**UNIVERSITY OF PIRAEUS - DEPARTMENT OF INFORMATICS**

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

**MSc «Distributed Systems, Security and Emerging Information Technologies»**

ΠΜΣ «Κατανεμημένα Συστήματα, Ασφάλεια και Αναδυόμενες Τεχνολογίες Πληροφορίας»

# MSc Thesis

Μεταπτυχιακή Διατριβή

| Thesis Title:<br>Τίτλος Διατριβής: | Evaluation of the security of embedded systems against fault injection attacks<br>Αξιολόγηση της ασφάλειας ενσωματωμένων συστημάτων σε επιθέσεις εισαγωγής σφαλμάτων |
|---|---|
| Student’s name-surname:<br>Ονοματεπώνυμο φοιτητή: | Nikolaos Tziris-Georgopoulos<br>Νικόλαος Τζιρής-Γεωργόπουλος |
| Father’s name:<br>Πατρώνυμο: | Georgios<br>Γεώργιος |
| Student’s ID No:<br>Αριθμός Μητρώου: | ΜΠΚΣΑ/20003 |
| Supervisor:<br>Επιβλέπων: | Michael Psarakis, Associate Professor<br>Μιχαήλ Ψαράκης, Αναπληρωτής Καθηγητής |

November 2023 / Νοέμβριος 2023

**3-Member Examination Committee**

Τριμελής Εξεταστική Επιτροπή

| **Michail Psarakis** | **Panayiotis Kotzanikolaou** | **Athanasios Papadimitriou** |
|---|---|---|
| **Associate Professor** | **Associate Professor** | **Assistant Professor** |
| Μιχαήλ Ψαράκης | Παναγιώτης Κοτζανικολάου | Αθανάσιος Παπαδημητρίου |
| Αναπληρωτής Καθηγητής | Αναπληρωτής Καθηγητής | Επίκουρος Καθηγητής |

# Acknowledgements

Combining my background in electrical and computer engineering with my professional experience as a cybersecurity professional, I embarked on an intriguing exploration of the security aspects within the realm of IoT and embedded systems. Having worked as a penetration tester, I have gained valuable insights into assessing the security of various systems. However, it was the convergence of my hardware background and my interest in cybersecurity that truly fascinated me. The integration of physical and digital realms in IoT and embedded systems presented a captivating challenge, driving me to further investigate their security vulnerabilities.

I have to admit that as I followed the lectures given by Dr. Papadimitriou in the course of "Internet of Things and Embedded Systems" in the first semester of my postgraduate studies, I was thrilled about the techniques on finding the AES encryption key using hardware attacks, and it sounded like pure magic. I hope that this thesis will help someone understand this magic better.

I would like to express my sincere gratitude to my supervisor, Assistant Professor Athanasios Papadimitriou, for his unwavering support, guidance, and invaluable contributions throughout this research endeavor. Additionally, I am grateful to my supervisor Associate Professor Mihalis Psarakis for supplementing the evaluation of my MSc Thesis, enriching the academic experience. Their expertise in the field has been instrumental in shaping the direction and depth of this thesis.

Lastly, I would like to acknowledge the support and encouragement of my family, friends, and colleagues throughout this journey.

October 2023

Nikolaos Tziris-Georgopoulos

## Περίληψη

Η αυξανόμενη χρήση ενσωματωμένων συστημάτων υπογραμμίζει τη σημασία της ασφάλειας υλικού, και συνεπώς της έρευνα στο αντικείμενο των επιθέσεων υλικού. Οι επιθέσεις εισαγωγής σφάλματος μέσω της διαταραχής της τάσης της τροφοδοσίας αποτελεί μια μεθοδολογία επίθεσης με ποικίλες εφαρμογές. Εν προκειμένω η επίθεση εισαγωγής σφάλματος μελετήθηκε σε επιθέσεις ενάντια σε μικροελεγκτή κατά την εκτέλεση του κρυπτογραφικού αλγορίθμου AES-128 με στόχο τον υπολογισμό του κρυπτογραφικού κλειδιού με την εφαρμογή της Διαφορικής Ανάλυσης Σφάλματος. Τα μοντέλα σφάλματος σε επίπεδο bit και επίπεδο byte μελετήθηκαν και υλοποιήθηκαν σε επίπεδο προσομοίωσης με τη χρήση MATLAB. Συνακόλουθα, υλοποιήθηκε με τη χρήση του Chipwhisperer nano, η εισαγωγή σφάλματος σε ένα μόνο μπάιτ. Τα αποτέλεσμα έδειξαν υψηλά ποσοστά επιτυχίας στην ανάκτηση του κρυπτογραφικού κλειδιού.


**Λέξεις Κλειδιά:** Κυβερνοασφάλεια, Ασφάλεια Υλικού, Επιθέσεις εισαγωγής σφάλματος, Διαφορική Ανάλυση Σφάλματος

---


## Abstract

The increasing use of embedded systems highlights the significance of hardware security, leading to research on hardware attacks. Fault injection attacks via voltage glitch represent a methodology with diverse applications. In this context, fault injection attacks were studied against a microcontroller executing the cryptographic algorithm AES-128, aiming to compute the cryptographic key using Differential Fault Analysis. The bit-level and byte-level fault models were examined and implemented at the simulation level using MATLAB. Subsequently, fault injection was implemented using Chipwhisperer nano attacking a single byte during the 8$^{th}$ round of AES. The results demonstrated high success rates in recovering the cryptographic key.


**Key Words:** Cybersecurity, Hardware Security, Fault injection attacks, Differential Fault Analysis, DFA

## Contents

Acknowledgements ..... 3
1 Introduction ..... 8
1.1 Hardware Attack Categories ..... 8
1.2 Fault Injection Attacks ..... 9
1.3 Thesis Objective & Outline ..... 10
2 The AES Block Cipher Algorithm ..... 12
2.1 Overview ..... 12
2.2 Block ciphers & encryption ..... 12
2.3 AES & Galois field arithmetic ..... 13
2.4 AES-128 flow ..... 14
2.5 Building blocks of AES ..... 16
2.6 KeyExpansion and Reversing KeyExpansion ..... 18
3 Differential Fault Analysis ..... 21
3.1 Introduction to block cipher security and cryptanalysis ..... 21
3.2 Differential Fault Analysis ..... 21
3.3 Differential Properties of AES ..... 22
3.4 Single Bit DFA against the Last Round of AES-128 ..... 22
3.5 Single Byte DFA against the $9^{th}$ round of AES-128 ..... 23
3.6 Single Byte DFA against the 8th round of AES-128 ..... 26
4 Framework ..... 29
4.1 The main components of our implementation ..... 29
4.2 Target: Victim Firmware ..... 32
4.3 ChipWhisperer API & Jupyter Notebook: The control center ..... 33
4.3.1 Implementation Details ..... 34
5 Analysis Platform & Simulation ..... 38
5.1 Simulation of AES Encryption ..... 38
5.2 Simulation of Faulty AES Encryption ..... 39
5.3 MATLAB Script for Bit-Level DFA ..... 40
5.4 MATLAB Script for Byte-Level DFA ..... 41
5.5 Reverse KeyExpansion Algorithm ..... 42
5.6 Simulation of Bit-Level & Byte-Level DFA Attacks ..... 43
6 DFA Implementation ..... 45
6.1 Fault Injection Campaigns ..... 45
6.1.1 Implementation Details ..... 45
6.1.2 Iterative Fault Injection Process ..... 45
6.1.3 Data Export to MATLAB ..... 47

6.2 Execution of DFA Attack ..... 48
7 Results ..... 50
7.1 Experiment Setting ..... 50
High-Level Algorithm for Experiment ..... 50
7.2 Success Rate ..... 50
7.3 Time Duration ..... 52
8 Conclusions ..... 54
8.1 Key Findings ..... 54
8.2 Implications and Future Directions ..... 54
8.3 Final Thoughts ..... 54
9 Bibliography ..... 55

Notation

| | |
|---|---|
| $\oplus$ | Exclusive OR |
| $d_i$ | The $i^{th}$ byte of the plaintext |
| $k_i$ | The $i^{th}$ byte of the key |
| S(x) | S-Box lookup result of value x |

Glossary

AES Advanced Encryption Standard
XOR Exclusive OR
DFA Differential Fault Analysis

# 1 Introduction

The rapid spread of embedded devices and the emergence of the Internet of Things (IoT) have significantly transformed the technology landscape over the past two decades, bringing about new security challenges. A wide range of devices, from smartphones to critical industrial systems, are now equipped with microprocessors, network connectivity, and Internet access, enabling them to perform diverse tasks such as medical equipment operation, military systems control, industrial control units, and smart home automation.

The increasing prevalence of hardware-based systems has elevated the importance of hardware security as a prominent research area. Security requirements and controls vary according to the specific application of various hardware components. The definition of cybersecurity by NIST as the "Prevention of damage to, protection of, and restoration of computers, electronic communications systems, electronic communications services, wire communication, and electronic communication, including information contained therein, to ensure its availability, integrity, authentication, confidentiality, and nonrepudiation" *(NIST Computer Security Resource Center 2023)* include the safeguard of hardware devices.

However, an in-depth understanding of hardware security requires distinguishing between and software attacks, as well as hardware and software targets. "The Hardware Hacking Handbook" defines software as anything consisting of bits and hardware as anything consisting of atoms *(Woudenberg and O'Flynn 2022)*. Consequently, a hardware attack uses an attack vector consisting of atoms and a software uses an attack vector consisting of bits. Likewise, the targets of these attacks can be either a software or hardware component of a given system.

This thesis aims to explore and evaluate the implementation of Differential Fault Analysis (DFA) attacks against a microcontroller executing an AES encryption algorithm, a hardware attack against a software target. The experimental setup utilizes ChipWhisperer Nano and MATLAB to perform fault injection techniques and analyze the obtained results. The primary objective of this research is to discuss the vulnerabilities of hardware systems to differential fault analysis attacks.

## 1.1 Hardware Attack Categories

As the focus of this thesis is hardware attacks, we should delve into the different types of these attacks. A fundamental categorization of hardware attacks is based on the physical intrusion degree and split into the following three categories *(Sakiyama, Sasaki and Li 2015)*:

- Invasive attacks: Invasive attacks involve physically opening the Integrated Circuit (IC) chip package, granting access to the silicon die through chemical methods. Attackers observe signals directly, making intermediate values transparent using special instruments. These attacks are potent but require expensive equipment and expertise, and the risk of permanent damage to the device is high.
- Semi-invasive attacks: Semi-invasive attacks also necessitate opening the package but do not involve direct contact with internal wires. Attackers alter intermediate values using methods like optical lasers, while these attacks reduce the risk of physical damage compared to invasive methods. Certain fault injection techniques fall into this class.
- Noninvasive attacks: These attacks exploit vulnerabilities without physically accessing the silicon die, making them less intrusive but still effective in compromising security. Attackers need limited contact with IC chip pins. Side-channel analysis and some fault injection attacks are classified as noninvasive methods.

Further categorization is performed according to the direction of the physical information between passive and active attacks. In a passive attack scenario, the attacker is restricted to receiving data from the physical device and uses the information gathered to perform an attack. Side-channel attacks is a typical example of passive attacks, as the attacker may observe the power consumption of the IC chip to perform calculations for extracting confidential information, like the encryption key of a cryptographic algorithm. In active attacks, the attacker disrupts the operations of the target device in order to alter an intermediate value. Fault injection attacks fall under this category.

## 1.2 Fault Injection Attacks

The injection of faults into computer systems serves as a common attack vector in both hardware and software attacks. Unexpected input values are often inserted to identify vulnerabilities in computer software. In the realm of hardware attacks, fault injections involve physical disturbances against a target device, potentially leading to the execution of incorrect instructions or the storage of faulty values in memory. These disturbances may consist of a rapid variation of voltage, temperature, clock frequency, as well as the exposure of the IC chip to optical laser beams or electromagnetic fields. The impact of fault injection attacks varies based on the target. A successful fault injection attack could bypass security mechanisms, such as secure boot processes. When combined with cryptanalysis, it may lead to cryptographic attacks, facilitating the recovery of encryption keys.

Specific fault injection methods are commonly employed due to their precise control over the injection process. For instance, rapidly increasing the temperature beyond the maximum threshold at which a device can operate might induce errors in application execution; however, the impact of this fault cannot be precisely targeted. The precision of a fault injection can be either temporal or spatial. Certain attacks can successfully target a specific strategic moment (temporal precision) or a certain memory register (spatial precision). Consequently, certain methods are predominantly utilized in fault injection attacks, as they offer a satisfactory level of spatial or temporal precision. The main reference for the description of these methods is the Hardware Hacking Handbook *(Woudenberg and O'Flynn 2022)*.

The method of *Clock Fault Injection* is a global attack as a clock glitch can interfere with different components of the microcontroller. Despite its global nature, it offers precise control over timing and is technically simple as noninvasive attack. This technique involves the insertion of additional rising edges into a device’s input clock to disrupt the timing of the timing constraints of the target. This is accomplished through the insertion of too-narrow or too-wide clock edges. This method relies on the condition that the internal core directly utilizes the external clock signal. An important limitation of this attack is that clock glitching is ineffective against devices equipped with internal oscillators or a Phase-locked loop (PLL) to generate a new clock from the external clock signal.

While Clock Fault Injection provides a global impact, other methods offer more localized effects but often require sophisticated equipment and might pose risks to the microcontroller as they fall into the semi-invasive category. *Electromagnetic (EM) Fault Injection*, for instance, involves the use of a strong electromagnetic pulse to cause a fault. A common technique to induce an electromagnetic glitch is to generate a strong electromagnetic pulse, by inducing a changing magnetic field through a wire loop. As per Faraday’s law, this changing magnetic field affects the wires on a chip, causing voltage spikes. These spikes can temporarily flip signal levels from 1 to 0 or vice versa. Achieving a successful EM fault injection demands high precision. Further, when dealing with Package-on-Package technology, the memory die is stacked over the processor die complicating the process. Despite its complexity, EM glitching offers a high level of control over specific registers, making it an excellent method for targeted attacks.

*Optical Fault Injection* stands out as a technique known for its exceptional spatial and temporal precision. This method involves the use of laser beams or other intense light sources to induce faults. When exposed to an optical pulse, transistors may switch as semiconductors are sensitive to light. However, the complexity of this attack increases significantly with the decapsulation of the target IC chip using acid, increasing the complexity of this attack and the overall cost of the required equipment.

*Voltage Fault Injection* emerges as our method of choice due to its excellent temporal precision and practicality when compared to Clock Fault Injection. It also requires significantly less complex equipment than EM and Optical Fault Injection methods. This technique involves purposefully manipulating a chip's power supply via momentarily underpowering the target device or inserting a positive or negative power spike. This manipulation disrupts the normal functioning of the chip during critical operations.

A chip's flip-flops require stable inputs before and after clock edges to capture values accurately. By changing the voltage, the chip's signals can change faster, potentially causing hold time violations if signals change before the required hold time is met, or setup time violations if signals change too close to the next clock edge.

In voltage fault injection, the objective is to create extremely brief voltage changes, typically ranging from less than a nanosecond up to a few nanoseconds, deep within the chip at the level of individual transistors. These changes must be of sufficient duration to influence specific areas of the circuitry effectively. It is important to note that these voltage variations occur at the transistor's power supply, located deep within the chip. Consequently, they must endure long enough to impact the targeted components effectively. Both the power supply and clock networks extend across the entire chip, allowing a voltage glitch to potentially affect multiple transistors simultaneously.

Within a chip, the power supply network incorporates decoupling capacitors designed to minimize fluctuations caused by a switching power supply and noise from the PCB. While these capacitors optimize regular chip functionality, they also influence deliberate voltage fluctuations introduced during fault injection experiments.

Generating voltage glitches employs diverse methods. One approach utilizes a programmable signal generator, where the generator's output passes through a voltage buffer before powering the target device. Another method involves switching between two power sources: the operating voltage and the fault voltage. Lastly, the crowbar method effectively induces voltage variations by shorting the chip's supplied operating voltage causing a spike.

The crowbar method to inject voltage glitches offer limited control over the fault voltage as it momentarily shorts the operating supply voltage to 0 V. However, it is the least complex method to implement and as it introduces large spikes to the power supply, it is an effective method to inject faults. The crowbar method can be implemented using a high-power or a low-power MOSFET depending on the target device. This method is the method of choice for our implementation because of its effectiveness and low cost.

## 1.3 Thesis Objective & Outline

The objective of this thesis, as stated in its title, is the "Evaluation of the security of embedded systems against fault injection attacks". Towards this direction we have implemented a voltage fault injection attack using Chipwhisperer nano. The ChipWhisperer nano has been used as a platform to generate crowbar voltage fault injection to attack the target microcontroller which is embedded in the device. The targeted firmware is AES-128 encryption and the implemented attack was Differential Fault Analysis (DFA). As *(O' Flynn 2016)* states "having a practical method of injecting faults into an embedded computer is of great importance to both these areas of research: understanding the vulnerability of systems to fault injection attacks, and validating design of fault-tolerant computing systems". Our direction towards the implementation of a DFA attack demonstrates the practicality and impact of fault injection attacks.

DFA can be described as specific type of fault injection technique that focuses on extracting secret information, such as cryptographic keys, from targeted hardware systems. To describe this implementation and provide the theoretical for this implementation, this thesis is divided in nine chapters, as follows.

- In Chapter 1, titled "Introduction", hardware attacks, with a specific focus on fault injection attacks, are discussed and the objective of this thesis is stated. This chapter also provides an overview of the thesis structure, outlining the organization of the subsequent chapters.
- Chapter 2, titled "The AES Block Cipher Algorithm", provides an overview of the fundamentals of the Advanced Encryption Standard (AES). The purpose of this chapter is a theoretical introduction to AES as a prerequisite for the understanding of the attacks.
- Chapter 3, titled "Differential Fault Analysis", focuses on a theoretical presentation of Differential Fault Analysis and introduces DFA attacks against AES-128 on the bit-level and byte-level.
- Chapter 4, titled "Framework", introduces the main components of our implementation. The focus of this chapter is Chipwhisperer and its use to implement our attack.
- Chapter 5, titled "Analysis Platform & Simulation", presents the MATLAB scripts developed for the simulation of fault injection against AES-128, as well as the scripts developed for the DFA attack itself.

- Chapter 6, titled “Experimental DFA”, presents how we orchestrate the different components of our implementation to perform a DFA attack using Chipwhisperer and MATLAB.
- Chapter 7, titled "Results”, presents an evaluation of the implemented fault injection platform for byte-level Differential Fault Analysis (DFA) attacks on a microprocessor executing AES. Key aspects of this chapter include the success rate of the DFA attacks and a comparison of time durations between actual and simulated data for both Byte-level and Bit-level DFA attacks.
- Chapter 8, titled “Conclusions”, discusses the key findings of this study and future directions.
- Chapter 9, titled “Bibliography”, lists the references for this study.

In summary, this introductory chapter has outlined the objectives of our thesis, as well as the significance of hardware attacks against modern embedded devices. The subsequent chapters will delve into deeper into the theoretical foundations, our practical implementation and our findings to explore this complex and exciting realm.

# 2 The AES Block Cipher Algorithm

## 2.1 Overview

The Advanced Encryption Standard (AES) is a specification of the Rijndael algorithm using data blocks of 128 bits and keys of 128, 192 or 256 bits length *(Dworkin 2001)*. The new federal encryption standard was publicly announced in the Federal Information Processing Standards Publication 197 on the 26th of November 2001 by the National Institute of Standard and Technology (NIST). The Rijndael algorithm, designed by the Belgian mathematicians Vincent Rijmen and Joan Daemen, won the competition which was held by NIST from 1997 to 2000. It provided the core algorithm for AES which would become the successor of the Data Encryption Standard (DES), the prior federal standard from 1979 to 2005 *(Aumasson 2018)*. AES is not only a federal encryption standard in the US, but also the de facto modern worldwide encryption standard.

## 2.2 Block ciphers & encryption

AES, as its predecessor DES, is a symmetric block cipher. A symmetric block cipher consists of an encryption and a decryption algorithm processing data sequence in blocks of a standard size which are encrypted and decrypted using the same cryptographic key. As it has been noted above, AES comes using three different key sizes. The number of rounds of AES depend on the key length. For a 128-bit, 10 rounds are iterated, for 196 bits 12 rounds and for 256 bits 14 rounds. However, the focus of this dissertation will be restricted to the AES-128, as NIST called the AES flavor using a 128-bit key.

A brief presentation of what an encryption algorithm (E) could be described, in general terms, as a function taking as input a key (K) and a plaintext block (P) producing a ciphertext block (C), while the decryption algorithm (D) is the exact inverse procedure as it consists of a function taking as input a key (K) and ciphertext block (C) to produce the original plaintext (P). It is important to also note that every intermediate value of plaintext (P) during the encryption process is called the state (S). These operations are depicted below in the form of block diagrams:

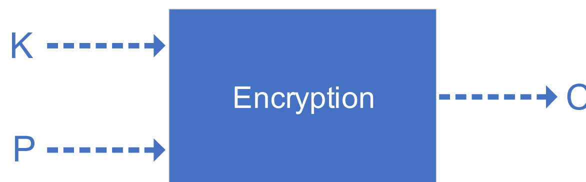



**Figure 2.1 Block diagram for the encryption process**

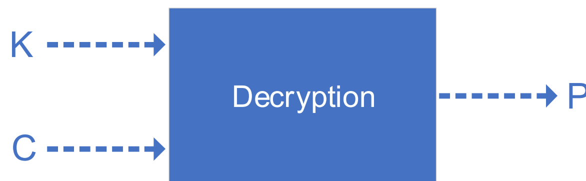



**Figure 2.2 Block diagram for the de process**

The need for block cipher security is achieved, if an attacker using cryptanalytic attacks is unable to determine the plaintext having access to the ciphertext. Likewise, the cryptographic key should not be possible to be determined having access to both the ciphertext and the plaintext. As a result, a block cipher should be a deterministic permutation which appears random and to achieve this goal confusion and diffusion are two key factors for success. Confusion suggests that non-linear transformations are applied to the input and diffusion that these transformations are equally dependent on all bytes of the input. More specifically, in block ciphers these operations are

implemented through a combination of operation in round called substitution-permutation networks (SPNs).

## 2.3 AES & Galois field arithmetic

Unlike its predecessor, DES, AES is not a Feistel cipher. Rijmen and Daemen chose to use Galois field arithmetic for the implementation of their algorithm. A brief introductory remark on Galois fields is useful for a deeper understanding of the inner mechanism of the encryption and decryption process.

A Galois field, or finite field, is a finite set of elements where the addition, subtraction, multiplication and inversion operations can be applied, and the output of the operation also belongs to the set. As we read in *(Paar and Pelzl 2010)* a field F is a set of elements with the following properties:

- All elements of F form an additive group with the group “+” and the neutral element 0
- All elements of F except 0 form a multiplicative group with the group “x” and the neutral element 1
- When the two-group operations are mixed, the distributive law holds

Given that the number of elements of a field is finite, we have a finite field or Galois field and this is the type of fields which are mostly used in cryptography. The number of elements defines the order of the field.

The fundamental theorem of existence of finite fields of order m is that a field with order m exists if m is a prime power. This suggests that m can be expressed as: $m = p^n, n \in Z^+, \ p \in P$. The prime number p is the characteristic of the field.

For example, the finite field used for AES is of order 256 and 256 can be expressed as $2^8$. According to the fundamental theorem for Galois fields, the finite field exists given that 2 is a prime number and 8 is positive integer. On the contrary, an example of the case of a nonexistent field would be a finite field of order 56. The reason is that 56 factored to prime number would be expressed as $2^3 \times 7$ -not a power of a prime.

A special type of finite fields are fields where n is equal to 1 and they are called prime fields. Prime field arithmetic provides the basis for computations on Galois Fields.

These four main operations are defined as following:

$Let\ a, b \ \in GF(p) = \{0, 1, \ldots, p-1\}$

Addition: $a + b = c \ mod \ p$

Subtraction: $a - b = d \ mod \ p$

Multiplication: $a \cdot b = e \ mod \ p$

The definition of inversion for $a \in GF(p)$ is $a^{-1} \cdot a = 1$. However, its computation is complex and can be calculated using the extended Euclidean algorithm.

The smallest possible prime field is GF(2)={0,1} and it has applications especially in computer science and cryptography. The possible operations for GF(2) are provided below as an example.

Addition:

$0 + 0 = 0 \ mod \ 2 = 0$

$1 + 0 = 0 + 1 = 1 \ mod \ 2 = 1$

$1 + 1 = 2 \ mod \ 2 = 0$

Subtraction:

$0 - 0 = 0 \ mod \ 2 = 0$

$1 - 0 = 1 \ mod \ 2 = 1$

$0 - 1 = (-1) mod \ 2 = 1$

$1 - 1 = 0 \ mod \ 2 = 0$

Multiplication:

$0 \cdot 0 = \ 0 \cdot 1 = 1 \cdot 0 = 0 \ mod \ 2 = 0$

$1 \cdot 1 = 1 \ mod \ 2 = 1$

Inversion:

$1^{-1} \cdot 1 = 1$

It is important to note that the addition and subtraction operations are equal.

The Galois fields $GF(p^n)$ where $n > 0$ is called extension field. As mentioned above, AES uses the Galois field $\mathrm{GF}(2^8)$. The elements of $\mathrm{GF}(2^m)$, including $\mathrm{GF}(2^8)$, are polynomials. In the special case of AES, the polynomial can be expressed as follows:

$a_7 \cdot x^7 + a_6 \cdot x^6 + a_5 \cdot x^5 + a_4 \cdot x^4 + a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x + a_0 = A(x) \mid a_i \in GF(2)$

This polynomial representation is suitable for AES as the eight coefficients can be expressed as the bit values of one byte.

For the addition and subtraction operations in $\mathrm{GF}(2^8)$, the polynomials are added or subtracted like in the case of regular polynomials, where the coefficients are computed in GF(2). Example given, if $A(x) = x^3 + x^2 + 1$ and $B(x) = x^4 + x^2 + 1$, $A(x) + B(x) = (0+1) \cdot x^4 + (1+0) \cdot x^3 + (1+1) \cdot x^2 + (1+1) = x^4 + x^3$ .

The multiplication is not as simple, involving only regular polynomial multiplication and GF(2) operations to calculate the value of coefficients. The evident reason explaining why this is not feasible is the fact that the product of two polynomial would not belong to the field. Example given, for two polynomials in $GF(2^8)$: $A(x) = x^7 + x$ and $B(x) = x^4 + x^2 + 1$, their product using regular arithmetic would be $A(x) \cdot B(x) = (x^7 + x)(x^4 + x^2 + 1) = x^{11} + x^9 + x^7 + x^5 + x^3 + x$ . However, this polynomial does not belong to $GF(2^8)$, given that it includes $x^{11}$ and $x^9$. This “regular” product $A(x) \cdot B(x) = C(x)'$ is called prime product which should be reduced modulo a polynomial that behaves like a prime. The polynomials used for reduction cannot be factored and are called irreducible polynomials. It is important to note that unlike prime fields and their characteristic prime number, extensions field may have more than one irreducible polynomial. As a result, in order to perform a multiplication in a given context the irreducible multiplication should be specified. The definition of multiplication in extension Galois fields is $C(x) = A(x) \cdot B(x) mod P(x)$.

In the case of AES, the irreducible polynomial is $P(x) = x^8 + x^4 + x^3 + x + 1$. Using the example polynomials given above, the product would be:

$$C(x) = (x^{11} + x^9 + x^7 + x^5 + x^3 + x) mod (x^8 + x^4 + x^3 + x + 1).$$

The division of the polynomials is described below in detail in order to extract the remainder.

$(x^{11} + x^9 + x^7 + x^5 + x^3 + x) : (x^8 + x^4 + x^3 + x + 1) = x^3 + x$

$+(x^{11} + x^7 + x^6 + x^4 + x^3)$

---

$x^9 + x^6 + x^5 + x^4 + x$

$+(x^9 + x^5 + x^4 + x^2 + x)$

---

$x^6 + x^2$

$\rightarrow C(x) = x^6 + x^2$

Concerning the inverse $A^{-1}(x)$ of an element $A(x) \in GF(2^m)$, it must satisfy the following condition $A(x) \cdot A^{-1}(x) = 1 mod P(x)$. However, its computation is again a complex procedure using the extended Euclidean algorithm and it will not be described here.

This rather extended note on finite field arithmetic is a prerequisite for the presentation of the inner mechanism of AES.

## 2.4 AES-128 flow

The initialization of the encryption process starts with the KeyExpansion function. This function creates 10 rounds keys of 16-bytes length for each round of encryption which are calculated as a combination of substitutions using the SubBytes function and XOR operations. A significant

property of the AES scheduling algorithm is that the initial encryption key, as well as every round key, can be recovered from the value of any given round key.

As it is visually presented in the schematic, AES-128 is consisted of ten rounds of encryption plus one initialization round. The initialization round, or round zero, is the shortest one, as only a XOR operation is applied between the initial encryption key and the plaintext. The output of the initial round, the state of the encryption, is provided to the first round of AES. The first round of AES includes four transformations (SubBytes, ShiftRows, MixColumns, AddRoundKey) and it is identical for the first nine rounds. It is the final round of AES, the tenth one, that differs. It includes only three permutations as the MixColumns permutation is omitted. The main reason for this design omitting MixColumns is the achievement of symmetry between the AES encryption and decryption algorithms, as the designers of AES supported that no improvement of the resistance of the algorithm against attacks would originate from the inclusion of the MixColumns permutation at this point *(Daemen and Rijmen 2002)*. Criticism of this argument has been risen *(Dunkelman and Keller 2010)* but this debate exceeds the scope of this thesis.

The substitutions and permutations of AES are applied to bytes of data, which are treated as 4X4 matrix -a two-dimensional array of bytes. The SPN structure for AES-128 is illustrated in the following schematic.

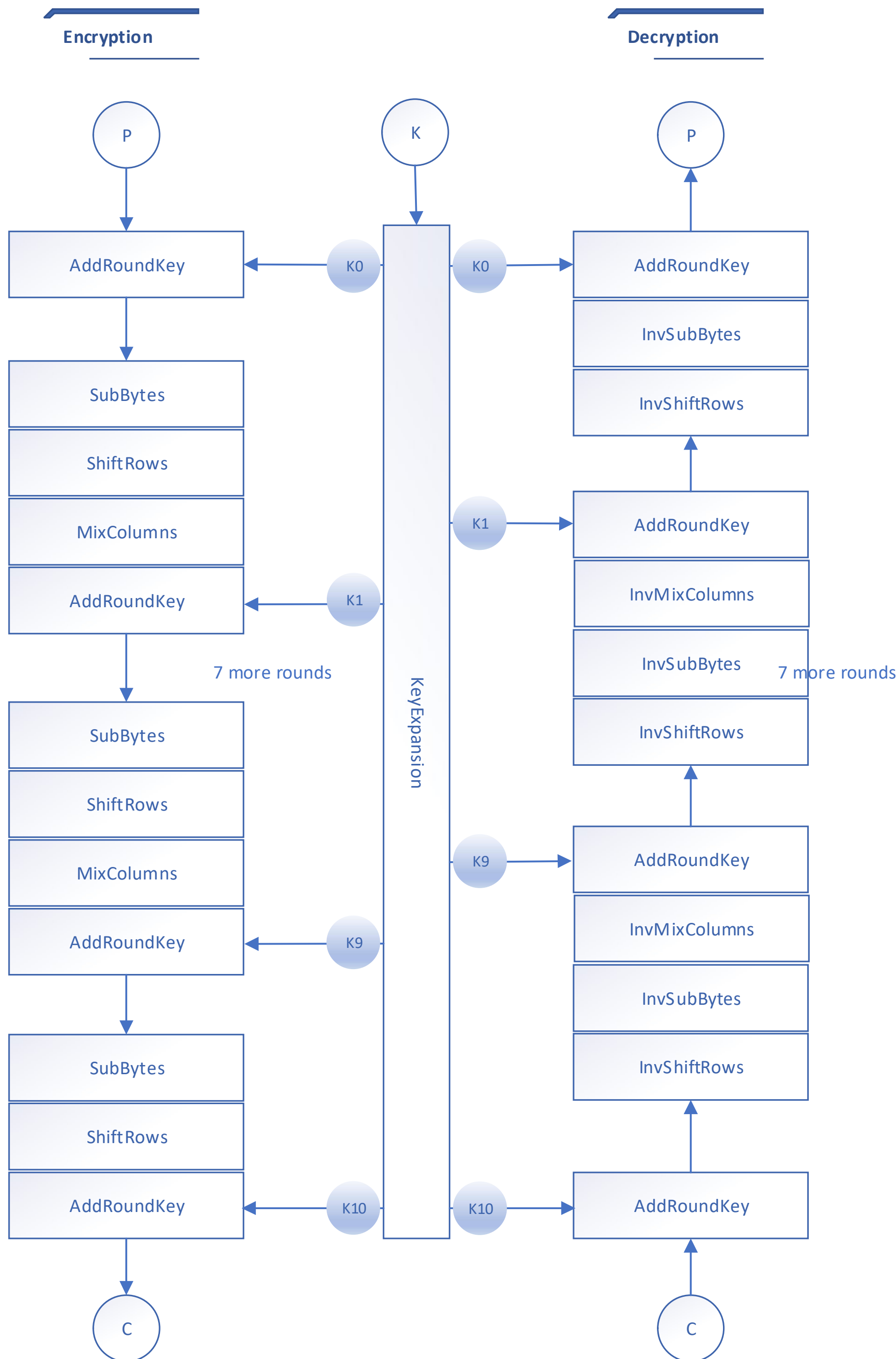



**Figure 2.3 AES encryption and decryption algorithm SPN**

## 2.5 Building blocks of AES

The **SubBytes** step consists of the only non-linear transformation applied to the plaintext in AES. Unlikely to what happens in the case of DES, the same S-box is applied to all 16 bytes of the state on all ten rounds, apart from the initial one. The design criteria for the SubBytes operation are non-linearity and algebraic complexity.

The S-box may be easily understood as a lookup table of 256 elements given some input $A_i \in GF(2^8)$: $SubBytes(A_i) = B_i$. The AES S-box is provided in the table below.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 63 | 7C | 77 | 7B | F2 | 6B | 6F | C5 | 30 | 1 | 67 | 2B | FE | D7 | AB | 76 |
| 1 | CA | 82 | C9 | 7D | FA | 59 | 47 | F0 | AD | D4 | A2 | AF | 9C | A4 | 72 | C0 |
| 2 | B7 | FD | 93 | 26 | 36 | 3F | F7 | C C | 34 | A5 | E5 | F1 | 71 | D8 | 31 | 15 |
| 3 | 4 | C7 | 23 | C3 | 18 | 96 | 5 | 9A | 7 | 12 | 80 | E2 | EB | 27 | B2 | 75 |
| 4 | 9 | 83 | 2C | 1A | 1B | 6E | 5A | A0 | 52 | 3B | D6 | B3 | 29 | E3 | 2F | 84 |
| 5 | 53 | D1 | 0 | ED | 20 | FC | B1 | 5B | 6A | CB | BE | 39 | 4A | 4C | 58 | CF |
| 6 | D0 | EF | AA | FB | 43 | 4D | 33 | 85 | 45 | F9 | 2 | 7F | 50 | 3C | 9F | A8 |
| 7 | 51 | A3 | 40 | 8F | 92 | 9D | 38 | F5 | BC | B6 | DA | 21 | 10 | FF | F3 | D2 |
| 8 | C D | 0C | 13 | EC | 5F | 97 | 44 | 17 | C4 | A7 | 7E | 3D | 64 | 5D | 19 | 73 |
| 9 | 60 | 81 | 4F | D C | 22 | 2A | 90 | 88 | 46 | EE | B8 | 14 | DE | 5E | 0B | DB |
| A | E0 | 32 | 3A | 0A | 49 | 6 | 24 | 5C | C2 | D3 | AC | 62 | 91 | 95 | E4 | 79 |
| B | E7 | C8 | 37 | 6D | 8D | D5 | 4E | A9 | 6C | 56 | F4 | EA | 65 | 7A | AE | 8 |
| C | BA | 78 | 25 | 2E | 1C | A6 | B4 | C6 | E8 | D D | 74 | 1F | 4B | BD | 8B | 8A |
| D | 70 | 3E | B5 | 66 | 48 | 3 | F6 | 0E | 61 | 35 | 57 | B9 | 86 | C1 | 1D | 9E |
| E | E1 | F8 | 98 | 11 | 69 | D9 | 8E | 94 | 9B | 1E | 87 | E9 | CE | 55 | 28 | DF |
| F | 8C | A1 | 89 | 0D | BF | E6 | 42 | 68 | 41 | 99 | 2D | 0F | B0 | 54 | BB | 16 |

To illustrate how the lookup table for the AES S-box functions, an element $A_i = F1_{16} = (x, y)$ in hexadecimal representation could be considered. The first hex digit is x and the second hex digit is y. The x and y are the coordinates, representing the row and column respectively, to detect the matching value for the S-box transformation of $SubBytes(F1_{16}) = A1_{16}$.

However, this representation of the SubBytes transformation as a lookup table of hexadecimal values ignores the inner mechanism of the calculation. According to the design criteria the S-box used in AES is based on the inversion function in GF($2^8$) and an affine mapping operation, as it is illustrated in the following figure.

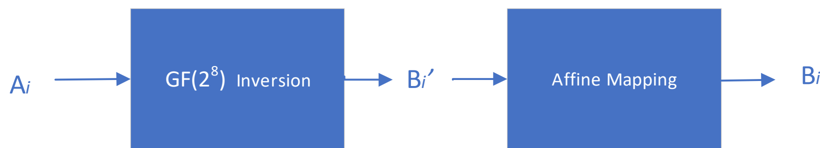



**Figure 2.4 Mathematical description of the AES S-box**

The GF($2^8$) inversion uses the following function

$$g : a \rightarrow b' = a^{-1}$$

Example given consider an element $A_i = (0100\,0101)_2 = x^6 + x^2 + 1$

In order to calculate the SubBytes transformation at this point, the first step is to find the inverse function of $A_i$ using the Extended Euclidean algorithm.

$$B_i' = A_i^{-1} = (110001)_2 = x^5 + x^4 + 1$$

$$A_i \cdot B_i' = 1 mod P(x) \leftrightarrow (x^6 + x^2 + 1) \cdot (x^5 + x^4 + 1) = 1 mod(x^8 + x^4 + x^3 + x + 1)$$

Given that the output of the SubBytes is the polynomial $B(x) = b_7 \cdot x^7 + b_6 \cdot x^6 + b_5 \cdot x^5 + b_4 \cdot x^4 + b_3 \cdot x^3 + b_2 \cdot x^2 + b_1 \cdot x + b_0$ and the output of the inversion is $B(x)' = b_7' \cdot x^7 + b_6' \cdot x^6 + b_5' \cdot$

$x^5 + +b_4' \cdot x^4 + b_3' \cdot x^3 + b_2' \cdot x^2 + b_1' \cdot x + b_0'$ , the output of the affine mapping is calculated based on the following transformation.

$$\begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} b_7' \\ b_6' \\ b_5' \\ b_4' \\ b_3' \\ b_2' \\ b_1' \\ b_0' \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

The coefficient of $B(x)$ are calculated using the regular operations of linear algebra to multiplicate matrices and $GF(2)$ operations for the multiplication addition of coefficients.

For instance, the calculation of the $b_0$ coefficient is as follows: $b_0 = (1 \cdot b_7' + 1 \cdot b_6' + 1 \cdot b_5' + 1 \cdot b_4' + 0 \cdot b_3' + +0 \cdot b_2' + 0 \cdot b_1' + 1 \cdot b_0') \oplus 0$

The ShiftRows and MixColumns permutations guarantee the diffusion property of AES.

The understanding of the **ShiftRows** step demands to represent the state as 4x4 matrix.

| | | | |
|---|---|---|---|
| $B_0$ | $B_4$ | $B_8$ | $B_{12}$ |
| $B_1$ | $B_5$ | $B_9$ | $B_{13}$ |
| $B_2$ | $B_6$ | $B_{10}$ | $B_{14}$ |
| $B_3$ | $B_7$ | $B_{11}$ | $B_{15}$ |

During the ShiftRows permutation the elements of the state matrix are shifted cyclically. The first row of the state matrix is not shifted, the second row of the column is shifted one position to the left, the third row two positions and the third is shifted three positions.

| | | | |
|---|---|---|---|
| $B_0$ | $B_4$ | $B_8$ | $B_{12}$ |
| $B_5$ | $B_9$ | $B_{13}$ | $B_1$ |
| $B_{10}$ | $B_{14}$ | $B_2$ | $B_6$ |
| $B_{15}$ | $B_3$ | $B_7$ | $B_{11}$ |

**MixColumns** reassures that a minor bit flip in any bit of the data path will have the maximum impact affecting the value of all four bytes.

$$\begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} B_0 \\ B_5 \\ B_{10} \\ B_{15} \end{bmatrix}$$

Given that all $B_i$ and $C_i$ representations are bytes the

$$C_0 = 02 \cdot B_0 + 03 \cdot B_0 + 01 \cdot B_{10} + 01 \cdot B_{15}$$

The multiplication and addition operations follow the $GF(2^8)$ operation rules.

Please note that the hexadecimal values of the MixColumns permutation matrix are hexadecimal representations of $GF(2^8)$ polynomials.

$01 = (0000\ 0001)_2 \leftrightarrow 1$

$02 = (0000\ 0010)_2 \leftrightarrow x$

$03 = (0000\ 0011)_2 \leftrightarrow x + 1$

## 2.6 KeyExpansion and Reversing KeyExpansion

From the last subkey to the initial key

The DFA attacks described above lead to the successful retrieval of the 10th round encryption key for AES-128. However, its importance would be rather limited if it was not possible to retrieve the original encryption key through reversing the AES KeyExpansion operation.

According to *(Dusart, Letourneux and Vivolo 2002)* we may denote the jth byte of the nth round key as $K_n[j]$ and as w[i] the output of the KeyExpansion where:

$$K_n = w[N_k \cdot n], w[N_k \cdot n], \dots, w[N_k \cdot n + N_b - 1]$$

Or given that $N_k$=4, $N_b$=4 and $N_r$=10 for AES-128:

$$K_n = w[4 \cdot n], w[4 \cdot n + 1], \dots, w[4 \cdot n + 15]$$

Thus, the table *w* will have 176 elements for AES-128.

For $i \in [0, N_b * (N_r + 1) - N_k,\ where\ i \neq 0\ mod\ N_k$:

$$w[i] = w[i - N_k] \oplus w[i - 1]$$
$$\leftrightarrow w[i - N_k] = w[i] \oplus w[i - 1]$$

$$\rightarrow w[i] = w[i + N_k] \oplus w[i + N_k - 1]$$

For $i = 0\ mod\ N_k$:

$$w[i] = w[i - N_k] \oplus SubWord(RotWord([i - 1] \oplus Rcon[i/N_k]$$
$$\leftrightarrow w[i - N_k] = w[i] \oplus SubWord(RotWord([i - 1] \oplus Rcon[i/N_k]$$

$$\rightarrow w[i] = w[i + N_k] \oplus SubWord(RotWord([i + N_k - 1] \oplus Rcon[(i + N_k)/N_k]$$

Likewise, for AES-128 specifically:

For $i \in [0, 40),\ where\ i \neq 0\ mod\ 4$:

$$w[i] = w[i + 4] \oplus w[i + 3]$$

For $i = 0\ mod\ 4$:

$$w[i] = w[i + 4] \oplus SubWord(RotWord([i + 3)])) \oplus Rcon[(i + 4)/4]$$

It is worth noting that RotWord is a permutation of four bytes which are cyclically permuted according to the following figure:

SubWord is the SubBytes substitution applied to each of the four bytes of the word and Rcon is the following constant matrix.

| | | | |
|---|---|---|---|
| 01 | 00 | 00 | 00 |
| 02 | 00 | 00 | 00 |
| 04 | 00 | 00 | 00 |
| 08 | 00 | 00 | 00 |
| 10 | 00 | 00 | 00 |
| 20 | 00 | 00 | 00 |

| | | | |
|---|---|---|---|
| 40 | 00 | 00 | 00 |
| 80 | 00 | 00 | 00 |
| 1B | 00 | 00 | 00 |
| 36 | 00 | 00 | 00 |

Which is derived from the following formula:

$$rcon_i = [rc_i \ \ 00\,00\,00]$$

Where $rc_i$ is defined as:

$$rc_i = \begin{cases} 1\,, if\ i = 1 \\ 2 \cdot rc_{i-1}\,, if\ i > 1\ and\ rc_{i-1} < 80 \\ 2 \cdot rc_{i-1} \oplus 11B\ and\ rc_{i-1} \geq 80 \end{cases}$$

The numbers above are given in hexadecimal format.

Consequently, it is possible to derive the elements of the expanded key tables using the last ones and reverse the KeyExpansion process.

# 3 Differential Fault Analysis

## 3.1 Introduction to block cipher security and cryptanalysis

Apart from a brief high-level remark in the previous chapter on block cipher security, a more precise definition of block cipher security requires the description of a series of relevant notions. More precisely, there are three key security classes according to *(Sakiyama, Sasaki and Li 2016)*: Key recover resistance, Plaintext recovery resistance, Indistinguishability. Key recover resistance is the ability to resist the recovery of the key value for any given choice of the key value. Plaintext recovery resistance is the ability to resist the recovery of the plaintext value for any given key value using the cipher text value. Indistinguishability is the property not allowing to distinguish the encoding process from a random permutation given any key value, disabling an attacker to calculate any valid plaintext-ciphertext pair without the knowledge of the key. As a result, the key recovery resistance is the most important security property because if the key is recovered, both plaintext recovery resistance and indistinguishability resistance should be also considered broken.

A native design property of block ciphers renders them vulnerable to generic attacks: brute force attacks and dictionary attacks. This design property is the use of keys and blocks of fixed size. A brute force attack can be defined as the attack through the iteration of all $2^N$ possibilities to recover the N-byte key. A dictionary attack, or codebook attack, for a block cipher of block size B demands the creation of a codebook of size $2^B$, a dictionary of all plaintexts and respective ciphertexts. This attack can recover the plaintext through the dictionary without having knowledge of the key.

It is evident that the generic attacks are costly with respect to the required data, time and memory. The time requirement for a brute-force attack can be defined as $2^N$ representing the time required to iterate all the possible keys. The data and memory requirement for the generic dictionary attack can be defined as $2^B$ representing the stored data and processed data required to encode all the possible plaintext blocks.

Attacks which require less time than $2^N$ or less data (or memory) than $2^B$ are shortcut attacks. The existence of such attacks consists of a critical security flaw for a block cipher. Differential cryptanalysis, impossible differential cryptanalysis and integral cryptanalysis are prominent cryptanalytic approaches for the discovery of shortcut attacks.

It is important to note that the discussed block cipher, AES, is considered safe against cryptanalytic attacks up to this day. However, differential cryptanalysis consists of the foundation for Differential Fault Analysis (DFA), an attack which can be successfully implemented against AES. Detailed description of differential analysis or cryptanalysis in general is out of the scope of this work. Nevertheless, it would be enlightening to mention that differential analysis is related to cryptanalytic techniques based on the difference of values of ciphertexts. A difference of two values is defined as a XOR of these values and is also important in differential fault analysis.

## 3.2 Differential Fault Analysis

Unlike cryptanalysis, fault analysis, as well as side-channel analysis, do not solely assume knowledge of the input and output of the cryptographic operation of the block cipher. In addition, hardware attacks do not depend on native flaws of the cryptographic algorithm. They depend on the hardware implementation which executes the given block cipher cryptographic operation and assume either knowledge of intermediate values of the cryptographic operation (side-channel analysis) or the possibility of intervention to the output of the operation (fault analysis).

This thesis focuses on noninvasive fault analysis and more specifically on differential fault analysis, as it is stated on the title of the current subject and the introduction. An attacker using fault analysis would attempt to disturb the cryptographic operation to inject a fault and calculate the encryption key through the comparison of the faultless and faulty values.

An important parameter concerning DFA is the adopted fault model used for the attack. The fault model describes the extent of the affected area by the fault injection. As a result, the fault model may fall under one of the following categories:

- Single bit or 1-bit fault model: The injected fault is localized to a single bit. This fault model is not realistic in practice, but it is of theoretical interest.
- Single byte or random byte fault model: The injected fault is propagated to multiple bits belonging to a single byte. This is a realistic scenario, and it is associated with powerful DFA techniques. A subclass of this fault model may also allow the attacker to control the position of the byte where the fault is injected to.
- Multiple byte fault model: The injected fault is spread to multiple bytes -more than one. It is assumed that the precision of the fault injection is more restricted. It is the most realistic scenario.

The byte or bytes which have been altered because of the fault injection are also referred as active bytes.

## 3.3 Differential Properties of AES

Supposing that an input to a given round of AES is X, the output from the S-Box can be expressed as $S(X)$. If a fault is injected before the S-Box, the input can be described as $X \oplus \varepsilon$ and the output of S-box as $S(X \oplus \varepsilon)$. Consequently, the difference between the faultless and faulty values can be expressed by the following equation:

$$S(X) \oplus S(X \oplus \varepsilon) = \delta$$

The association between ε and δ is non-linear as it is illustrated by the equation above. However, according to *(Nyberg 1993)*, this equation may have 0, 2 or 4 solutions for $X$ given a ε and δ. For a given ε, 126 out of 256 values of δ lead to 2 solutions for $X$, one value of δ leads to 4 solutions and the remaining 129 have no solution for $X$. As a result, the average number of solutions per equation is 1. This property of AES is crucial for the success of DFA attacks.

## 3.4 Single Bit DFA against the Last Round of AES-128

The most intuitively understandable case as an introduction to Differential Fault Analysis is the one of a single bit fault injected to the last round of AES-128. More precisely, let us assume that the fault is injected after the AddRoundKey phase of the 9$^{th}$ round causing a bit flip to the input value inserted in the 10$^{th}$ round S-Box. The fault is illustrated on Figure 3.1 with the symbol of a lightning supposing that the value of one single bit of a single byte of the state has been altered. It is important to note that the attacker has knowledge of both the fault-free (C) and faulty ciphertext (C*). Given the fact that the MixColumns permutation is omitted in the last of AES, the discovery of the byte where the fault has been injected would become obvious through comparing C and C*. The initial value of the injected state byte can be represented as $x_{i,j}$ and the same value after the injection as $x_{i,j} \oplus \varepsilon$.

The fault-free ciphertext can be represented as:

$$C = SR\big(S(X)\big) \oplus K_{10}$$

The faulty ciphertext can be represented as:

$$C^{*} = SR\big(S(X \oplus \varepsilon)\big) \oplus K_{10}$$

If the fault was injected to the byte positioned in the i$^{th}$ and j$^{th}$ column the fault will be transferred to the i$^{th}$ and $l$ $^{th}$ byte of the ciphertext because of the 10 round ShiftRows operation. Let us note

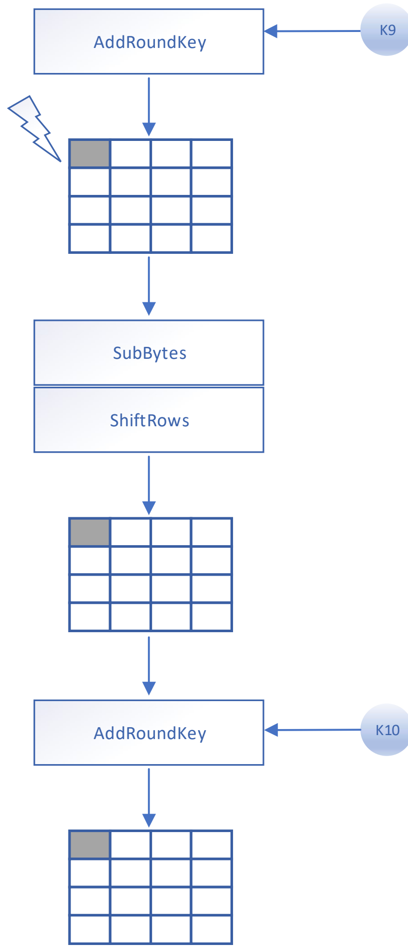



Figure 3.1 Bit-level DFA

that $l = (j - i)\, mod\, 4$, and as a result the fault inserted in the first byte of the state is not propagated to any other byte in Figure 3.1, because $i = j = 0 \rightarrow l = 0$.

The following equation describes the difference between C and C* for the fault-injected byte -given that $K_{10} \oplus K_{10} = 0$:

$$C_{i,l} \oplus C^*_{i,l} = S(x_{i,j}) \oplus S(x_{i,j} \oplus \varepsilon)$$

Given that the value $C_{i,l} \oplus C^*_{i,l}$ is known according to our assumption and ε corresponds to the position of the bit where the fault was injected, $x_{i,j}$ can deduced. The value of ε is considered known and even if it is not, there are only eight possible values for ε. Besides, it was pointed out in the previous section the average number of solutions for the equation is one -according to the differential properties of AES. Consequently, $x_{i,j}$ can be calculated and the $K^{10}_{i,l}$, the subkey byte used for the AddRoundKey operation of the $10^{th}$ round, can be also easily calculated.

This attack is not feasible in practice, because it is very difficult to restrict a fault to flip a single bit of the state, but it is important as a proof of concept from a theoretical point of view to introduce more complex attacks.

## 3.5 Single Byte DFA against the 9th round of AES-128

More realistic scenarios of DFA attacks against AES-128 are based on the single-byte fault model. The fault injected is restricted to one or more bit of a single byte of the state matrix. One interesting case of such an attack involves the alteration of value of one byte during the penultimate round of AES-128, namely the $9^{th}$ round.

A successful attack at this case requires the utilization of the differential properties of both the S-box and the MixColumns operations. The propagation of active bytes during the MixColumns consists of an important difference in comparison with the single-bit fault during the last round of the AES and explains the higher complexity of the attack. One active byte at the input of MixColumns will be propagated to all four bytes of the same column of the matrix state, given the 4x4 matrix multiplication which takes place during the MixColumns operation.

Assuming the existence of an active byte at the first byte of the state matrix at the input of the $9^{th}$ round. If the active byte at the input of MixColumns is represented as $f$, the difference matching the initial active byte will be $2f, f, f, 3f$ as the initial active byte will be diffused to four rows, creating four active bytes. The multipliers 2, 1, 1 and 3 are the values of the first row of the MixColumns matrix. However, the 4 bytes difference will be converted during the non-linear bijective operation of SubBytes to $f_0$, $f_1$, $f_3$ and $f_4$. Consequently, the four bytes difference will be transposed due to the ShiftRows operation. The AddRoundKey operation of the last round will only apply an XOR operation between each byte of the state matrix -including the active bytes-

and the value of the Round Key. A graphical illustration of the fault propagation is presented on Figure 3.2.

The following four equations describe the difference between the fault-free (C) and faulty ciphertext (C*). It is important to highlight that the adversary has access to both the faulty and fault-free ciphertexts, so the following system of 4 equations contains 5 unknown variables.

$$2f = S^{-1}(C_{0,0} \oplus K_{0,0}^{10}) \oplus S^{-1}(C_{0,0}^{*} \oplus K_{0,0}^{10})$$
$$f = S^{-1}(C_{1,3} \oplus K_{1,3}^{10}) \oplus S^{-1}(C_{1,3}^{*} \oplus K_{1,3}^{10})$$
$$f = S^{-1}(C_{2,2} \oplus K_{2,2}^{10}) \oplus S^{-1}(C_{2,2}^{*} \oplus K_{2,2}^{10})$$
$$3f = S^{-1}(C_{3,1} \oplus K_{3,1}^{10}) \oplus S^{-1}(C_{3,1}^{*} \oplus K_{3,1}^{10})$$

In their book, *(Mukhopadhyay and Chakraborty 2014, 215)*, explain that each of the equations above can be expressed generally as $A = B \oplus C$ . The aforementioned variables belong to the GF($2^8$) hence could have $2^8$ possible values each. According to the differential properties of AES, introduced in Chapter 3.3, for every random simultaneous combination of A, B, and C the probability that an equation will be satisfied is $\frac{1}{2^8}$, but the total number of possible combinations is $(2^8)^3 = 2^{24}$. Subsequently, the total number of combinations satisfying an equation is $2^{24} \times \frac{1}{2^8}$=$2^{16}$.

For a total of M equations consisted of N random byte variables, the probability that these M equations will be satisfied by N random byte variables is $\left(\frac{1}{2^8}\right)^M$. The total number of combinations can be calculated using the formula: $\left(\frac{1}{2^8}\right)^M * (2^8)^N = (2^8)^{N-M}$. The four equations above contain the following five unknown random byte variables: $f, K_{0,0}^{10}, K_{1,3}^{10}, K_{2,2}^{10}, K_{3,1}^{10}$. Thus, the total number of possible solutions is $(2^8)^{5-4} = 2^8$, reducing the total search space to $2^8$ combinations. The reduction of the search space suggests that out of the total possible hypotheses of the 4 key bytes, there are only $2^8$ hypotheses satisfying the 4 equations. One fault can significantly reduce the search space for the possible four bytes of the key to $2^8$ possible encryption keys. Two faults can determine the four bytes of the key. The retrieval of all four key quartets shaping the entire AES key demands two faults to be injected to the appropriate locations, suggesting a total of 8 faulty ciphertexts and a fault-free ciphertext.

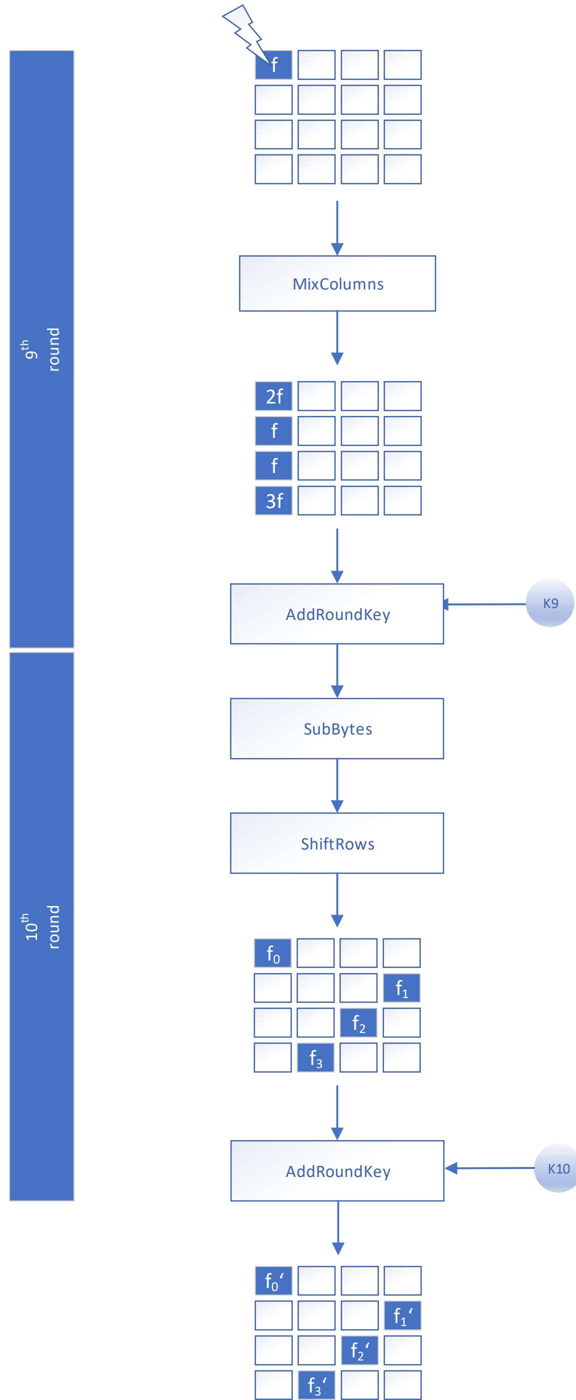



**Figure 3.2 Byte-level DFA against 9th round of AES**

For the sake of completeness of the presentation of AES key retrieval, the rest of three possible systems of four equations are cited below.

For an active byte in the second row of the state matrix:

$$3f = S^{-1}\left(C_{0,1} \oplus K_{0,1}^{10}\right) \oplus S^{-1}\left(C_{0,1}^{*} \oplus K_{0,1}^{10}\right)$$
$$2f = S^{-1}\left(C_{1,0} \oplus K_{1,0}^{10}\right) \oplus S^{-1}\left(C_{1,0}^{*} \oplus K_{1,0}^{10}\right)$$

$$f = S^{-1}(C_{2,3} \oplus K_{2,3}^{10}) \oplus S^{-1}(C_{2,3}^{*} \oplus K_{2,3}^{10})$$
$$f = S^{-1}(C_{3,2} \oplus K_{3,2}^{10}) \oplus S^{-1}(C_{3,2}^{*} \oplus K_{3,2}^{10})$$

For an active byte in the third row of the state matrix:

$$f = S^{-1}(C_{0,2} \oplus K_{0,2}^{10}) \oplus S^{-1}(C_{0,2}^{*} \oplus K_{0,2}^{10})$$
$$3f = S^{-1}(C_{1,1} \oplus K_{1,1}^{10}) \oplus S^{-1}(C_{1,1}^{*} \oplus K_{1,1}^{10})$$
$$2f = S^{-1}(C_{2,0} \oplus K_{2,0}^{10}) \oplus S^{-1}(C_{2,0}^{*} \oplus K_{2,0}^{10})$$
$$f = S^{-1}(C_{3,3} \oplus K_{3,3}^{10}) \oplus S^{-1}(C_{3,3}^{*} \oplus K_{3,3}^{10})$$

For an active byte in the fourth row of the state matrix:

$$f = S^{-1}(C_{0,3} \oplus K_{0,3}^{10}) \oplus S^{-1}(C_{0,3}^{*} \oplus K_{0,3}^{10})$$
$$f = S^{-1}(C_{1,1} \oplus K_{1,1}^{10}) \oplus S^{-1}(C_{1,1}^{*} \oplus K_{1,1}^{10})$$
$$3f = S^{-1}(C_{2,1} \oplus K_{2,1}^{10}) \oplus S^{-1}(C_{2,1}^{*} \oplus K_{2,1}^{10})$$
$$2f = S^{-1}(C_{3,0} \oplus K_{3,0}^{10}) \oplus S^{-1}(C_{3,0}^{*} \oplus K_{3,0}^{10})$$

## 3.6 Single Byte DFA against the 8th round of AES-128

In this section, we present a Differential Fault Analysis (DFA) attack against the 8th round of AES-128. The objective of this attack is to recover the entire 128-bit secret key by exploiting a single-byte fault injection. This attack assumes that the adversary has control over the plaintext being encrypted and targets the same fault model and location as the previous attack.

The DFA attack against the 8th round of AES-128 takes advantage of the fault propagation characteristics observed in the S-Box and the MixColumns operation. By injecting a fault into the first column of the state matrix, the adversary can derive differential equations that relate the fault values to the key bytes and ciphertext bytes.

The fault propagation across the last three rounds of AES-128 is illustrated in Figure 3.3. By analyzing this fault propagation, we can establish the following set of four equations for the first column of the state matrix of the 9th round after the MixColumns operation.

$$2f_0 = S^{-1}(C_{0,0} \oplus K_{0,0}^{10}) \oplus S^{-1}(C_{0,0}^{*} \oplus K_{0,0}^{10})$$
$$f_0 = S^{-1}(C_{1,3} \oplus K_{1,3}^{10}) \oplus S^{-1}(C_{1,3}^{*} \oplus K_{1,3}^{10})$$
$$f_0 = S^{-1}(C_{2,2} \oplus K_{2,2}^{10}) \oplus S^{-1}(C_{2,2}^{*} \oplus K_{2,2}^{10})$$
$$3f_0 = S^{-1}(C_{3,1} \oplus K_{3,1}^{10}) \oplus S^{-1}(C_{3,1}^{*} \oplus K_{3,1}^{10})$$

The key search space is significantly reduced due to the differential properties of AES and the resulting equations. The variable $f_0$ in the equations can take $2^8$ possible values. Consequently, the overall search space for the quartet of key bytes is also reduced to $2^8$ choices. This reduction applies independently to all four columns of the state matrix of the 9th round, resulting in a combined search space of $2^{32}$ for the entire last round key $K^{10}$.

To further narrow down the search space, the relationships between the fault values in the aforementioned state matrix are considered. Assuming the fault location is known, additional equations are derived that exploit these relationships. These equations allow for the reduction of the search space for the 9th round key $K^9$, which is a precursor to further key recovery.

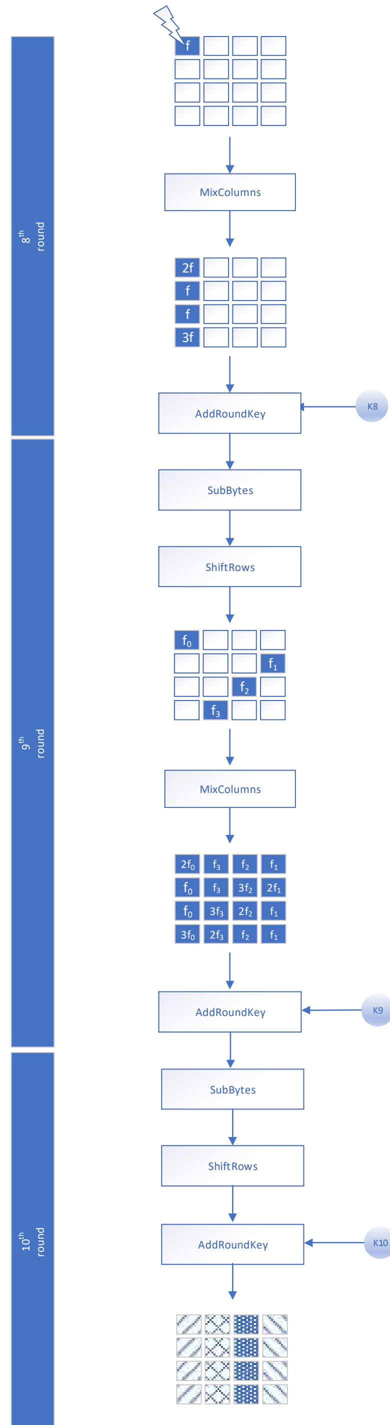



**Figure 3.3 Byte-level DFA against 8th round of AES**

The equations relating the fault value in the first column of the state matrix of the 8th round of AES following the MixColumns operation to $K^9$, fault-free output of the 9th round $C^9$, and faulty output of the 9th round $C^{*9}$ are the following:

$$2f_0 = S^{-1}\left(14(C_{0,0}^9 \oplus K_{0,0}^9) \oplus 11(C_{1,0}^9 \oplus K_{1,0}^9) \oplus 13(C_{2,0}^9 \oplus K_{2,0}^9) \oplus 9(C_{3,0}^9 \oplus K_{3,0}^9)\right) \oplus S^{-1}\left(14(C_{0,0}^{*9} \oplus K_{0,0}^9) \oplus 11(C_{1,0}^{*9} \oplus K_{1,0}^9) \oplus 13(C_{2,0}^{*9} \oplus K_{2,0}^9) \oplus 9(C_{3,0}^{*9} \oplus K_{3,0}^9)\right)$$

$$f_0 = S^{-1}\left(9(C^9_{0,3} \oplus K^9_{0,3}) \oplus 14(C^9_{1,3} \oplus K^9_{1,3}) \oplus 11(C^9_{2,3} \oplus K^9_{2,3}) \oplus 13(C^9_{3,3} \oplus K^9_{3,3})\right)$$
$$\oplus S^{-1}\left(9(C^{*9}_{0,3} \oplus K^9_{0,3}) \oplus 14(C^{*9}_{1,3} \oplus K^{*9}_{1,3}) \oplus 11(C^{*9}_{2,3} \oplus K^9_{2,3}) \oplus 13(C^{*9}_{3,3} \oplus K^9_{3,3})\right)$$

$$f_0 = S^{-1}\left(13(C^9_{0,2} \oplus K^9_{0,2}) \oplus 9(C^9_{1,2} \oplus K^9_{1,2}) \oplus 14(C^9_{2,2} \oplus K^9_{2,2}) \oplus 11(C^9_{3,2} \oplus K^9_{3,2})\right)$$
$$\oplus S^{-1}\left(13(C^{*9}_{0,2} \oplus K^9_{0,2}) \oplus 9(C^{*9}_{1,2} \oplus K^9_{1,2}) \oplus 14(C^{*9}_{2,2} \oplus K^9_{2,2}) \oplus 11(C^{*9}_{3,2} \oplus K^9_{3,2})\right)$$

$$3f_0 = S^{-1}\left(13(C^9_{0,1} \oplus K^9_{1,0}) \oplus 9(C^9_{1,1} \oplus K^9_{1,1}) \oplus 14(C^9_{2,1} \oplus K^9_{2,1}) \oplus 11(C^9_{3,1} \oplus K^9_{3,1})\right)$$
$$\oplus S^{-1}\left(13(C^{*9}_{0,1} \oplus K^9_{0,1}) \oplus 9(C^{*9}_{1,1} \oplus K^9_{1,1}) \oplus 14(C^{*9}_{2,1} \oplus K^9_{2,1}) \oplus 11(C^{*9}_{3,1} \oplus K^9_{3,1})\right)$$

By applying these equations, the search space for the 9th round key $K^9$ is reduced to $2^{32}$. For each hypothesis of $K^{10}$ that survives the first phase of the attack, and for a fault-free and faulty ciphertext pair (C, $C^*$), a unique triplet ($K^9$, $C^9$, $C^{*9}$) can be obtained. These triplets are then tested using the derived system of equations.

The attack complexity is proportional to the search space for $K^{10}$, which is $2^8$ choices. However, the time complexity of the attack remains $2^{32}$, as all hypotheses of $K^{10}$ need to be exhaustively tested using the equations. In conclusion, the DFA attack against the 8th round of AES-128 allows for the recovery of the entire secret key by exploiting a single-byte fault injection.

# 4 Framework

## 4.1 The main components of our implementation

Hardware security research projects often require a significant budget for equipment. However, this implementation was accomplished using a low-budget platform designed for educational purposes by NewAE Technology, called ChipWhisperer-Nano (CW-Nano). This allowed the project to be completed within a budget of approximately 50 euros.

According to its creators, ChipWhisperer (CW) is an open-source toolchain designed for hardware research, with the Nano version being the most affordable option. With its built-in target microcontroller and the ability to connect to external targets through its connectors, CW-Nano provides an effective platform for conducting side channel analysis and voltage fault injection attacks.

A brief overview of the various layers of ChipWhisperer would provide insights not only into the tool itself, but also into the different layers of work involved in this thesis. These layers consist of hardware, firmware, and software.

At the hardware level, CW-Nano is physically divided into two sections: the capture board, highlighted in red in the following image, and the target board, highlighted in purple. The target board features an STM32F030F4P6 microcontroller, while the capture board houses an ATSAM4SD16B microcontroller that supports the USB interface and sampling, along with an 8-bit 20 MS/s Analog-to-Digital Converter and a crowbar that allows for voltage fault injection. It is worth noting that while ChipWhisperer Nano supports performing tests against external targets, our implementation was limited to testing against the built-in target.

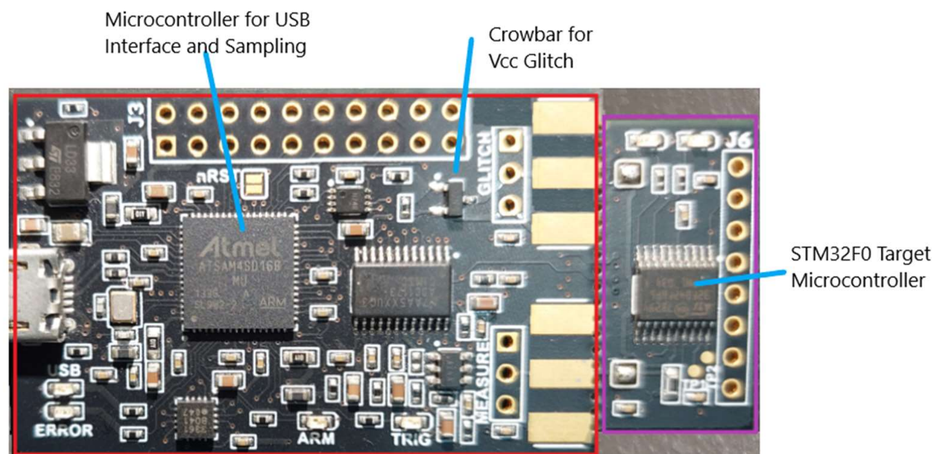



**Figure 4.1 ChipWhisperer Nano**

The main properties of ChipWhisperer-Nano[1], as listed in the official documentation, are summarized in the following table.

| Feature | Notes/Range |
|---|---|
| **ADC Specs** | 8-bit 20MS/s |
| **ADC Clock Source** | Internally generated, external input |
| **Analog Input** | AC-Coupled, fixed gain of ~10dB |
| **Sample Buffer Size** | 50 000 samples |
| **ADC Decimation** | No |

[1] https://rtfm.newae.com/Capture/ChipWhisperer-Nano/ [last accessed on 14/1/2023]

| | |
|---|---|
| **ADC Offset Adjustment** | No |
| **ADC Trigger** | Rising-edge |
| **Presampling** | No |
| **Phase Adjustment** | No |
| **Capture Streaming** | No |
| **Clock Generation Range** | 60MHz, divisible by 1, 2, 4, 8, or 16 |
| **Clock Output** | Regular only |

The fault injection specific features of ChipWhisperer-Nano are described below.

| Feature | Notes/Range |
|---|---|
| **Voltage Glitching** | Yes |
| **Clock Glitching** | No |
| **Glitch Outputs** | Glitch-Only |
| **Glitch Width** | Time increments between [0, 2^32) increments (Actual glitch width will be affected by cabling used for glitch output) |
| **Glitch Width Increments** | ~8.3ns |
| **Glitch Offset** | Time increments between [0 , 2^32) increments, ~200ns jitter |
| **Glitch Offset Increments** | ~8.3ns |
| **Glitch Cycle Offset** | N/A |
| **Glitch Cycle Repeat** | N/A |
| **Voltage Glitch Type** | Low-power crowbar |
| **Voltage Glitch Pulse Current** | 4A |
| **Glitch Trigger** | Rising-Edge |
| **Glitch Cycle Offset** | N/A |
| **Glitch Cycle Repeat** | N/A |
| **Voltage Glitch Type** | Low-power crowbar |
| **Voltage Glitch Pulse Current** | 4A |
| **Glitch Trigger** | Rising-Edge |

As it is presented on the tables above, ChipWhisperer Nano is able to inject a fault with a minimum width of 8.3 ns which can increment at integer multiples of this minimum width up to $2^{32}$. Additionally, the glitch can be introduced after a trigger is received in the form of a rising edge. The offset from a rising clock edge trigger to a glitch pulse rising edge can also be adjusted and increment from a minimum of 8.3 ns up to $2^{32}$ times of this minimum duration. The exact type of the supported voltage glitch type is low-power crowbar. However, a restriction concerning ChipWhisperer Nano is that it does not support clock fault injections.

The USB controller for the target device was written in C, while an FPGA written in Verilog enabled high-speed power trace captures. The device had its own firmware examples written in C, and a custom firmware was developed specifically for this project.

On the software side, the ChipWhisperer Python API was used for communication with the device. A pre-configured Virtual Machine (VM) running Oracle VM VirtualBox as a Type-2 Hypervisor was used, with all necessary software pre-installed, including version 5.6.0 of ChipWhisperer. The VM used Debian GNU/Linux 9.13 as the operating system, and Jupyter Notebook version 6.4.0 was installed. Jupyter Notebook was used as a web application hosted on the VM to write and execute Python scripts, which communicated with the device using the ChipWhisperer API.

The following diagram illustrates the building blocks of the setup used to implement the fault injection attacks, as well as the interconnection among these components. The Host Machine is a laptop computer equipped with an AMD Ryzen 9 4900HS Central Processing Unit (CPU) and 16 GB of RAM. Oracle VM VirtualBox is running on the Host Machine, and as a Type-2 Hypervisor abstracts the guest operating system from the host.

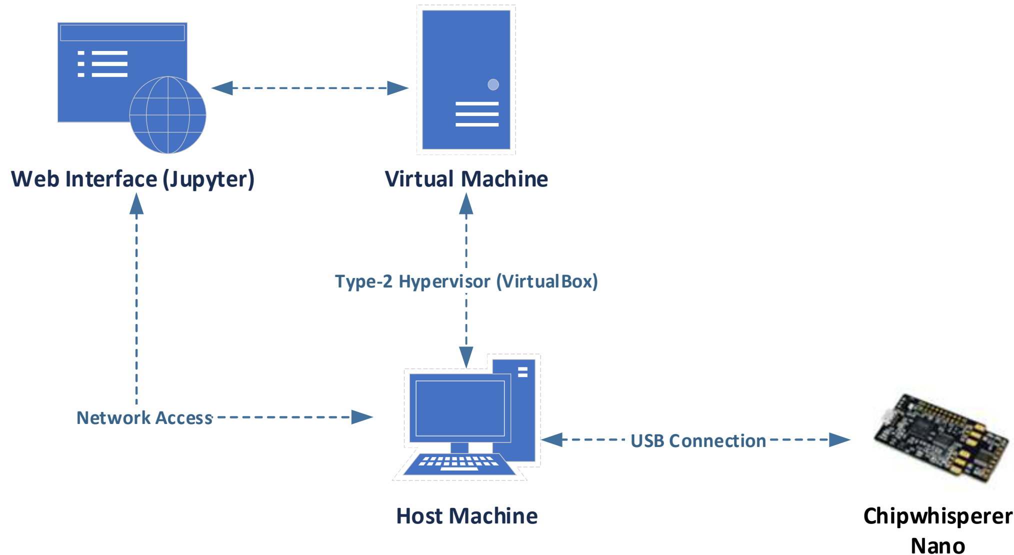



**Figure 4.2 Building blocks of our setup**

The Jupyter Notebook is running as an HTTP service accessible by default on port 8888 of the Virtual Machine, which is then forwarded to the localhost of the host machine. The following screenshot illustrates the web interface of the Jupyter Notebook, which includes the initial interface and a file browser. The file browser allows users to navigate the file system of the Virtual Machine and manage the files and folders associated with the Jupyter Notebook.

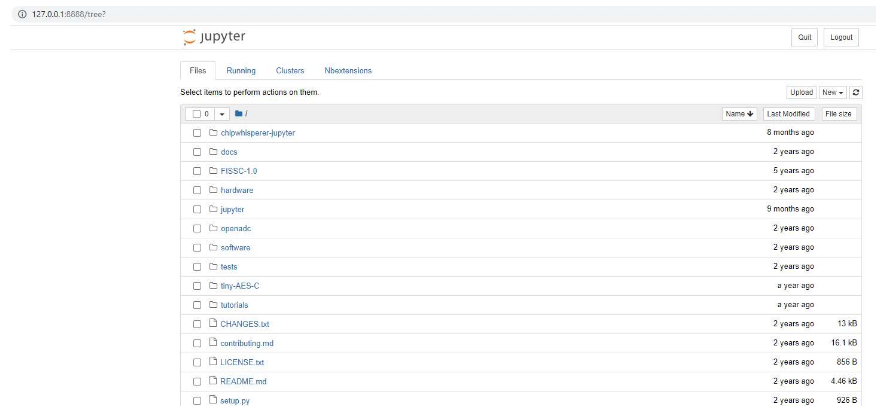



**Figure 4.3 The Jupyter Notebooks Interface – Browsing the Filesystem**

In addition to being able to browse the filesystem of the virtual machine, Jupyter enables users to edit text files, run Linux terminal commands, and execute Python scripts dynamically through Jupyter notebooks (.ipynb files). The preconfigured ChipWhisperer VM offers a plethora of Jupyter notebooks with examples of use cases for different models of its hardware. While Jupyter notebooks offer a wide range of features, a detailed presentation of Jupyter Notebook and its features is outside the scope of this thesis.

## 4.2 Target: Victim Firmware

Our attack targets Tiny AES in C[2] running in ECB mode on the STM32F030F4P6 microcontroller of the CW Nano. The cipher key used is: “2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c” (in hexadecimal representation) which is the default key provided by Tiny AES, as well as the cipher key presented in the definition of AES-128 by NIST *(Dworkin 2001)*.

For the communication with the microcontroller the SimpleSerial communication protocol has been used. The SimpleSerial is written in C and it is the default communication used by NewAE for the demo material provided with ChipWhisperer.

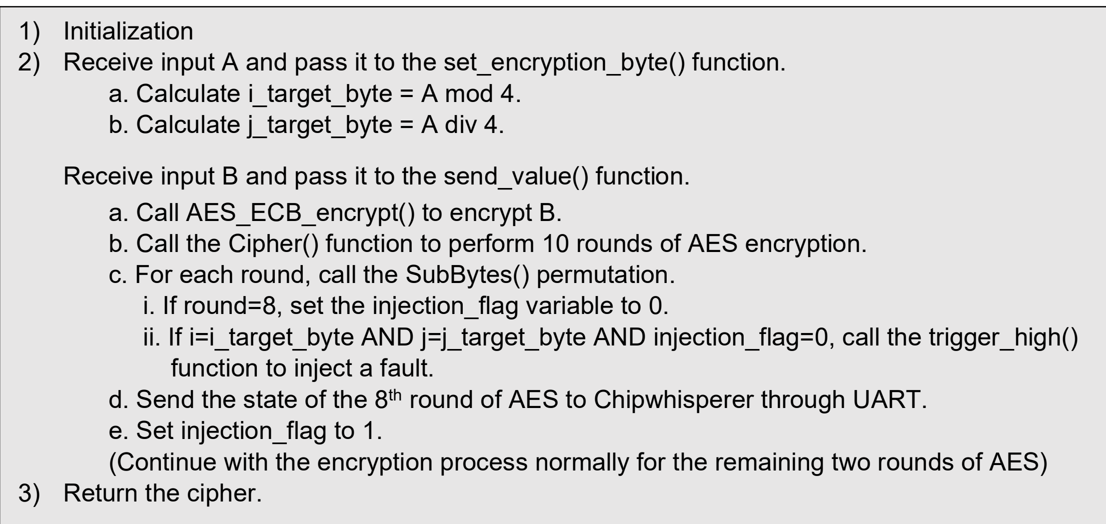
Moving on to the presentation of our custom firmware, the victim firmware developed for our implementation is initialized through setting up the UART protocol for communication, the trigger mechanism, and the reset functionality. As soon as the target device is ready, it awaits for an input value to its UART. The first input value is awaited to be an integer defining the exact byte at which the glitch will be targeted. This value is then passed to the set_encryption_byte() function. This function takes as input the byte value where the fault will be inserted and calculates to which value of the AES state array it corresponds. This is achieved by calculating the integer quotient and remainder when dividing the byte value by 4, since the state array is a 4x4 matrix.

After receiving the first input value, the second value received is the plain text, which is an array of 16 integer bytes. This array is provided to the send_value() function, which initializes the state buffer and starts the encryption process by calling AES_ECB_encrypt(). Once the encryption is complete, the function sends the contents of the buffer, which now contains the cipher, to the receiving end.

During the encryption process, the AES_ECB_encrypt() function adheres to the standard AES-128 algorithm with modifications that are specific to our implementation. These modifications involve the expansion of the encryption key and the invocation of the Cipher() function to execute the 10 rounds of AES. Notably, we have customized the Cipher() function to set an "injection_flag" to zero when the "round" variable reaches 8. If, during the SubBytes() permutation, the "injection_flag" remains at zero and the byte is the intended target of the fault, the trigger_high() function is triggered to generate a rising edge and initiate the fault injection. The specifics of this fault injection will be discussed in greater detail in the subsequent section on the CW Python API as an integral part of our implementation.

Once the SubBytes() permutation has been executed during the 8th round of AES, the value of the targeted byte in the state matrix is replaced with its corresponding value in the S-box substitution. At this point, the trigger_low() function is invoked to generate a signal that indicates to ChipWhisperer that the process has concluded. Subsequently, the "injection_flag" is reset to 1, and the state at the end of the 8th round of AES is transmitted via UART. This enables us to determine the nature of the fault injection, whether it is a single-bit, single-byte, or multiple-byte fault. The remaining steps of the encryption process continue as usual, and the resulting cipher is returned to the sender through the send_value() function, as previously described.

[2] https://github.com/kokke/tiny-AES-c (Last accessed on 11/3/2023)



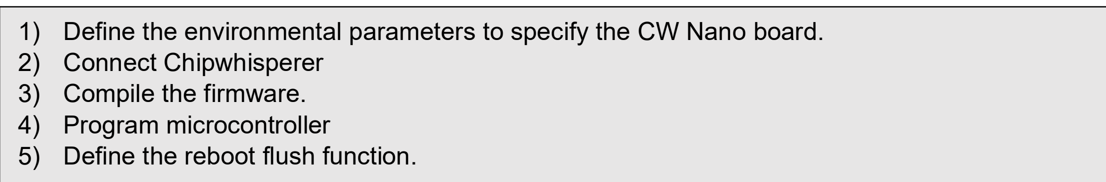
1) Initialization
2) Receive input A and pass it to the set_encryption_byte() function.
    a. Calculate i_target_byte = A mod 4.
    b. Calculate j_target_byte = A div 4.

   Receive input B and pass it to the send_value() function.
    a. Call AES_ECB_encrypt() to encrypt B.
    b. Call the Cipher() function to perform 10 rounds of AES encryption.
    c. For each round, call the SubBytes() permutation.
        i. If round=8, set the injection_flag variable to 0.
        ii. If i=i_target_byte AND j=j_target_byte AND injection_flag=0, call the trigger_high() function to inject a fault.
    d. Send the state of the 8th round of AES to Chipwhisperer through UART.
    e. Set injection_flag to 1.
    (Continue with the encryption process normally for the remaining two rounds of AES)
3) Return the cipher.

**Figure 4.4 Algorithm description**

## 4.3 ChipWhisperer API & Jupyter Notebook: The control center

The ChipWhisperer API and the related Jupyter Notebooks act as a control center for the entire process, including the compilation of the firmware, programming of the microcontroller, initiation of the encryption process, and control of the glitching process. This process always begins with an initialization stage that defines environmental parameters, such as the board in use -being the CW Nano, and executes a setup script to ensure proper connection to the USB. The firmware, as described in the previous section, is then compiled and programmed onto the flash memory of the STM32 target. Additionally, a reboot flush function is defined where the clock frequency and the sampling rate of the ADC are defined. This function is used to reset the microcontroller in case of errors or timeouts during the encryption and glitching process. A summary of the first steps of the initialization of every Jupyter Notebook used throughout this project would be in algorithmic format as follows:



1) Define the environmental parameters to specify the CW Nano board.
2) Connect Chipwhisperer
3) Compile the firmware.
4) Program microcontroller
5) Define the reboot flush function.

**Figure 4.5 Initialization algorithm**

The initialization process via Jupyter including the connection of ChipWhisperer and firmware compilation is illustrated on the following screenshot.

**Figure 4.6 The Jupyter Notebooks Interface – Initialization process**

After establishing the hardware connection and initializing the firmware, we move on to describing our single fault injection test against AES-128. We first define the encryption parameters, including the cipher key and encryption mode (ECB). Next, we generate a random 16-byte hex array using the "random" Python library, which serves as the plaintext for encryption. We then use the "pycrypto" library to execute the exact same encryption against the plaintext in the Python script. This same encryption process will later take place on the microcontroller. By comparing the received cipher from the microcontroller with the expected output, any encryption faults caused by injected voltage glitches can be identified.

1) Define the encryption key.
2) Generate random plaintext.
3) Encrypt plaintext.

**Figure 4.7 Encryption process algorithm**

Once the initialization steps are complete, we proceed with a test run of the fault injection procedure. This involves sending the plaintext to the microcontroller, performing a voltage glitch on a specific clock cycle during the encryption process, and then retrieving the resulting cipher. We repeat this process several times with different glitch settings and compare the obtained ciphers with the correct one. If a fault was injected successfully, the received cipher will differ from the expected one. This procedure is a basic test that allows us to ensure that our setup is working correctly and to identify any potential issues with the fault injection process. It also serves as a reference for further analysis of more complex glitching schemes.

### 4.3.1 Implementation Details

The script begins by calling the reboot_flush() function to reset the microcontroller and set the clock frequency and the sampling rate of the ADC. The required libraries are then imported, including the numpy library for working with arrays, the sys library for accessing system-specific parameters and functions, the pycrypto library for executing the encryption, and the matplotlib library for visualizing the results.

The glitching parameters are set using the ChipWhisperer API. The width of the glitch in cycles and the offset from the rising edge of the trigger are two key parameters that determine the

duration and timing of the fault injection. After performing multiple tests, appropriate values for these parameters were determined.

The scope.arm() function is called to arm the ChipWhisperer for glitching. The injection byte variable is set to a single byte of data which defines the byte of the state where the injection will occur and is sent to the microcontroller using the target.simpleserial_write() function. The plaintext is also sent to the microcontroller using the same function.

The simpleserial_read_witherrors() function is used to read the microcontroller's response. The state variable receives the microcontroller's 16-byte response, which is the state of the AES encryption at the end of the 8th round -after the voltage injection. This variable is important as it enables inspecting the impact of the voltage glitch, if it affects a single bit, a single byte or multiple bytes of the state. The valid variable receives the cipher from the microcontroller after the completion of the encryption.

If the response is valid, the "val" variable receives the full response from the microcontroller, including the 16-byte cipher. If the received cipher matches the expected cipher generated by the Python script, the message "Correct" is printed to the console. Otherwise, the message "Fault" is printed.

Finally, the ChipWhisperer captures the trace data using the scope.capture() function, and the last trace is retrieved using the scope.get_last_trace() function. The trace is plotted using the matplotlib library, and the result is shown in a pop-up window.

This single fault injection test is an essential component of the overall glitching procedure, as it allows us to verify the integrity of the encryption process and identify any faults caused by voltage glitches injected during the encryption process.

In the following test run, the glitch width was set to zero (0) cycles, so there was not any fault injection.

```
State: {'valid': True, 'payload': CWbytearray(b'b2 67 87 38 ec 49 50 a8 97 f4 08 59 fe 3f e7 c9'), 'full_response': 'rB2678738E
C4950A897F40859FE3FE7C9\n', 'rv': None}
Cipher: {'valid': True, 'payload': CWbytearray(b'a3 fe 52 9a cd cc cb cf 0c bb bb d0 ba 79 2b d2'), 'full_response': 'rA3FE529A
CDCCCBCF0CBBBBD0BA792BD2\n', 'rv': None}
Correct
```

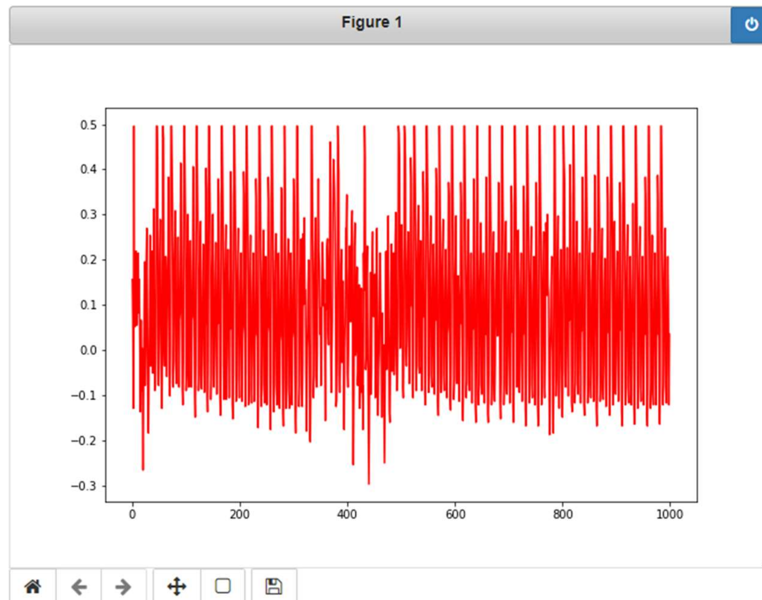



**Figure 4.8 No Fault Injected**

In the following test run, the glitch width was set to nine (9) cycles, which corresponds to a duration of approximately 74.7 nanoseconds (8.3 ns * 9), and the offset was set to 2419 cycles. The width of the glitch was so large that caused the encryption process to crash and a reset response was received. "The reset caused by the glitch is clearly visible as a flat line starting just before sample 400 in the acquired trace. The glitch itself should only last around 1.5 samples at a 20 MS/s

sampling rate with a time interval between each sample of 50 ns. However, the flat line was found to be connected with the reset caused by the glitch. Further analysis was conducted to locate the exact timing and location of the glitch within the trace.

```
State: {'valid': False, 'payload': None, 'full_response': 'rRESET   \n', 'rv': None}
Cipher: {'valid': False, 'payload': None, 'full_response': '', 'rv': None}
reset
```

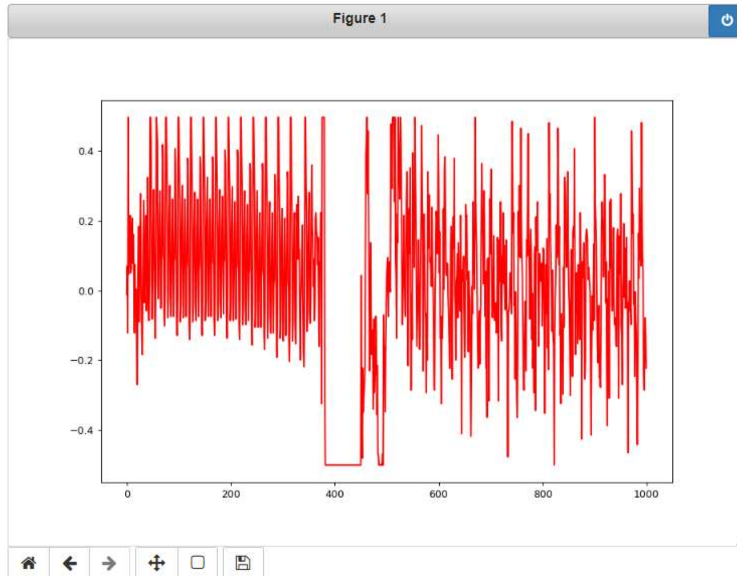



**Figure 4.9 Large glitch causing a reset**

Figure 4.5 depicts the attempted fault injection with a zoomed-in view before the reset. The Y-axis represents the current going through the shunt resistor, with a constant increase in current from sample 376 to sample 381.

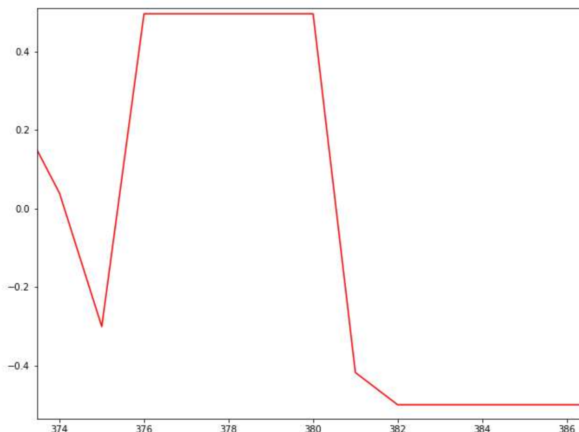
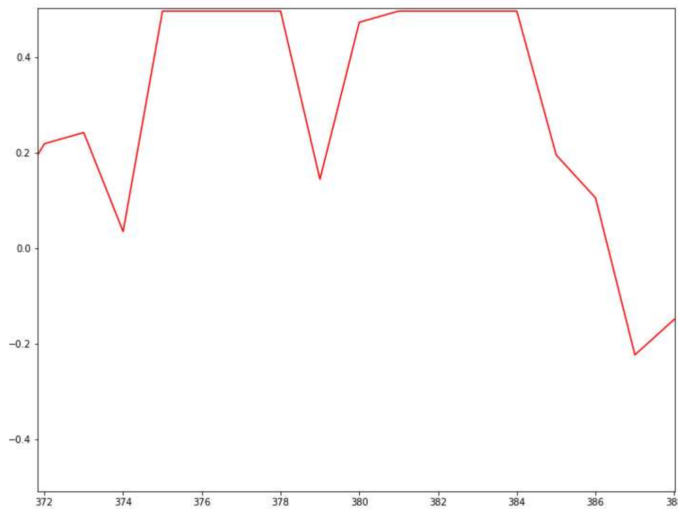



**Figure 4.10 Voltage glitch spike (width=9)**

Figure 4.6 presents another fault injection with a glitch width of three (3) cycles and the same offset of 2419. The disturbance caused by the fault injection begins at sample 376 and lasts until sample 378, indicating that the crowbar is shorter. However, the expected voltage peak of 1.5 samples for a glitch width of 9 cycles or 0.5 samples for a glitch width of 3 cycles is not observed, as the disturbance seems to have a greater amplitude than expected.




.

**Figure 4.11 Voltage glitch spike (width=3)**

A current peak of such magnitude is not observed in samples taken without a fault injection.

Having discussed the methodology for successfully performing a fault injection, we have taken a step towards achieving our goal of Differential Fault Analysis and obtaining the encryption key of AES. However, our task is not yet complete as we need to engineer a method for obtaining multiple fault injections to create a comprehensive framework. Additionally, we need to add a module for performing the actual Differential Fault Analysis. In the following chapter, we will focus on the use of MATLAB, which will serve as the missing link in this chain by providing us with the necessary analysis platform for Differential Fault Analysis.

# 5 Analysis Platform & Simulation

An analysis platform is essential to experiment on the implementation of Differential Fault Analysis techniques described during the previous chapter. Prior to applying these techniques to actual data obtained from fault injection attacks on a microcontroller, it is essential to establish a simulation platform. In this regard, MATLAB, a versatile programming language and numerical computing environment, was employed as a practical tool to facilitate the implementation of various aspects related to AES-128 encryption.

This chapter provides an in-depth exploration of how MATLAB was utilized to realize the following objectives:

1. AES-128 encryption was successfully implemented. The implementation adhered to the specifications and guidelines outlined in the AES standard, ensuring the accurate transformation of plaintext into ciphertext.
2. To investigate the impact of fault injection attacks on the AES encryption process, bit-level and byte-level fault injection simulations were carried out. MATLAB provided a flexible environment to simulate and analyze the effects of injecting faults at different levels of granularity.
3. Bit-Level and Byte-Level DFA attacks were developed to recover the encryption key of the last round of AES. These attacks aimed to exploit vulnerabilities introduced by fault injection, allowing for the retrieval of sensitive information.
4. The key scheduling algorithm was reverse engineered by analyzing the round keys and applying reverse operations to retrieve the initial encryption key.

Throughout the implementation and simulation processes, MATLAB version R2020b was utilized as the preferred software tool. By leveraging its capabilities, the analysis platform provided an effective means to explore the security aspects of AES-128 encryption, assess the vulnerability to fault injection attacks, and evaluate the effectiveness of DFA techniques.

Overall, this chapter sheds light on the methodology employed to implement AES-128 encryption, simulate fault injection attacks, and execute DFA techniques using MATLAB. The insights gained from this analysis platform lay the foundation for the subsequent chapters, which delve deeper into the experimental results and insights derived from the application of DFA techniques on real-world data.

## 5.1 Simulation of AES Encryption

The implementation of an AES encryption is based on two functions: initialization() and encrypt(). The first function defines the necessary parameters for the encryption, and the second function performs the encryption itself against the provided plaintext.

The initialization() function does not take any input parameter and returns five variables as output. More specifically, the function begins with the initialization of the transformation arrays: sbox, inv_sbox, aesmult_table, polymat, inv_polymat, rcon. An important enhancement is the inclusion of the aesmult_table, which is a hardcoded matrix used to implement AES multiplications efficiently. The aesmult_table is a precomputed lookup table that contains the results of multiplication operations for all possible combinations of values from 0 to 255. This table eliminates the need for performing multiplication calculations during the encryption process, resulting in faster and more efficient execution. The aesmult_gen() function is responsible for generating the aesmult_table matrix. It iterates over all possible values of i and j from 0 to 255 and calls the aesmult() function to calculate the multiplication result of i and j. The calculated result is then stored in the corresponding entry of the aesmult_table matrix.

Likewise, the polynomial matrices polymat and inv_polymat are also included hardcoded in order to be used during MixColumns permutation, as well as the rcon matrix used in the KeyExpansion. Furthermore, the sbox and inv_sbox matrices are hardcoded lookup tables consisting of 256 values each. These matrices provide the substitution values required during the encryption and decryption processes of the AES S-box operation. Each value corresponds to a specific input, facilitating the transformation of plaintext to ciphertext during encryption and vice versa during

decryption. Next, the encryption key is defined in hexadecimal format and transformed to decimal for further calculations. Finally, within the initialization() function, the key_schedule() function is invoked using the input variables key, sbox, and rcon.

The key_schedule() function is an important component of the AES encryption process, responsible for generating the round keys used in each round of the encryption. The function takes in the key, sbox, and rcon parameters, and outputs the w array, which contains the expanded key.

The function begins by copying the 16-byte key vector row-wise into the first four rows of the w array. Then, it loops over the remaining 40 rows of w to generate the additional round keys. For each row i in w, the function first copies the previous row of w into a buffer called temp. If i is a multiple of 4 (i.e., the start of a new round), temp is shifted one byte to the left, and the sbox is applied to each element of temp. A round constant r is then generated using the rcon array and XORed with the first element of temp. Finally, the new w(i,:) row is generated by XORing temp with the row i-4 of w. The resulting 44x4 table calculated by the key scheduling function is assigned to the w variable based on the defined encryption key.

Once the initialization() function has completed its execution, the encrypt() function is called to perform the encryption on the provided plaintext. The encrypt() function takes the plaintext, w, sbox, poly_mat, and aesmult_table as input parameters. The plaintext, which represents a 16-byte array containing the message or data to be encrypted, is reshaped into a 4x4 state matrix. The initial round key, derived from the expanded key, is extracted and applied to the state matrix using bitwise XOR.

Next, a loop is executed for 10 rounds (excluding the final round). Within each iteration, the state matrix undergoes the following operations:

1. Substitution Bytes (S-box): Each element in the state matrix is replaced with its corresponding value from the sbox lookup table.
2. Shift Rows: After applying the Substitution Bytes (S-box) operation to each element in the state matrix, the shift_rows() function is then called. This function cyclically shifts the last three rows of the state matrix, providing diffusion and increasing the complexity of the encryption. The shifted state matrix is passed on to the mix_columns() function.
3. Mix Columns: The mix_columns() function iterates over each column of the state matrix. For each element in a column, the function performs a modulo multiplication operation with a constant from the aesmult_table lookup table. The result of each multiplication is accumulated, and the final value represents the transformed element in the column. This transformation further increases the diffusion and strengthens the encryption algorithm.

After the 10 rounds are completed, the state matrix undergoes the final round, which includes Substitution Bytes and Shift Rows operations, but excludes Mix Columns. The resulting state matrix represents the ciphertext.

Finally, the ciphertext is reshaped into a 1x16 vector and returned as the output of the encrypt() function.

The encrypt() function serves as the core component of the AES encryption process, applying the necessary operations and transformations to the plaintext using the round keys generated by the key_schedule() function.

## 5.2 Simulation of Faulty AES Encryption

In this chapter, we expand the simulation framework to include fault injection in the AES encryption process. Instead of physical perturbations such as voltage or clock jitter, we simulate the process programmatically by implementing the faulty AES encryption. This implementation is achieved through the introduction of the faulty_encrypt() function, which shares similarities to the encrypt() function discussed in the previous chapter but incorporates additional functionality for injecting faults into the encryption process. In other words, it can be regarded as a faulty version of the encryption function.

The faulty_encrypt() function takes the plaintext, w, sbox, poly_mat, aesmult_table, roundToInject, byteToInject, numberOfBitsToInject, and bitToInject as input parameters. It

performs the encryption while selectively injecting faults into the encryption process. The roundToInject parameter defines the round of AES where the fault is going to be injected. The byteToInject specify the byte of the AES round. If the byteToInject value is 0 or exceeds 16, a random byte is selected for fault injection. In addition, the numberOfBitsToInject and bitToInject parameters determine the number of bits and the specific bit within the selected byte where the fault is injected, respectively. If these parameters are set to 0 or exceed the appropriate limits, random values are chosen for fault injection.

Similar to the encrypt() function, the faulty_encrypt() function begins by reshaping the plaintext into a 4x4 state matrix. The initial round key, derived from the expanded key, is applied to the state matrix using bitwise XOR. A loop is then executed for 10 rounds (excluding the final round), just like in the encrypt() function.

Within each iteration, if the round matches the provided roundToInject parameter, the fault injection is taking place before the SubBytes permutation. Then the encryption process follows the same steps performed in the normal encrypt() function.

Fault injection occurs at a specific round and byte level, as determined by the input parameters. The fault injection process involves calculating the row and column of the state matrix where the fault is injected based on the byteToInject value. The corresponding element within the state matrix is then modified by applying a bitwise XOR operation with the fault injection mask. The fault injection mask has an 1 in the position of every bit which should be shifted.

After the fault injection, the AES encryption algorithm is performed normally until the completion of all rounds. The faulty ciphertext is reshaped into a 1x16 vector and returned as the output of the faulty_encrypt() function.

The introduction of fault injection in the encryption process allows us to evaluate the impact of injected faults on the integrity and security of the ciphertext. By selectively injecting faults at specific rounds and byte levels, we can analyze potential vulnerabilities and assess the resilience of AES against fault injection attacks.

## 5.3 MATLAB Script for Bit-Level DFA

In this section, we explore the implementation of Bit-Level Differential Fault Analysis (DFA) on the AES encryption algorithm using a MATLAB script. While Bit-Level DFA is challenging to achieve in practical scenarios, simulation-based approaches can be used to study its feasibility.

The MATLAB script begins by executing the initialization(), encrypt(), and faulty_encrypt() functions to prepare the encrypted data for the attack. The faulty_encrypt() function is called three times with different parameters to generate three distinct faulty ciphertexts. Each faulty ciphertext targets a different single bit, as specified in the previous chapter.

The core of the attack lies in the crack_bitDFA() function, which takes the original ciphertext, the three faulty ciphertexts, and the byte position of the injected fault as input. The function employs the following equation, which has been discussed thoroughly in section 3.4.

$$C_{i,l} \oplus C_{i,l}^{*} = S(x_{i,j}) \oplus S(x_{i,j} \oplus \varepsilon)$$

It calculates the XOR difference between the original and faulty ciphertexts to determine the left part of the equation for each case. Next, the script iterates through possible positions of the injected fault to guess the value of ε, representing the injected error. It also considers the 256 possible values of the given byte during the 9th round of AES, denoted as $x_{i,j}$. In each iteration, the script calculates the right part of the equation and compares it with the left part obtained earlier. If a match is found, a potential solution to the equation is added.

During this iteration, the script generates multiple possible solutions to the equation, storing them in three separate arrays of possible solutions. These arrays represent the sets of potential solutions obtained from the comparisons between the left and right parts of the equation for the three faulty ciphertexts.

To identify the correct byte value of $x_{i,j}$ and eliminate extraneous solutions, the script loops through the solutions again. It compares the solutions from the different sets of solutions, searching for the common value that satisfies the equation across all arrays. This process ensures that the identified solution accurately represents the byte of the encryption key.

Unlike the need for two faulty ciphertexts described in theory, using only two ciphertexts led to false positives in many situations. The utilization of three faulty ciphertexts is essential to enhance the reliability of the analysis. By comparing the solutions obtained from different fault injections, the script can eliminate false positives and identify the unique solution that consistently satisfies the equation for all faulty ciphertexts.

It is important to note that the recovery of the entire 16 bytes in an AES encryption necessitates the repetition of the procedure 16 times. Given that the recovery of a single byte demands three successful bit-level fault injections, the completion of this process would, therefore, require a total of 48 successful fault injections.

The insights gained from this script contribute to a deeper understanding of the security aspects and weaknesses of the AES encryption algorithm. By simulating Bit-Level DFA and employing various fault injections, the script enables the analysis of potential vulnerabilities and assesses the resilience of AES against fault injection attacks.

## 5.4 MATLAB Script for Byte-Level DFA

In this chapter, we present the implementation of Byte-Level Differential Fault Analysis (DFA) on the AES-128 encryption algorithm using a MATLAB script. The script is designed to perform fault injections and analyze the resulting ciphertexts to recover the secret key used in the encryption process. Specifically, the fault injections are performed during the SubBytes operation of the 8$^{th}$ round of AES-128.

The implementation is built upon the theoretical background presented in sections 3.5 and 3.6, which outline the practical approach to Differential Fault Analysis. The core of the implementation is the crackDFA() function, which takes several inputs, including the ciphertext, and two different faulty ciphertexts performed against the same byte (faultyciphertext8a and faultyciphertext8b). Additionally, the injectedByte parameter specifies the byte position to target during the fault injection. The inverse S-box (inv_s_box) and AES multiplication table (aesmult_table) arrays are defined as part of the initialization phase.

The script utilizes a FaultMap matrix that maps the injected byte positions to their corresponding positions in the ciphertext. This mapping ensures accurate alignment of the fault injections with the correct bytes in the encryption process following the propagation of the fault during the operations of AES following the glitch.

Through nested loops, the script iterates through possible values of the encryption key bytes (K1, K2, K3, and K4) in the range of 0 to 255. It calculates intermediate values (f1, f2, f3, and f4) using the find_candidate_keys() function which calculates the following equations: $S^{-1}\left(C_{x,y} \oplus K_{x,y}^{10}\right) \oplus S^{-1}\left(C_{x,y}^{*} \oplus K_{x,y}^{10}\right)$ , where $C_{x,y}$ is the corresponding byte of ciphertext and the $C_{x,y}^{*}$ is the respective byte of the faulty ciphertext. $K_{x,y}^{10}$ is the candidate key examined during the given iteration.

In other words, the find_candidate_keys() performs:

- XOR operation between the possible key values and the corresponding ciphertext
- XOR operation between the possible key values and faulty ciphertext bytes
- Inverse SubBytes of the result of these operations
- XOR operation between the result of the inverted SubBytes

By comparing these intermediate values, the script identifies potential candidate key values that satisfy the expected conditions. These conditions should satisfy the equations described in sections 3.5 and 3.6. For example, for bytes belonging to the first row of the ciphertext the system of equations which should be satisfied is the following:

$$2f_0 = S^{-1}\left(C_{0,0} \oplus K_{0,0}^{10}\right) \oplus S^{-1}\left(C_{0,0}^{*} \oplus K_{0,0}^{10}\right)$$
$$f_0 = S^{-1}\left(C_{1,3} \oplus K_{1,3}^{10}\right) \oplus S^{-1}\left(C_{1,3}^{*} \oplus K_{1,3}^{10}\right)$$
$$f_0 = S^{-1}\left(C_{2,2} \oplus K_{2,2}^{10}\right) \oplus S^{-1}\left(C_{2,2}^{*} \oplus K_{2,2}^{10}\right)$$
$$3f_0 = S^{-1}\left(C_{3,1} \oplus K_{3,1}^{10}\right) \oplus S^{-1}\left(C_{3,1}^{*} \oplus K_{3,1}^{10}\right)$$

Satisfaction of the expected conditions are examined progressively to avoid unnecessary operations. So if, in the given example, $S^{-1}\left(C_{1,3} \oplus K_{1,3}^{10}\right) \oplus S^{-1}\left(C_{1,3}^{*} \oplus K_{1,3}^{10}\right)$ is equal to

$S^{-1}(C_{2,2} \oplus K_{2,2}^{10}) \oplus S^{-1}(C_{2,2}^{*} \oplus K_{2,2}^{10})$ , we may assume that it is a possible $f_0$. The given value of $f_0$ should be multiplied using the aesmult_table to perform the multiplication in the Galois Field and calculate $2f_0$. Hence the value of $S^{-1}(C_{0,0} \oplus K_{0,0}^{10}) \oplus S^{-1}(C_{0,0}^{*} \oplus K_{0,0}^{10})$ is also calculated using the find_candidate_keys() and compared to $2f_0$. If they are also equal, we follow the same process with $S^{-1}(C_{3,1} \oplus K_{3,1}^{10}) \oplus S^{-1}(C_{3,1}^{*} \oplus K_{3,1}^{10})$ which should be equal to $3f_0$.

If all a set of four candidate keys satisfies the condition, they are considered candidate keys. The candidate key values are stored in separate arrays (candK1, candK2, candK3, and candK4).

To determine the correct key bytes, the script performs another loop, repeating the procedure described above but using only the sets of candidate keys identified previously and the second faulty ciphertext. By comparing these intermediate values across the candidates, the script identifies the common values of K1, K2, K3, and K4 that consistently satisfy the expected conditions. This process eliminates false positive and produces a single set of solutions.

The script returns the arrays solutionK1, solutionK2, solutionK3, and solutionK4, which represent the solutions for each byte of the encryption key.

In order to retrieve all sixteen bytes of the $10^{th}$ round key, the crackDFA() function should be called four times. Successful retrieval of the encryption key was possible using simulated data and real data from the microcontroller. It should be mentioned that the retrieval works using a total of four faulty ciphertexts, 2 pairs byte-level faulty ciphertext where the glitch has targeted the same byte of the state.

The implementation has been successfully tested using both simulated data and real data from a microcontroller, demonstrating its practical viability for fault analysis. The approach effectively reveals insights into the security aspects and weaknesses of the AES encryption algorithm.

By implementing Byte-Level DFA and analyzing the resulting data, we gain valuable insights into the security aspects and weaknesses of the AES encryption algorithm. The practical demonstration of this technique contributes to the field of fault analysis and highlights the importance of understanding and addressing potential vulnerabilities in cryptographic systems.

## 5.5 Reverse KeyExpansion Algorithm

In this section, we present a MATLAB script that implements the reverse KeyExpansion process for AES-128. The script takes the crackedKey, which represents the 10th round key obtained through the DFA attack and uses it to calculate the expanded key used for the encryption and consequently the initial encryption key.

The MATLAB script begins by initializing the necessary variables and constants. More specifically the key length Nk is set to 4. It also defines the S-box (sbox) and rcon matrices, which contain the S-box permutations table and the round constants used in the KeyExpansion respectively.

The script then iterates through the key schedule in reverse order, starting from the last round key and working backwards. It follows the reverse KeyExpansion equations and formulas to derive the previous round keys.

For each round key, the script performs the following steps:

- Retrieves the previous round key (tempb) from the current round key.
- Checks if the current index is a multiple of 4. If it is a multiple of 4, it applies a permutation (RotWord) and substitution (SubWord) operation on tempb, like the forward KeyExpansion process. It also XORs tempb with the corresponding Rcon value.
- XORs the current round key with the derived tempb to obtain the previous round key.

By repeating these steps for all round keys in reverse order, the script reconstructs the original expanded key.

The MATLAB script for reverse KeyExpansion presented in this section complements the DFA attack by enabling the retrieval of the full encryption key used in AES-128. This script, along with the DFA attack implementation, demonstrates the practicality and effectiveness of the proposed technique in recovering the AES encryption key.

## 5.6 Simulation of Bit-Level & Byte-Level DFA Attacks

Having presented the functions to simulate bit-level and byte-level Differential Fault Analysis attacks in earlier sections, the execution of these simulated attacks requires minimal explanation.

A successful bit-level DFA attack using MATLAB is illustrated on Figure 5.1. The encrypt() function is used to produce the valid ciphertext. Next, a loop is created iterating the bit-level fault injection process three times for each byte. As theory indicates it is important to note that $l = (j - i)\ mod\ 4$ in order to take into account the ShiftRows and the propagation of the fault during the 10th round. However, our implementation of AES in MATLAB has shifted the rows with columns. As a result, for each injection byte b is necessary, it is necessary to calculate the byte where the injection has propagated to using the following process:

- The parameter i is the integral result of the division of b minus 1 by 4.
- The parameter j is the remainder of the integer division of b minus 1 by 4.
- The parameter l is the remainder of the integer division of j minus 1 by 4.
- Finally, the parameter c equals i plus l times 4 plus 1 and represents the position of the byte where the fault has propagated to.

The faulty_encrypt() function is used three times in order to simulate three bit-level fault injections in the corresponding byte indicated by the parameter b, shifting the values of three different bits of the state. The ciphertexts created by the aforementioned functions are used as input for the crack_bitDFA() function, which resolves the respective byte c of the encryption key of the final round.

When the process is completed, a message is displayed on the screen showing which byte of the encryption key was acquired and the position of the initial single bit injection which led to its retrieval.

The same process is repeated 16 times until the successful recovery of all 16 bytes of the encryption key. As the final round key is successfully identified, the rev_key_schedule() function is called to calculate the initial encryption key. In the end, the print_key() function is called to print the encryption in hexadecimal format. It is a function which was not explained earlier, but it was implemented exactly for this cause taking as input the expanded decimal key as input, transposing it, converting the decimal to hexadecimal representation and printing the initial encryption key in hexadecimal format.

```
7                      '88' '99' 'aa' 'bb' 'cc' 'dd' 'ee' 'ff'};
8      % Convert plaintext from hexadecimal (string) to decimal representation
9 -    plaintext = hex2dec (plaintext_hex);
10
11 -   crackedKey=[];
12 -   tic;
13
14     %Byte to inject
15 -   ciphertext = encrypt (plaintext, w, sbox, polymat, aesmult_table);
16 -   for b=1:16
17
18 -   i=fix((b-1)/4); %0
19 -   j=mod((b-1),4); %0 1 2 3
20 -   [1,2,3,4];
21 -   [4,1,2,3];
22 -   l=mod((j-i),4);
23
24 -   c=i+l*4+1;
25
26
27 -   faultyciphertext10a = faulty_encrypt (plaintext, w, sbox, polymat, aesmult_table, 10,b,1,7);
28 -   faultyciphertext10b = faulty_encrypt (plaintext, w, sbox, polymat, aesmult_table, 10,b,1,3);
29 -   faultyciphertext10c = faulty_encrypt (plaintext, w, sbox, polymat, aesmult_table, 10,b,1,5);
30
31 -   K=crack_bitDFA(ciphertext,faultyciphertext10a,faultyciphertext10b,faultyciphertext10c,c);
32
33 -   if K>0
34 -       fprintf('\nThe element %d of the encryption is %d\nThe attack was successful using fault injection on %d. \n',c,K,b);
35 -   end
36
37 -   crackedKey(l+1,i+1)=K;
38 -   end
39 -   w2=rev_key_schedule(crackedKey);
40
41     %Print the encryption key in hex format using the expanded decimal key
42 -   print_key(w2);
43 -   toc;

demo_crack_dfa_bitfault_full.m | shift_rows.m | crack_bitDFA.m | demo_crack_dfa.m | demo_crack_dfa_bitfault_one.m | demo_crack_k_dfa.m | faulty_encrypt.m

Command Window
 The attack was successful using fault injection on 16.
 Initial AES-128 encryption key in hexadecimal format:
 2B7E151628AED2A6ABF7158809CF4F3C
 Elapsed time is 0.171493 seconds.
 >>
```

**Figure 5.1 Simulation of a successful bit-level DFA attack**

Similarly, Figure 5.2 demonstrates a successful DFA attack against AES-128 using a byte-level fault injection in a simulated manner. The process is initiated with the use of encrypt() function again in order to produce the valid ciphertext. Additionally, the faulty_encrypt() function is called four times to inject twice 3 random bits in the $0^{th}$ and $1^{st}$ byte of the state of the $8^{th}$ round. The crackDFA() function is called four times using the valid ciphertext and the two faulty ones with different parameters in the FaultMap parameter in order to formulate the complete key of the last round of AES. The faulty ciphertexts associated with the fault injection against the $1^{st}$ byte of AES are used three times with different settings.

As in the bit-level process above, the rev_key_schedule() function is called to calculate the initial encryption key, and the print_key() function is called to print the encryption in hexadecimal format.

```
3
4 -   s_box=[99,124,119,123,242,107,111,197,48,1,103,43,254,215,171,118,202,130,201,125,250,89,71,240,173,212,162,175,156,164,114,192,183,253,147,38,5
5 -   inv_s_box=[82,9,106,213,48,54,165,56,191,64,163,158,129,243,215,251,124,227,57,130,155,47,255,135,52,142,67,68,196,222,233,203,84,123,148,50,166,
6
7 -   [sbox, inv_sbox, w, poly_mat, aesmult_table] = initialization();
8 -   plaintext_hex = {'00' '11' '22' '33' '44' '55' '66' '77' ...
9                    '88' '99' 'aa' 'bb' 'cc' 'dd' 'ee' 'ff'};
10
11    % Convert plaintext from hexadecimal (string) to decimal representation
12 -  plaintext = hex2dec (plaintext_hex);
13 -  crackedKey=[];
14 -  tic;
15 -  ciphertext = encrypt (plaintext, w, s_box, poly_mat, aesmult_table);
16
17    %perform fault injection and DFA attack
18 -  faultyciphertext8a = faulty_encrypt (plaintext, w, s_box, poly_mat, aesmult_table, 8,1,3);
19 -  faultyciphertext8b = faulty_encrypt (plaintext, w, s_box, poly_mat, aesmult_table, 8,1,3);
20 -  [crackedKey(1,3),crackedKey(2,2),crackedKey(4,4),crackedKey(3,1)]=crackDFA(ciphertext,faultyciphertext8a,faultyciphertext8b,3);
21 -  [crackedKey(1,1),crackedKey(2,4),crackedKey(3,3),crackedKey(4,2)]=crackDFA(ciphertext,faultyciphertext8a,faultyciphertext8b,1);
22 -  [crackedKey(3,4),crackedKey(4,3),crackedKey(2,1),crackedKey(1,2)]=crackDFA(ciphertext,faultyciphertext8a,faultyciphertext8b,4);
23
24 -  faultyciphertext8a = faulty_encrypt (plaintext, w, s_box, poly_mat,aesmult_table, 8,2,3);
25 -  faultyciphertext8b = faulty_encrypt (plaintext, w, s_box, poly_mat,aesmult_table, 8,2,3);
26
27 -  [crackedKey(2,3),crackedKey(3,2),crackedKey(4,1),crackedKey(1,4)]=crackDFA(ciphertext,faultyciphertext8a,faultyciphertext8b,2);
28
29
30 -  w2=rev_key_schedule(crackedKey);
31 -  toc;
32
33    %Print the encryption key in hex format using the expanded decimal key
34 -  print_key(w2);
35
36
```

demo_crack_dfa.m | crackDFA.m | random_demo_crack_dfa.m

Command Window

```
>> demo_crack_dfa
Elapsed time is 3.348773 seconds.
Initial AES-128 encryption key in hexadecimal format:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
fx >>
```

**Figure 5.2 Simulation of a successful byte-level DFA attack**

As we have showcased a successful DFA attack for the retrieval of the complete encryption key using Differential Fault analysis, it is possible to demonstrate the same process using data from real fault injection experiments.

# 6 Experimental DFA

Building upon the foundations established in the preceding chapters, this section will explore the experimental implementation of Differential Fault Analysis (DFA) attacks. Commencing from the voltage fault injection phase and advancing towards the recovery of the initial AES encryption key, this section will provide a comprehensive examination of the DFA attack execution process.

It is important to note that despite multiple attempts, practical experiments aimed at replicating single-bit fault injections were unsuccessful. As a result, this implementation focuses solely on the execution of necessary fault injection attacks required for successful byte-level DFA.

## 6.1 Fault Injection Campaigns

In this chapter, we will fully utilize the functions and parameters introduced in Chapter 4.3, “ChipWhisperer API & Jupyter Notebook: The control center.” While Chapter 4.3 presented a Proof-of-Concept for a single fault injection attack, our objective here is to expand our framework to conduct a fault injection campaign, covering a series of fault injection attacks suitable for practical applications.

Having discussed thoroughly the implementation of MATLAB scripts to retrieve the encryption using Byte-level DFA attacks, it is evident that in order to retrieve the initial encryption key, more than one successful fault injections is needed, and the glitch should be shaped in order to produce faults restricted to either a single bit or a single byte of the state array.

Consequently, to acquire multiple faults we need to run a fault injection campaign. A fault injection campaign is a set of fault injections using different parameters concerning the position or duration of the glitch (voltage glitch in our experiment) It is essential to retrieve necessary data for the evaluation of the security of a certain hardware against fault injection attacks and the successful execution of such an attack.

### 6.1.1 Implementation Details

The implementation of such a campaign follows the same preparatory steps described in the Chapter 4 for compiling the firmware and programming the microcontroller, as well as the initialization of the AES. However, instead of performing a single fault injection, it iterates through different parameters. The process is represented graphically using Jupyter and the matplotlib library. Moreover, our implementation exports the faulty ciphertexts to MATLAB-compatible files, so that we can use them to derive the encryption key using DFA.

In the initialization phase of the fault injection campaign, the **chipwhisperer.common.results.glitch** module library is imported and an instance of this class is created. The **repeat** and **ext_offset** parameters are defined within a range of values that will be iterated, with a step size of one. Additionally, we specify the bytes that will be targeted with the fault injection. The time of initiation of the process is displayed.

### 6.1.2 Iterative Fault Injection Process

A loop is initiated iterating through the defined bytes targeted by the glitch. Within this loop, a **counter** parameter is set to zero, as it counts the number of successful fault injections accomplished for the given byte. A second loop is implemented, iterating through the values in **gc.glitch_values(),** which consists of an object containing multiple lists of values for the **repeat** and **ext_offset** parameters. This second loop is using different parameters to achieve a useful fault injection.

Subsequently, a third *for* loop begins. This *for* loop iterates ten times to validate the reliability of the process, ensuring that potential variations are accounted for. In this loop the fault injection is executed using the defined parameters. The core process has been already described in Chapter 4.3, so an extensive presentation will be omitted at this point.

The Chipwhisperer predefined functions handle the fault injection using the defined parameters. These functions manage various aspects of the fault injection process, including sending the cleartext to the device, receiving the ciphertext and resetting the device in case of a time-out. The algorithm identifies if the ciphertext has been corrupted due to the glitch, and if it is unique for the

specific targeted byte, the **counter** parameter is incremented by 1. The faulty ciphertext is then added to an array, accompanied by associated data, such as the contents of the state array post-injection and the injection parameters. These collected data will be exported upon the process's completion.

If the **counter** parameter now holds the value of 2, indicating that a total of 2 unique faults have been successfully collected for a specific byte, the current loop terminates with a break statement, given that two faults are necessary at every targeted byte to facilitate the retrieval of the encryption key, as previously described in the context of byte-level DFA. Following this, the injection process proceeds to target another byte within the AES encryption, specifically, it advances to the next target byte.

Furthermore, if the ciphertext remains uncorrupted and the **ciphertext_flag** parameter is set to True, the ciphertext is stored in a separate array. In addition to the core fault injection process, we have implemented actions to visually represent successful and unsuccessful fault injections. These graphical representations enhance our ability to identify patterns and anomalies effectively, as it is illustrated on Figure 6.1.

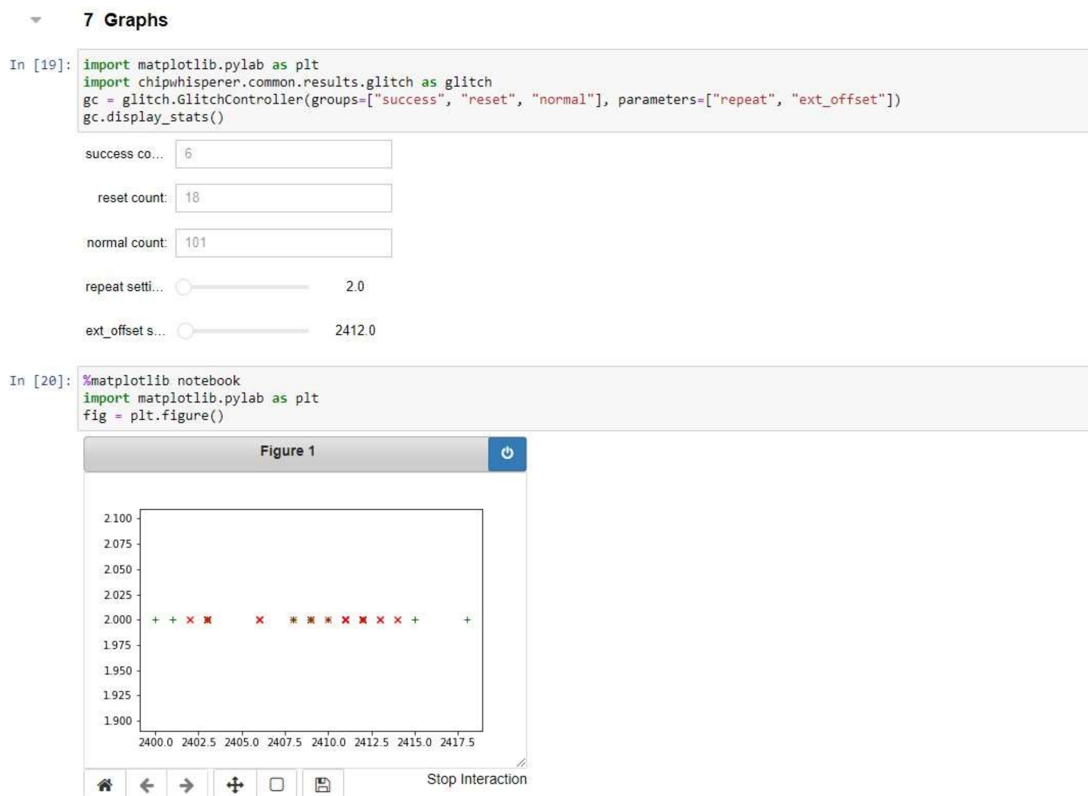



**Figure 6.1 The results are represented graphically in real-time**

Upon completion of the process when two faulty ciphertexts have been collected for the targeted bytes, or when all possible injection parameter combinations have been exhausted, the execution concludes. The completion time is then displayed. A sample output illustrating this scenario is presented in Figure 6.2, where two faulty ciphertexts have been successfully retrieved for the targeted bytes 0, 3, and 5.

```
2023-09-17 21:12:17.317131 Start
plaintext: CWbytearray(b'b7 7d b4 71 63 6e 5d 9f 2b 1f 85 ff 22 96 69 ac')
State: {'valid': True, 'payload': CWbytearray(b'51 ab d2 f5 65 09 54 13 c1 80 18 d5 a0 f7 06 9a'), 'full_response': 'r51ABD2F56
5095413C18018D5A0F7069A\n', 'rv': None}
{'valid': True, 'payload': CWbytearray(b'06 a2 09 30 f8 2a 8c 49 4c 54 77 ca 51 5a 5b 98'), 'full_response': 'r06A20930F82A8C49
4C5477CA515A5B98\n', 'rv': 0}
CWbytearray(b'06 a2 09 30 f8 2a 8c 49 4c 54 77 ca 51 5a 5b 98')
2 2401

repeat: 2 ext offset: 2401 byte injected: 0
State: {'valid': True, 'payload': CWbytearray(b'd1 63 63 63 65 09 54 13 c1 80 18 d5 a0 f7 06 9a'), 'full_response': 'rD16363636
5095413C18018D5A0F7069A\n', 'rv': None}
{'valid': True, 'payload': CWbytearray(b'0c 71 aa 7a f5 82 8d ef f7 e3 36 29 db 20 85 3e'), 'full_response': 'r0C71AA7AF5828DEF
F7E33629DB20853E\n', 'rv': 0}
CWbytearray(b'0c 71 aa 7a f5 82 8d ef f7 e3 36 29 db 20 85 3e')
2 2418

repeat: 2 ext offset: 2418 byte injected: 0
State: {'valid': True, 'payload': CWbytearray(b'd1 ab d2 77 65 09 54 13 c1 80 18 d5 a0 f7 06 9a'), 'full_response': 'rD1ABD2776
5095413C18018D5A0F7069A\n', 'rv': None}
{'valid': True, 'payload': CWbytearray(b'1e 52 90 50 3d 5b 83 7e ec 4e e8 31 5f 7f ef 1a'), 'full_response': 'r1E5290503D5B837E
EC4EE8315F7FEF1A\n', 'rv': 0}
CWbytearray(b'1e 52 90 50 3d 5b 83 7e ec 4e e8 31 5f 7f ef 1a')
2 2400

repeat: 2 ext offset: 2400 byte injected: 3
State: {'valid': True, 'payload': CWbytearray(b'd1 ab d2 e0 65 09 54 13 c1 80 18 d5 a0 f7 06 9a'), 'full_response': 'rD1ABD2E06
5095413C18018D5A0F7069A\n', 'rv': None}
{'valid': True, 'payload': CWbytearray(b'30 b2 44 d8 aa 05 79 60 45 ed f7 d1 e1 0d c6 ae'), 'full_response': 'r30B244D8AA057960
45EDF7D1E10DC6AE\n', 'rv': 0}
CWbytearray(b'30 b2 44 d8 aa 05 79 60 45 ed f7 d1 e1 0d c6 ae')
2 2429

repeat: 2 ext offset: 2429 byte injected: 3
State: {'valid': True, 'payload': CWbytearray(b'd1 ab d2 f5 65 40 54 13 c1 80 18 d5 a0 f7 06 9a'), 'full_response': 'rD1ABD2F56
5405413C18018D5A0F7069A\n', 'rv': None}
{'valid': True, 'payload': CWbytearray(b'75 7f fa d3 c9 1f 94 33 57 b9 04 87 72 f7 54 41'), 'full_response': 'r757FFAD3C91F9433
57B9048772F75441\n', 'rv': 0}
CWbytearray(b'75 7f fa d3 c9 1f 94 33 57 b9 04 87 72 f7 54 41')
2 2404

repeat: 2 ext offset: 2404 byte injected: 5
State: {'valid': True, 'payload': CWbytearray(b'd1 ab d2 f5 65 3b 54 13 c1 80 18 d5 a0 f7 06 9a'), 'full_response': 'rD1ABD2F56
53B5413C18018D5A0F7069A\n', 'rv': None}
{'valid': True, 'payload': CWbytearray(b'51 21 6d df d0 8d 7e bf 31 22 1d 0f 8b 89 f1 75'), 'full_response': 'r51216DDFD08D7EBF
31221D0F8B89F175\n', 'rv': 0}
CWbytearray(b'51 21 6d df d0 8d 7e bf 31 22 1d 0f 8b 89 f1 75')
2 2433

repeat: 2 ext offset: 2433 byte injected: 5
2023-09-17 21:17:52.833257 End
```

**Figure 6.2 Sample fault injection process output in Jupyter**

### 6.1.3 Data Export to MATLAB

The final step in completing the fault campaign involves exporting data to a .mat file, a MATLAB-compatible format. The file will be used for further computations and analysis. The exported data includes:

More specifically, the following are exported:

- The AES encryption key used.
- The plaintext.
- The ciphertext.
- The state array after the SubBytes operation at the 8$^{th}$ round of AES.
- The six faulty ciphertexts.
- The faulty state arrays after the SubBytes operation at the 8$^{th}$ round of AES.
- The offset parameter for each successful injection.
- The sample rate.

The exported data will serve as the essential input for the application of the byte-level Differential Fault Analysis (DFA) attack, enabling a detailed and systematic analysis of the fault injection campaign's results.

## 6.2 Execution of DFA Attack

The exported file including the data from the AES encryption under the effect of the fault injection during the 8th round of the encryption process is imported to MATLAB to our analysis platform, MATLAB.

A significant finding in the exported file was the corruption of multiple bytes during the voltage fault injection process targeting the 0th byte of the AES. The following tables depict the uncorrupted and corrupted state arrays at the 8th round of AES. It is evident that three bytes have been altered due to the insertion of the voltage glitch. The altered bytes were specifically bytes 1, 2, and 3, with values of 99 in decimal or 63 in hexadecimal.

| **108** | **228** | **15** | **40** |
|---|---|---|---|
| **53** | 33 | 134 | 10 |
| **170** | 147 | 131 | 109 |
| **48** | 7 | 98 | 243 |

**Figure 6.3 Original state array**

| **108** | **99** | **99** | **99** |
|---|---|---|---|
| **53** | 33 | 134 | 10 |
| **170** | 147 | 131 | 109 |
| **48** | 7 | 98 | 243 |

**Figure 6.4 Corrupted state array**

Therefore, it is conceivable that overcoming this failure could have been achieved through the extraction of an additional faulty ciphertext for the 0th byte of AES, as the failed cases exhibited this specific pattern, or re-execute the experiment using a different duration for the glitch on the 0th byte.

However, it was possible to perform a byte-level DFA attack successfully against the collected data using the faulty ciphertexts from the fault injection against 5th byte of the state array instead of the 0th byte refactoring the MATLAB script described in Chapter 5.6 for Byte-Level DFA.

The algorithm for the successful retrieval of the AES is the following:

1) Import .mat file from Chipwhisperer.
2) Perform byte-level DFA attack using the crackDFA() function with the FaultMap parameter 1,2 and 4 against the faulty ciphertexts from the fault injection targeting the 5th byte of AES.
3) Perform byte-level DFA attack using the crackDFA() function with the FaultMap parameter 3 against the faulty ciphertexts from the fault injection targeting the 3rd byte of AES.
4) Reverse the KeyExpansion and display the retrieved encryption key.

The Figure 6.5 illustrates a successful execution of the DFA attack leading to the retrieval of the encryption key which was used by our firmware.

```
1 -     clear all;
2 -     addpath(genpath('AES'));
3
4 -     load('results/dfa8_result1.mat')
5
6 -     [sbox, inv_s_box, w, poly_mat, aesmult_table] = initialization();
7
8 -     crackedKey=[];
9 -     tic;
10
11 -    [crackedKey(1,1),crackedKey(2,4),crackedKey(3,3),crackedKey(4,2)]= ...
12           crackDFA(ciphertext,faultycipher51,faultycipher52,1);
13
14 -    [crackedKey(2,3),crackedKey(3,2),crackedKey(4,1),crackedKey(1,4)]= ...
15           crackDFA(ciphertext,faultycipher31,faultycipher32,2);
16
17 -    [crackedKey(1,3),crackedKey(2,2),crackedKey(4,4),crackedKey(3,1)]= ...
18           crackDFA(ciphertext,faultycipher51,faultycipher52,3);
19
20 -    [crackedKey(3,4),crackedKey(4,3),crackedKey(2,1),crackedKey(1,2)]= ...
21           crackDFA(ciphertext,faultycipher51,faultycipher52,4);
22
23 -    w2=rev_key_schedule(crackedKey);
24 -    cracked_AES_key =  reshape (w2(1:4,1:4)', 16, 1)';
25 -    if cracked_AES_key==AESkey
26 -        display("AES encryption key has been found")
27 -         print_key(w2);
28 -    end
29
30 -    toc;
```

demo_crackdfa_cw.m

Command Window

```
>> demo_crackdfa_cw
    "AES encryption key has been found"

Initial AES-128 encryption key in hexadecimal format:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
Elapsed time is 2.401000 seconds.
```

**Figure 6.5 Successful run of DFA against real fault injection data**

# 7 Results

This thesis has presented a comprehensive framework for conducting fault injections against a microprocessor executing AES and performing a byte-level Differential Fault Analysis (DFA) attack. In this section, we evaluate the efficiency and reliability of the implemented platform while addressing potential weaknesses. Achieving a voltage fault injection campaign at the single-bit level proved challenging. However, a more precise targeting is possible through alternative fault injection methods, such as optical fault injection.

## 7.1 Experiment Setting

To assess the byte-level DFA attack, we conducted one hundred (100) fault injection campaigns targeting the 0th, 3rd, and 5th bytes of the AES encryption process. The fault injections were executed randomly, with the **ext_offset** parameter range between 2400 and 3000 clock cycles from the trigger point. Additionally, the **repeat** parameter was set to 2, resulting in a voltage glitch duration of 16.6 nanoseconds (2 x 8.3 ns). These experiment parameters were chosen following extensive iterations, deemed optimal for achieving successful fault injections.

### 7.1.1 High-Level Algorithm for Experiment

The implementation of this experiment necessitated a slight modification of the algorithm previously described for executing fault injection campaigns. The high-level algorithm can be summarized as follows:

```
1) Repeat for 100 times:
       Set a random plaintext.
       Execute a fault injection campaign.
       Export the results.
2) Perform byte-level DFA attack.
```

## 7.2 Success Rate

An essential metric for evaluating the results of our experiments is the success rate. Following the completion of the aforementioned experiment, our findings demonstrated a notably high success rate, with 86% of the executions resulting in the successful retrieval of the encryption key. This achievement is visually represented in the following pie chart presented below as Figure 7.1.

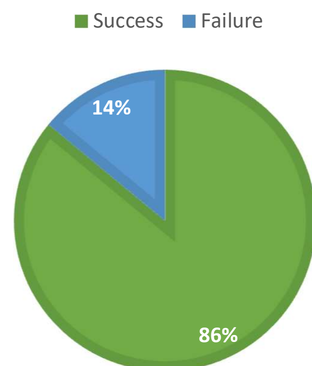



**Figure 7.1 Success rate in Byte-level DFA (ChipWhisperer)**

This failure could be associated with either the reliability of the algorithm used for the DFA attack or the fault injection technique. To answer this question the same experiment was executed for random data occurred during a simulated attack from MATLAB using both Byte-level and Bit-level DFA attacks. The following pie charts demonstrate the success rate.

**BYTE-LEVEL DFA ATTACK USING SIMULATION**

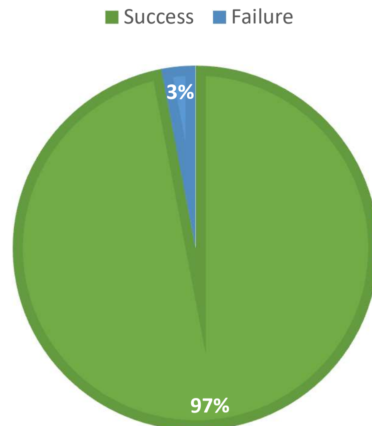



**Figure 7.2 Success rate in Byte-level DFA (Simulation)**

**BIT-LEVEL DFA ATTACK USING SIMULATION**

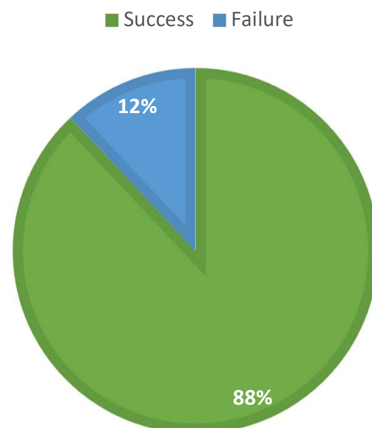



**Figure 7.3 Success rate in Bit-level DFA (Simulation)**

Consequently, the existence of failures in both actual hardware generated data and simulated data suggest that the fault injection technique used is trustworthy. The slight differentiation of the success rate between the simulated and experimental execution of the byte-level DFA, could be an indication that certain state bytes may have been corrupted during the voltage fault injection. The reason for the existence of failures is that certain ciphertexts lead to multiple candidate keys. The repetition of the attack with additional faulty ciphertexts would eliminate supplementary candidate keys and lead to the successful retrieval of the encryption key. Further, it is important to note that even in cases of failure, the algorithm did not produce any inaccurate encryption keys; it simply did not find the correct key.

## 7.3 Time Duration

The time required for the successful execution of a DFA attack is an important metric for the evaluation of its consistency and practical feasibility. The plot in Figure 7.4 demonstrates the time duration of the fault injection and DFA attack applied while implementing the attack with ChipWhisperer. The table of average values is illustrated in Figure 7.5

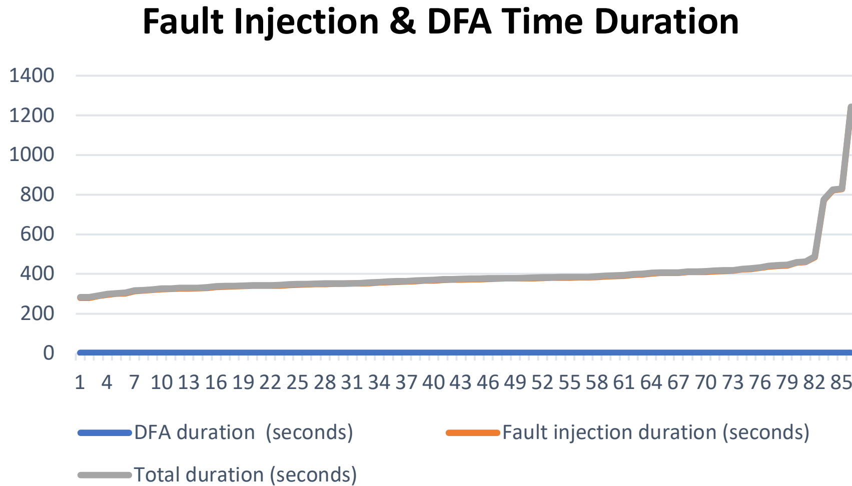



**Figure 7.4 Time duration of the attack using ChipWhisperer**

| Parameter | Value |
|---|---|
| Average Fault Injection Duration | **394.9 seconds** |
| Average DFA Attack Duration | **2.38 seconds** |
| Total Duration Average | **397.34 seconds** |

**Figure 7.5 Average duration per attack phase**

It is important to note that only 4.6% of the values deviate significantly from the average total duration of 397.34 seconds, with a difference of more than 129.8 seconds, the standard deviation of our values. This observation is visually represented by the standard deviation graph in Figure 7.6, indicating a relatively minor deviation from the average duration. Furthermore, it is evident that the required amount of time for the execution of the DFA attack is insignificant in comparison with the time required for the execution of the voltage fault injection.

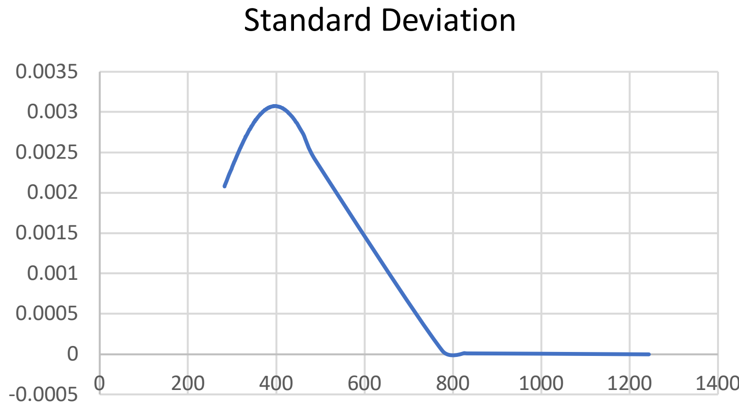



**Figure 7.6 Standard deviation of DFA attack total time duration**

Furthermore, it is intriguing to compare the time durations required for executing core DFA calculations in three different scenarios: Byte-level DFA on actual data, Byte-level DFA using simulated data, and simulated Bit-level DFA. The simulated Byte-level DFA attack takes slightly more time than the one using actual data. However, it is important to note that this calculation is not precisely accurate as it includes the time for both the actual encryption and fault injection simulation. On the other hand, the simulated Bit-level DFA is remarkably fast, with an average execution time of 0.1 seconds.

| Parameter | Value |
| --- | --- |
| Byte-level DFA duration (seconds) | **2.380 seconds** |
| Simulated Byte-level DFA duration (seconds) | **3.321 seconds** |
| Simulated Bit-level DFA duration (seconds) | **0.106 seconds** |

**Figure 7.7 Average durations of core DFA calculation per scenario**

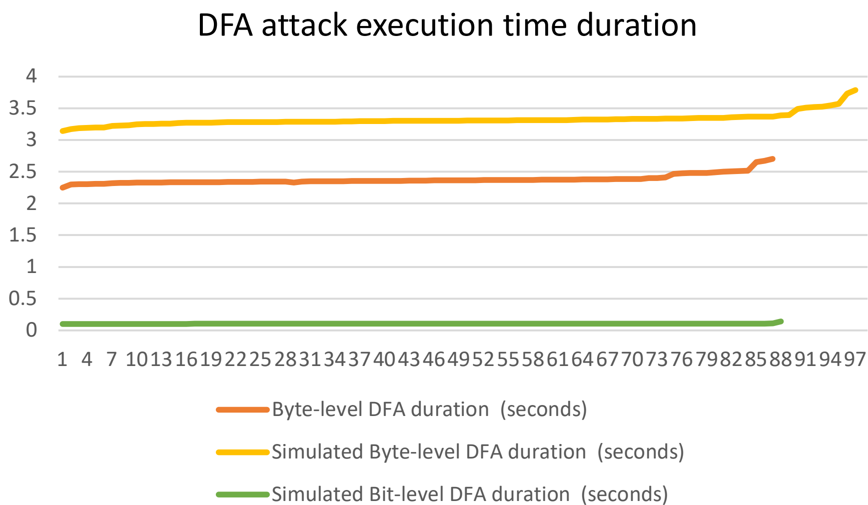



**Figure 7.8 DFA attack execution time per scenario**

# 8 Conclusions

In this thesis, we delved into Differential Fault Analysis (DFA) attacks on AES encryption implemented using ChipWhisperer and MATLAB. The process of the completion of this thesis, although marked by challenges, has provided valuable insights into the complexities and strengths of fault injections.

## 8.1 Key Findings

The experiments conducted revealed a success rate of 86% in retrieving encryption keys through byte-level DFA attacks. Simulated executions of the byte-level and bit-level DFA attacks led to a success rate of 97% and 88% respectively. The cases of failures to retrieve the encryption key are associated with the existence of more than one encryption keys. However, even in cases of failure, the algorithm did not produce any inaccurate results, such as the retrieval of a wrong encryption key.

Furthermore, an experimental implementation of bit-level DFA using voltage fault injection was not practically feasible.

However, it is important to underline the efficiency of bit-level DFA computations, taking merely 0.1 seconds. In comparison, byte-level DFA, though slightly more time-intensive at 2.4 seconds, demonstrated remarkable efficiency.

## 8.2 Countermeasures, Implications and Future Directions

Various implementations against fault injection and fault analysis exists. As we used an educational platform to perform voltage fault injections, voltage glitch detectors and hardening techniques are beyond the scope of our thesis. Other techniques, like Spatial, Temporal or Parity Redundancy *(Patranabis and Mukhopadhyay 2018)* introduce countermeasures within the AES execution itself which poses challenges against the implementation of countermeasures attacks. Spatial and Temporal Redundancy perform duplicate execution of AES calculations in parallel to validate the encryption process. Likewise, the Parity Redundancy technique involves the use of one or more parity bits for the detection of fault injections.

These techniques can be bypassed using multiple fault injections in parallel but require very precise glitches for optimal results. Such techniques have been efficient even against the most sophisticated hardware-level countermeasures as it was demonstrated by *((Saß, Mitev and Sadeghi 2023).*

The study of such countermeasures would pose new challenges to our research. Furthermore, understanding patterns in injection failures could lead to interesting observation, such as the possibly deterministic nature of these events.

Other future directions could include the study of other fault injection techniques, like optical or electromagnetic fault injections.

## 8.3 Final Thoughts

In conclusion, it is essential to acknowledge that this thesis has underlined the realistic nature of voltage fault injection and byte-level differential fault analysis attacks against modern cryptographic algorithms. While our study may not have led into uncharted territories of cybersecurity, it does prove the efficiency of well-established methodologies and presents an experimental implementation and framework for fault analysis. In closing, I trust this research offers valuable insights into fault injection attacks, benefiting future hardware security studies.

# 9 Bibliography

Aumasson, Jean-Philippe. *Serious Cryptography*. San Francisco, CA: No Starch Press, 2018.

Bar-El, Hagai, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. *The Sorcerer's Apprentice Guide to Fault Attacks*. Proceedings of the IEEE, vol. 94, no. 2, pp. 370–382, 2006.

Daemen, Joan, and Vincent Rijmen. *The design of Rijndael*. Vol. 2. New York: Springer-verlag, 2002.

Dunkelman, Orr, and Nathan Keller. "The effects of the omission of last round's MixColumns on AES." *Information Processing Letters* 110, no. 8-9 (2010): 304-308.

Dusart, Pierre, Gilles Letourneux, and Olivier Vivolo. "Differential fault analysis on AES." In *International Conference on Applied Cryptography and Network Security*, pp. 293-306. Springer, Berlin, Heidelberg, 2003.

Dworkin, Morris. "NIST Special Publication 800-38A 2001 Edition." *NIST Special Publication* 800 (2001): 38A.

Mukhopadhyay, Debdeep, and Rajat Subhra Chakraborty. *Hardware security: design, threats, and safeguards*. CRC Press, 2014.

NIST Computer Security Resource Center. *Cybersecurity*. National Institute of Standards and Technology, Accessed October 21, 2023, https://csrc.nist.gov/glossary/term/cybersecurity.

Nyberg, Kaisa. "Differentially uniform mappings for cryptography." In *Workshop on the Theory and Application of Cryptographic Techniques*, pp. 55-64. Springer, Berlin, Heidelberg, 1993.

O' Flynn, Colin. *Fault Injection using Crowbars on Embedded Systems*. Cryptology ePrint Archive, Paper 2016/810. 2016.

Paar, Christof, and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2010.

Patranabis, Sikhar, and Debdeep Mukhopadhyay, eds. *Fault tolerant architectures for cryptography and hardware security*. Singapore: Springer, 2018.

Sakiyama, Kazuo, Yu Sasaki, and Yang Li. *Security of block ciphers: from algorithm design to hardware implementation*. John Wiley & Sons, 2016.

Woudenberg, Jasper van, and Colin O'Flynn. *The Hardware Hacking Handbook*. San Francisco, CA: No Starch Press, 2022.

Saß, Xhani Marvin, Richard Mitev, and Ahmad-Reza Sadeghi. *Oops..! I Glitched It Again! How to Multi-Glitch the Glitching-Protections on ARM TrustZone-M*. In 32nd USENIX Security Symposium (USENIX Security 23), 6239-6256. Anaheim, CA: USENIX Association, 2023. https://www.usenix.org/conference/usenixsecurity23/presentation/sass