}$, considered here as polynomial over GF(2^8), and a fixed polynomial $c(x)$ given by

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

The multiplication $b_{*,j}(x) = c(x) \times S_{*,j}(x)$ can be written as a matrix multiplication given by:

$$\begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{0,j} \\ S_{1,j} \\ S_{2,j} \\ S_{3,j} \end{bmatrix}$$

Based on that, the `MixColumns` output of the r^{th} round is denoted $S_{r,MC}$.

$$S = (S_{i,j}) \in \left(GF(2^8)\right)^{4\times 4} \quad \rightarrow \quad S_{r,MC} \in \left(GF(2^8)\right)^{4\times 4}$$

$$S = \begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix} \rightarrow \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} = S_{r,MC}$$

2.3 Fault model and Differential Fault Analysis against AES

Powerful attacks based on fault introduction exist against block cipher implementations. In this purpose, an attacker may exploit Trojan to deliberately introduce such faults [28],[31],[16], [13], [29] in order to produce a Denial of Service (DOS) or to recover sensitive information.

Depending on the methodology used, the attacker may have more or less control on the introduced faults. In general, they aim to find an attack optimizing the following features:

- **The amount of faults needed for the attack to success**: Depending on the attacker model, introducing efficiently some faults may be a challenging or a costly operation. A minimal amount of faults is thus desirable.
- **The type of faults**: Introduced faults may need to be very accurate for the attack to succeed. From the more to the less restrictive, the different faults types are the following:
  - *Set bit*: A bit is set to a defined value, introducing a fault in comparison to the correct specifications.
  - *Toggle bit*: A bit toggles from its original value, introducing a fault in comparison to the correct specifications.
  - *Random bit*: A bit has a random value, introducing a fault in comparison to the correct specifications.
- **The accuracy of the fault**: The fault may need to be introduced from a specific bit to any byte. One may prefer minimum accuracy requirement since it is less restrictive.
- **The post processing complexity**: The algorithm needed to take advantage of the introduced faults may require different complexity. A low complexity is obviously desirable.

More specifically, this work will consider a unique fault model that one may appreciate thanks to its randomness and byte location: the *Random byte error* model. This models specifies that, at some point in the processed data, a byte has not the expected value regarding the specifications but is instead considered as a random value. Attacks exist for other models, but they are out of the scope of this work.

AES hardware implementations are vulnerable to faults insertion attacks [28],[34]. Using the inserted faults, an attacker may take advantage of information leaked on the key by a faulty/correct ciphertexts pair. This process is called a differential fault analysis (DPA). The main features of the attack depend on the location of the fault introduction. In order to properly understand the mechanisms of DPA, one may cryptanalyse more into the details the behaviour of AES internal states if a fault occurs at some point in the encryption process. Thanks to the *confusion-diffusion paradigm*, an error appearing in the first rounds will lead to a random modification in the output ciphertext. Possible exploits begin thus at the end of the encryption process. The different operations of the r^{th} round are denoted AK_r (`AddRoundKey`), SB_r (`SubBytes`), SR_r (`ShiftRows`) and MC_r (`MixColumns`).

Fault insertion before MC_9. The *Random byte error* model becomes useful when a fault is introduced before MC_9. Note however that DFA may be useful at earlier stages considering other faults models. First of all, remember that the `MixColumns` operation is multiplying every columns $S_{*,j}$ of the input by a given matrix:

$$\begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{0,j} \\ S_{1,j} \\ S_{2,j} \\ S_{3,j} \end{bmatrix}$$

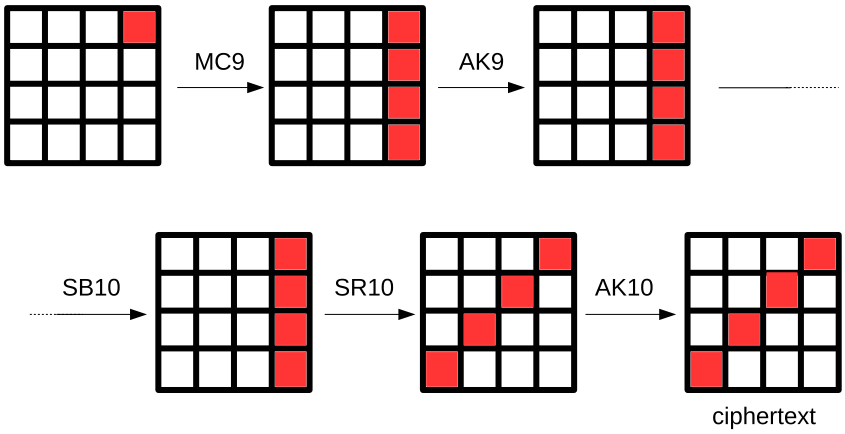

Figure 2.5: *Random byte error at* $[0, 3]$ *propagation pattern*

Considering a random byte error appearing in $S_{0,3}$ (as depicted in Fig.2.5), one may see such an error as a value δ that is added in GF(2^8) to the original correct byte, as depicted by:

$$\begin{bmatrix} \tilde{b_{0,3}} \\ \tilde{b_{1,3}} \\ \tilde{b_{2,3}} \\ \tilde{b_{3,3}} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{0,3} \oplus \delta \\ S_{1,3} \\ S_{2,3} \\ S_{3,3} \end{bmatrix} = \begin{bmatrix} b_{0,3} \\ b_{1,3} \\ b_{2,3} \\ b_{3,3} \end{bmatrix} \oplus \begin{bmatrix} 02 \times \delta \\ 01 \times \delta \\ 01 \times \delta \\ 03 \times \delta \end{bmatrix}$$

After MC_9, the faulty columns differs from the theoretical correct one according to a defined difference pattern. There are 4 different patterns, based on the position of the faulty byte in the altered column. Here, the altered column position is assumed to be known by the attacker. In practice, error may appear in any position, leading to 4 different patterns for each altered column. In every cases, a faulty byte appearing before MC_9 leads to 4 faulty bytes in the ciphertext. The altered column position may thus be recovered using the propagation pattern of the error (as shown in Fig.2.6) An attacker may take advantage of this property by mounting a divide and conquer attack, finding candidates for 32bits parts of the key for which a difference pattern holds. More formally, one may consider the following representation for the last round key:

$$\begin{bmatrix} RK_{15} & RK_{11} & RK_7 & RK_3 \\ RK_{14} & RK_{10} & RK_6 & RK_2 \\ RK_{13} & RK_9 & RK_5 & RK_1 \\ RK_{12} & RK_8 & RK_4 & RK_0 \end{bmatrix}$$

Based on the propagation error pattern and considering a faulty byte in the i^{th} column before MC_9, information about the following 32bits key parts $\mathcal{P}_i$ of the can be obtained:

$$\begin{aligned} \mathcal{P}_3 &= [RK_3, RK_6, RK_9, RK_{12}] \\ \mathcal{P}_2 &= [RK_7, RK_{10}, RK_{13}, RK_0] \\ \mathcal{P}_1 &= [RK_{11}, RK_{14}, RK_1, RK_4] \\ \mathcal{P}_0 &= [RK_{15}, RK_2, RK_5, RK_8] \end{aligned}$$

Considering a ciphertext C and its corresponding faulty ciphertext $\tilde{C}$, the attack against a key part $\mathcal{P}_i$ takes place as follows:

1. Generate the 4 possible difference sets for each of the 255 initial δ.

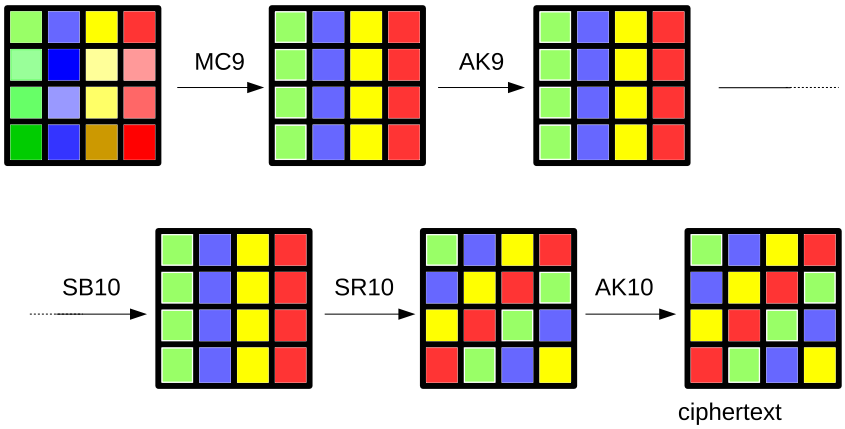

Figure 2.6: *Error propagation patterns with fault introduction before* MC_9. Before MC_9, every color shade represents a different random byte fault appearing in each positions of each columns. Eventually, each fault contaminate only one column after `MixColumns`.

2. Take a guessed value $\tilde{\mathcal{P}}_i$ for the key part $\mathcal{P}_i$.

3. Reverse AK_{10}, SR_{10} and SB_{10} for $\tilde{C}$ and C using $\tilde{\mathcal{P}}_i$ to obtain the internal states $\tilde{t}$ and t. There is no need to reverse AK_9 since it is a linear operation.

4. Compute $\Delta t = \tilde{t} \oplus t$

5. Check if the i^{th} column of Δt is present in a difference set. If yes then $\tilde{\mathcal{P}}_i$ is kept in $\mathcal{L}_i$

6. If all the possible values of $\mathcal{P}_i$ have been tested, stop. Else, go to 2.

Once all the different values over 32bits have been tried for $\mathcal{P}_i$, it remains 255 key parts candidates $\tilde{\mathcal{P}}_i$ per difference set. If the attacker does not know which byte in the column is faulty, that left 1020 candidates for the attacked part $\mathcal{P}_i$. In this case, the attack filters 2^{32} possible values to keep $1020 \approx 2^{10}$ remaining possibilities. Using information theory entropy, one may intuitively consider that each attack using a pair $(C, \tilde{C})$ allows to recover 22 bits of the key part $\mathcal{P}_i$. From this, by using the same method with a second pair $(C', \tilde{C}')$ for the same part $\mathcal{P}_i$ and only keeping the matching value between $\mathcal{L}_i$ and $\mathcal{L}'_i$, one may expect on average 1 remaining value $\tilde{\mathcal{P}}_i = \tilde{\mathcal{P}}'_i$.

In order to recover the full key, the attack needs to be performed on every key parts $\mathcal{P}_i$, requiring thus a total of 2 correct/faulty ciphertexts pairs for each part $\mathcal{P}_i$. Note that, since there is no `MixColumns` operation involved at this stage, one may look for 1020 possible values of $\tilde{\mathcal{P}}_i$ byte by byte in order to reduce the time complexity. Indeed, in this case, the faults are related to only one byte at a time and is not mixed amongst the byte value of a unique column, leading to a search complexity of $\mathcal{O}\left(4 \cdot 2^8\right)$.

Fault insertion before SR_9, SB_9 or AK_8. The `ShiftRows` operation only consists in routing modifications and the bytes values are thus not modified. Moreover, any random byte fault appearing before the `SubByte` or the `AddRoundKey` operations is similar to one appearing after those. From that, a fault appearing before these operations is the same as one appearing before MC_9.

Fault insertion before MC_8. At this stage, the state-of-the-art fault attack against AES is reached. The main difference in comparison to a faulty byte appearing before MC_9 is the number of fault needed to recover the complete key. First of all, one may be interested in looking at the error propagation pattern (cfr Fig.2.7). Thanks to the 2 `MixColumns` operations, a random byte fault appearing before MC_8 may be seen as 4 random bytes errors appearing in the different columns of the state before MC_9. Remembering the attack describe above, the whole key is thus recoverable using 2 faults injections before MC_8, using the same timing complexity to perform the DPA.

Moreover, considering a Chosen Plaintext Attack (CPA) model, fault insertion at this stage allows a key recovery using a unique correct/faulty ciphertexts pair. As said previously, one can find for each key part $\mathcal{P}_i$ some candidates $\tilde{\mathcal{P}}_i$ that fulfill the difference pattern after MC_9. In addition to the previous attack, more information on the key is available since the candidates must also fulfill the difference pattern after MC_8. Considering the correct/faulty ciphertexts pair $\left[C, \tilde{C}\right]$, an attack takes place as follows:

1. Generate the 4 possible difference sets for each of the 255 initial δ.
2. For each key part $\mathcal{P}_i$, generate as set $\mathcal{L}_i$ of key part candidates $\tilde{\mathcal{P}}_i$ that fulfilling a possible difference pattern after MC_9.
3. Take a 128 bits long key candidate $k = [\mathcal{P}_{0,a}, \mathcal{P}_{1,b}, \mathcal{P}_{2,c}, \mathcal{P}_{3,d}]$, where $\mathcal{P}_{0,a} \in \mathcal{L}_0$; $\mathcal{P}_{1,b} \in \mathcal{L}_1$; $\mathcal{P}_{2,c} \in \mathcal{L}_2$ and $\mathcal{P}_{3,d} \in \mathcal{L}_3$.
4. Inverse $AK_{10}, SR_{10}, SB_{10}, AK_9$,$MC_9$,$SR_9$ and SB_9 for $\left[C, \tilde{C}\right]$ by using k to obtain the internal states t' and $\tilde{t}'$.
5. Compute $\Delta t' = \tilde{t}' \oplus t'$.
6. Check if the i^{th} column of $\Delta t'$ is present in a difference set. If yes then k is kept in $\mathcal{L}'$.
7. If all the possible values of k have been tested, continue. Else, go to 3.
8. Try exhaustively each remaining candidates $k' \in \mathcal{L}'$ by checking if $(D_{k'}(C) == p)$.

Since the error appearing before MC_8 is of the random byte model, one may except 2^8 remaining key candidates after the second filtering phase.

Fault insertion before SR_8, SB_8 or AK_7. Again, `ShiftRows` is only a routing operation and any random byte fault appearing before the `SubByte` or the `AddRoundKey` operations is similar to one appearing after those. In conclusion, a fault appearing before these is the same as one appearing before MC_8.

2.4 Setup timing constraints in ICs design

In ICs design, one aims to meet the correct specifications at a targetted clock frequency. However, if not done properly, the design may lead to unstable processed signals, potentially producing faults or even deadlocks if the clock frequency exceeds a specific threshold.

To understand why such failures appear, one may be interested in checking a general situation

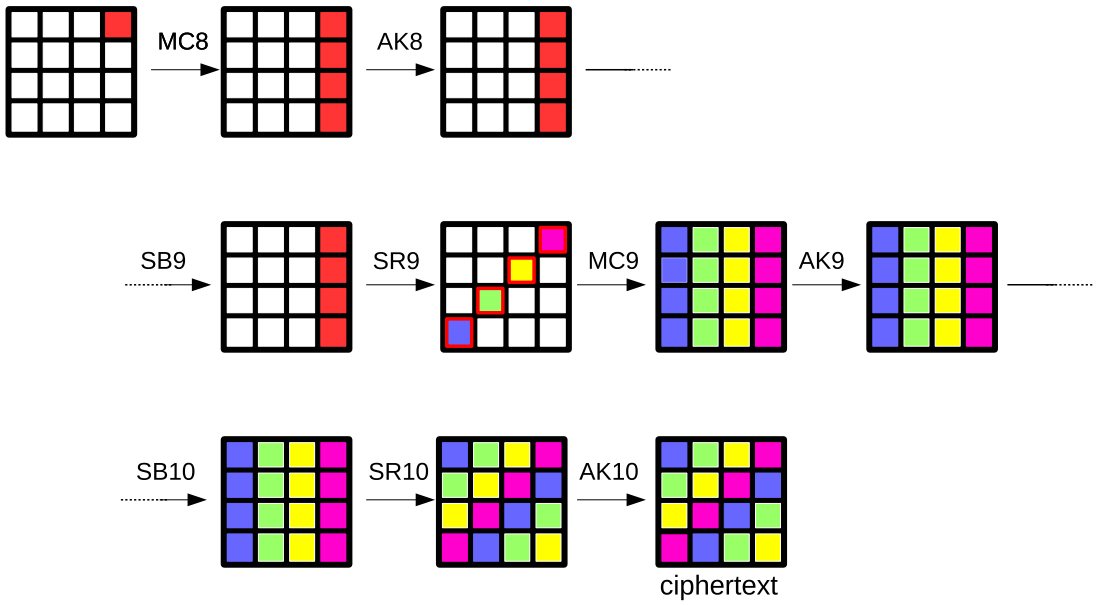

Figure 2.7: Error propagating pattern in AES

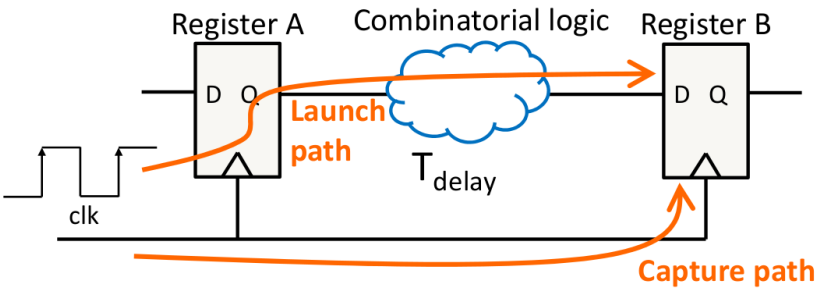

Figure 2.8: REG2REG data path [6]

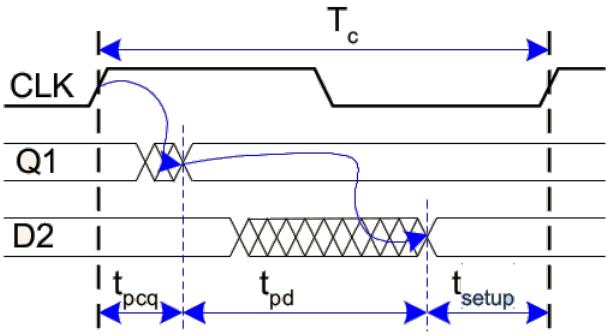

Figure 2.9: Setup timing constraint [19]

where data takes a path from a starting register to an ending register while passing through a combinatorial logic (shown in Fig.2.8). This situation is common in synchronous design and in this case the path is called a register-to-register path. From that, two main paths may be defined:

1. **The launch path**: this is the path taken by the data going from the data launching register A to data capturing register B at the clock edge.

2. **The capture path**: this is the path taken by the control clock signal from its source to the data capturing register B.

A basic property that must be fulfilled is that the propagation over the launch path must be faster that the propagation over the capture path. The propagation over the launch path is composed of two main delays: t_{pcq} and t_{pd}. The first is the delay needed after the clock edge to guarantee that the output Q of the register is stable. The second is the propagation delay of the combinatorial logic. Both are depicted in Fig.2.9. If the propagation over the capture path is considered as being ideal there is thus no delay over it and the property can be noted as:

$$t_{cycle} > t_{pcq} + t_{pd}$$

with t_{cycle} the clock period.

Additionally, another delay required by the capturing register needs to be considered in order to properly drive the data at the clock edge. This delay is known as the setup time and is depicted t_{setup}. It is the time during which the data at the input D of the capturing register needs to remain stable. This allows D-flip-flop registers to charge internal capacitive loads in order to have a stable signal at the clock edge. The setup time value is thus dependent of the technology used in the design and changes according to the libraries used by the Electronic Design Automation (EDA) tool. Including it, the property becomes:

$$t_{cycle} > t_{pcq} + t_{pd} + t_{setup}$$

In practice, more advanced models taking into account buffers and wiring delays are used. If the setup timing constraints as described above is not fulfilled, the value captured by the data capturing register B is said to be *metastable*. The value captured by the register in such a case stays in the metastable case during a short random period of time before setting to a random logical value. Finally, another value called the slack time is commonly used. It is defined by:

$$t_{slack} = t_{cycle} - (t_{pcq} + t_{pd} + t_{setup})$$

Based on the setup contraint time, one may see that a positive slack time is required in order to fulfill the constraint.

Chapter 3

Hardware-Trojan Targeting Block Cipher Implementations

Implementing efficient Trojans against block ciphers first requires to establish the context around these. In particular, the Trojan concept needs to be defined, as well a the threat menace related to its insertion. This section begins with a theoretical framework, formally defining the goals and properties of such Trojans. Based on that, it follows by presenting the insertion power of potential adversaries and continues by specifying the chip validating environment. Finally, it characterizes an additionnal protection layer, provided by the Trojan resilient framework proposed in [12].

3.1 Trojans Objectives And Properties

3.1.1 The Ideal Trojan Implementation Framework

As mentioned in Sec. 1, one may see the Trojan problematic as follows: malicious actors in the ICs manufacturing process can introduce a Trojan by bringing some discrete modifications to a design. Once put on the market, these actors can take advantage of the Trojan in order to obtain results that are not attended by the chip specifications, using a known trigger mechanism. In the case of block cipher, adversary aims to recover the secret key used for messages encryptions.

More formally, one may imagine a game, denoted as TRO, consisting in two phases: the *validation* and the *execution*. The game is involving an honest designer $\mathcal{D}$, an honest design and manufacturing process flow $\mathcal{M}_{gold}$, a design and manufacturing process flow $\mathcal{M}_{\mathcal{A}}$ involving an adversary $\mathcal{A}$ and a chip validator $\mathcal{V}$.

In addition, a *manipulation* is defined as a set of interactions made with the device. The interactions involve the digital inputs $\vec{x}$ provided to the device and the physical interactions $\vec{f}$. Formally, a manipulation is denoted by $\vec{m} = \left[\vec{x}, \vec{f}\right]$.

Moreover, a *view* $\mathbb{V}(D[\vec{m}])$ is defined as the observable behaviors set of a device D while running under the manipulation $\vec{m}$. The behavior set involves two observation types. First, the outputs $\vec{y}$ obtained from D running under $\vec{m}$. Second, the side-channel leakages $\vec{l}$ of D again running under $\vec{m}$. Formally we have $\left[\vec{y}, \vec{l}\right] \leftarrow \mathbb{V}(D[\vec{m}])$

The game takes place as shown by Alg.1.

Algorithm 1 TRO$(\mathcal{D}, \mathcal{M}_{gold}, \mathcal{M}_{\mathcal{A}}, \mathcal{V})$

procedure VALIDATION
  $\Gamma \leftarrow \mathcal{D}$
  $D_{gold} \leftarrow \mathcal{M}_{gold}(\Gamma)$
  $D \leftarrow \mathcal{M}_{\mathcal{A}}(\Gamma)$
  if $\mathcal{V}(D, D_{gold}, \Gamma) == 0$ **then return** 0
procedure EXECUTION
  $\vec{m}_1 \leftarrow \mathcal{A}$
  while $i = 1$ to ∞ **do**
    $\left[\vec{y}_1, \vec{l}_1\right] \leftarrow \mathbb{V}\left(D\left[\vec{m}_i\right]\right)$
    $\left[\vec{y}_2, \vec{l}_2\right] \leftarrow \mathbb{V}\left(D_{gold}\left[\vec{m}_i\right]\right)$
    $b_y \leftarrow (\vec{y}_1 \neq \vec{y}_2)$
    $b_l \leftarrow \mathrm{MI}\left(k, \vec{l}_1\right) \neq \mathrm{MI}\left(k, \vec{l}_2\right)$
    if $(b_y | b_l)$ **then return** 1
    $\vec{m}_{i+1} = \mathcal{A}(\vec{m}_i)$

First, $\mathcal{D}$ produces an honest circuit specification Γ. He sends then Γ to $\mathcal{M}_{gold}$ and $\mathcal{M}_{\mathcal{A}}$, producing respectively a honest chip D_{gold} and a malicious chip D supposely implementing the specification Γ.

Then, D must pass the validation process. If not, $\mathcal{A}$ looses the game. Here, D_{gold} is assumed to success in the validation since he is honest.

Once the validation succeeded, $\mathcal{A}$ can interact with D and the execution phase begins. The game is won by $\mathcal{A}$ if he manages to induce a difference between the behaviors of D_{gold} and D by using a specific manipulation $\vec{m}_i$. These can be defined as differences between the chips outputs or the Mutual Information MI between the secret key and the side-channel leakages of each chip.

By considering the game TRO$(\mathcal{D}, \mathcal{M}_{gold}, \mathcal{M}_{\mathcal{A}}, \mathcal{V})$, an adversary designs a Trojan taking into account the following objective:

$$\Pr\left[\mathrm{TRO}\left(\mathcal{D}, \mathcal{M}_{gold}, \mathcal{M}_{\mathcal{A}}, \mathcal{V}\right) = 1\right] \geq 1 - \epsilon(\zeta)$$

where ζ is the size of the field $\mathbb{M}$ such as $\vec{m} \in \mathbb{M}$ and ϵ is a negligible function for a growing ζ. Intuitively, it is decreasingly probable to trigger a Trojan using randomly chosen manipulation for an increasing number of manipulations. However, it is difficult to formally define the manipulations amount ζ while dealing with physical manipulations since these are continuous physical quantities. On the other hand, dealing with digital manipulations seems practical. For example, considering that the device input is over b bits, the size ζ of $\mathbb{M}$ equals 2^b.

3.1.2 Trojan Targetted Properties

An inspection of the game presented in Section 3.1.1 shows that an adversary has two main issues while designing Trojans. First, the infected chips D has to successfully pass the validation process. Second, once put on the market, the adversary must be able to make the Trojan deliver its malicious payload. These two conditions can be expressed as two properties, introduced in [16] and generalized here, that Trojan need to fulfill: the **stealthiness** and the **triggerability**.

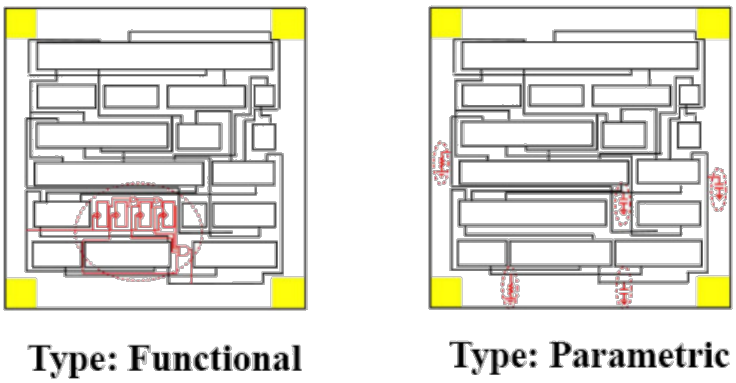

Figure 3.1: Functional and parametric modifications examples [35]

Stealthiness The stealthiness property assures that Trojans are able to pass the validation phase. It states that for randomly chosen manipulations, the probability that the Trojan is triggered is negligible. This is expressed as follows:

$$\Pr\left[\mathcal{V}(D, D_{gold}, \Gamma) = 0\right] \leq \epsilon(\zeta)$$

For example, a validator, executing functionality tests of a device D with input size of b bits, has after t test runs a probability to have found the cheat code given by $t/2^b$, which is negligible for a growing value b.

Triggerability The triggerability property assures that an adversary can trigger the Trojan when it desires once the infected chip is deployed on the market. It states that for a secret manipulation known by the adversary, the latter is able to lead the Trojan to carry out its malicious action with a probability close to the unity.

$$\Pr\left[(b_y|b_l) = 1\right] \approx 1$$

For example, considering a digitally triggered Trojan using a cheat code, an adversary knowing this cheat code as a probability of triggering the Trojan equals to 1.

3.2 Threat Model

Considering a circuit specification Γ, a design and manufacturing flow involving a malicious actor $\mathcal{A}$ gets the specification Γ and produces a device D, supposely implementing the honest functionality Γ. Since $\mathcal{A}$ is in the production flow, D may involves a Trojan.

Remembering Alg.1, a Trojan is a design modification resulting in the undesired behaviour of an electronic device when this one is in operation. Such a modification may appear anywhere in the IC design and manufacturing flow. Adversary $\mathcal{A}$ may use two strategies [35] in order to insert Trojans: **functional modifications** or **parametric modifications**. A representation for both mechanisms can be seen in Fig.3.1.

- **Functional modifications**: by using this mechanism, the Trojan is implemented through modifications of the design existing logic, by adding or removing logical gates

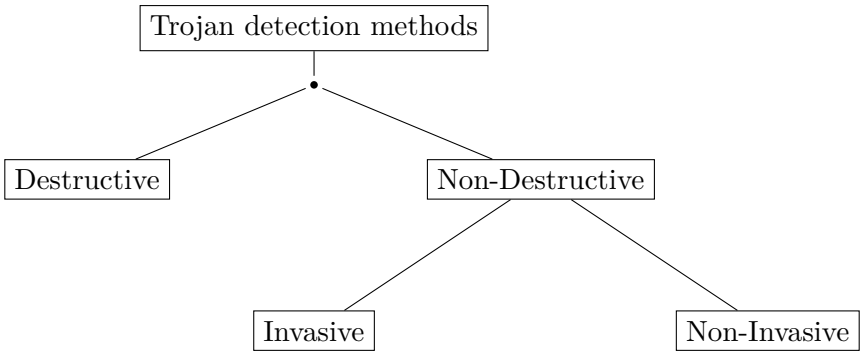

Figure 3.2: Detection methods taxonomy

- **Parametric modifications**: by using this mechanism, the Trojan is implemented through parameters modifications of the existing logic such as the sizing of components or wirings. The existing logic is thus kept unmodified.

Depending on its position in the design and manufacturing flow, the adversary has more or less power over potential design alterations [4],[5]. More into the details, the following cases are considered:

- **Design adversary**: this kind of adversary has access to the design files and sources code. For example it can be an insider or a hacker that compromises a computer system. He may add or remove design logic. This kind of adversary includes the malicious IP providers too.

- **Synthesis adversary**: this kind of adversary can compromise the EDA tool or the scripts runned by it in order to modify design or IP at any level from prepossessing the HDL to the netlist generation. The attacker may add and remove logic or modify constraints and libraries.

- **Place and Route adversary**: this kind of adversary can compromise the EDA tool or the scripts runned by it in order to modify design or IP at any level from prepossessing the netlist to the layout generation. The attacker may add and remove logic or modify constraints and libraries.

- **Fabrication adversary**: it is usually assumed that this kind of adversary is not taking part into the design or synthesis phase because of the reasons stated in Sec. 1. Foundries have access to the layout geometry and masks, they may thus add or remove components to every ICs through layout modification. Moreover, they can modify created ICs using Focus Ion Beams (FIB).

3.3 Validating Environment: Tests And Detection Mechanism

Once produced, the device D need to pass the validation process. The goal of the validator is to ensure that the device D produced by the malicious design and manufacturing flow is Trojan-free. In this purpose, he uses nowadays Trojan detection mechanisms, which are presented here. The detection methods are divided in two main categories (as shown in Fig.3.2): *destructive* and *non-destructive* [30],[5].

Destructive methods These methods aim to detect Trojans using reverse engineering methods. These usually takes place with a depackaging using Chemical-Mechanical Planarization (CMP) followed by a design analysis using Scanning Electron Microscope (SEM) or other visualization techniques [9],[30]. Such processes are known to be costly and time consuming. Indeed, current Very-Large-Scale Integration (VLSI) circuits may contain hundreds of thousands logic gates which implies a tremendous amount of time to verify a complete design. Another important issue is that the adversary $\mathcal{A}$ may insert Trojan randomly in chips. This implies that the detection result should not be based on a single chip analysis.

Non-destructive methods These methods aim to detect Trojans while preserving the chip. These methods can be *invasive* or *non-invasive*. The first ones modify the design at some points to embed features that helps in Trojan detection [18],[5]. The second ones leave the design unaltered. These are based on comparison between a suspicious chip with a honest one, known to be Trojan-free. First comparison method is based on Automatic Test Pattern Generation (ATPG). Using this method, the validator checks that the device D is implementing properly Γ. The output obtained from D using the input $\vec{x}$ is denoted $\vec{y} \leftarrow D(\vec{x})$. The validator runs tests checking if $\vec{y_i} = \Gamma(\vec{x_i})$ for $i \in [1, ..., t]$ and uniformly random chosen input $\vec{x_i}$. However, this method is limited because of the Trojan stealthiness property.

Another promising method is based on side-channel analysis and fingerprints [33], [21]. In comparison to an unaltered design, Trojan insertion may lead the infected chip to leak different information through side-channels such as power traces, EM emissions or sounds variations. Side-channel analysis aims to detect Trojan based on the leakage variations between an honest and a malicious device. A typical example is to verify that a malicious chip does not leak key information when running. This may be done using traditional Differential Power Analysis (DPA) using correlation [24] or MI [27] metrics between the key and the leakages of the malicious device D and a trusted one D_{gold}. Another example is based on comparison between side-channel fingerprints of trusted and malicious chip in order to reveal the presence of functional Trojans that are waiting for trigger [26]. Nevertheless, the variability of the chip properties appearing during the manufacturing process may be a bottleneck of such methods [9]. Because of this variability, small design logic and parametric modifications may pass through such validation mechanisms without being detected.

Formally, we denote the validation process based on the circuit specifications Γ, the honest device D_{gold} implementing Γ and validating the malicious device D as $\mathcal{V}(D, D_{gold}, \Gamma)$.

3.4 Countermeasure: Hardware Trojan-Resilience circuit via Testing Amplification

In practice, as in a vicious circle, one may not be ensured of the honesty of a golden device D_{gold}. Moreover, due to the difficulties met in the deployment of efficient Trojan detection mechanisms, validator may prefer validation processes based on specifications verification using ATPG methods. In this context, sensitive information leakage are thus not checked. In order to avoid the harmful actions of Trojans, one may move from detection mechanisms to protection mechanism. The goal here is no more to detect the Trojan before the deployment but to avoid its exploitation once deployed. In [12], authors propose a generic countermeasure against digitally triggered Trojans based on Multi-Party Computation (MPC) protocols using *testing amplification*.

The Trojan resilient framework proposed by [12] relies on two components: a *circuit transformation*

TR and a *tester* T.

1. **Circuit Transformation** The circuit transformation TR aims to compile a functionality described as the arithmetic circuit Γ into a protected specification involving a trusted *master circuit* M an a set of circuits $\Gamma_1, ..., \Gamma_\lambda$, called the *sub-circuits*. Put together, M and $\Gamma_1, ..., \Gamma_\lambda$ emulate Γ using a passively secure 3-party computation protocol. The master M is independent of $\Gamma_1, ..., \Gamma_\lambda$ and is assumed to be implemented in a trusted way and not by the malicious hardware manufacturer $\mathcal{A}$. Moreover, it involves a majority vote between the outputs of each sub-circuits.

2. **Tester** The tester T verifies in PPT if the device D_i produced by $\mathcal{A}$ correctly implement the functionality Γ_i. The uniformly random t_i tests made during the testing phase compare the input/output behavior of D_i regarding the honest specifications Γ_i.

The 3-party secure protocol protect against cheat codes Trojans while the random number of tests for each device protect against time-bombs. The majority vote amplifies the security bounds granted by the construction. Using this scheme, for at most n real runs after the testing phase of $t_i \leq t$, the authors prove that the probability that inserted Trojans deliver their payload during an execution phase does not exceed $\left(\frac{n}{t}\right)^{\lambda/2}$.

The following points show formally how the protection scheme proposed by [12] is working, what is its security model and how the security bound is computed. In a first time, the 3-party protocol is considered as a protocol that securely emulate a specification Γ, each party involved working with *shares* denoted as $(\vec{r}, \vec{s})$. Then, the formal specification of it can be found in Sec.3.4.7.

3.4.1 Threat Model

Considering a circuit functionality Γ a potential malicious manufacturer $\mathcal{A}$ get the specifications Γ and produces the device D that is supposed to implement the functionality Γ. The device D produces $\vec{y}$ based on its input $\vec{x}$. Since produced by $\mathcal{A}$, the device D can be infected by a Trojan. The security model involves digitally triggered Trojans and thus allows logic modification. Remember that these Trojans deliver their payload when a specific cheat-code is provided to the device or when a time bomb triggers.

3.4.2 Circuit Transformer

The goal of the circuit transformer (or compiler) specification Γ' securely emulating Γ. The Γ' specification is composed of two parts: a set of *sub-circuit* $\Gamma_1, ..., \Gamma_\lambda$ and the *master circuit* M (cfr Fig.3.3). The master circuit needs to be as small as possible in order to be ensured of its honesty more easily. In [12], authors proposed to use secure 3-party protocol in order to emulate Γ. One may thus see each Γ_i as a triplet of *mini-circuits* $[\Gamma_i^0, \Gamma_i^1, \Gamma_i^2]$. The role of M is to manage to communication between the sub-circuits $\Gamma_1, ..., \Gamma_\lambda$ and the user. A special communication protocol is added to $\Gamma_1, ..., \Gamma_\lambda$ in order to allow the mini-circuits of a same subcircuits to communicate with the others through the master circuit M. This is required in order to properly operate the 3-party protocol.

Formally, the compilation process is denoted by $\Gamma' \leftarrow \text{TR}\left(1^k, \lambda, \Gamma\right)$, where k is the computational security parameter of the Pseudo Random Number Generator (PRNG) required by the secure MPC protocol used and λ is the amount of independent subcircuits implementing a 3-parties

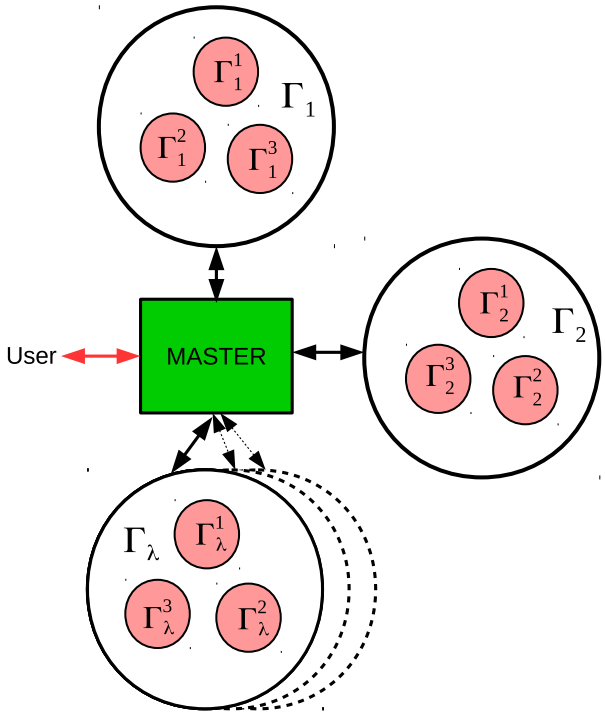

Figure 3.3: Γ' circuit obtained by the compiler

computation emulation of Γ.

3.4.3 Functionality Tester

Once the devices D_i produced, they are tested by the PPT tester in order to verify that they properly implement the specifications Γ_i. They are considered as black-boxes and the test unit can communicate with these, imitating the master M. Their output $\vec{y_i}$ are compared to the results produced by $\Gamma_i(\vec{x_i})$.

Formally, the testing phase is denoted $b \leftarrow \mathrm{T}^{D_1,\ldots,D_\lambda}\left(1^k, \Gamma\right)$ where T is the PPT tester and b a bit representing the output status of the testing phase. The tester T is called *t-bounded* if each of the D_i is tested at most t times.

3.4.4 Trojan Protection Scheme Security Framework

Considering the Trojan protection scheme $\Pi = (\mathrm{TR}, \mathrm{T})$, the security is modeled using the concept of *robustness*. The robustness is modeled by a game involving two phases, denoted by ROB_Π and presented in Alg.2.

The game begins by running the compiler TR to obtain the protected specification $\Gamma' = (M, [\Gamma_i])$ of the circuit functionality Γ. Then, Γ' is given to the malicious manufacturer $\mathcal{A}$ who produces a set of device D_i, each device implementing a functionality Γ_i.

Following that, the D_i are tested by the tester T.

If the testing phase succeeds, the devices D_i are deployed and the real execution begins. The adversary $\mathcal{A}$ can specify an input $\vec{x_i}$ and interact n times with the master M to obtain the

output $\vec{z_i} \leftarrow (M \Leftrightarrow D_1, ..., D_\lambda)(\vec{x_i})$. The adversary $\mathcal{A}$ wins ROB_Π if and only if, after a successful testing, he manages to produce an output $\vec{z_i}$ different from $\vec{y_i} \leftarrow \Gamma(\vec{x_i})$, which is the output of a correct computation.

Algorithm 2 $\mathrm{ROB}_\Pi(\mathcal{A}, \lambda, n, t, k, \Gamma)$

1: $(M, [\Gamma_i]) \leftarrow \mathrm{TR}\left(1^k, \lambda, \Gamma\right)$
2: $D_i \leftarrow \mathcal{A}\left(1^k, (M, [\Gamma_i])\right)$
3: **if** $\mathrm{T}^{D_1,...,D_\lambda}\left(1^k, (M, [\Gamma_i])\right) = \text{false}$ **then return** 0
4: $\vec{x_1} \leftarrow \mathcal{A}\left(1^k\right)$
5: **for** $i = 1$ to n **do**
6: $\quad \vec{z_i} \leftarrow (M \Leftrightarrow D_1, ..., D_\lambda)(\vec{x_i})$
7: $\quad \vec{y_i} \leftarrow \Gamma(\vec{x_i})$
8: $\quad$ **if** $\vec{y_i} \neq \vec{z_i}$ **then return** 1
9: $\quad \vec{x}_{i+1} \leftarrow \mathcal{A}\left(1^k, \vec{z_i}\right)$

Considering λ, n,t and k as natural parameters, the Trojan protection scheme $\Pi = (\mathrm{TR}, \mathrm{T})$ is (t,n,ϵ)-Trojan robust if:

1. The tester T is *t-bounded*
2. For any $\mathcal{A}$ and Γ we have
$$\Pr[\mathcal{A}, \lambda, n, t, k, \Gamma = 1] \leq \epsilon$$

Intuitively, the probability that an adversary $\mathcal{A}$ manages to make an inserted Trojan deliver its payload during a real execution is bounded by ϵ.

3.4.5 Parameters and Scheme Construction Intuitions

Some discussions can be made about value of n and t parameters regarding ϵ. By considering a device D, amongst the D_i designed by $\mathcal{A}$, such that it behaves correctly regarding the specifications Γ except with a probability ϵ. Moreover, Bad_i is define as the event according to which D behaves badly at the i^{th} run of its life (testing phase and real runs combined). One may assume that every Bad_i are independent or in another way that $\forall i, \Pr[\mathrm{Bad}_i] = \epsilon$. Beside, p is defined as the probability that a device D successfully passed the test and that an event Bad_i occurs during one of the n execution runs. In [12], the authors prove that the probability p is bounded as follows:

$$p \geq \frac{n}{e \cdot (n + t)} \tag{3.1}$$

Note that a more detailed explanation is shown in Appx.A.1. Intuitively, Eq.3.1 shows that $t >> n$ is needed in order to avoid device failure during the real execution. Moreover, in order to improve the bound depicted by Eq.3.1, an output processing located in the master circuit M is needed since any device D_i may output wrong results with a least probability p without it. In the proposed scheme, the output processing mechanism is a majority vote $\mathrm{MAJ}^{D_1,...,D_\lambda}$ between the output of the D_i.

However, even if the 3-party protocol combined with $\mathrm{MAJ}^{D_1,...,D_\lambda}$ gets rid of cheat code Trojan, additional measures need to be taken to avoid time-bombs. In this purpose, every device D_i is

tested a amount of time $t_i \leq t$ uniformly chosen at random in order to prevent any synchronization between them.

3.4.6 Security Bound Computation

Remember that the aim of the protection is to bound the probability that an adversary $\mathcal{A}$ wins the game $\mathrm{ROB}_{\Pi}(\mathcal{A}, \lambda, n, t, k, \Gamma)$ presented in Alg.2. In addition to the game, an additional majority vote $\mathrm{MAJ}^{D_1,...,D_\lambda}$ and random amount of testing $t_i \leq t$ for each device D_i are considered.

In order to win, $\mathcal{A}$ must be able to induce a Bad event in a device D_i during one of the n executions runs once the malicious chip has passed the test process. In [12] the authors have proved that this event has a probability $\Pr(\mathrm{Bad})_n$ bounded by:

$$\Pr(\mathrm{Bad})_n \leq \frac{n}{t}$$

Then, thanks to the majority vote $\mathrm{MAJ}^{D_1,...,D_\lambda}$, an attacker must in fact induce Bad events in $\lambda/2$ devices (or $\lambda/2+1$ depending on the parity of λ) simultaneously in order to induce a global Bad event at the output returned by the master. Because of the independence of the t_i, it holds:

$$\Pr(\mathrm{Bad})_n \leq \left(\frac{n}{t}\right)^{\lambda/2}$$

Note that a more detailed description of the security bound computation is given in Appx.A.2.

3.4.7 Multi-Party Computation Protocol Involving 3 Parties

As mentioned earlier, the protected specifications Γ' produced by the compiler emulates a specification Γ using a secure 3-party computation protocol proposed by [3]. The concept behind it is that the functionality $\Gamma(\vec{x})$ using the input $\vec{x}$ is computed by 3 different parties working jointly $[\Gamma_1, ..., \Gamma_3]$ while the input $\vec{x}$ is kept private from these parties. This avoids that a potential malicious party $\mathcal{A}$ gets the input value while keeping its functionality.

The privacy of the inputs is ensured using a *scrambling* mechanism, which intuitively makes the data appear random. The result of such a scrambling is called a *share*. The main trouble met by secure multi-party computation scheme is the keep correctness of operations while computing with shares. For this purpose, specific algorithms are used in order to implement basic operations. These algorithms may require communication of temporary shares between the parties in order to properly implement the targetted functionality. The main steps involved in the secure 3-party protocol are the *inputs sharing mechanism* in order to obtains scrambled inputs, the *functionality computation* done using different algorithms working on the shares and the *reconstruction mechanism* to recover the processed functionality output from the shares.

3.4.8 Input Sharing and Reconstruction Mechanisms

First, remember that the master circuit M is the interface between the user and the subcircuits $[D_1, ..., D_3]$ implementing the functionalities $[\Gamma_1, ..., \Gamma_3]$. Also, in this case, each D_i implements the functionality Γ_i using 3 minicircuits implementing $[\Gamma_i^1, ..., \Gamma_i^3]$. The master sends the inputs

v, w to the $[D_1, ..., D_3]$ and must thus be in charge of the inputs sharing to scramble the data before these are acquired by the malicious subcircuits.

The sharing mechanism is divided in two steps: the *random generation* followed by the *secret sharing*. The process is shown in Alg.3.

First, the minicircuits $[\Gamma_i^1, ..., \Gamma_i^3]$ obtain a random value $\alpha_l \in [\alpha_1, \alpha_2, \alpha_3]$. The α_l values are correlated such that $\alpha_1 \oplus \alpha_2 \oplus \alpha_3 = 0$. This operation is denoted GetCorrRandom. Once their random value obtained, the $[\Gamma_i^1, ..., \Gamma_i^3]$ send these to the master M that holds thus $[\alpha_1, \alpha_2, \alpha_3]$.

Second, the master scrambles the input v (called the *secret*) using the random values $[\alpha_1, \alpha_2, \alpha_3]$ to obtain $[x_1, x_2, x_3]$ where $x_l = v \oplus \alpha_l$. He then sends each x_l to a mini-circuit Γ_i^j with $l \neq j$ in order to avoid any mini-circuit from recovering the secret v. After that, each mini-circuit Γ_i^j holds a share $\tilde{v}_j$.

Algorithm 3 Sharing(v)

procedure STEP1 : RandomGeneration
    $\Gamma_i^1, \Gamma_i^2, \Gamma_i^3$ run GetCorrRandom
    Γ_i^1 sends α_1 to M
    Γ_i^2 sends α_2 to M
    Γ_i^3 sends α_3 to M
procedure STEP2 : SecretSharing(v)
    M sends $x_3 = v \oplus \alpha_3$ to Γ_i^1 that then holds $\tilde{v}_1 = (\alpha_1, x_3)$.
    M sends $x_1 = v \oplus \alpha_1$ to Γ_i^1 that then holds $\tilde{v}_2 = (\alpha_2, x_1)$.
    M sends $x_2 = v \oplus \alpha_2$ to Γ_i^2 that then holds $\tilde{v}_3 = (\alpha_3, x_2)$.

On the other hand, the reconstruction mechanism aims to retrieve the functionality output $\Gamma_i(v)$ from shares. In this case, the shares are designed such as the secret is recoverable from the shares of two mini-circuits out of three after being processed. Based on Alg.3, the algorithm for the reconstruction mechanism is presented in Alg.4.

First, the minicircuits $[\Gamma_i^1, ..., \Gamma_i^3]$ send their processed shares $[\tilde{v}'_1, ..., \tilde{v}'_3]$ to the master M.

Then, from these shares, the master computes three reconstructed values $[o_1, ...o_3]$ and checks if they are equal. If yes, he outputs one chosen arbitrarily. If not, he raises an error status.

Algorithm 4 Reconstruction

Γ_i^1 sends $\tilde{v}'_1 = (\alpha'_1, x'_3)$ to M
Γ_i^2 sends $\tilde{v}'_2 = (\alpha'_2, x'_1)$ to M
Γ_i^3 sends $\tilde{v}'_3 = (\alpha'_3, x'_2)$ to M
M computes $o_1 = \tilde{v}'_1[0] \oplus \tilde{v}'_2[1] = \alpha'_1 \oplus x'_1$
M computes $o_2 = \tilde{v}'_2[0] \oplus \tilde{v}'_3[1] = \alpha'_2 \oplus x'_2$
M computes $o_3 = \tilde{v}'_3[0] \oplus \tilde{v}'_1[1] = \alpha'_3 \oplus x'_3$
if $(o_1 == o_2 == o_3)$ **then return** o_1
elseRaises an error.

For example, considering a value v data has just been shared using Sharing(v), one may reconstruct the secret v from the shares as shown here:

$$\begin{aligned} o_1 &= \alpha'_1 \oplus x'_1 = \alpha_1 \oplus x_1 = \alpha_1 \oplus v \oplus \alpha_1 = v \\ o_2 &= \alpha'_2 \oplus x'_2 = \alpha_2 \oplus x_2 = \alpha_2 \oplus v \oplus \alpha_2 = v \\ o_3 &= \alpha'_3 \oplus x'_3 = \alpha_3 \oplus x_3 = \alpha_3 \oplus v \oplus \alpha_3 = v \end{aligned}$$

Since all the o_i are similar, the master M returns $o_1 = v$.

In order to keep correctness, functionality implementation must thus operate over the shares such that their output stay be recoverable using Alg.4.

3.4.9 Field Addition Functionality Implementation Using Shares

The operation presented in this section is the addition in $GF(2^8)$ using shared inputs. Remembering the Section 2.2, this operation is used to have key dependency in block ciphers such as AES. The algorithm used to perform it using a secure 3-party computation is presented in Alg.5.

First, the inputs v, w are shared using the SHARING algorithm. Note that this step is considered here as an implicit operation: the algorithm must be able to work on shares already kept by the mini-circuits from previous operations.

Second, each mini-circuits Γ_i obtains a share of the addition result $\tilde{a}_i$ by computing $\tilde{a}_i = \tilde{v}_i \oplus \tilde{w}_i$.

It remains to verify the correctness of the solution, which is actually: using the RECONSTRUCTION algorithm for the shares $[\tilde{a}_1, \tilde{a}_2, \tilde{a}_3]$, it holds:

$$o_1 = (x_1 \oplus y_1) \oplus (\alpha_1 \oplus \beta_1) = (v \oplus \alpha_1 \oplus w \oplus \beta_1) \oplus (\alpha_1 \oplus \beta_1) = v \oplus w$$
$$o_2 = (x_2 \oplus y_2) \oplus (\alpha_2 \oplus \beta_2) = (v \oplus \alpha_2 \oplus w \oplus \beta_2) \oplus (\alpha_2 \oplus \beta_2) = v \oplus w$$
$$o_3 = (x_3 \oplus y_3) \oplus (\alpha_3 \oplus \beta_3) = (v \oplus \alpha_3 \oplus w \oplus \beta_3) \oplus (\alpha_3 \oplus \beta_3) = v \oplus w$$

Since all the o_i are similar, the master M returns $o_1 = v \oplus w$.

Algorithm 5 ADDITION

1: $\rightarrow \Gamma_i^1$ holds $\tilde{v}_1 = (\alpha_1, x_3)$ and $\tilde{w}_1 = (\beta_1, y_3)$
2: $\rightarrow \Gamma_i^2$ holds $\tilde{v}_2 = (\alpha_2, x_1)$ and $\tilde{w}_2 = (\beta_2, y_1)$
3: $\rightarrow \Gamma_i^3$ holds $\tilde{v}_3 = (\alpha_3, x_2)$ and $\tilde{w}_3 = (\beta_3, y_2)$
4: Γ_1 computes $\tilde{a}_1 = (\alpha_1 \oplus \beta_1, x_3 \oplus y_3)$ and holds it
5: Γ_2 computes $\tilde{a}_2 = (\alpha_2 \oplus \beta_2, x_1 \oplus y_1)$ and holds it
6: Γ_3 computes $\tilde{a}_3 = (\alpha_3 \oplus \beta_3, x_2 \oplus y_2)$ and holds it

Other algorithms exist in order to implement the operations needed for example to perform the AES block cipher using a 3-party protocol. These are out of the scope of the work and are thus not presented here.

Chapter 4

First Trojan: Differential Fault Analysis Against AES Using Timing Violation

As seen in Sec. 2.3, the DFA is a powerful tool that may be used in order to break the AES block cipher security, by recovering the secret key used for the encryption of sensitive data. However, this kind of attack requires that an adversary introduces faults at some precise positions during the encryption process. Remembering Sec. 2.9, some faults may appears in synchronous systems if the setup constraint over data propagation paths are not met. Based on that, one may take advantage of this property in order to induce an error for a specific path in a system.

In this chapter, a Trojan architecture aiming to introduce faults in an AES hardware design is described. Induced error are attended to be used for performing a DFA, leading to the key recovery. It begins with the threat model description, then moves to the targetted design architecture. It continues with the description of the Trojan implementation, theoretically in ASICs and practically in FPGAs, and with the key recovery method used. Finally, some discussions related to the detection and possible improvement are made.

4.1 Threat model

In this first case, one may consider a high-level circuit description Γ implementing the AES block cipher. The Γ is designed by an honest designer $\mathcal{D}$ and is sent to a design and manufacturing flow $\mathcal{M}_{\mathcal{A}}$ involving a malicious actor $\mathcal{A}$. Here, $\mathcal{A}$ may alter the design from the synthesis step to the manufacturing step included. As seen in Sec. 3.2, this let to the adversary the possibility to include a functional or parametric Trojan. The device D produced by $\mathcal{M}_{\mathcal{A}}$ is then sent back to $\mathcal{D}$ and is supposely implementing Γ. Once the infected device deployed, the adversary is considered to have a physical access to it in order to take advantage of the inserted Trojan.

4.2 Targetted Architecture Description

The targetted design is the AES block cipher with key size of 128 bits. As said in Sec. 2.3, this block cipher has been adopted by the NIST and is used a lot worldwide. A common example

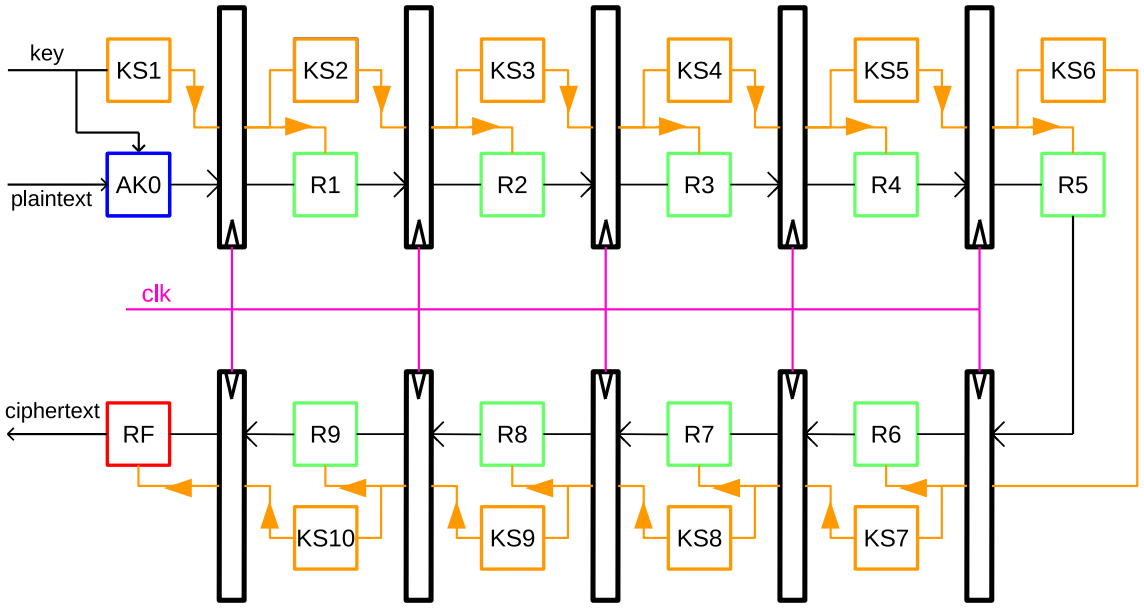

Figure 4.1: AES round pipelined

is its use in the Wi-Fi Protected Access II (WPA2) protocol in order to secure data over a wireless network. More specifically, a round level pipelined architecture of the AES block cipher is considered. The round level pipelined is an architecture where the pipelining is done between each main rounds of the AES, as shown in Figure 4.1. This means that each pipelined level involves a logic for the `AddRoundKey`,`SubBytes`, and `MixColumns` operations. The `ShiftRows` is in fact not considered as a logic in itself since it is a routing operation.

This specific architecture is chosen because of the difficulties encountered by the implementation of the proposed Trojan in loop and full-pipelined AES architecture. Indeed, remembering Sec. 2.3, the fault needs to appear at very specific location in the AES block cipher in order to be useful. Using only logic for one round with controls and looping data, the loop architecture makes the fault unusable because of the confusion-diffusion paradigm. On the other hand, since the Trojan is implemented using timing violations, a sufficient logic depth is necessary, making the implementation with full-pipelined architecture difficult.

4.3 Fault Introduction

As said earlier, specific fault introduction is needed in order to perform properly a DFA against AES. The following points present a top-down approach describing a Trojan architecture inducing such exploitable errors.

4.3.1 Fault Insertion in AES

First, one may define the location of the fault insertion. As explained in Sec.2.3, powerful attacks take place by taking advantage of the `MixColumns` operation since it propagates defined difference patterns between a correct/faulty ciphertexts pair $\left[C, \tilde{C}\right]$. The most powerful ones takes advantage of faults insertions before the `MixColumns` operation of the 8^{th} round. The corresponding key recovery methods require algorithms with $O\left(2^{32}\right)$ or $O\left(4 \cdot 2^{8}\right)$ complexity,

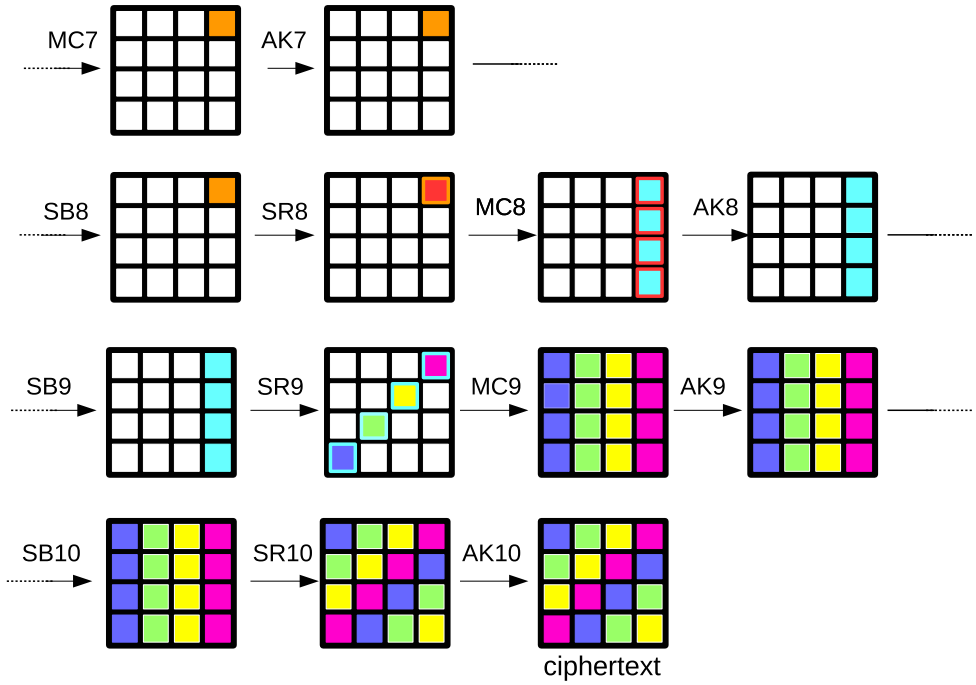

Figure 4.2: Targeted bytes

depending on the number of error used to perform the DFA. Moreover, the fault model is the *Random byte error* model, allowing to use random fault value insertion methods which are less restrictive and so more practical.

In addition, efficient DFA using faulty byte before MC_8 can take place if and only if there is at most one faulty byte before MC_8. In order to ensure the unicity, a specific byte $b_{t,MC8}$ is targeted, the 4^{th} byte of the processed encryption state (which is depicted in red in Figure 4.2). Remember that the faulty $b_{t,MC8}$ appearing before MC_8 may come from a faulty byte appearing before SR_8, SB_8 or AK_7 denoted $[b_{t,SR8}; b_{t,SB8}; b_{t,AK7}]$ (which are depicted in orange in Figure 4.2).

4.3.2 Trojan Implementation Principle

Now that the possible targets b_t are defined, one may specify the fault introduction method. As mentioned in Sec. 3.2, adversary may use two mechanisms in order to introduce a Trojan: modifying the original design logic using functional modifications or modifying circuitry parameters using parametric modifications.

The proposed Trojan implementation consists in a delay insertion via parametric modifications of the design. The additional delay leads the targetted path to fail because of setup timing constraint violation. This failure may be induced by the adversary $\mathcal{A}$ by increasing the clock frequency f_{clk} above the maximal frequency f_{trig} that the targeted path can tolerate while still meeting the setup constraint. In order to be sure that the fault induced is the only error appearing in the system when $f_{clk} \geq f_{trig}$, the adversary $\mathcal{A}$ must add delay properly. More specifically, this has to be done such that the targetted path become the worst critical path of the design with a significant negative slack time in comparison to the other critical paths for which the latter needs to stay positive.

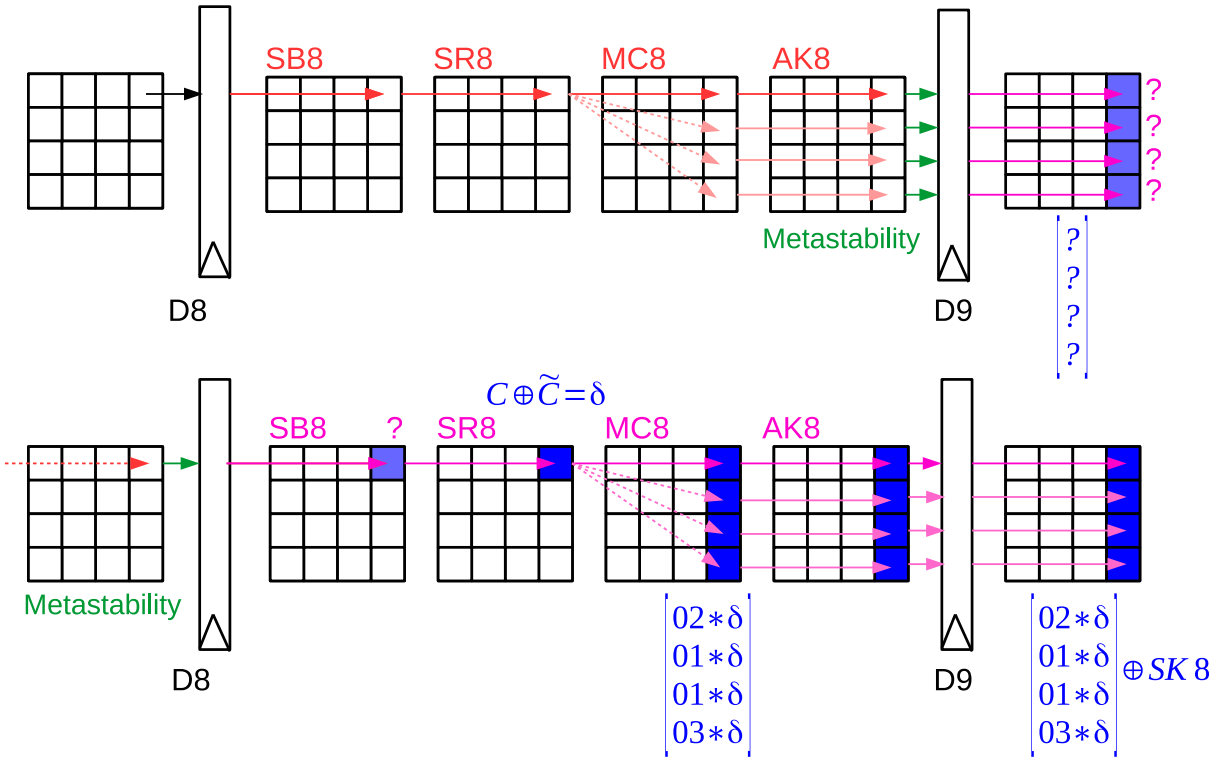

Figure 4.3: Metastability analysis for Trojan insertion location choice

Before going into the implementation details, one may specify which byte to attack amongst the $[b_{t,MC8}; b_{t,SR8}; b_{t,SB8}; b_{t,AK7}]$. Indeed, now that the Trojan implementation method is defined, some of the b_t are not valid anymore, as described below. In order to properly perform the DFA, the differences appearing after the `MixColumns` operations between C and $\tilde{C}$ need to strictly respect a theoretical patterns presented in Sec.2.3. Because of metastability, delay introduced over any path in the 8^{th} round may lead to random byte error after the MC_8 operation once the data is captured (as seen in Fig.4.3). The randomness may thus induce the difference patterns to not be respected anymore, making the DFA impossible. The random byte error must thus be stabilized before entering into the MC_8 operation. In this purpose, the proposed solution introduces delay in the 7^{th} round, targeting the byte $b_{t,SB8}$.

Remembering that Trojan need to be triggerable, one may encounter some limitations by trying to induce fault while targeting an individual bit. Since the fault is induced at the 7^{th} round, the logic values transitions appearing at the encryption state can be considered as random, thanks to the diffusion-confusion paradigm. Because of this, transition at the targetted bit is not ensured and fault apparition neither. Besides, in the case of a transition, the random nature of the metastability induced by timing violation may also prevent fault apparition. Based on the previous discussions, the proposed solution is targeting the 8bits of the targetted byte in order to increase the probability to have a fault at each clock cycle when $f_{clk} \geq f_{trig}$ holds.

4.3.3 Theoretical Trojan Implementation in ASIC

In ASIC, delay insertion can be made at the sub-transistor level by slightly modify parameters of some transistors . A delay faults is usually induced by consecutive opposite data that are modifying the state of the gates capacitance by charging or discharging them. The objective of the modification is thus to change the charging or discharging speed of the gates capacitance in

Index	Sources	Ends	Original delay	Modified delay
1	$D_{7,out}[31:24]$	$SB_{7,in}[31:24]$	$\approx$ 1.8ns	$\approx$ 3.5ns
2	$SB_{7,out}[31:24]$	$MC_{7,in}[31:0]$	$\approx$ 1ns	$\approx$ 2.6ns
3	$MC_{7,out}[31:24]$	$AK_{7,in}[31:24]$	$\approx$ 1.8ns	$\approx$ 7.5ns
4	$AK_{7,out}[31:24]$	$D_{8,in}[31:24]$	$\approx$ 1.8ns	$\approx$ 7.6ns

Table 4.1: LUT Placement modifications

order to not meet anymore the setup constraint on the targetted path. To do so, three methods may be used:

1. **Increase the threshold voltage**: Usually, transistor are driven using a constant voltage. By increasing their threshold voltage, the the current flowing through the transistors is reduced, increasing thus the time needed for their gate capacities to be charged. Generally, technologies using low threshold voltage are fast while those using high threshold voltage are slow. Since mixing threshold voltage technologies are common in ICs design, an adversary may thus use high threshold technology for the targetted paths.

2. **Decrease the transistor width**: The current flowing through a transistor is directly proportional to the width of the latter. Reducing it leads thus to a reduction of the current flowing through the transistor, increasing thus the time needed for its gate capacitance to be charged.

3. **Increase the transistor length**: The current flowing through a transistor is inversely proportional to the width of the latter. Increasing it leads thus to a reduction of the current flowing through the transistor, increasing thus the time needed for its gate capacitance to be charged.

4.3.4 Practical Trojan Implementation in Virtex VI FPGA

The proposed Trojan was implemented using the ML605 Virtex VI Evaluating Board from Xilinx. Delay insertion in FPGA is quite different compared to ASICs since the combinatorial logic is implemented using Look-Up Tables (LUTs). The LUTs available in the Virtex VI have fixed delay that cannot be changed by user design parameters. Another methodology is needed to introduce delay on specific paths.

In this case, delay is introduced by modifying the post place and route design. As said in the Sec.4.3.2, the error needs to be introduced on paths in the 7^{th} round. In this purpose, the placement of specific LUTs as well as the signals routing related to them are manually modified.

More into the details, one need to find the LUTs introducing delay over the paths computing the byte $b_{t,SB8}$ value while minimizing the impact on the other paths. The post place and route layout inspection reveals that each round is composed of 4 LUT ledgers: the round inputs registers, the `SubBytes/ShiftRows` LUTs, the `MixColumns` LUTs and the `AddRoundKey` LUTs. The `MixColumns` LUTs are not good candidates since they have influence over a whole column, in place of the targetted byte taken individually. Instead, the placements of the 13 LUTs computing `SubBytes/ShiftRows` and `AddRoundKey` operations leading the byte value of $b_{t,SB8}$ are modified. The delay modifications induced by these modifications are shown in Table 4.1. Moreover, as example, the visualization of the path for the 31^{th} bit along the 7^{th} round can be seen in Figs. 4.4,4.5,4.6 and 4.7.

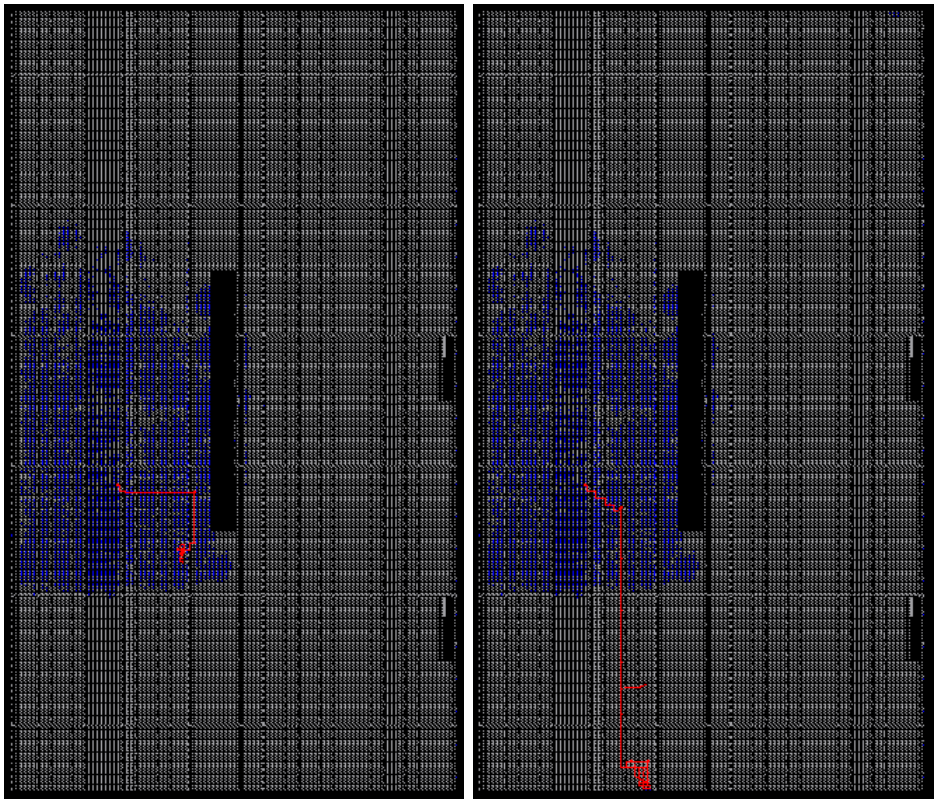

Figure 4.4: Original and modified path 1 in Table4.1 for the $31^{th} bit$

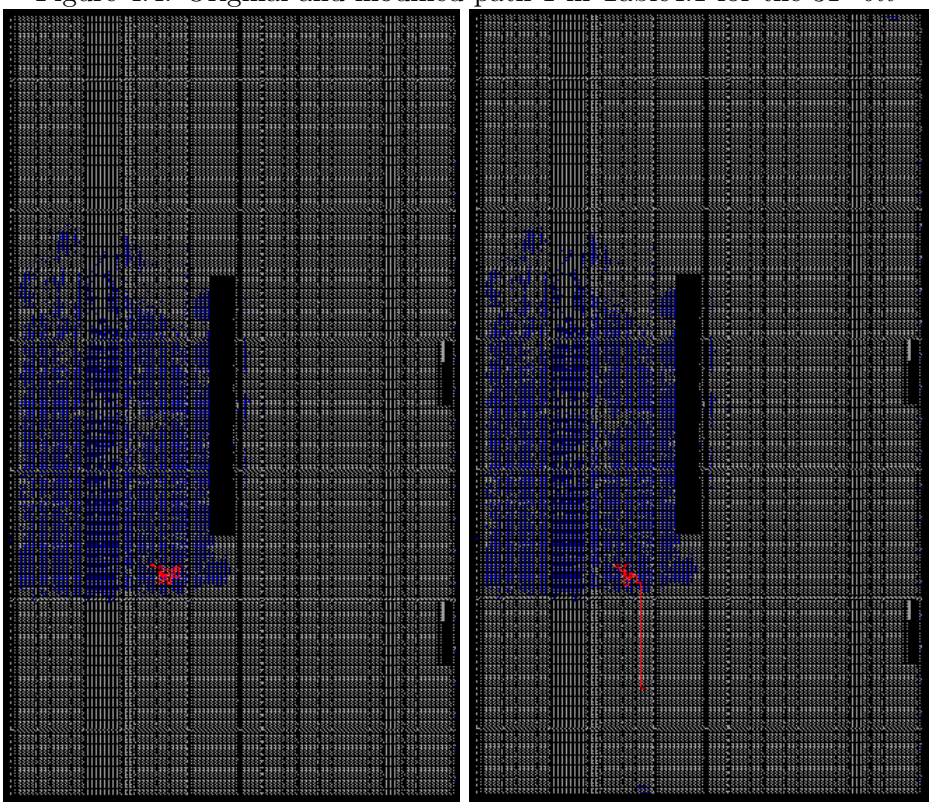

Figure 4.5: Original and modified path 2 in Table4.1 for the $31^{th} bit$

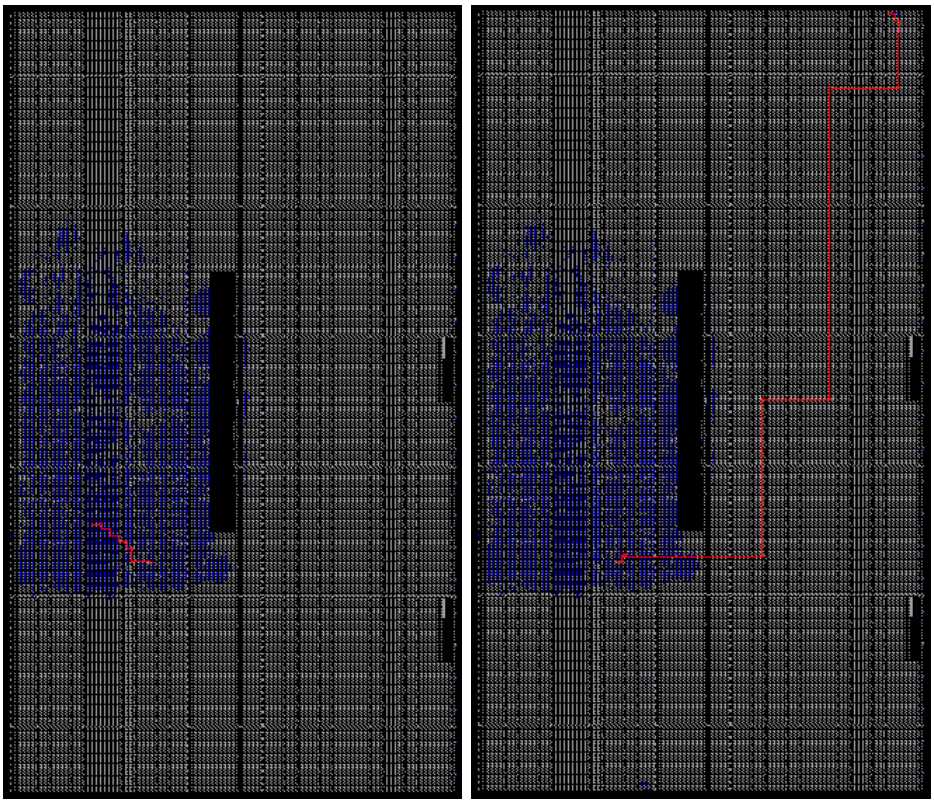

Figure 4.6: Original and modified path 3 in Table4.1 for the $31^{th} bit$

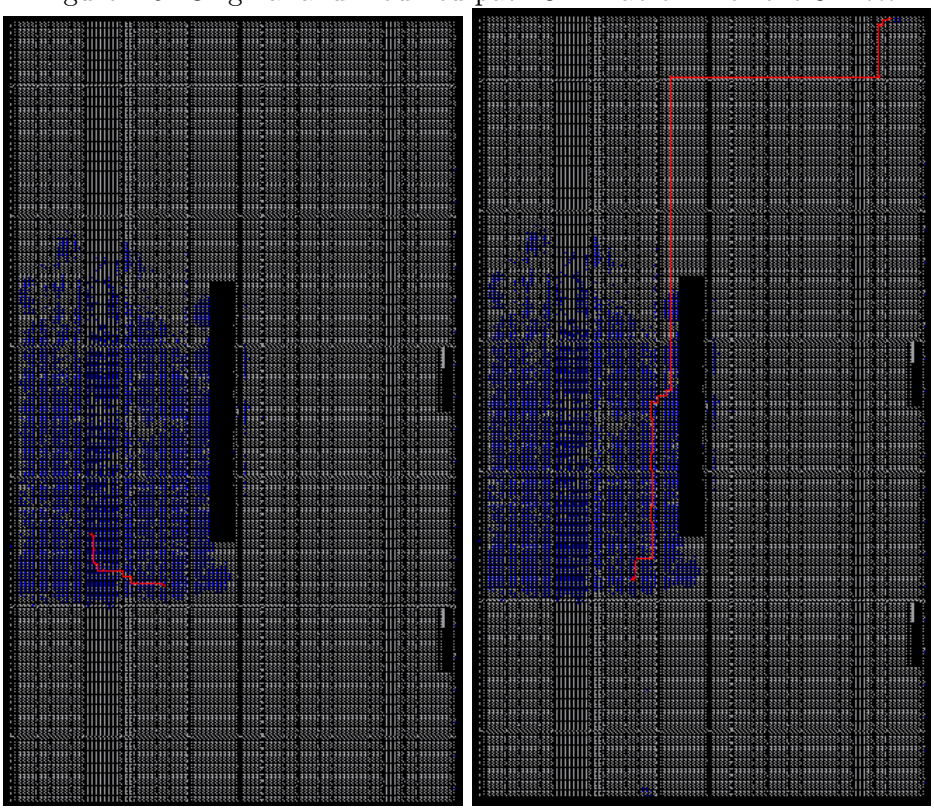

Figure 4.7: Original and modified path 4 in Table4.1 for the $31^{th} bit$

Sources	Ends	Modified delay	Modified Slack time
$D_{7,out}[31:24]$	$D_{8,in}<26>$	≈ 23.3ns	≈ -13.3ns
$D_{7,out}[31:24]$	$D_{8,in}<25>$	≈ 23ns	≈ -13ns
$D_{7,out}[31:24]$	$D_{8,in}<28>$	≈ 23ns	≈ -13ns
$D_{7,out}[31:24]$	$D_{8,in}<29>$	≈ 23ns	≈ -13ns
$D_{7,out}[31:24]$	$D_{8,in}<31>$	≈ 22.5ns	≈ -12.5ns
$D_{7,out}[31:24]$	$D_{8,in}<27>$	≈ 22.5ns	≈ -12.5ns
$D_{7,out}[31:24]$	$D_{8,in}<30>$	≈ 22.5ns	≈ -12.5ns
$D_{7,out}[31:24]$	$D_{8,in}<24>$	≈ 21ns	≈ -11ns

Table 4.2: Delay impacts of LUT placement modifications

The original design is synthesized using the fastest clock frequency f_{max} constraint that does not result in a slack time violation. In this case, $f_{max} = 100$MHz (or $T_{min} = 10$ns) and the resulting slack time for the worst critical path equals 0.975ns. Introducing the delay as done above meets the goal stated in Sec.4.3.2: the targetted byte paths are the only ones to have a significant timing violation. More into the details, the different slack times for the targetted paths are shown in Table.4.2. Note that the following critical path has a slack time of 0.5ns.

Based on the slack times obtained, the frequency f_{trig} at which errors begin to occur can be estimated at $f_{trig} \approx 43$MHz. Distribution of the appearing error Hamming weight in function of the clock frequency f_{clk} are shown in Fig.4.8. These distributions were built using 5000 random input vectors each. Based on these, one may observe that in practice, the trigger frequency is observed to be at 65MHz, which is higher that the prediction. This may be explained by the following phenomenon. Timing analysis is based on models using components properties approximations. In order to avoid errors caused by these approximations, models designers may have considered a margin of error, leading to stricter theoretical timing constraints which distorts the approximation. Starting from 65MHz, the distribution slightly evolves to eventually stabilizes into a Gaussian distribution of mean 4, which corresponds to the random byte error model targetted by the Trojan. According to the measured distributions, using random inputs, an attacker is assured to have an error appearing more than 99% of the time with a clock frequency bigger than 80MHz. Note that in practice, there is always a probability that no error appears given by $\frac{1}{2^8}$, which leads to a theoretical maximal bound of 99.6% of fault appearance.

4.4 Key recovery

In order to recover the key, the two DFA based on fault introduction before in MC_8 described in Sec.2.3 are implemented in C. In both cases, the random fault introduced before MC_8 by the Trojan induces 4 random byte errors before MC_9. Since the attacker knows the precise location of the fault before MC_8, only 1 difference set is used to find candidates for each key parts $\mathcal{P}_i$. The last round key is represented by:

$$\begin{bmatrix} RK_{15} & RK_{11} & RK_7 & RK_3 \\ RK_{14} & RK_{10} & RK_6 & RK_2 \\ RK_{13} & RK_9 & RK_5 & RK_1 \\ RK_{12} & RK_8 & RK_4 & RK_0 \end{bmatrix}$$

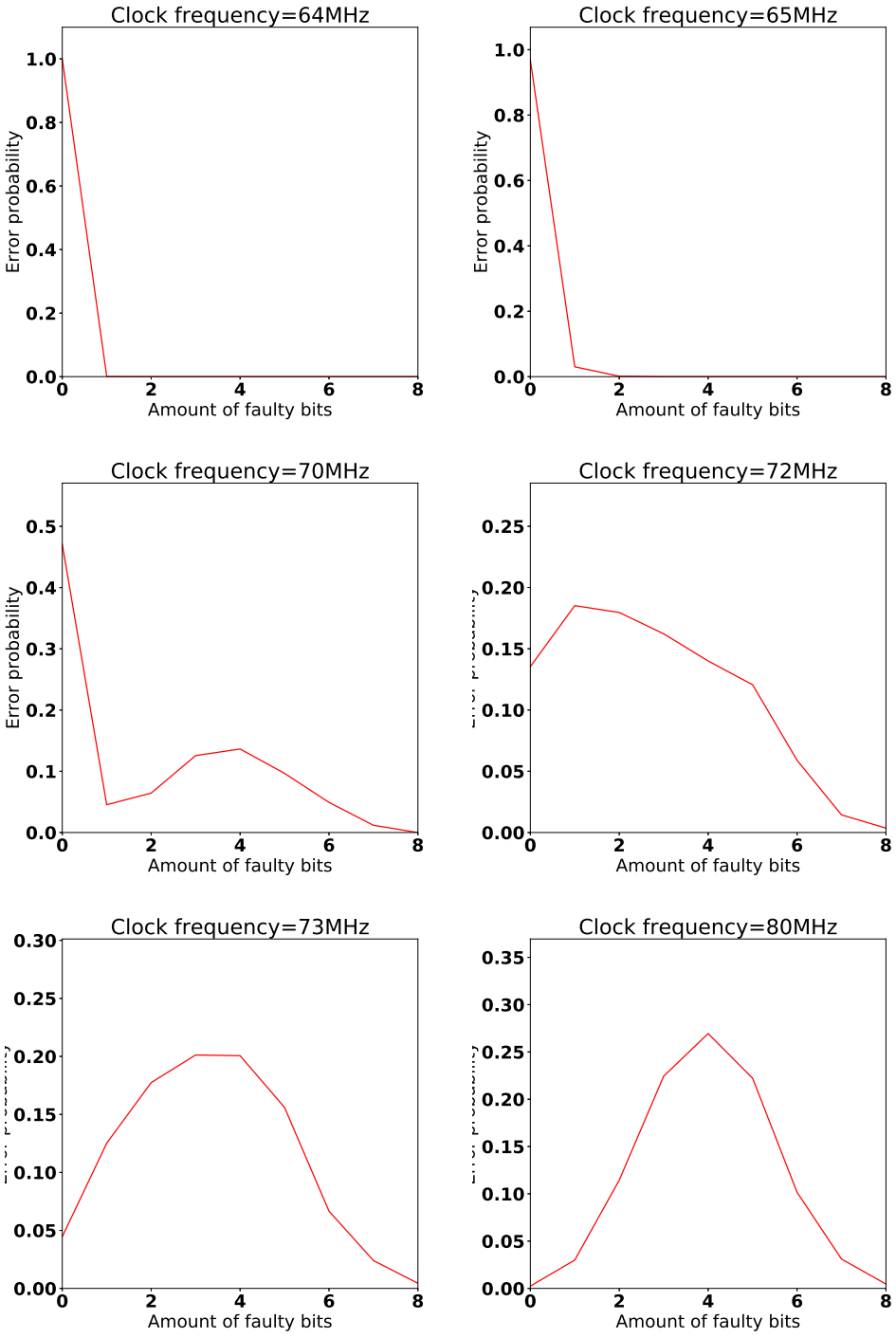

Figure 4.8: Fault Hamming weight probability density distribution in function of the clock frequency

Based on the propagation error pattern and considering a faulty byte in the i^{th} column before MC_9 , the following 32bits key parts $\mathcal{P}_i$ are targeted:

$$\begin{aligned}\mathcal{P}_3 &= [RK_3, RK_6, RK_9, RK_{12}]\\ \mathcal{P}_2 &= [RK_7, RK_{10}, RK_{13}, RK_0]\\ \mathcal{P}_1 &= [RK_{11}, RK_{14}, RK_1, RK_4]\\ \mathcal{P}_0 &= [RK_{15}, RK_2, RK_5, RK_8]\end{aligned}$$

DFA against AES using 2 correct/faulty ciphertexts pairs The first DFA requires two correct/faulty ciphertexts pairs $\left[C_1, \tilde{C}_1\right]$ and $\left[C_2, \tilde{C}_2\right]$, obtained by the encryptions of 2 different plaintexts p_1 and p_2 using the same unknown key k. The algorithm implemented is presented in Alg.6.

The attack follows a divide and conquer strategy, recovering 32bits key parts one by one. For each parts $\mathcal{P}_i$, the following steps are performed.

First, the difference set Ω for each initial error δ is created based on the difference pattern θ_i.

Considering the 32bits key parts $\mathcal{P}_i = [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \mathcal{P}_{i,3}]$ (where $\mathcal{P}_{i,*} \in [0, 255]$), a first guess $\tilde{\mathcal{P}}_i = [\mathcal{P}_{i,0,a}, 0, 0, 0]$ is taken. The last round R_{10} is reversed using $\tilde{\mathcal{P}}_i$ for $\left[C_1, \tilde{C}_1\right]$ in order to obtain the corresponding internal states t_0 and $\tilde{t}_0$. If the difference between t_0 and $\tilde{t}_0$ does not corresponds to a difference in Ω, then another value of $\tilde{\mathcal{P}}_i = \left[\mathcal{P}'_{i,0}, 0, 0, 0\right]$ is tried. On the other case, a second byte is added in order to obtain $\tilde{\mathcal{P}}_i = [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, 0, 0]$ and the last round R_{10} is reversed for $\left[C_1, \tilde{C}_1\right]$ using this new value. Similarly, if the difference between the corresponding internal states is not involved in Ω then another value $\tilde{\mathcal{P}}_i = \left[\mathcal{P}_{i,0}, \mathcal{P}'_{i,1}, 0, 0\right]$ is tested. The algorithm continues to add one byte at a time when the key parts value used leads to a match between internal states difference and Ω. When guessing for a 32bits value $\tilde{\mathcal{P}}_i = [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \mathcal{P}_{i,3}]$, reversing the last round is done for both correct/faulty ciphertexts pairs. If the resulting differences of internal states match with Ω for each pair, then $\tilde{\mathcal{P}}_i$ is added to the $\mathcal{P}_i$ key part candidates set $\mathcal{L}_i$.

Once this filtering part done, a unique remaining key part value $\tilde{\mathcal{P}}_i = [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \mathcal{P}_{i,3}]$ is expected. The whole key is obtained by recombining the different value obtained for each $\mathcal{P}_i$.

Every key recovery attempts performed by the proposed solution implementation succeeded. Using an Intel(R) Core(TM) i7-3630QM (2.4GHz) CPU, the attack approximated duration is 0.5s to recover the full 128 bits key.

DFA against AES using 1 correct/faulty ciphertexts pair The second DFA requires one unique correct/faulty ciphertexts pair $\left[C_1, \tilde{C}_1\right]$, obtained by the encryption of a known plaintext p using an unknown key k. The algorithm implemented is presented in 7.

As the first one, the attack follows a divide and conquer strategy. It involves three main phases: two filtering phases and a exhaustive search.

The first filtering phase, describe in Alg.8, is based on the same principle that the attack developed for 2 correct/faulty pairs. The only difference occurs when a 32bits key part value $\tilde{\mathcal{P}}_i = [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \mathcal{P}_{i,3}]$ is guessed. Previously, this key part candidate was kept iff internal

Algorithm 6 DFA Using 2 Faults

1: **procedure** KeyRecovery($C_1, C_2, \tilde{C_1}, \tilde{C_2}$)
2: $\mathcal{L} = [\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3]$
3: $\theta = [\theta_0, \theta_1, \theta_2, \theta_3]$
4: **for** $i \in [0 \rightarrow 3]$ **do**
5: $\Omega = \emptyset$
6: $\mathcal{L}_i = \emptyset$
7: **for** $\delta \in [1 \rightarrow 255]$ **do**
8: $\Omega = \Omega \cup (\delta \times \theta_i)$
9: **for** $\mathcal{P}_{i,0} \in [1 \rightarrow 255]$ **do**
10: $t_0 \leftarrow {\mathrm{R}_{10}}^{-1}(C_1, [\mathcal{P}_{i,0}, 0, 0, 0])$
11: $\tilde{t_0} \leftarrow {\mathrm{R}_{10}}^{-1}\left(\tilde{C_1}, [\mathcal{P}_{i,0}, 0, 0, 0]\right)$
12: **if** $(t_0 \oplus \tilde{t_0}) \in \Omega$ **then**
13: **for** $\mathcal{P}_{i,1} \in [1 \rightarrow 255]$ **do**
14: $t_1 \leftarrow {\mathrm{R}_{10}}^{-1}(C_1, [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, 0, 0])$
15: $\tilde{t_1} \leftarrow {\mathrm{R}_{10}}^{-1}\left(\tilde{C_1}, [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, 0, 0]\right)$
16: **if** $(t_1 \oplus \tilde{t_1}) \in \Omega$ **then**
17: **for** $\mathcal{P}_{i,2} \in [1 \rightarrow 255]$ **do**
18: $t_2 \leftarrow {\mathrm{R}_{10}}^{-1}(C_1, [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, 0])$
19: $\tilde{t_2} \leftarrow {\mathrm{R}_{10}}^{-1}\left(\tilde{C_1}, [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, 0]\right)$
20: **if** $(t_2 \oplus \tilde{t_2}) \in \Omega$ **then**
21: **for** $\mathcal{P}_{i,3} \in [1 \rightarrow 255]$ **do**
22: $t_3 \leftarrow {\mathrm{R}_{10}}^{-1}(C_1, [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \mathcal{P}_{i,3}])$
23: $\tilde{t_3} \leftarrow {\mathrm{R}_{10}}^{-1}\left(\tilde{C_1}, [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \mathcal{P}_{i,3}]\right)$
24: $t_3' \leftarrow {\mathrm{R}_{10}}^{-1}(C_2, [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \mathcal{P}_{i,3}])$
25: $\tilde{t_3}' \leftarrow {\mathrm{R}_{10}}^{-1}\left(\tilde{C_2}, [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \mathcal{P}_{i,3}]\right)$
26: **if** $\left(((t_3 \oplus \tilde{t_3}) \in \Omega) \wedge \left((t_3' \oplus \tilde{t_3}') \in \Omega\right)\right)$ **then**
27: $\mathcal{L}_i = \mathcal{L}_i \cup [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \mathcal{P}_{i,3}]$
28: **if** $(\forall i \in [0 \rightarrow 3], |\mathcal{L}_i| == 1)$ **then**
29: **return** $\mathcal{L}$
30: **else**
31: **return** Failure

states of two pairs match with the difference set Ω. In this case, $\tilde{\mathcal{P}}_i$ is added to $\mathcal{L}$ every time it matches with Ω, leading to 2^{32} 128 bits long key candidates.

A second filtering phase, presented in Alg.9, is used to find the key candidates amongst $\mathcal{L}$ that respect the `MixColumns` error property after MC_8. These are added to $\mathcal{L}'$. Once done, one may except 2^8 remaining key candidates.

The final phase consists in an exhaustive search using the correct ciphertext C_1, the known plaintext p and the remaining key candidates amongst $\mathcal{L}'$.

Algorithm 7 DFA Using 1 Fault

procedure KeyRecovery($C_1, \tilde{C}_1, p$)
  $\mathcal{L} \leftarrow$ FirstFilter($C_1, \tilde{C}_1$)
  $\mathcal{L}' \leftarrow$ SecondFilter($C_1, \tilde{C}_1, \mathcal{L}$)
  for $k \in \mathcal{L}'$ **do**
    if Decrypt(C_1, k) == p **then**
      return k
  return Failure

Algorithm 8 First Filtering Phase

procedure FirstFilter($C_1, \tilde{C}_1$)
  $\mathcal{L} = [\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3]$
  $\theta = [\theta_0, \theta_1, \theta_2, \theta_3]$
  for $i \in [0 \rightarrow 3]$ **do**
    $\Omega = \emptyset$
    $\mathcal{L}_i = \emptyset$
    for $\delta \in [1 \rightarrow 255]$ **do**
      $\Omega = \Omega \cup (\delta \times \theta_i)$
    for $\mathcal{P}_{i,0} \in [1 \rightarrow 255]$ **do**
      $t_0 \leftarrow \mathrm{R}_{10}{}^{-1}(C_1, [\mathcal{P}_{i,0}, 0, 0, 0])$
      $\tilde{t_0} \leftarrow \mathrm{R}_{10}{}^{-1}\left(\tilde{C}_1, [\mathcal{P}_{i,0}, 0, 0, 0]\right)$
      if $(t_0 \oplus \tilde{t_0}) \in \Omega$ **then**
        for $\mathcal{P}_{i,1} \in [1 \rightarrow 255]$ **do**
          $t_1 \leftarrow \mathrm{R}_{10}{}^{-1}(C_1, [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, 0, 0])$
          $\tilde{t_1} \leftarrow \mathrm{R}_{10}{}^{-1}\left(\tilde{C}_1, [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, 0, 0]\right)$
          if $(t_1 \oplus \tilde{t_1}) \in \Omega$ **then**
            for $\mathcal{P}_{i,2} \in [1 \rightarrow 255]$ **do**
              $t_2 \leftarrow \mathrm{R}_{10}{}^{-1}(C_1, [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, 0])$
              $\tilde{t_2} \leftarrow \mathrm{R}_{10}{}^{-1}\left(\tilde{C}_1, [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, 0]\right)$
              if $(t_2 \oplus \tilde{t_2}) \in \Omega$ **then**
                for $\mathcal{P}_{i,3} \in [1 \rightarrow 255]$ **do**
                  $t_3 \leftarrow \mathrm{R}_{10}{}^{-1}(C_1, [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \mathcal{P}_{i,3}])$
                  $\tilde{t_3} \leftarrow \mathrm{R}_{10}{}^{-1}\left(\tilde{C}_1, [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \mathcal{P}_{i,3}]\right)$
                  if $((t_3 \oplus \tilde{t_3}) \in \Omega)$ **then**
                    $\mathcal{L}_i = \mathcal{L}_i \cup [\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \mathcal{P}_{i,3}]$
  return $\mathcal{L}$

Note that every exhaustive search can be done using multi-threading techniques. This is not done in the solution implemented. Despite that, every attack attempts succeeded, taking approximately 1h15 to recover the 128bits key using an Intel(R) Core(TM) i7-3630QM (2.4GHz) CPU. Timing analysis of the program execution shows that 98% of the time is used to perform the second

Algorithm 9 Second Filtering Phase

```
1: procedure SecondFilter(C_1, C̃_1, L = [L_0, L_1, L_2, L_3])
2:     L' = ∅
3:     Ω' = ∅
4:     θ_3
5:     for δ ∈ [1 → 255] do
6:         Ω' = Ω' ∪ (δ × θ_3)
7:     for L^0 ∈ L_0 do
8:         for L^1 ∈ L_1 do
9:             for L^2 ∈ L_2 do
10:                for L^3 ∈ L_3 do
11:                    k_10 ← [L^0, L^1, L^2, L^3]
12:                    s ← R_10^{-1}(C_1, k_10)
13:                    s̃ ← R_10^{-1}(C_1, k_10)
14:                    k_9 ← KS_10^{-1}(k_10)
15:                    s' ← R_9^{-1}(s, k_9)
16:                    s̃' ← R_9^{-1}(s̃, k_9)
17:                    if ((s' ⊕ s̃') ∈ Ω') then
18:                        L' = L' ∪ k_10
       return L'
```

filtering phase. Multi-threading in this case could significantly improve the time needed to perform the key recovery, dividing approximately the attack duration time by the number of physical threads used.

4.5 Trojan Properties Discussions

As mentioned in Sec. 3.1.2, Trojans must fulfill two properties: the stealthiness and the triggerability.

Based on the measurements performed, the triggerability is obvious in this case: with a clock frequency $f_{clk} = 80$MHz, an adversary may trigger the Trojan with a probability between 0.99 and 0.996.

The stealthiness is more problematic. One may at the first sight say that, since the trigger mechanism is physical it has an infinite number of manipulation possibilities. Moreover, the Trojan is parametric and thus difficult to detect using both side channel and destructive detection methods. Last but not least, due to its parametric condition, the global alteration may be spread across several transistors and confused with potential process variability. However, in practice, a validator can detect an error with a probability equals to 1 by increasing the clock frequency. That is why, in order to keep stealthy its Trojan, a malicious manufacturer may specify that the produced infected chips tolerates a maximal clock frequency given by $f_{clk} < f_{trig}$. Considering this mechanism, the error induced by the Trojan can be confound with a normal setup timing violation appearing somewhere in the design. Indeed, thanks to the confusion-diffusion paradigm, the resulting ciphertext looks random when a fault appears. This holds as long as the validating process does not check the MixColumns difference properties, as done in classical differential fault analysis and which can be done in practice with a $\mathcal{O}(2^8)$ complexity. Precise EM side-channel analysis may in this case detect the Trojan. By analyzing the chip, one may create a mapping of the strongest emission points. The placements of the targeted LUTs, appearing far from the rest

of the other LUTs, could be classified as malicious modifications of the design.

Chapter 5

Second Trojan: key recovery under Trojan countermeasure mechanism

Remembering the Chapter 3.4, the protection mechanism introduced by [12] is a efficient way to protect block cipher implementations from cheat code and time bombs Trojan, more generally known as digitally triggered Trojan. However, as seen in Chapter 1, Trojan may be triggered by physical and side-channel means. Despite the fact that this kind of Trojan does not theoretically enter into the threat model proposed by [12], such Trojans can in practice be used in a convincing way while allowing to defeat the protection scheme.

This chapter describes the implementation of a side-channel triggered Trojan using timing modulation. Delivering its payload, the Trojan allows key recovering of protected implementation of AES and MOE block cipher implementations. It begins with the description of the threat model, followed by the protection scheme architecture description. It continues with the Trojan description and implementation results. Finally, some discussions are made about the stealthiness and triggerability properties of the proposed Trojan.

5.1 Threat model

Considering a circuit functionality Γ , a potential malicious manufacturer $\mathcal{A}$ get the specifications Γ and produces the device D that is supposed to implement the functionalities Γ. Since produced by $\mathcal{A}$, the device D may be infected by a Trojan. The security model involves digitally triggered Trojans and inputs side-channel triggered Trojan. Logic modifications are thus allowed. Remember that these Trojans deliver their payloads when a specific cheat-code is provided to the device, when a time bomb triggers or when a side-channel trigger is used through the inputs.

This is in fact the threat model considered in [12], but extended with a specific type of side-channel triggered Trojans. Formally, the threat is thus not considered by the protection scheme. However, this small side step out of the model leads to a huge modification in the security bounds of the Trojan resilient framework, since the adversary $\mathcal{A}$ is always able to win the game ROB_{Π} as shown in the following sections.

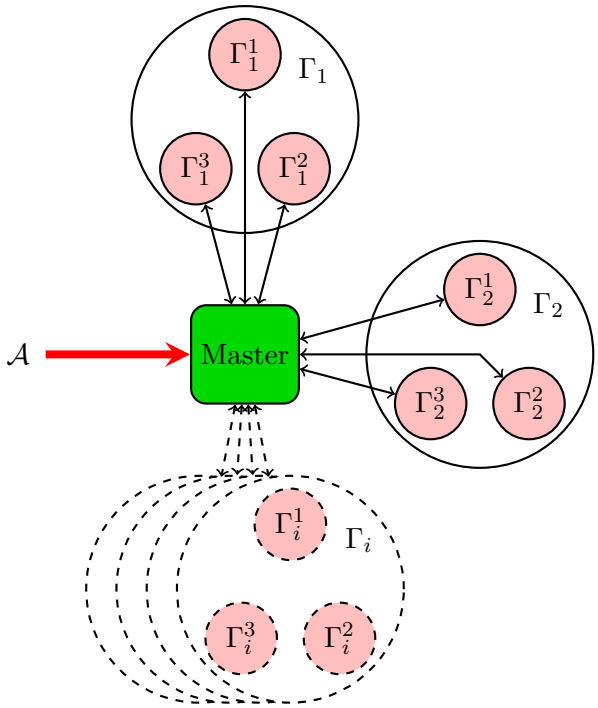

Figure 5.1: Trojan Resilient Framework Architecture [7]

5.2 Targetted Architecture Description

The targetted architecture is the protected implementation of AES. As mentioned in Chapter 3.4, it involves a secure 3-party computation protocol and a random testing phase combined with a majority vote in order to avoid cheat codes Trojan and time-bombs.

More specifically, the protected version is obtained by using some devices $[D_1, ..., D_\lambda]$ called the subcircuits, each of them securely emulating the AES block cipher. This emulation is done in a secure way using a set of three mini-circuits $[\Gamma_i^1, ..., \Gamma_i^3]$ that jointly implement a secure 3-party computation protocol. An additional trusted master circuit M is used in order to share the inputs, send them to the different sub-circuits, manage the communication of shares between mini-circuits of a same subcircuit during the 3-party computation protocol and reconstruct the secret once the encryption is done. Finally, it implements the majority vote for the reconstructed output of each sub-circuit.

Practically, the sub-circuit is more conceptual: the master is in fact directly communicating with sets of three minicircuits (as shown in Fig.5.1). This set of mini-circuits represents the subcircuit. Indeed, if a unique device D produced by $\mathcal{A}$ is supposed to implement the emulating secure 3-party computation protocol, nothing avoid him to recombine the shares internally. By doing that, the secret is recovered and the secure 3-party protocol is useless. In this purpose, a honest designer $\mathcal{D}$ provide the specifications $[\Gamma_i^1, ..., \Gamma_i^3]$ for each desired sub-circuit to a malicious manufacturer $\mathcal{A}$ that produces a set of 3 devices supposedly implementing $[\Gamma_i^1, ..., \Gamma_i^3]$. Thanks to that, the only communication allowed between the devices is through the trusted master circuit.

The general architecture of the protection scheme is a trusted master device M communicating with λ sets of three slaves devices $\left[D_{\Gamma_i^1}, ..., D_{\Gamma_i^3}\right]_{i=1\to\lambda}$ representing the mini-circuits sets (as depicted in Figure 5.2). Operation by operation, the system performs the required algorithms in order to compute the ciphertext. The master M manages the different algorithms using control

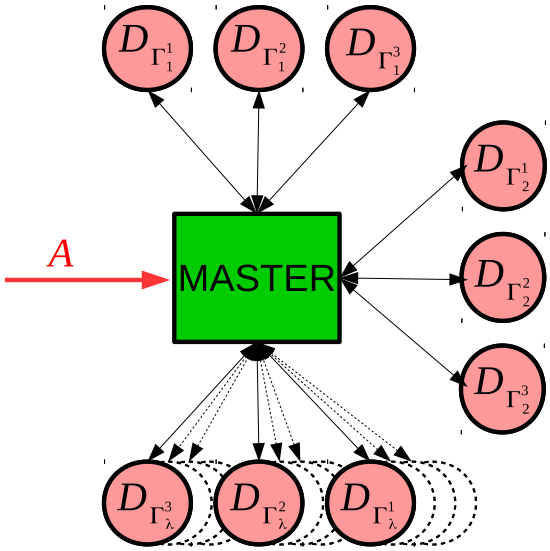

Figure 5.2: Trojan Resilient Framework Implementation

Figure 5.3: Trojan Payload Insertion in AES

signals and by relaying the data between the different slaves amongst each sets $\left[D_{\Gamma_i^1}, ..., D_{\Gamma_i^3}\right]$. Finally, the system is working with a clock frequency of 66MHz.

5.3 Trojan Implementation

As said in Section 5.1, considering side-channel triggers through the inputs allows an adversary to recover the key from a protected AES implementation. In this purpose, this section presents a Trojan triggered by plaintext arrival time modulation. A description of the Trojan principle is given, followed by the practical implementation and results obtained with a Virtex VI FPGA evaluation board.

5.3.1 Trojan Description

Despite the fact that the protection scheme proposed in [12] allows to avoid exploitation of cheat code and time bombs Trojans, it does not consider side-channel information that may pass through the inputs. For example, when the master device receives a plaintext at its inputs, it proceeds to the encryption process at the next clock edge. The timing information relating to the interval between two plaintext receptions is thus transmitted to the different minicircuits devices at the same time as the shared inputs.

An adversary can take advantage of this information in order to trigger a Trojan implemented inside the mini-circuit devices. More into the details, the trigger mechanism of the proposed Trojan is based on the modulation of the inputs arrival time. If we consider l modulation levels (from slow to the fast), $\mathcal{A}$ can design a Trojan that triggers if and only if a specific modulation sequence of inputs arrival time, long of q modulation levels, is sent to the device. By doing that, the protection scheme is override and the Trojan can proceed to its malicious behavior. This method let the adversary free to choose the values of the parameters l and q, depending on the trade-off he faces.

Once the trigger mechanism established, the Trojan payload needs to be defined, with the aims to eventually let the adversary $\mathcal{A}$ recover the secret key. In this purpose, first remember the AES block cipher architecture described in Chapter 2.2.1: it consists in a initial `AddRoundKey` operation followed by 9 rounds, each composed of `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` operations. A final round without `MixColumns` operation ends the encryption process. Since design modification are allowed in the threat model, $\mathcal{A}$ can simply add logic to output the subkey used in the last `AddRoundKey` operation instead of the correct ciphertext when the trigger sequence is received. He can do it by using multiplexers driving 0 at the encryption state input of the `AddRoundKey` operation or directly drive the subkey value at the output instead of the correct ciphertext (as shown in Figure 5.3).

Algorithm 10 ADDITION

$\rightarrow \Gamma_i^1$ holds $\tilde{v}_1 = (\alpha_1, x_3)$ and $\tilde{w}_1 = (\beta_1, y_3)$
$\rightarrow \Gamma_i^2$ holds $\tilde{v}_2 = (\alpha_2, x_1)$ and $\tilde{w}_2 = (\beta_2, y_1)$
$\rightarrow \Gamma_i^3$ holds $\tilde{v}_3 = (\alpha_3, x_2)$ and $\tilde{w}_3 = (\beta_3, y_2)$
Γ_1 computes $\tilde{a}_1 = (\alpha_1 \oplus \beta_1, x_3 \oplus y_3)$ and holds it
Γ_2 computes $\tilde{a}_2 = (\alpha_2 \oplus \beta_2, x_1 \oplus y_1)$ and holds it
Γ_3 computes $\tilde{a}_3 = (\alpha_3 \oplus \beta_3, x_2 \oplus y_2)$ and holds it

Algorithm 11 RECONSTRUCTION

Γ_i^1 sends $\tilde{v}_1' = (\alpha_1', x_3')$ to M
Γ_i^2 sends $\tilde{v}_2' = (\alpha_2', x_1')$ to M
Γ_i^3 sends $\tilde{v}_3' = (\alpha_3', x_2')$ to M
M computes $o_1 = \tilde{v}_1'[0] \oplus \tilde{v}_2'[1] = \alpha_1' \oplus x_1'$
M computes $o_2 = \tilde{v}_2'[0] \oplus \tilde{v}_3'[1] = \alpha_2' \oplus x_2'$
M computes $o_3 = \tilde{v}_3'[0] \oplus \tilde{v}_1'[1] = \alpha_3' \oplus x_3'$
if $(o_1 == o_2 == o_3)$ **then return** o_1
elseRaises an error.

Second, the AES block cipher is protected. The operations are thus computed with shares by a secure 3-party computation protocol. The 3-party key addition algorithm presented in Section 3.4.9 need thus to be taken into account while designing the payload. It is remembered in Alg.10. Based on it, it emerges that the techniques proposed before still work. For example, considering the encryption state shares $\tilde{v}_i$ and the key shares $\tilde{w}_i$, the followings computations hold by driving $\tilde{v}_i$ to 0:

$$\begin{aligned} \tilde{a}_1 &= (\beta_1, y_3) \\ \tilde{a}_2 &= (\beta_2, y_1) \\ \tilde{a}_3 &= (\beta_3, y_2) \end{aligned}$$

Using the RECONSTRUCTION algorithm remembered in Alg.11, the different reconstructed values are obtained as follows:

$$\begin{aligned} o_1 &= \beta_1 \oplus y_1 = \beta_1 \oplus k \oplus \beta_1 = k \\ o_2 &= \beta_2 \oplus y_2 = \beta_2 \oplus k \oplus \beta_2 = k \\ o_3 &= \beta_3 \oplus y_3 = \beta_3 \oplus k \oplus \beta_3 = k \end{aligned}$$

and since all the reconstructed values are similar, the master considers the value as valid. As mentioned in Section 5.2, the malicious manufacturer produces all the sets of mini-circuits devices.

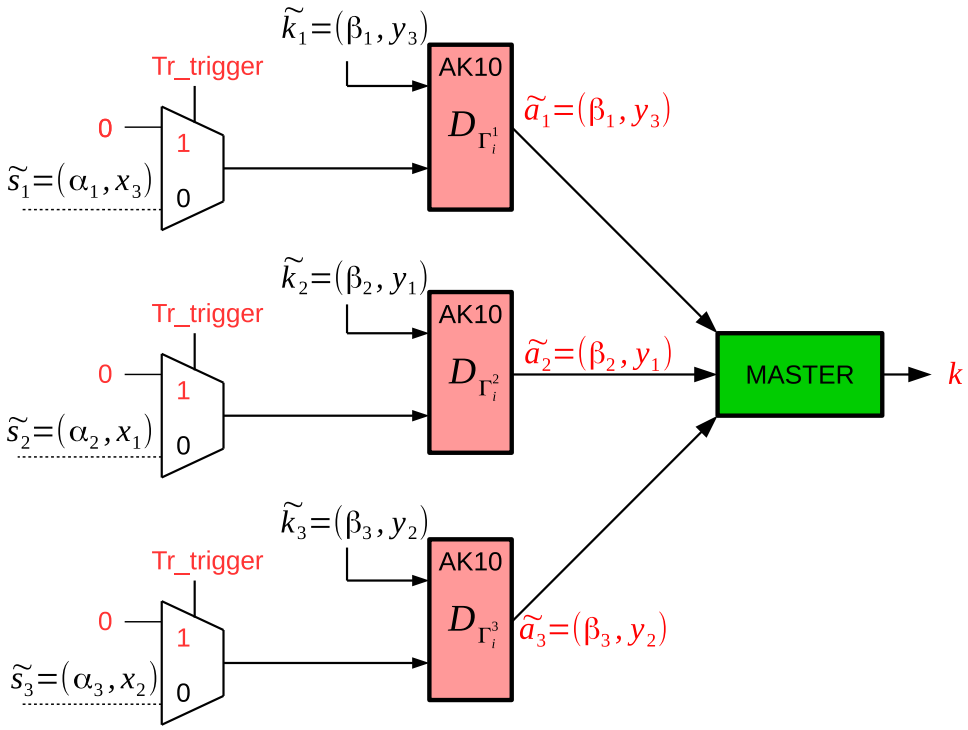

Figure 5.4: Payload Implementation in AES

Because of that, the Trojan is present in all the mini-circuits, which induces that all the values returned by the different sets at the reconstruction step are similar. The majority vote keeps thus the value and the master return the key as output value of the encryption process.

5.3.2 Trojan Practical Implementation

The proposed Trojan is implemented on a ML605 Virtex VI Evaluation Board from Xilinx. Note that in this practical case, the targetted ciphertext is not the AES but the Multiplication Operated Encryption [8] block cipher, which is also based on the protection scheme presented in [12]. The presented Trojan works against this block cipher since, as for AES, it finishes with a key addition. The only difference in the Trojan implementation between both is the location of the payload introduction (cfr Figs.5.4, 5.5). Indeed in MOE the last key addition is done by the trusted master circuit after the secret reconstruction. The objective of the Trojan in the MOE block cipher is thus to output from the sub-circuit a share composed of 0, leading to a reconstructed value validated by the majority vote also equal to also 0. The key addition done by the master before outputting the value lead thus to an output of the secret key value. Excepting the payload mechanism location, all the following discussions and constructions hold for a Trojan implementation against the AES block cipher.

The Trojan implementation can be divided in two main parts: the trigger and the payload mechanisms.

Plaintext Arrival Detection Mechanism The trigger mechanism needs to be able to detect a plaintext arrival. This is done in the implementation using the active-high control signal needed to reset the internal Final State Machine (FSM) each time a new encryption process start. This signal is denoted here by FSM_RST. Usually, a input validity signal is used and propagate in

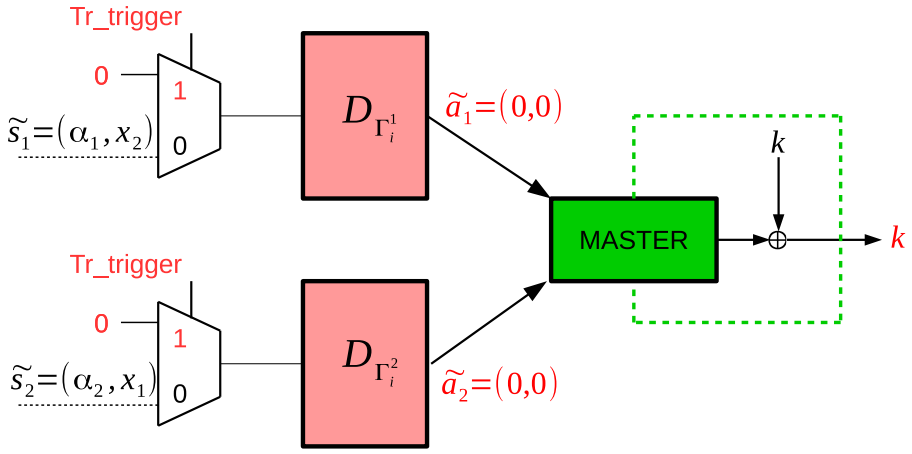

Figure 5.5: Payload Implementation in MOE

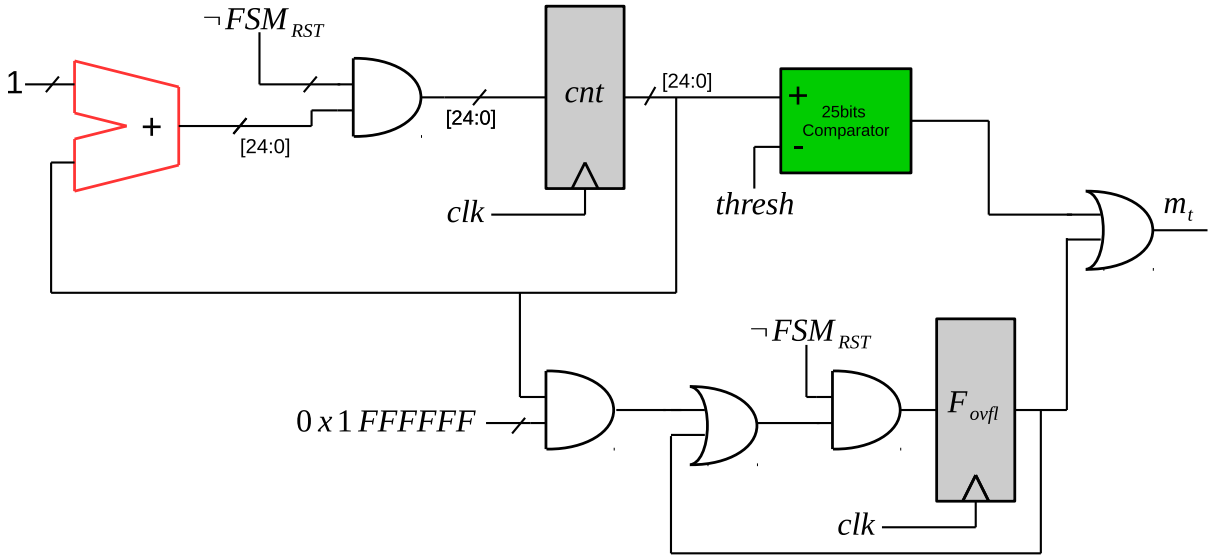

Figure 5.6: Modulation Level Detection Mechanism Implementation

order to check the encryption completion status. This signal can be used too in the plaintext detection purpose.

Modulation Levels Detection Mechanism Based on the plaintexts arrivals, the trigger mechanism has to be able to detect the modulation type. The proposed construction used in order to achieve this goal is presented in Fig.5.7. A specific counter *cnt* is reset each time a new plaintext arrival is detected and is incremented at each clock cycle. Each time the counter value exceeds a specific threshold *thresh*, the resulting modulation type m_t is updated. The proposed implementation is considering 2 modulation types in order to avoid consequent additional logic required to distinguish between more distinction levels. The threshold is arbitrarily chosen such that every delay above 0.5s between two plaintexts arrivals is considered as a low modulation. In this case, using a clock frequency of 66MHz, the threshold is thus set to 33000000. The corresponding counter *cnt* must thus be at least 25bits long. An additional logic using a flag F_{ovfl} is used in order to manage the case when the counter overflows. This flag keeps the information that the modulation is low when such an overflow happens.

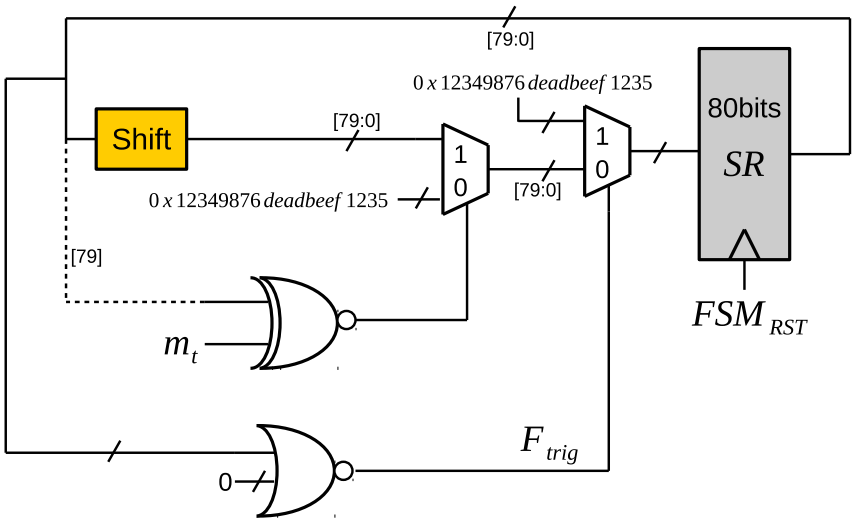

Figure 5.7: Modulation Sequence Detection Mechanism Implementation

Modulation Sequence Detection Mechanism Based on the two mechanisms described above, a final component is added to the trigger mechanism. Indeed, it must be able to detect sequences of modulation. For this purpose, the binary pre-encoded 80bits long modulation sequence 0x12349876deadbeef1235 is hardcoded in a shift register. Every time a plaintext arrival is detected with the FSM_RST, the binary modulation type m_t is derived and compared with the MSB of the shift register. If the values are similar, the register is shifted of one bit. If not, its value is reset to the binary modulation sequence. Once the value of the shift register is 0, a flag F_{trig} announcing the completeness of the trigger sequence is set. If the flag F_{trig} is set, the shift register is reset to the binary modulation sequence at the next clock cycle.

Payload Mechanism The payload mechanism is activated if and only if the F_{trig} flag is set. It consists in a multiplexer controlled by F_{trig}, driving the targeted shared value to 0 if the flag is set.

5.4 Key Recovery

The key recovery mechanism involves two steps: the Trojan reset sequence and the Trojan trigger sequence.

5.4.1 Reset Sequence

Before sending plaintext according to the trigger sequence, the adversary need to reset the value of the shift register. Indeed, unless having stored all the previous modulation times of the previous encryptions since the last global reset of the device, he has no idea of the current state of the register file. The only way to be sure of its value is to force a reset to the modulation sequence. To do so, he sends a so called reset sequence. This sequence is specifically chosen by the adversary such that at the end of it, the shift register has as value the trigger sequence. Considering the trigger sequence as a binary word s_w, he can compute a reset sequence using the following algorithm:

1. Find MSB value of s_w, denoted as b_m.
2. Compute the bit value b_s by computing $b_s = \neg b_m$.
3. Find the longest subword s'_w of s_w involving only the value b_s.
4. Compute the reset sequence as $s_{reset} = s'_w || b_s$.

The reset sequence is composed of bits values opposed to the trigger sequence MSB in order to avoid that the register shifts once reset. Using a sequence created by the concatenation of s'_w and b_s assures that the shift register is reset at least once during the sequence. For example, considering the trigger sequence $0b101110100$, the reset sequence found using the presented algorithm is $0b000$. Note that in this case, the MSB and the LSB of the trigger sequence have not the same value. This may lead the Trojan to be triggered by sending the reset sequence. Although it is not critical since the attacker is still able to trigger it as he pleases, this effect may be undesirable to assure a total control on the Trojan triggering. Moreover, since the flag F_{trig} is set when the shift register is at the value $0b0$, finishing trigger sequence with a sequence of $0b0$ reduces in fact the real trigger sequence size. Based on that, proper trigger sequences need to begin and finish with $0b1$ values, as done for the proposed solution. Its corresponding reset sequence found by the algorithm is $0b00000$.

5.4.2 Trigger Sequence and Key Recovery

After having sent plaintexts according to the reset sequence to the device, the adversary uses the trigger sequence. Once done, he knows that the following encryption process output is the secret key, thanks to the activation of the Trojan. Every key recovery attempt on the implemented block cipher succeeded.

5.5 Trojan Properties Discussion

As mentioned earlier, Trojans must fulfill two properties: the stealthiness and the triggerability. In this case, triggerability of a single device is obvious: by using the reset and the triggering sequences, the adversary is always able to trigger the Trojan. However, the key recovery attack is only possible if the value returned by the malicious mini-circuits sets is the one kept after the majority vote. This implies that at least $\lambda/2$ mini-circuit devices sets need to be infected. This can be avoided if the different devices sets are produced by different manufacturer, provided that they are not in collusion.

One may argue on the stealthiness: First, by considering a trigger sequence of $q+2$ bits such that its MSB and LSB are equal to $0b1$, a validator a probability p to trigger the Trojan using random modulation sequences given by $p = \frac{1}{2^q}$, which becomes negligible for a growing q. The complexity of the detection grows exponentially with a linear growth in the complexity of the Trojan implementation: once the detection mechanism implemented, adding bit to the shift register size is a linear cost comparing to the exponentially decreasing probability p. Second, a comparison of the Trojan implementation impact on the logic is shown in Tab.5.1. A 3.2% additional logic is an appreciable result. Despite that, one may consider that the Trojan could be detected using precise side-channel power analysis.

	Honest design	Malicious design	Additional Logic
Slice registers amount	7,292	7,899	8.3%
Slice LUTs amount	38,729	39,614	2.2%
Global logic amount	46,021	47,513	3.2%

Table 5.1: Trojan implementation impact on logic

Chapter 6

Conclusion

Because of the nowadays globalization of ICs manufacturing, verification of chips specifications is an increasingly difficult problem. In addition, some manufacturers could not be trustworthy. Having access to the design of the systems they produce, these could bring modifications at some points in the manufacturing process. In particular, malicious logic or parametric alteration could lead to intentional specifications change of the chip. Such alterations are known as hardware Trojans. Not trusting manufacturer is the ultimate threat model in ICs design and one may wonder about the feasibility of such Trojan introduction.

Contributions

Two Trojans, respectively targeting protected and unprotected block cipher implementation, were designed and implemented on a FPGA Virtex6 evaluation board in order to recover the secret key used by the encryption schemes.

The first proposed Trojan targets a round pipelined implementation of 128bits key size. Based on the ideas proposed in [16] and [13], it induces random faults caused by setup timing constraint violation when the clock frequency is exceeding a particular threshold. Strategically introduced at the 8^th round of the AES by adding delay a the post place and route stage, an individual induced fault enables a key recovery using a differential fault analysis . The whole recovery takes approximately 1h15 using a non-multithreaded architecture. Implemented using parametric delay modification and with a trigger based on the frequency, this Trojan is assumed to be hard to detect regarding the current verification methods.

The second proposed Trojan targets AES and MOE block ciphers implementations under the Trojan resilient framework proposed in [12]. Being slightly out of the threat model considered by the framework, the malicious design proposed shows the difficulties faced in the design of countermeasure against side-channel triggered Trojan. Based on detection of the plaintext arrival time modulation, the Trojan bypasses the security mechanisms used to protect block ciphers implementations against digitally triggered Trojans, allowing online key recovery. The modulation sequence has been designed in order to assure a bound of trigger probability under random input of $\frac{1}{2^{80}}$, with an additional logic of 3.2% comparing to the original design.

Impact

This work presents two Trojan implementations that may be used in practical cases against block cipher architectures. More generally, it shows that hardware Trojans are serious threats regarding the amount of design alterations and exploitations possibilites. Indeed, since a Trojan insertion may occur at every steps in the design and manufacturing flow, its implementation is theorically unbounded. The need of efficient prevention mechanisms in ICs manufacturing process is thus highlighted.

In addition, because of their stealthy property, Trojans are in practice difficult to detect. If no prevention methods are set up, chips validation and protection are the following protection ledgers against Trojans. It therefore appears that there is a need for effective detection methods and Trojan resilient scheme, both based on digital input testing and side-channel protection mechanisms. This seems to be challenging, again considering the amount of possible manipulations used to trigger such Trojans. A modification in the testing paradigm may be needed, in order to consider specific Trojan presence checks in addition to classical verifications testing. Remaining question are about the bound of such verification process while dealing with continuous physical quantities.

Bibliography

[1] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM side-channel(s). In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, pages 29–45, 2002.

[2] AnySilicon. *Semiconductor Fabrication Plant*. https://anysilicon.com/semipedia/semiconductor-fabrication-plant/.

[3] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 805–817, 2016.

[4] Alex Baumgarten, Michael Steffen, Matthew Clausman, and Joseph Zambreno. A case study in hardware trojan design and implementation. *Int. J. Inf. Sec.*, 10(1):1–14, 2011.

[5] Mark R G Beaumont, Bradley D Hopkins, and Tristan Newby. *Hardware Trojans – Prevention, Detection, Countermeasures*. DSTO Defence Science and Technology Organisation.

[6] David Bol. Lelec2570 - synthesis of digital integrated circuits (digital ics) lecture, 2017.

[7] Olivier Bronchain, Louis Dassy, Sebastian Faust, and François-Xavier Standaert. Implementing trojan-resilient hardware from (mostly) untrusted components designed by colluding manufacturers. 2017.

[8] Olivier Bronchain, Sebastion Faust, Virginie Lallemand, Gregor Leander, Léo Perrin, and François-Xavier Standaert. Moe: Multiplication operated encryption with trojan resilience. 2017.

[9] Rajat Subhra Chakraborty, Seetharam Narasimhan, and Swarup Bhunia. Hardware trojan: Threats and emerging solutions. In *IEEE International High Level Design Validation and Test Workshop, HLDVT 2009, San Francisco, CA, USA, 4-6 November 2009*, pages 166–171, 2009.

[10] Joan Daemen and Vincent Rijmen. The block cipher rijndael. In *Smart Card Research and Applications, This International Conference, CARDIS '98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings*, pages 277–284, 1998.

[11] Joan Daemen and Vincent Rijmen. Rijndael for AES. In *AES Candidate Conference*, pages 343–348, 2000.

[12] Stefan Dziembowski, Sebastian Faust, and François-Xavier Standaert. Private circuits

III: hardware trojan-resilience via testing amplification. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 142–153, 2016.

[13] Maik Ender, Samaneh Ghandali, Amir Moradi, and Christof Paar. The first thorough side-channel hardware trojan. *IACR Cryptology ePrint Archive*, 2017:865, 2017.

[14] Ericsson. *Internet of Things forecast.* https://www.ericsson.com/en/mobility-report/internet-of-things-forecast, Nov 2016.

[15] Dave Evans. *The Internet of Things: How the Next Generation of the Internet is Changing Everything.* Cisco Internet Business Solution Group (IBSG).

[16] Samaneh Ghandali, Georg T. Becker, Daniel Holcomb, and Christof Paar. A design methodology for stealthy parametric trojans and its application to bug attacks. *IACR Cryptology ePrint Archive*, 2016:600, 2016.

[17] Andrew Grochowski, Debashis Bhattacharya, T. R. Viswanathan, and Ken Laker. Integrated circuit testing for quality assurance in manufacturing: History, current status, and future trends. *IEEE Transactions On Circuits And Systems-II: Analog And Digital Signal Processing*, 44:610, 1997.

[18] Syed Kamran Haider, Chenglu Jin, Masab Ahmad, Devu Manikantan Shila, Omer Khan, and Marten van Dijk. Hatch: Hardware trojan catcher. *IACR Cryptology ePrint Archive*, 2014:943, 2014.

[19] Sarah Harris. *Digital design and computer architecture.* Elsevier/Morgan Kaufmann, Amsterdam, 2016.

[20] Yier Jin, Nathan Kupp, and Yiorgos Makris. Experiences in hardware trojan design and implementation. In *IEEE International Workshop on Hardware-Oriented Security and Trust, HOST 2009, San Francisco, CA, USA, July 27, 2009. Proceedings*, pages 50–57, 2009.

[21] Yier Jin and Yiorgos Makris. Hardware trojans in wireless cryptographic ics. *IEEE Design & Test of Computers*, 27(1):26–35, 2010.

[22] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography.* Chapman and Hall/CRC Press, 2007.

[23] Samuel T. King, Joseph Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang, and Yuanyuan Zhou. Designing and implementing malicious hardware. In *First USENIX Workshop on Large-Scale Exploits and Emergent Threats, LEET '08, San Francisco, CA, USA, April 15, 2008, Proceedings*, 2008.

[24] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 388–397, 1999.

[25] Lang Lin, Markus Kasper, Tim Güneysu, Christof Paar, and Wayne Burleson. Trojan side-channels: Lightweight hardware trojans through side-channel engineering. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, pages 382–395, 2009.

[26] Yu Liu, Ke Huang, and Yiorgos Makris. Hardware trojan detection through golden chip-free

statistical side-channel fingerprinting. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, pages 155:1–155:6, 2014.

[27] Stefan Mangard, Elisabeth Oswald, and François-Xavier Standaert. One for all - all for one: Unifying standard DPA attacks. *IACR Cryptology ePrint Archive*, 2009:449, 2009.

[28] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against SPN structures, with application to the AES and KHAZAD. In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, pages 77–88, 2003.

[29] Falk Schellenberg, Markus Finkeldey, Nils Gerhardt, Martin Hofmann, Amir Moradi, and Christof Paar. Large laser spots and fault sensitivity analysis. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2016, McLean, VA, USA, May 3-5, 2016*, pages 203–208, 2016.

[30] Ehsan Sharifi, Kamal Mohammadiasl, Mehrdad Havasi, and Amir Yazdani. Performance analysis of hardware trojan detection methods. *INJOIT International Journal of Open Information Technologies*, 3:39–44, 2015.

[31] John Shield, Bradley D. Hopkins, Mark R. Beaumont, and Chris J. North. Hardware trojans - A systemic threat. In *13th Australasian Information Security Conference, AISC 2015, Sydney, Australia, January 2015*, pages 45–51, 2015.

[32] François-Xavier Standaert. *LELEC2760 - Secure Electronic Circuits and Systems Lecture*. 2018.

[33] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Design & Test of Computers*, 27(1):10–25, 2010.

[34] Michael Tunstall and Debdeep Mukhopadhyay. Differential fault analysis of the advanced encryption standard using a single fault. *IACR Cryptology ePrint Archive*, 2009:575, 2009.

[35] Xiaoxiao Wang, Mohammad Tehranipoor, and Jim Plusquellic. Detecting malicious inclusions in secure hardware: Challenges and solutions. In *IEEE International Workshop on Hardware-Oriented Security and Trust, HOST 2008, Anaheim, CA, USA, June 9, 2008. Proceedings*, pages 15–19, 2008.

Appendix A

Protection Scheme: Security Bound Computation

A.1 Detailled Parameters and Scheme Construction Intuitions

The probability that a Bad event is not detected during $t_0 \leq t$ test runs is bounded according to Eq.A.1. On the other hand, the probability that a Bad event occurs during one of the n real runs bounded according to Eq.A.2. From Eq.A.1 and Eq.A.2, we can bound the probability p that D passed the testing phase and fails during the real execution as shown in Eq.A.3.

$$\begin{aligned} \Pr\left(\neg \mathrm{Bad}_1 \wedge ... \wedge \neg \mathrm{Bad}_{t_0}\right) &\geq (1-\epsilon)^t && \text{(A.1)} \\ \Pr\left(\mathrm{Bad}\right)_n &= 1-(1-\epsilon)^n && \text{(A.2)} \\ p &\geq (1-\epsilon)^t \cdot (1-(1-\epsilon)^n) && \text{(A.3)} \end{aligned}$$

Since the goal of the adversary $\mathcal{A}$ is to induced a Bad event during the real runs once the testing succeded, he sets

$$\begin{aligned} \epsilon &= \epsilon_{max} && \text{(A.4)} \\ \epsilon_{max} &= 1-\left(\frac{t}{n+t}\right)^{\frac{1}{n}} && \text{(A.5)} \end{aligned}$$

with ϵ_{max} the ϵ value that maximizes p. Considering Eq.A.4, the Eq.A.3 evolves as follows

$$\begin{aligned} p &\geq (1-\epsilon)^t \cdot (1-(1-\epsilon)^n) && \text{(A.6)} \\ &\geq \left(\frac{t}{n+t}\right)^{\frac{t}{n}} \cdot \frac{n}{n+t} && \text{(A.7)} \\ &\geq \left(1+\frac{n}{t}\right)^{\frac{-t}{n}} \cdot \frac{n}{n+t} && \text{(A.8)} \\ &\geq \frac{n}{e \cdot (n+t)} && \text{(A.9)} \end{aligned}$$

where e is the Euler's number. The Eq.A.9 comes from the fact that $(1+n/t)^{-t/n} \geq e^{-1}$.

A.2 Security Bounds Computation

In order to find the security bound, we first need to update the security framework presented in Section 3.4.4 according to the architecture of the protection scheme proposed.

Indeed in addition to Alg.2, we need to consider the 3-party protocol, the majority vote $\text{MAJ}^{D_1,\ldots,D_\lambda}$ and the random amount of test t_i for the device D_i. Moreover, the scheme protects against time-bombs. We should thus not consider anymore that a Bad event during the real execution only occurs because of a input $\vec{v}$ choosen by $\mathcal{A}$. Instead, random inputs are used. The updated version of $\text{ROB}_\Pi(\mathcal{A}, \lambda, n, t, k, \Gamma)$ is shown [1] in Alg.12.

Base on that, we can compute the probability that an index i is added to Λ denoted by $\Pr[i \in \Lambda]$. This only occurs when the following conditions are met:

1. The devices D_i pass the testing phase of $t_i \leq t$ runs according to Γ_i, which is denoted by

$$D_i(\vec{\alpha}, \vec{x}) = \Gamma'_i(\vec{\alpha}, \vec{x})$$

2. Some Bad event occur during the execution phase with at most n runs, which is denoted by

$$D_i(\vec{\alpha}, \vec{x}) \neq \Gamma'_i(\vec{\alpha}, \vec{x})$$

We denote the first time that $D_i(\vec{r}, \vec{s}) \neq \Gamma'_i(\vec{r}, \vec{s})$ by $\nu_i \in [n + t_i]$. It follows that

$$\Pr[i \in \Lambda] = \Pr[\nu_i \in [t_1 + 1, t_i + n]] \leq \frac{n}{t}$$

Considering now the majority vote, there must be a least $\lambda/2$ Bad event occuring simultaneously in a run in order to fool $\text{MAJ}^{D_1,\ldots,D_\lambda}$. The final bounds is thus given by:

$$\Pr[|\Lambda| \geq \lambda/2] \leq \left(\frac{n}{t}\right)^{\lambda/2}$$

[1] Note that the presented game is vulgarized compared to REF. It should normally consider testing phase based on view of D_i consisting of messages exchanges tuples with the tester.

Algorithm 12 $\mathrm{ROB}'_{\Pi}(\mathcal{A}, \lambda, n, t, k, \Gamma)$

$(M, (\Gamma_i)) \leftarrow TR\left(1^k, \lambda, \Gamma\right)$
$D_i \leftarrow \mathcal{A}\left(1^k, (M, \Gamma_i)\right)$
for $i \in [\lambda]$ **do**
    Sample $t_i \leftarrow [t]$
    for $j = 1$ to t_i **do**
        Sample random sharing of inputs $\vec{r}, \vec{s} \leftarrow \mathbb{F}$
        if $D_i(\vec{r}, \vec{s}) \neq \Gamma'_i(\vec{r}, \vec{s})$ **then return** 0
$\vec{x_1} \leftarrow \mathcal{A}\left(1^k\right)$
for $j = 1$ to n **do**
    Set $\Lambda = \{\}$
    for $i \in [\lambda]$ **do**
        Sample random sharing inputs $\vec{r}, \vec{s} \leftarrow \mathbb{F}$
        if $D_i(\vec{r}, \vec{s}) \neq \Gamma'_i(\vec{r}, \vec{s})$ **then** add i to Λ
    if $|\Lambda| \geq \lambda/2$ **then return** 1
return 0

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