Analysis and Design of Clock-glitch Fault Injection within an FPGA

by

Masoumeh Dadjou

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2013

Abstract

In modern cryptanalysis, an active attacker may induce errors during the computation of a cryptographic algorithm and exploit the faulty results to extract information about the secret key in embedded systems. This kind of attack is called a fault attack. There have been various attack mechanisms with different fault models proposed in the literature. Among them, clock glitch faults support practically dangerous fault attacks on cryptosystems. This thesis presents an FPGA-based practical testbed for characterizing exploitable clock glitch faults and uniformly evaluating cryptographic systems against them. Concentrating on Advanced Encryption Standard (AES), simulation and experimental results illustrates proper features for the clock glitches generated by the implemented on-chip glitch generator. These glitches can be injected reliably with acceptably accurate timing. The produced faults are random but their effect domain is finely controllable by the attacker. These features makes clock glitch faults practically suitable for future possible complete fault attacks on AES. This research is important for investigating the viability and analysis of fault injections on various cryptographic functions in future embedded systems.

Acknowledgements

I would like to express my sincere thanks to my supervisor, Prof. Cathy Gebotys, for her valuable advices, guidance and encouragement throughout my graduate studies. This thesis would not have been possible without her kind support and motivation.

In addition, thanks to my colleagues in Prof. Gebotys's laboratory at the University of Waterloo: Dr. Edgar Mateos Santillan, Dr. Marcio Juliato, Farhad Haghighizadeh, and Najma Jose for their encouragement and friendship. I would like to especially thank Edgar Mateos Santillan for his help with the laboratory part of this thesis.

Finally, I would like to express my deepest gratitude to my parents, Behrooz Dadjou and Zohreh Amiri, and my siblings, Hossein, Mahsa and Mahla, for their unconditional love and irreplaceable support throughout my life.

This thesis is dedicated to my parents

for their unconditional love, support and encouragement

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Security is a major concern in computing, communication systems, and e-commerce among others and significant ongoing research is underway to address this concern. Cryptographic algorithms such as symmetric ciphers, public-key ciphers, and hash functions are used to construct security mechanisms. Modern cryptography relies on the assumption that cryptography primitives are perfectly secure; however, it is an unproven assumption. For example, public-key cryptosystem is based on the idea of one-way functions–functions which are easy to compute but computationally infeasible to invert. But, it has not been proven that the functions used in public-key cryptosystem are one-way functions or even whether one-way functions really exist. Still, some cryptographic algorithms are unbroken and remain widely used. Hence, some algorithms are assumed to be secure and unbreakable.

Cryptographic algorithms can be viewed from two perspectives: classical and side channel. From a classical perspective, they are perceived as a mathematical object which receives a plaintext or message and using a key, transforms the input into a ciphertext or vice-versa. The adversaries are assumed to have a complete understanding of the underlying algorithm and access to both ciphertext and plaintext. In addition, they may have more control over the exchanged data between the two parties e.g., by selecting the plaintexts in a chosen-plaintext attack, or by selecting the ciphertexts in a chosen-ciphertext attack. The attackers are only lacking knowledge of the secret key. This is the black-box view depicted in figure 1.1 in which the attacker attempts to solve the underlying problems–assumed to be computationally intractable–in the cryptographic protocol by exploiting the mathematical specifications without necessarily having to implement the algorithms and use the system.

input
Encryption/signature
output

Figure 1.1: The Black Box view in traditional cryptography

1.1 Side Channel Attacks

In the real world, cryptographic protocols have to be implemented in software or hardware on a device which interacts with its environment. This interaction with the environment can be monitored by the adversary. For example, the device gives information about the time or power consumption of operations and produces sounds or electromagnetic radiations during the computations. In addition, the device may have some unintentional inputs such as voltage or clock frequency which may be modified to create predictable or faulty outputs. Such information leakage during the protocols execution is called the side channel which is not considered in the classical security model. Figure 1.2 represents the cryptographic view including side channels. In side channel analysis (SCA), the adversary goes beyond the pure mathematics and takes advantage of side channel information to recover the secret key. This kind of attack, which was presented in 1996 by Kocher [33] for the first time, works because, the state of the cryptographic device, which is related to the secret key, can be correlated with the measured side channel information.

1.1.1 Classifications of Side Channel Attacks

Side-channel attacks can be classified into the following three categories where these categories are orthogonal i.e. a specific type of attack can be in more than one category [49].

Control over the process: Passive vs. Active

The first category is based on the level of control over the process execution and divides the SCA attacks into ***passive attacks*** and ***active attacks***. In a passive attack, the attacker does not interfere with the device's computation and gains information by simply observing the device behavior, which is exactly as if there is no attack. Passive attacks are

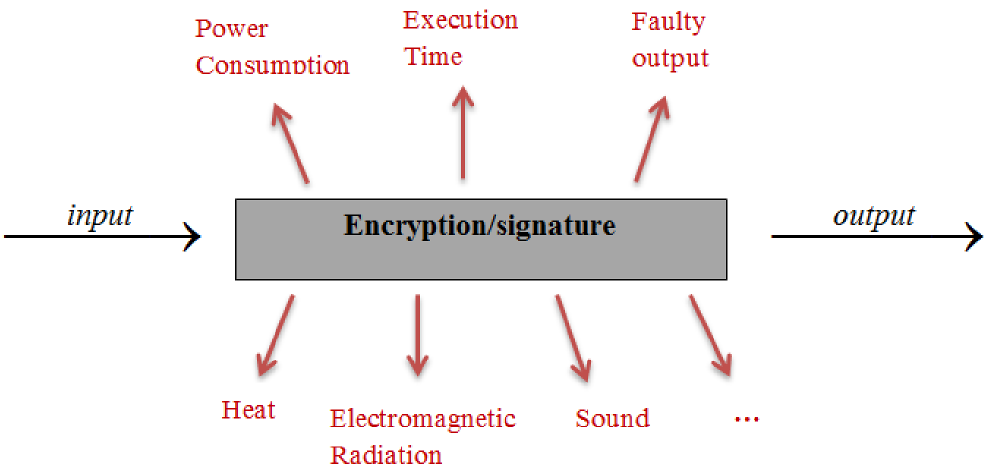


Figure 1.2: The Side channel view of the cryptography

based on the observation and analysis of various measurable side channels including power consumption, computation time, and electromagnetic or radio frequency emanation of the device. They use the correlation between the operations, processed data and collected measures. While in an active attack, the attacker tampers with the functioning device and tries to affect its behavior. Active attacks are based on interrupting the algorithm process to cause an abnormal behavior or erroneous result that can be exploited to recover the secret key.

Access to the device: Invasive vs. Non-invasive

Depending on the attack surface (the set of physical, electrical and logical interfaces that are exposed to a potential adversary) Anderson et al. in [4] divided the attacks into three classes: ***invasive attacks***, ***semi-invasive attacks*** and ***non-invasive attacks***. Invasive attacks require depackaging the device to have direct access to its components, for example, placing a probing needle on a data bus to see the data transfer. A non-invasive attack is usually an undetectable and low-cost attack that only uses the side-channel information which is externally available. Timing analysis and power consumption analysis are in this class. The semi-invasive attack involves depackaging of the device to get access to its surface, but does not require electrical contact to the metal layer. For example, in a fault induction attack, the attacker may use a laser beam to ionize a device and change some of its memories to change the output.

Analysis process: Simple vs. Differential

The sampled data in a side-channel attack needs to be analyzed to reveal information about the secret key. In the analysis process, if a single side-channel trace is used and the secret key is directly related to that trace, the attack is referred to as a ***simple side channel attack (SSCA)***. While, if due to too much noise in the measurements SSCA is not feasible, a ***differential side channel attack (DSCA)*** is used. In DSCA many traces are used to exploit the correlation between the processed data and the side-channel leakage. Since this correlation is usually small, statistical methods are needed to exploit it efficiently.

1.1.2 Fault Attacks

A fault attack is a fundamentally different kind of side-channel attack. Here, the adversary induces faults into the device during the computation of a cryptographic algorithm and observes its behavior. Other side-channel attacks are passive in which there is no tampering with the attacked device. While, a fault attack is an active attack which aims at recovering the secret key by altering the computation process and analyzing the faulty output or reaction of the system.

The first successful fault attacks have been reported by Boneh et al. in 1997 [14] and later in 2001 on the RSA signature scheme, the Fiat-Shamir and Schnorr identification protocols [15]. They injected random hardware faults and showed that an RSA implementation based on the Chinese Remainder Theorem (CRT) can be broken using a single erroneous RSA signature. A non-CRT implementation of RSA needs a larger number (e.g. 1000) of erroneous signatures. Their results also illustrates that the secret key can be revealed in Fiat-Shamir and Schnorr identification protocols after a small number (e.g. 10) and a larger number (e.g. 10000) of faulty executions respectively. Fault attack on ElGamal, Schnorr and digital signature algorithm (DSA) was presented by Bao et al. in 1998[9]. Biehl et al. in 2000[11] presented the fault attacks on elliptic curve public-key encryption. In 1997 Biham and Shamir[12] described fault attacks on the DES symmetric-key encryption. Transient faults or glitches were discussed by Anderson and Huhn in 1997[7]. In 2002 Skorobogatov and Anderson[46] proposed a powerful and practical optical fault attack using inexpensive equipment on a smartcard.

The fault attacks mentioned above indicate that this kind of attack may be practical and very realistic to mount, although more advanced knowledge and tools are needed.

Table 1.1: Key-Block-Round Combinations [37]

	Key Length (Nk words)	Block Size (Nb words)	Number of Rounds (Nr)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Thus, these fault attacks may be much more effective and dangerous. Fault attacks can break an unprotected system more quickly than any other kind of side-channel attack such as power analysis or electromagnetic analysis. For example, the attacker can break RSA-CRT with one faulty result, and DES and AES with two. Furthermore, the fault attack countermeasures are more costly in terms of chip area. Hence, they must be taken into account during the design and test of secure systems and devices.

1.2 Introduction to AES

The Advanced Encryption Standard (AES) algorithm is a symmetric 128-bit block cipher with a key of length 128, 192 or 256, and the corresponding number of rounds for each is 10, 12, and 14 rounds respectively. The Key-Block-Round combinations is given in Table 1.1. From the original key, a different round key is computed for each of these rounds. Consider the key length of 128 bits and hence 10 rounds. AES operates on a 4×4 array of bytes named a state. This state undergoes 4 transformations in each round, in order, called ***SubBytes***, ***ShiftRows***, ***MixColumns***, and ***AddRoundKey***. Before the first round (round 0), the block is processed by AddRoundKey. The last round (round 10) skips the MixColumns step. The complete description of the algorithm can be found in [37]. Figure 1.3 shows the encryption process in AES algorithm.

1.2.1 AES Functions

SubBytes

SubByte transformation is a highly nonlinear byte substitution where each byte is replaced with another from a look up table called Sbox. This nonlinear function involves finding the

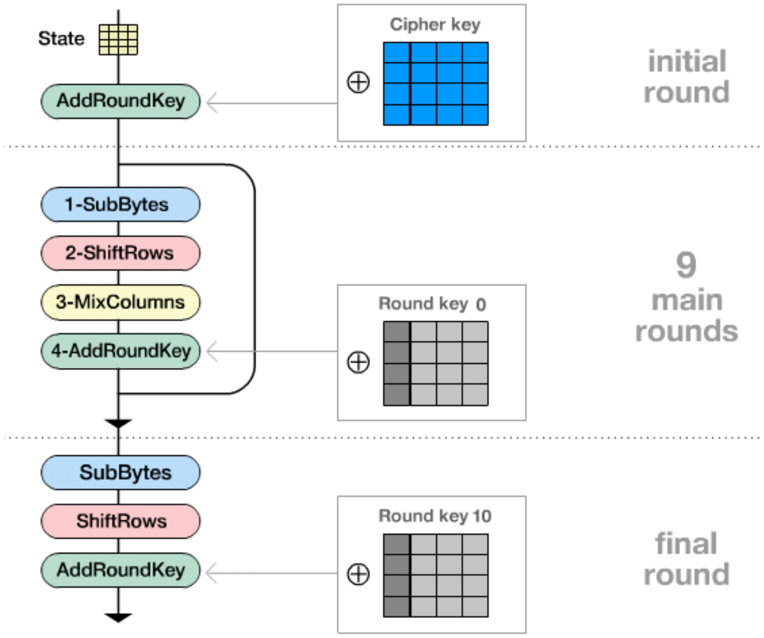


Figure 1.3: AES encryption process

inverse of the 8-bit numbers as elements of the Galois Field $GF(2^8)$. The Sbox function of an input byte a is defined by two substeps:

1. $b = a^{-1}$: Multiplication inverse in $GF(2^8)$ (inverse of 0 is 0)
2. $a' = M.b + c$: Affine transformation (linear plus a constant)

Figure 1.4 depicts the SubBytes transformation on the State. The affine transformation element of the Sbox can be expressed as:

$$\begin{pmatrix} a'_0 \\ a'_1 \\ a'_2 \\ a'_3 \\ a'_4 \\ a'_5 \\ a'_6 \\ a'_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

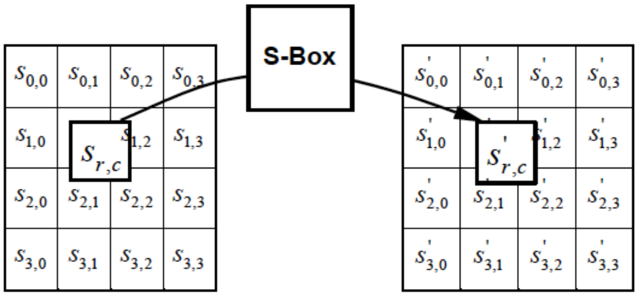

S-Box
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{r,c}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$
$s'_{r,c}$

Figure 1.4: SubBytes [37]

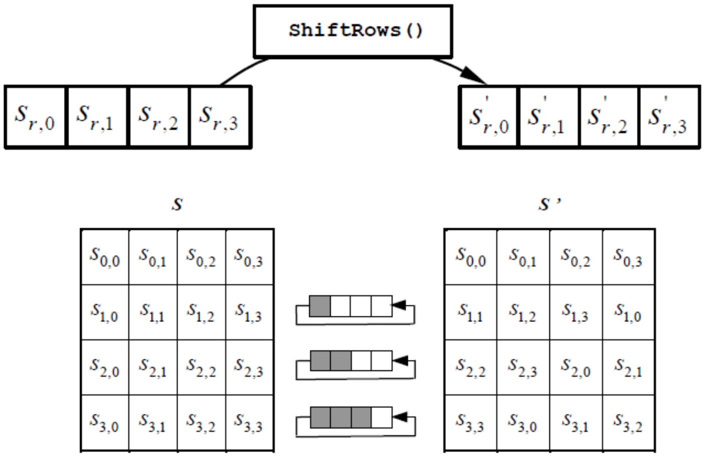

ShiftRows()
$s_{r,0}$ $s_{r,1}$ $s_{r,2}$ $s_{r,3}$
$s'_{r,0}$ $s'_{r,1}$ $s'_{r,2}$ $s'_{r,3}$
s
s '
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,1}$ $s_{1,2}$ $s_{1,3}$ $s_{1,0}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,1}$
$s_{3,3}$ $s_{3,0}$ $s_{3,1}$ $s_{3,2}$

Figure 1.5: ShiftRows transformation [37]

ShiftRows

The ShiftRows transformation cyclically shifts the last three rows in the State and the first row, $r = 0$, is not shifted. The effect is moving the bytes to lower position in the row as illustrated in Figure 1.5.

MixColumns

In MixColumns, each column of the State is treated as a polynomial over $GF(2^8)$ and multiplied with a fixed polynomial $a(x)$ modulo $x^4 + 1$ where $s'(x) = a(x) \otimes s(x)$ and:

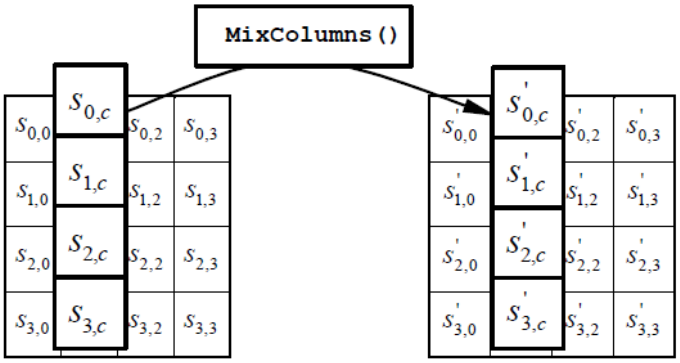

MixColumns()
$s_{0,0}$ $s_{0,c}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,c}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,c}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,c}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,c}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,c}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,c}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,c}$ $s'_{3,2}$ $s'_{3,3}$

Figure 1.6: MixColumns transformation [37]

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}.$$

The matrix multiplication is as follows and the MixColumns transformation is illustrated in Figure 1.6

$$\begin{pmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{pmatrix} for 0 \le c < Nb$$

AddRoundKey

As Figure 1.7 shows, AddRoundKey transformation is simply a bitwise XOR between each column of the State and a word of the key matrix from the key schedule.

KeyExpansion

The KeyExpansion schedule computes the round keys. Figure 1.8 shows the pseudo code of KeyExpansion and a block diagram of this function is displayed in Figure 1.9. The KeyExpansion consists of SubWord and RotWord functions and Rcon Table. The SubWord accepts a word as input, performs the SubBytes on each of the four bytes and returns a word. The Rotword applies a cyclic permutation on a word (s_0, s_1, s_2, s_3) and outputs the word (s_1, s_2, s_3, s_0). Rcon is a constant word array which contains the values $(x^{i-1}, 0, 0, 0), i \in 1, 2, 3, 4$.

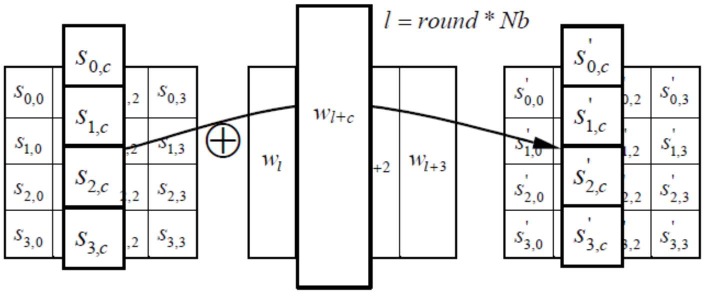


Figure 1.7: AddRoundKey transformation [37]

```
key[]: The input 16-byte key.
w[]: The resulting round keys, stored in an array of four-byte words.

KeyExpansion(byte key[16], word w[44]){
    word temp;
    for (i = 0; i < 4; i++)
      w[i] = (key[4*i], key[4*i+1], key[4*i+2], key[4*i+3]);
    for (i = 4; i < 11; i++){
      temp = w[i-1];
      if (i mod 4 == 0)
        temp = SubWord(RotWord(temp)) ⊕ Rcon[i/4];
      w[i] = w[i-4] ⊕ temp;
    }
}
```

Figure 1.8: Pseudo code of the KeyExpansion schedule in AES_128 [17]

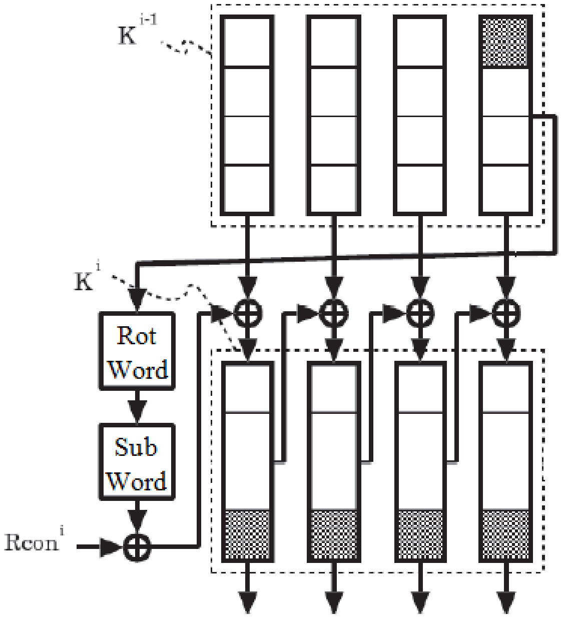


Figure 1.9: The block diagram of the KeyExpansion schedule in AES_128

1.2.2 Sbox Implementation

One of the most common and straight forward implementations of the S-box was to have the pre-computed values stored in a ROM based look-up table (LUT) in which the substitution bytes of all possible bytes are stored in a table. However, there are two drawbacks for the LUT-based approach:

- All 256 possible values of a byte are stored in a ROM and the input byte would be wired to the ROM's address bus. This method suffers from a fixed delay since ROMs have a fixed access time for read and write operations.
- Each copy of the table requires 256 bytes of storage, along with the circuitry to address the table. Each of the 16 bytes can go through the S-box function independently. This then effectively requires 16 copies of the S-box table for one round. To fully pipeline, the encryption would require unrolling the loop of 10 rounds into 10 sequential copies of the round calculation. This needs 160 copies of the S-box table, a significant allocation of hardware resources.

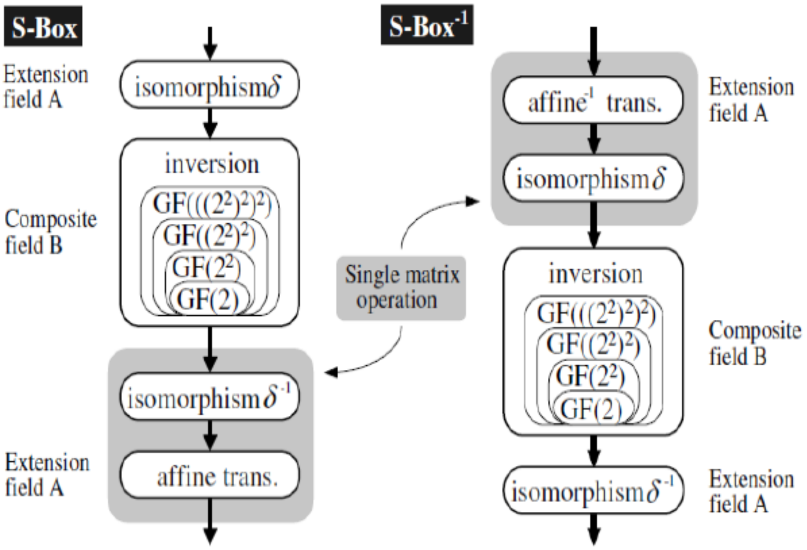


Figure 1.10: Computation sequence of S-box implementation [43].

Designing a compact S-box is one of the most critical problems for reducing the total circuit size of AES hardware. It is possible to implement the S-box as a particular circuit based on its functional specification by using automatic logic synthesis tools. However, in [42] a significant reduction in the size of the S-box was achieved, by using composite field arithmetic. In [43], they have proposed further optimization of S-box by introducing a new composite field based on Polynomial Basis (PB). Figure 1.10 shows the outline of such S-box implementation.

The most costly operation in the S-box is the multiplicative inversion over a field A , where A is an extension field over $GF(2^8)$ with the irreducible polynomial $m(x) = x^8+x^4+x^3+x+1$. We performed the composite field arithmetic to calculate the multiplication inversion of the Sbox under Normal Basis (NB) rather than polynomial basis, which is done in previous works. This is done by decomposing $GF(2^8)$ to $GF(2^8)/GF(2^4)/GF(2^2)/GF(2)$, performing the required calculations and then, bringing the result to $GF(2^8)$. The following 3-stage method is to be adopted to reduce the cost of the inversion operation [43]:

1. Map all elements of the field A to a composite field B , using an isomorphism function δ^{-1}.

2. Compute the multiplicative inverse over the field B.

3. Re-map the computation results to A, using the function δ.

The composite field B in stage 2 is constructed not by applying a single degree-8 extension to GF (2), but by applying multiple extensions of smaller degrees. To reduce the cost of stage 2 as much as possible, the composite field B is built by repeating degree-2 extensions under a normal basis through the following steps:

1. Isomorphism between $GF(2^8)$ and $GF(2^8)/GF(2^4)$:
   NB=$\{U^{16}, U\}$, $A \in GF(2^8) \Rightarrow A = r_1U^{16} + r_0U$, where $r_0, r_1 \in GF(2^4)$.
   Irreducible polynomial: $u^2 + u + \lambda = (u + U)(u + U^{16})$.

2. Isomorphism between $GF(2^4)$ and $GF(2^4)/GF(2^2)$:
   NB= $\{V^4, V\}$, $r \in GF(2^4) \Rightarrow r = x_1V^4 + x_0V$, where $x_0, x_1 \in GF(2^2)$.
   Irreducible polynomial: $v^2 + v + \phi = (v + V)(v + V^4)$.

3. Isomorphism between $GF(2^2)$ and $GF(2^2)/GF(2)$:
   NB= $\{W^2, W\}$, $x \in GF(2^2) \Rightarrow x = p_1W^2 + p_0W$ where $p_0, p_1 \in GF(2) = \{0, 1\}$.
   Irreducible polynomial: $w^2 + w + 1 = (w + W)(w + W^2)$.

To change bases we need an 8 × 8 multiplication bit matrix. δ^{-1} refers to the matrix that converts a given byte $(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$ as an element of $GF(2^8)$, from the standard basis into the subfield basis. Likewise, δ represents the matrix to convert from the subfield basis into the standard basis.

$$\begin{pmatrix} p_7 \\ p_6 \\ p_5 \\ p_4 \\ p_3 \\ p_2 \\ p_1 \\ p_0 \end{pmatrix} = \delta^{-1} \begin{pmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{pmatrix}, and \begin{pmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{pmatrix} = \delta \begin{pmatrix} p_7 \\ p_6 \\ p_5 \\ p_4 \\ p_3 \\ p_2 \\ p_1 \\ p_0 \end{pmatrix}$$

Where,

$$\delta^{-1} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}, and, \delta = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

1.3 Thesis Overview

This thesis focuses on a particular fault attack called the glitch attach in which a glitch is generated in the input clock signal and the targeted cryptosystem is the Advanced Encryption Standard (AES). The effect of the glitchy clock cycle on the registered values in the output registers and how it can be used to recover the secret key is explored in this thesis.

The thesis is organized as follows: Chapter 2 introduces the fault type and models, and presents a survey on the most important approaches to create fault injection, including: power spikes, clock glitches, electromagnetic and optical fault injections. This is followed by an literature review on previous fault attacks applied on AES focused on the attack method used, complexity and results. Chapter 3 illustrates the experimental setup by explaining the fault model, how to use the glitch characteristics, and how to generate the desired glitch accurately. It continues by introducing the cryptographic board and the target AES architecture. The simulation results for an Sbox architecture based on normal basis, and experimental result for real fault injection in AES are presented in chapter 4. Finally, Chapter 5 provides a summary, conclusions and recommendations for future work.

Chapter 2

Background and related work

This chapter provides a review of fault injection methods by introducing fault types and models. This is followed by a discussion of the most common fault injection mechanisms and surveying previous fault attacks against AES. Finally, attacking the AES datapath rather than key schedule is discussed.

2.1 Fault injection method: The model and type of the fault

There has been a large variety of fault attacks reported in the literature. The difference between them is based on their ***fault types*** and ***fault models***.

2.1.1 Fault types

There are two different fault types based on the durability of the effect of the attack: permanent faults and transient faults.

Permanent faults A permanent fault certainly changes the value of a cell and the behavior of the device, and returning to the initial state is impossible. It can be caused by injecting a fault in a ROM or by using a laser cutter to cut a wire inside the chip.

Transient faults A transient fault is a provisional fault in which the device can return to its initial state and resume its original behavior after a short time, for example, by resetting the circuit or ceasing the fault's stimulus. The transient fault in a memory block, RAM, causes the fault to be memorized and the modified value lasts until overwriting the variable holding it.

2.1.2 Fault models

Attackers must know the fault model they are going to inject in a proposed attack and then use it in practice. This knowledge helps to distinguish whether or not the proposed attack is practical. The most important factors indicating a fault model are as follows[29]:

Bit vs. byte fault In fault attacks it is plausible to affect the value of only one bit or one byte. Modifying a single bit is more difficult and needs more precise equipments but makes it possible to break almost all ciphers. On the other hand, since the storage or transferring the data is done in the byte level, the attacks modifying a byte are more practical.

Specific vs. random fault In this category the attackers determine if they assume to change the bit or byte value to a specific or random value. In practice, it is easier to inject a random value fault.

Static vs. computational fault If the attack aims to change the value of the memory itself, for example flipping a secret key bit in the memory, it is of the static model, which is usually difficult in practice. On the other hand, computational faults occur when inducing a fault during an operation computation and is easier than the static one to inject.

Data vs. control fault When the fault causes an omission of an instruction or iteration, it is of the control fault group. Whereas, modifying the data is more common in attacking on the secret key or the intermediate states.

2.2 Fault Injection Mechanisms

In a cryptographic device there are flip flops to define the current state, and combinational logic to calculate the next state from the current state in each clock cycle [31]. There are also input clock and voltage, transistor current, among others with many analog effects that can be used in non-invasive attacks like the following:

2.2.1 Power Spikes

Embedded systems such as smartcards need power supply which is provided externally. Variations in the supply voltage can impede the device's functionality with misinterpretation or omission of instructions as well as data misread. It enables an adversary to both tamper with the power feeding and measuring the power consumption.

According to the "Electronic signals and transmission protocols" standard, ISO/IEC 7816-3, a smartcard must tolerate the power supply variation of ±10% of the standard voltage. For variations higher than 10%, the system no longer works properly. ***Spikes*** are short massive variations of the power supply and can be used to induce faults during the computation of the smartcard. Spikes do not require a modification of the device itself but provoke faults by modifying the working conditions. They have different effects depending on different parameters including time, voltage value, and the transition shape. A spike attack is a non-invasive, or at least semi-invasive, method which can lead to wrong computation result when the device is still able to complete the computation. Experimental results for injecting faults into smartcards using spikes are described in details in [8].

2.2.2 Optical Fault Injection

Due to the photoelectric effects, the semiconductor transistors and the EEPROM are sensitive to lasers and ion-beams and the memory cells can be set to 0[46]. In [31], they depackage the device and use an optical microscope to reconstruct the layout and apply invasive fault attacks including laser cutting and focused ion-beam manipulation to recover the covered data. But it was not sufficiently precise to change the chosen bits. The attacker needs to be able to control the light's wavelength, energy, location, and emission time. In 2008, Hutter at al. [24] described a precise locolized fault injection method with affordable equipments which enables them to interfere data, control lines, memory blocks and driver circuits. This method is the underlying method used in a fault analysis attack against HB+ authentication protocol used in radio-frequency identification (RFID) tags [16]. In 2010, Agoyan et al.[5] showed that it is possible to reproduce single-bit faults on SRAM which was considered unfeasible before. In addition, today the laser attack can be applied to the back side of the chip where there is no protecting mechanism.

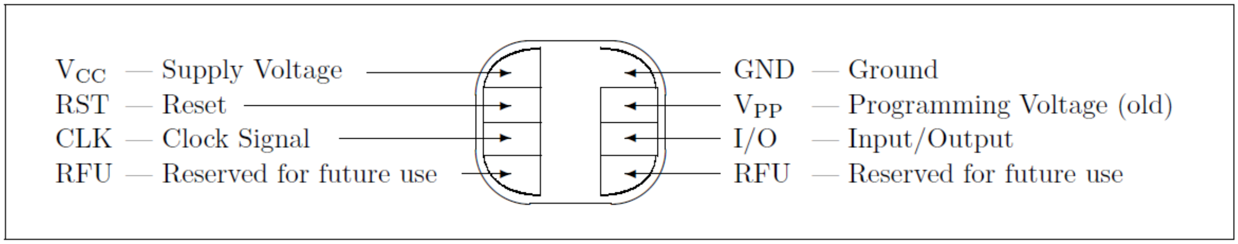

V_{CC} — Supply Voltage
RST — Reset
CLK — Clock Signal
RFU — Reserved for future use
GND — Ground
V_{PP} — Programming Voltage (old)
I/O — Input/Output
RFU — Reserved for future use

Figure 2.1: An smartcard contact assignment according to ISO 7816-2 [1]

2.2.3 Electromagnetic Attack

Emitting a powerful magnetic pulse close to the silicon area of the cryptographic device is another way to inject faults. This emission creates local current called Eddy current on the component's surface. Eddy current can modify the number of electrons inside a transistor's oxide grid and change its threshold voltage. As a result it impedes the transistor's switching and ensures the attacker that a memory cell contains the value 1 or 0[38]. Another usage of the Eddy current is to heat a material uniformly and hence, to generate transient or permanent faults by inducing the heat.

This attack can be performed on small parts of the device if the attacker has the knowledge of the layout of the device to control the targeted area precisely. As reported in [40], Eddy current can enable the adversary to induce faults very accurately and even set or reset the individual selected bits. Hence, it has been considered a practical method [25].

2.2.4 Clock Glitches

As shown in figure 2.1, cryptographic devices such as smartcards have an external clock signal. According to the ISO/IEC 7816-3, a smartcard must work properly in the clock signal with voltage variation in the range of 0.7 Vcc to Vcc for the high signal and 0 to 0.5 Vcc for low signal. Another constraint is to tolerate the clock rise and the clock fall times of 9% from the period cycle. A deviation of the external clock which is out of the specified tolerance scope is called ***glitch***. Glitches may be generated by temporarily increasing the clock frequency in one or a half cycle. Clock glitch can cause data misread in which the circuit samples the input before it gets the new state or reads the values from the data bus before being latched into the memory. The other possible effect can be the omission of instructions in which the next instruction starts executing before finishing the current one. As presented by Agoyan et al. in [6], injecting the fault by clock glitches is currently the simplest and most practical fault attack.

2.3 Previous Fault Attacks on AES

This sections reviews past research in fault analysis attacks on AES, attacking both the AES datapath as well as the AES key expansion circuitry.

Fault based cryptanalysis of the Advanced Encryption Standard (AES), Blomer-Seifert [13]

A fault attack against AES using an optical fault injection method (from [46]) has been demonstrated in [13]. The attack was implementation independent and it recovered the complete 128-bit AES cipher key by generating 128 faulty ciphertexts. First, they consider the case where the block length is greater than or equal to the key length 128 ($Nb \geq Nk$). In this case the complete cipher key is used in the initial AddRoundKey. The cipher key is stored in a 4×4 array of bytes k_{ij} and the $l - th$ bit of the byte is denoted by k_{ij}^{l} where $0 \leq l \leq 7$ and $0 \leq i, j \leq 3$. The attack encrypts a block of plaintext where each bit has the value of 0. Then the initial AddRoundKey transformation will be performed as

$$s_{ij} = 0 \oplus k_{ij} = k_{ij}.$$

Before the next transformation, the attacker tries to set s_{ij}^{l} to 0 and let the encryption process proceed without further fault injection. If $k_{ij}^{l} = 0$ then the encryption results the correct ciphertext as if there was no fault. But if $k_{ij}^{l} = 1$, setting the state bit to 0 affects the temporary state and causes an incorrect ciphertext or resetting the device–if the device can detect the corruption and reset itself . Hence, with one encryption process the value of k_{ij}^{l} can be deduced and the attacker can determine all bits of the cipher key by encrypting the null plaintext for 128 times, each time inducing a single fault.

If the key size is greater than the input block ($Nk > Nb$), the attack obtains the first 128 bits of the key as described above and then continues up to the next AddRoundKey in round 1. The plaintext can be chosen such a way that all bits of the state before the AddRoundKey are equal to 0. Then the attacker can set the remaining bits of the state one by one after the AddRoundKey of round 1. As before, if the ciphertext is correct the key bit k^l_{ij} is 0, otherwise $k^l_{ij} = 1$.

DFA on AES, Giraud [21]

Giraud [21] presents two different differential fault analysis (DFA) attacks on AES. In the first attack, the fault model is a single-bit fault which must occur right before the SubByte transformation in the last round. If C denotes the correct ciphertext and C^* the faulty ciphertext, the following equations hold:

$$\begin{cases} \delta = C^i \oplus C^{*i}, \\ \delta = (SubBytes[S] \oplus K^{10}) \oplus (SubBytes[S'] \oplus K^{10}). \end{cases}$$

where fault is injected in $i - th$ byte, S is the correct byte and S' is the faulty byte. If the following two conditions hold:

- $\delta = SubBytes[S] \oplus SubBytes[S']$
- one bit fault :$S \oplus S' = 2^i, i \in 0, ..., 7$

then, by an exhaustive search for the couple of (S, S'), a list of possible key bytes can be computed such that $K^i = SubBytes[S] \oplus C^i$.

Each pair of (S, S') gives one possible key which is put in a list α. If the number of guessed keys is more than one, this process will be repeated with a new plaintext and generate a new set β. Taking the intersection of these two sets and repeating the operation will eventually result in only one candidate left for the key. With the probability of about 97%, using three ciphertexts will be sufficient to recover one byte of the last round key. So, by less than 50 faulty ciphertexts the 128-bit AES key can be obtained. If the attacker has the ability to choose the affected byte, this attack can reach the key with only 35 faulty ciphertexts.

The second attack is more realistic because it assumes that the temporary fault affects a whole byte. The attacker aims to induce the fault in the Key Schedule unit and pursues the attack in 3 steps:

1. Before computing K^{10}, a fault is inserted in K^9 and using the resulting ciphertexts, the last 4 bytes of K^9 are obtained.
2. Before computing K^9, a fault is inserted in K^8 and using the resulting ciphertexts, another 4 bytes of K^9 are obtained.
3. Before starting round 9, a fault is inserted in S^8 and using the resulting ciphertexts and the recovered 8 bytes of K^9 the AES key will be obtained.

According to he author's claim, the complete key can be reached by less than 250 ciphertexts and in an extended scenario where the attacker can choose which byte to affect, the number of required faulty ciphertext is 31.

Differential fault analysis on AES key schedule and some countermeasures, Chen and Yen

The attack presented in [17] uses the same method as the second attack in Giraud's publication [21] , on the keypath instead of the datapath, to disclose the secret key but with less required ciphertexts. A fault is injected into one of the last four bytes (last word) of K^9 which results in 5 faulty bytes in the last round key and consequently in the final output. Repeating this process for a few different faults and intersecting the results of solving the equations and statements from each process gives the exact values of these 5 key bytes. Then a fault is injected in the round key K^8 in a different position (e.g. the penultimate word). It affects 6-7 output bytes and leads to guess another 8 key bytes. The remaining 3 bytes can be reached by a light exhaustive search. With an accurate fault inducing method, the whole AES secret key can be disclosed by exploiting less than 44 fault injections.

Differential Fault Analysis on A.E.S, Dusart et al. [18]

The idea in this work is to first retrieve the last round key and then, using the invertibility feature of the AES key schedule, to retrieve the initial cipher key. It is assumed that the attacker can inject a random fault in a single byte before the last MixColumns operation. The fault is spread over only four bytes of the ciphertext which according to their location, the location of the fault can be deduced. Hence, the location of fault injection is not essential and the attack is highly realistic. Suppose F is the faulty state and the unknown fault ε is injected in the first element of the state. Then we will have

$$F_{9,ShiftRows} = S_{9,ShiftRows} + \begin{pmatrix} \varepsilon & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Then the effect of the fault will be as follows:

After the MixColumns:

$$F_{9,MixColumns} = S_{9,MixColumns} + \begin{pmatrix} 2\cdot\varepsilon & 0 & 0 & 0 \\ \varepsilon & 0 & 0 & 0 \\ \varepsilon & 0 & 0 & 0 \\ 3\cdot\varepsilon & 0 & 0 & 0 \end{pmatrix}.$$

After the AddRoundKey:

$$F_{9,AddRoundKey} = S_{9,AddRoundKey} + \begin{pmatrix} 2\cdot\varepsilon & 0 & 0 & 0 \\ \varepsilon & 0 & 0 & 0 \\ \varepsilon & 0 & 0 & 0 \\ 3\cdot\varepsilon & 0 & 0 & 0 \end{pmatrix}.$$

After the last round SubBytes:

$$F_{10,SubBytes} = S_{10,SubBytes} + \begin{pmatrix} \acute{\varepsilon}_0 & 0 & 0 & 0 \\ \acute{\varepsilon}_1 & 0 & 0 & 0 \\ \acute{\varepsilon}_2 & 0 & 0 & 0 \\ \acute{\varepsilon}_3 & 0 & 0 & 0 \end{pmatrix}.$$

After the last round ShiftRows:

$$F_{10,ShiftRows} = S_{10,ShiftRows} + \begin{pmatrix} \acute{\varepsilon}_0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \acute{\varepsilon}_1 \\ 0 & 0 & \acute{\varepsilon}_2 & 0 \\ 0 & \acute{\varepsilon}_3 & 0 & 0 \end{pmatrix}.$$

And finally after the last round AddRoundKey:

$$F_{10,AddRoundKey} = S_{10,AddRoundKey} + \begin{pmatrix} \acute{\varepsilon}_0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \acute{\varepsilon}_1 \\ 0 & 0 & \acute{\varepsilon}_2 & 0 \\ 0 & \acute{\varepsilon}_3 & 0 & 0 \end{pmatrix}.$$

As shown above, when a random byte fault is injected before the MixColumns operation, the fault will be spread over all four bytes of the same column. The subsequent AddRoundKey and the last round operations modify the bytes independently. The only operation that could exploit information about the last round key is the last SubBytes transformation because due to the nonlinearity, some assumptions can be made on the error value. We have the general equation

$$s(x + c.\varepsilon) + s(x) = \acute{\varepsilon}, \tag{2.1}$$

in which $c \in 01, 02, 03$ and $\varepsilon \in GF(2^8) - 0$. The following four equations are driven from the above general equation:

$$\begin{cases} s(x_0 + 2 \cdot \varepsilon) = s(x_0) + \acute{\varepsilon}_0 \\ s(x_1 + \varepsilon) = s(x_1) + \acute{\varepsilon}_1 \\ s(x_2 + \varepsilon) = s(x_2) + \acute{\varepsilon}_2 \\ s(x_3 + 3 \cdot \varepsilon) = s(x_3) + \acute{\varepsilon}_3 \end{cases}$$

Solving each equation results in a set of possible values for the error. The intersection of these sets is a smaller set which reduces the number of required ciphertexts for the full analysis. The corresponding four key bytes can be guessed for each possible value of fault. According to this paper, intersecting the solution sets will retrieve four bytes of the round key quickly and the full 128-bit key can be found by analysing less than 50 ciphertexts.

A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD, Piret and Quisquater [39]

Piret and Quisquater use a realistic fault model as they consider a random fault affecting a byte. In the first attack scenario, the fault is induced between the MixColumns of round 9 and 8. The MixColumn of round 9 spreads the fault over the whole column. After the non-linear computation of SubBytes the 4 faulty bytes are scattered over different columns according to the ShiftRows transformation. This leads to obtain a set of candidates for the 4 key bytes among which the unique correct candidate can be found by a couple of well-located faults. Suppose a one-byte fault occurring before the MixColumns of round 9 on one of the state bytes $S_{0,0}$, $S_{1,1}$, $S_{2,2}$, or $S_{3,3}$. Using one pair of correct and faulty ciphertexts $(C; C^*)$, the attacker obtains about 1036 candidates for $(K^{10}_{0,0}, K^{10}_{1,3}, K^{10}_{2,2}, K^{10}_{3,1})$. By exploiting two pairs of $(C; C^*)$, the only correct candidate for these 4 key bytes is remained. Hence, for the whole 16 bytes of the key, only 8 faults at carefully chosen locations are required, which in comparison with [21], [18], and [13], this attack scenario requires a considerable less number of faulty ciphertexts. The basic attack algorithm is as follows:

1. Consider a one-byte difference at the input of the MixColumns of round 9. There are 255 possible differences for a byte and 4 possible locations for fault that affect the same 4 bytes of the output.

2. Compute $255 \cdot 4$ possible differences at the output of the MixColumns and store them in a list D.

3. Take a guess on the $(K^{10}_{0,d}, K^{10}_{1,(1-d)mod4,}, K^{10}_{2,(2-d)mod4}, K^{10}_{3,(3-d)mod4})$.

4. Compute the difference $\Delta = SubBytes^{-1}((C \oplus K^{10})_{*,d}) \oplus SubBytes^{-1}((C^* \oplus K^{10})_{*,d})$. Verify if Δ is in list D. If yes, add the key bytes to the list L of potential key candidates.

5. Repeat the above steps with a new plaintext P and its corresponding C and C^* with the key guesses only from the list L, until only one candidate remains.

In a second attack scenario, a complete 128-bit key can be obtained by only 2 faulty ciphertexts, assuming that the fault occurs in between the 7-th and 8-th round MixColumns operations. As depicted in Figure 2.2, a one-byte fault before the MixColumns of round 8 leaves 4 faulty bytes at the output of round 8. Then the last MixColumns spreads the 4 faults over the whole state and affects all 16 bytes. This makes it possible to recover the complete 128-bit key using only 2 faults.

Differential Fault Analysis of the Advanced Encryption Standard using a Single Fault, Tunstall et al. [48]

In 2009, Tunstall et al. reported a differential fault analysis on AES using only a one-byte random fault at the input of the round 8. The attack includes two steps where in the first one the number of possible key candidates is reduced to 2^{32}, and then to 2^8 in the second step. Considering C_i and C'_i as the correct and faulty bytes of the ciphertext, sixteen equations can be obtained from MixColumns operation. This results in 2^{32} hypotheses for

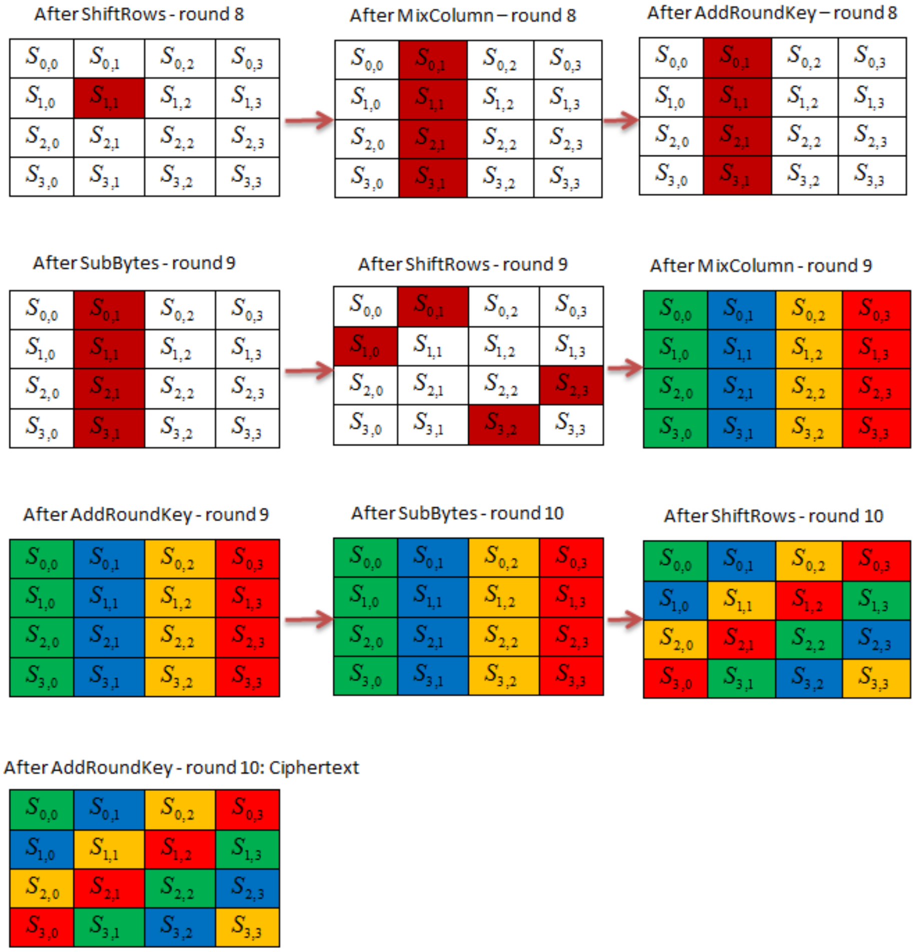

After ShiftRows - round 8
After MixColumn – round 8
After AddRoundKey – round 8
After SubBytes - round 9
After ShiftRows - round 9
After MixColumn - round 9
After AddRoundKey - round 9
After SubBytes - round 10
After ShiftRows - round 10
After AddRoundKey - round 10: Ciphertext

Figure 2.2: The effect of the fault injected between rounds 8 and 7

the key. By exploiting the relationship between round 9 and 10, and using the Inverse MixColumns, a set of four other equations is made which tests all 2^{32} candidates reached from step one. From these potential keys, 2^8 candidates remain among which a simple brute-force search can determine the right cipher key.

Note that the attacks explained above are not implemented on a hardware such as FPGA. They are attack descriptions assuming that there exist fault injection mechanisms to induce the desired fault into the targeted part of the AES architecture with a wanted timing.

2.4 Attacking the AES State rather than Key schedule

In reviewing the fault attacks on AES, as discussed above, we have attacks on both datapath and key schedule. These attacks are classified into two tables in table 2.1. This research only focuses on injecting faults into the datapath. As it will be discussed in next sections, we use clock glitch which causes the fault in the output by violating the setup time. Since generally the path delay of the key schedule in AES is much smaller than that of the datapath, attacking the key schedule using clock glitches is not easy in practice.

To investigate the attack limits on the State, the attacker injects a random single byte fault between the MixColumns of rounds 8 and 7. In the case of the attack on the Key schedule, it is assumed that the same fault model can be induced in the first column of K^8. Another assumption is that the AES is theoretically unbreakable i.e. almost all attacks have the time complexity of an exhaustive search. If the security level of AES is denoted by K_s, then $K_s = 2^{128}$.

In a DFA, when attacking the State, a fault is induced during a round computation, causes a difference ΔS and leads to a faulty ciphertext C'. Suppose that instead of a fault injection, there is a classical collision based adversary Adv_{col} who tries different plaintexts in order to find a pair (P, P') which generate two states with a specific difference ΔS after a target round r. Note that the ciphertext pair (C, C') is exactly the same as in DFA. If the probability of reaching such a pair is $Pr(\Delta S)$ then the number of required plaintext pairs would be $\frac{1}{Pr(\Delta S)}$. If the DFA reduces the key search space to K_l then $K_s \leq \frac{1}{Pr(\Delta S)} \cdot K_l \Rightarrow K_l \geq K_s \cdot Pr(\Delta S)$. This means that a DFA in the best case decreases

Table 2.1: Classification of fault attacks into two classes: DFA on datapath, DFA on key schedule.

(a) DFA on datapath

Attack	Model	Number of required faults
Blomer et al [13].	Byte	90,112
Giraud [21]	Byte	250
Dusart et al [18].	Byte	50
Piret et al. [39]	Byte	2,1

(b) DFA on key schedule

Attack	Model	Number of required faults
Blomer et al [13].	Bit	128
Chen et al. [17]	Byte	44
Takahashi et al. [47]	Byte	7,4,2
Kim et al. [30]	Byte	2

the search space to $K_s \cdot Pr(\Delta S)$. In a single byte fault model, ΔS is a byte difference and the probability of obtaining a plaintext pair generating a state with one byte difference (15 similar bytes) is $2^{-15\times 8} = 2^{-120}$. Therefore an *optimal* one-byte fault DFA reduces the search space to $2^{-120} \cdot 2^{128} = 2^8$. It can be derived that if the fault affects i bytes in the State, then the minimum search space would be $2^{8\cdot i}$ [45].

With a similar analysis, the search space limits for the DFA on Key schedule can be computed. Here, the attacker tries to find a plaintext pair such that they generate a required difference ΔK in the r^{th} round key. This creates a difference ΔK_p after the next round operation. Therefore, the number of choices for P' is $\frac{1}{Pr(\Delta K_p)}$ and the optimal DFA is at $K_s \cdot Pr(\Delta K_p)$ level. With the assumption of a one-byte fault in the first column of K^9 in AES-128, which causes a four-byte difference in the ninth round key, the attack reduces the key space to $2^{128} \cdot (\frac{255}{(2^8)^4} \cdot \frac{1}{(2^8)^{12}}) = 2^8$. However, there is no reported attack which reaches this limit. In [30] a fault affects three bytes in the first column of the ninth round key while generating the key and thus causes a 12-byte difference in the round key 9. Therefore in the best case the key space reduces to $2^{128} \cdot (\frac{255^3}{(2^8)^4} \cdot \frac{1}{(2^8)^{12}}) = 2^{24}$.

Table 2.2 shows the optimal limits for DFA on AES-128. The second column corresponds to the case in which the attacker has access to plaintext and can apply a brute force

Table 2.2: Optimal results of DFA on AES-128 [45]

	Number of faults	Number of remaining keys	Number of faults for unique key
State	1	2^8	2 [39]
Key schedule	1	2^8	2 [27]

search on the key hypotheses, as explained above. The third column represents the results when the attacker does not have access to the plaintext and thus must uniquely determine the key. It can be seen that there is no reported attack which reaches the AES-128 limits for the number of the remaining keys. However, in the case of results for unique key, Piret's attack[39] obtains the best result to recover the AES-128 unique correct key by injecting only two faults in the State. There are also attacks on the State for the AES-192 and AES-256 which eventuates only one key after injecting 3 faults in 5 minutes, and 2 faults in 10 minutes respectively[26]. For the DFA on Key schedule, there is only a recent attack on AES-128 which requires 2 pairs of faulty and correct ciphertexts[28].

2.5 Summary

In this chapter fault types and models along with the most common fault injection mechanisms were introduced and analysed. Then various DFA attacks targeting the AES algorithm were synthesized. Among them, two attacks, Piret's attack [39] and Tunstall's attack [48], are particularly powerful because the number of required faults to disclose the 128-bit AES key is only 1 or 2. This increases the attack feasibility as usually it is difficult to generate different faulty outputs from the same fault injections. Moreover, they induce the fault into the datapath as it is easier to disturb than the key schedule. In the next chapter, the experimental setup to inject a glitch fault in a FPGA-based implementation of AES is described. The design of the glitch generator and how it is used for a setup time violation in a glitch attack is described. This attack targets the critical paths which are mostly in the datapath than in the key schedule path.

Chapter 3

Experimental Setup

This chapter describes the experimental setup of a FPGA-based fault attack testbed. The details of the FPGA board and AES circuit, clock glitch creation and implementation, and final experimental setup is described and compared to previous research.

3.1 Using the clock glitch to inject the fault

In this thesis, the goal is to experimentally characterize faults generated by clock glitches, and then determine how it can be used to implement theoretical fault analysis attacks. Clock glitch attacks are known as a serious and practical threat since they are easy to implement and repeatable. Being repeatable is a significant feature since as discussed in the previous chapter, most of the practical attacks rely on the possibility of having more than one fault occurrence to obtain the unique key. This section will briefly discuss how clock faults occur, in purticular how a setup time violation can be created from a clock glitch.

ICs usually process the data in combinatorial blocks and the D-flip-flops are used in between the combinatorial blocks to separate them. For example a few hundred flip-flops may define the IC's current state and the combinatorial logic calculates the next state from the current state during each clock cycle. In a synchronous circuit(figure 3.1), the flip-flops use the same clock input and registers may latch the data at the rising edge of the clock. The intermediate combinatorial logic modifies the data while traveling between the registers. There are some features that can be used by an attacker including:

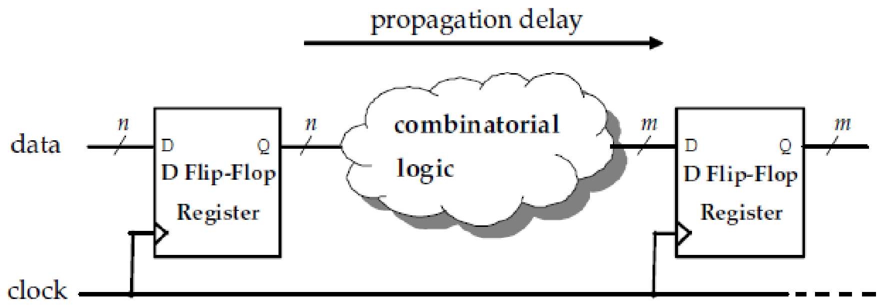


Figure 3.1: Combinatorial logic and synchronous representation of a digital IC [6]

- The propagation delay can vary within a chip or between same-type chips.
- If the result of a combinatorial logic is not stabilized on a previous state, the flip-flops do not accept the correct new state.
- During a short time slot, the flip-flops sample their input and compare it with the power supply voltage. This sampling time is fixed, however, it can vary between different flip-flops.

The ***Propagation delay*** (time needed for the data to propagate through the combinatorial logic blocks) depends on the data, the logic performed on the data, and the capacitance and resistance of the transistors and interconnections. It also varies with the temperature and power supply voltage [6]. The maximum propagation delay is called ***Critical path***, which imposes a limit on the maximum speed of the circuit. A second parameter affecting the clock speed is the ***Setup time*** which applies to flip-flops. The setup time is the minimum time before the clock event during which the data must be stable in order to be reliably sampled by the clock.

In order to ensure proper circuit functionality, the clock period must be greater than the critical path propagation delay plus the register's setup time.

$$T_{clk} > t_{critical} + t_{setup}. \tag{3.1}$$

In a glitch attack the attacker intentionally causes one or more flip-flops to accept the wrong state. This may modify an instruction, or corrupt the data or state. One method to do this is ***Overclocking*** in which the clock period decreases (or the frequency increases).

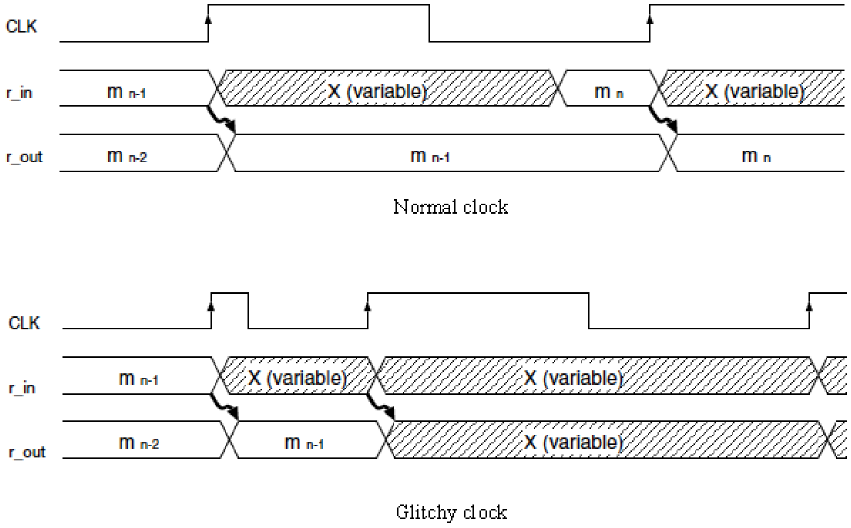


Figure 3.2: Latching the data with normal and glitchy clock [20]

Decreasing the clock period potentially can violate the condition in equation 3.1. Therefore, the register's input does not have the sufficient time to be correctly latched, which causes the faulty data to be latched instead. It is crucial to have the ability to precisely control the clock period, especially when the temperature and power supply change. In this thesis, a single transient clock glitch is generated by overclocking only in a specific time interval. As shown in Figure 3.2, in the normal case, clock cycles are longer than the maximum delay plus the setup time and the intermediate states are correctly stored. But, when the glitch occurs and the generated clock cycle is shorter than the sum of maximum delay and the setup time, a timing violation called a ***setup time violation*** happens. This causes the output of the combinational logic (r-in)to be latched in the register (r-out) before the correct output value is produced.

3.2 Clock glitch generating on FPGA

Generating the faulty clock with a sufficiently accurate glitch is crucial to launch an attack on AES. To do so, there are some conditions that the glitch generator must fulfill. For example:

- The attacker should be able to induce the glitchy cycle into any position of the clock.

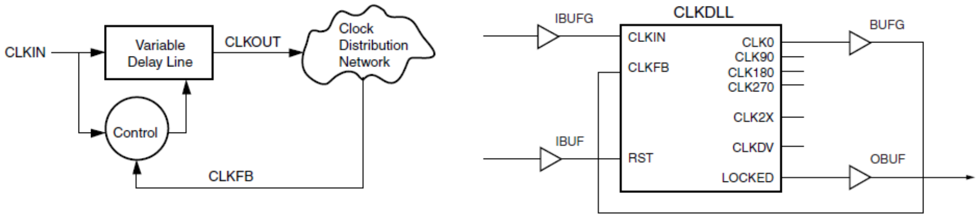
This makes it plausible to inject the fault into any desired round of the cipher. Since as we saw in the previous sections, each fault model requires a specific AES round to inject the fault into.

- The attacker should be able to change the glitch specifications to support a wide range of control on the effects.
- In order to be able to acquire power traces during an attack, timing of the target operation should be available and an internal clock counter is needed.

To generate a glitchy clock with the above mentioned features, the embedded Delay Locked Loop (DLL) of an FPGA of Xilinx Virtex-5 family is used. The DLL circuit is implemented by Digital Clock Manager (DCM) in Xilinx FPGAs. The Virtex-5 user guide (UG190) provides information on clock management technology. Alternatively, in Altera FPGAs the Phase Locked Loop (PLL) can be used. The basic function of the DLL is to remove the clock distribution delay. It also provides additional applications such as frequency multiplication and division, duty cycle correction, and phase shifting. In the simplest form, a DLL consists of a variable delay line and control logic. The delay line provides a delayed version of the input clock CLKIN. The DLL inserts a delay between the input clock and the feedback clock until the two clocks are in the same phase. After aligning the input and feedback clocks, the DLL locks and as long as the circuit is processing the data, there is no difference between the two clocks (see Figure 3.3).

To achieve the lock, DLL needs to sample several clock cycles. After achieving the lock, the LOCKED signal, which is one of the DLL outputs, activates. Until this signal activates, the DLL outputs are not valid and cannot be used or analyzed. They can exhibit unwanted glitches, spikes or other imprecise movements. For instance it might affect the duty cycle, or the CLK2X output appears as a 1X clock with a 25/75 duty cycle. Figure 3.4 shows a timing simulation waveform representing the output signals before reaching the LOCKED signal.

In the glitch generator the DCM's phase shifting feature which provides various phase shift options was used. The dedicated phase shift output signals including CLK0, CLK90, CLK180, and CLK270 always maintain their relationship. Using the "PHASE_SHIFT" attribute an arbitrary fixed delay value was set and the phase relationship of all DCM clock outputs was adjusted. The "CLKOUT_PHASE_SHIFT" attribute determines the phase shift mode and can be initialized as "NONE" (no delay) or "FIXED" (a fixed arbitrary



DLL block diagram

Standard DLL implementation

Figure 3.3: Delay Locked Loop (DLL) in Virtex FPGAs [2]

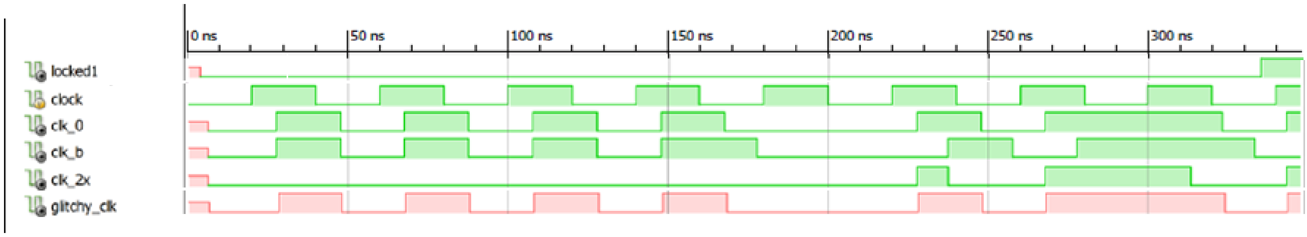


Figure 3.4: DLL output signals from ISE software before the LOCKED signal activates.

delay).

As shown in Figure 3.5 there are a number of taps in the delay line each of 30 ps to 60 ps. The number of taps and the maximum guaranteed delay line changes per data sheet. For the Virtex5-XC5VLX30 FPGA, there are 255 taps for the negative phase shifts and 1023 taps for the positive phase shifts, and the maximum guaranteed delay line or the FINE_SHIFT_RANGE is 7ns. This means that if the period of the CLKIN input clock is T_{CLKIN}, then this condition must be met: $PHASE\_SHIFT \cdot T_{CLKIN}/256 < 7ns$ [3]. for example if the input clock period is 40ns the maximum PHASE_SHIFT we can choose is $\pm[integer(256 \cdot \frac{7ns}{40ns})] = \pm 44$, and the corresponding attributes to be set are:

defparam dcm.CLKOUT_PHASE_SHIFT = "FIXED";
defparam dcm.PHASE_SHIFT = 44;

Note that the DCM has a fixed phase shift of $10\,^{\circ}$. This means that, for instance, the phase relationship between CLKIN and CLK0 is $0\,^{\circ} + 10\,^{\circ} = 10\,^{\circ}$, or the phase relationship between CLKIN and CLK90 is actually $90\,^{\circ} + 10\,^{\circ} = 100\,^{\circ}$. But, the output clocks hold

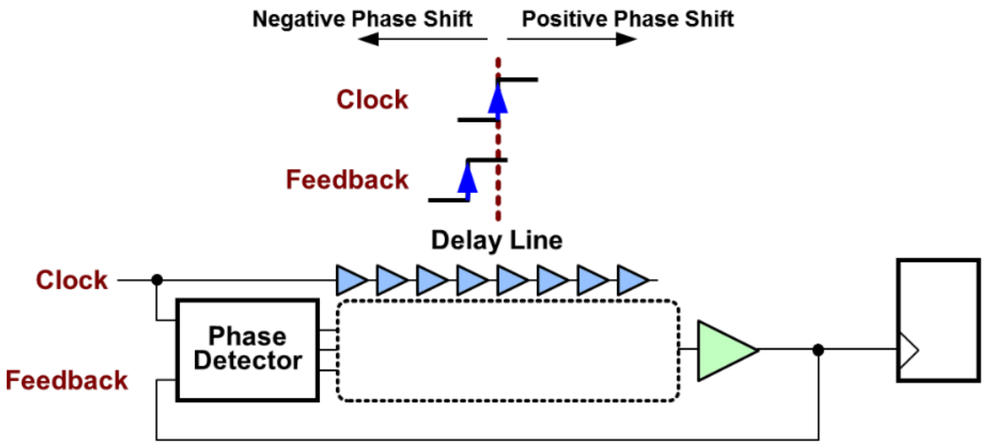


Figure 3.5: Delay line taps [3]

their phase difference i.e. the phase relationship between CLK0 and CLK90 is exactly 90 °.

3.2.1 Comparison to previous clock glitch generators

The glitch generator proposed in [20] uses two clock sources with the same frequency and different phases, generated by an external pulse generator. When switching between the two clock sources at the right time, a glitch occurs in the desired round. An oscilloscope is employed to control the switching time in order to select or change the fault injection round, and to measure the shortened interval of the clock cycle. However, we want to have the whole environment integrated on a single FPGA without the need to an external pulse generator or oscilloscope. In addition, we prefer to control the timing and shape of the generated glitch more precisely using parameters via a control PC. Therefore, the glitch generator we utilized, has the design idea of the one used in [19] to perform a safe-error attack against RSA. In their attack, the clock glitch characteristics and the fault effect is not explored. Instead it is just checked whether or not the fault has an effect on the output power trace.

The glitch generator uses a DLL to generate a shifted clock named clk_b which will be switched with the input clock at the proper time. Another DLL is used to generate clk_c with a delay less than that of the clk_b. A counter is activated by the input clock and outputs 1 when it reaches the required clock cycle. The 1 output lasts for a half clock cycle. The AND operation of clk_c and counter output gives the selector of the MUX and

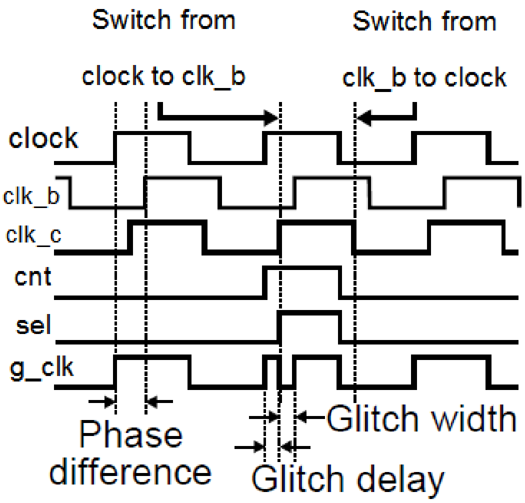


Figure 3.6: Timing chart of the glitch generator

the output of the mux is the demanded glitchy clock. Figure 3.6 shows the timing chart of the signals.

3.3 Experimental setup

This section provides an overview of the evaluation board on which the AES code is implemented, and other software and hardware equipments needed for the experiment. The on-chip glitch generator, which is designed on the evaluation board to generate the desired glitch in the clock signal, is discussed and the target AES cryptosystem is introduced.

3.3.1 The target cryptographic device

In order to have a uniform evaluation environment for fault injection, the glitch generator is implemented on the same FPGA where the AES cipher is located. Such an on-chip glitch generator makes it possible to reproduce glitchy clock signals. The hardware utilized the Side-channel Attack Standard Evaluation Board (SASEBO-GII) which is designed for side-channel attack experiments. This board is used in the DPA Contest where Side-channel

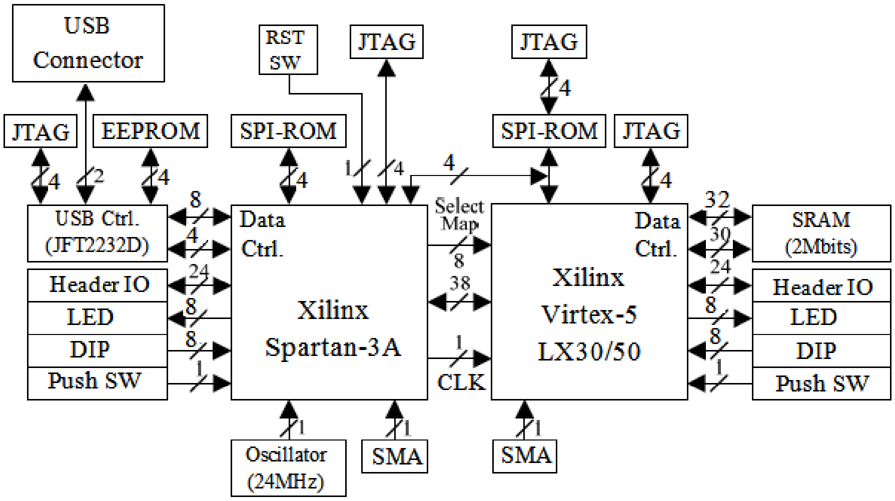


Figure 3.7: Block diagram of SASEBO-GII board [4]

attacks are publicly evaluated and compared. It is suitable for security evaluations of comprehensive cryptographic systems or the implementation of large circuits with various countermeasures. Figure 3.7 displays a block diagram of this board.

The board features two FPGAs: the the cryptographic FPGA (Xilinx Virtex5 XC5VLX30 or 50) on which the AES and the glitch generator are implemented, and the control FPGA (Xilinx Spartan3A XC3S400A). The cryptographic FPGA extends the logic area and provides various ways to access the reconfiguration function of the FPGA. The FPGAs are connected through a 38-bit common input/output bus with fully-flexibility in signal assignment. There is an oscillator on the board that provides the control FPGA with a 24MHz clock signal. An external clock input is also supported. External power source supplies the on-board power regulators and the FPGAs with 5.0 V. The power regulators convert the 5-V input into 3.3 V, 1.8 V, 1.2 V, and 1.0 V for the FPGAs. The core voltage of 1.0 V of the cryptographic FPGA can also be applied directly through the external power connector. The power trace measurements are possible with the shunt resistors by inserting them on the core VDD and/or ground lines of the virtex5 FPGA. A USB interface is provided for power supply and configuration. The host PC uses this USB port to control and communicate with the board. There are two configuration mechanisms for the cryptographic FPGA: SPI-ROM and Slave-SelectMap, and the control FPGA controls these configuration methods.

In addition to the SASEBO-GII board, a USB cable and a host PC, the Xilinx ISE software, the Microsoft .Net framework 3.5, the SASEBO_AES_Checker software and the FPGA configuration cable are needed. The cable is used to program the flash ROMs connected to the FPGAs. Furthermore, the driver software D2XX provided by FTDI is needed to use USB communication. The SASEBO_AES_Checker software written in C# defines the plaintexts and the cipher key, computes the correct cyphertext and provides a comparison of this correct value and the corresponding ciphertext from the FPGA after fault injection.

3.3.2 The target AES architecture

The AES design to be loaded on the SASEBO-GII board is publicly provided by the AIST web site for the Differential Power Analysis (DPA) contest. Figure 3.8 depicts a hierarchy of the code. This AES architecture implements a 128-bit key AES and supports both encryption and decryption. It uses a composite field based Sbox which uses polynomial basis. The output handshake signals including BSY (busy signal), Kvld (key output valid) and Dvld (data output valid) are used to communicate with the control part. The BSY signal is active when the cryptographic block starts encrypting or decrypting. We use this signal as a trigger that shows the start of the round operation, and directly connect it to the reset input of the counter in the glitch generator. Thus, as soon as the BSY signal is activated, the counter starts counting the clock cycles. Each round takes one clock cycle to be run. Hence, the counter indeed counts the number of AES rounds. This makes it possible to inject the fault in the desired round in practice.

3.4 Summary

In this chapter, the focus was to develop a uniform hardware environment as a testbed, in order to inject clock glitch faults into an FPGA-implemented AES cryptosystem, to explore the suitable fault characteristics in fault attacks. For this purpose, the glitch generator module and AES code provided in DPA contest, are implemented on the Xilinx Virtex5 FPGA located on the Side-channel Attack Standard Evaluation Board (SASEBO-G-II). In the next chapter, the simulation results for the normal basis Sbox architecture, and the experimental results for the complete AES are presented and discussed.

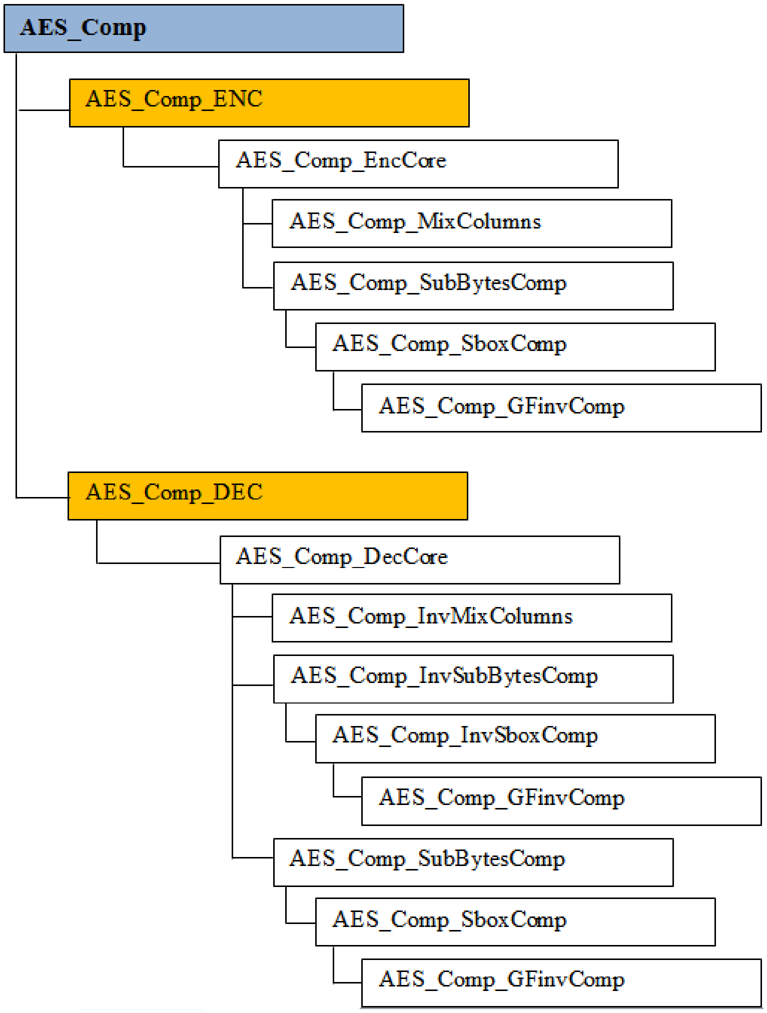


Figure 3.8: The hierarchy of the AES code used in fault injection experiments

Chapter 4

Experimental Results

This chapter provides the post-route timing simulation results of injecting the clock glitch faults into an Sbox architecture to be able to confirm the theoretical fault models and compare to the future experimental results. Then, an FPGA-based testbed was utilized to inject the glitch faults into the full AES implementation and examine the feasibility of fault attacks in practice.

4.1 Fault injection simulation in an Sbox architecture

Prior to experimentally injecting faults into the FPGA-implemented AES, a post-route timing simulation was utilized to observe the effect of the fault. This approach supports confirming the theoretical expectations and comparison of the experimental results to the timing simulation. In addition, we can gradually change the glitch characteristics and observe the result. To do so, an Sbox block is designed which includes an input register, the combinatorial logic to calculate the substitution byte for each input byte, and an output register. Before the Sbox operation, the plaintext is XORed with a round key.

4.1.1 Simulation result for the Sbox with a glitchy-cycle clock

The glitch generator block is added to the Sbox block and then the timing simulation is run. The timing waveform displays the exact shape of the glitchy clock and its effect on the output bytes. To be able to compare the correct and faulty outputs, there are two

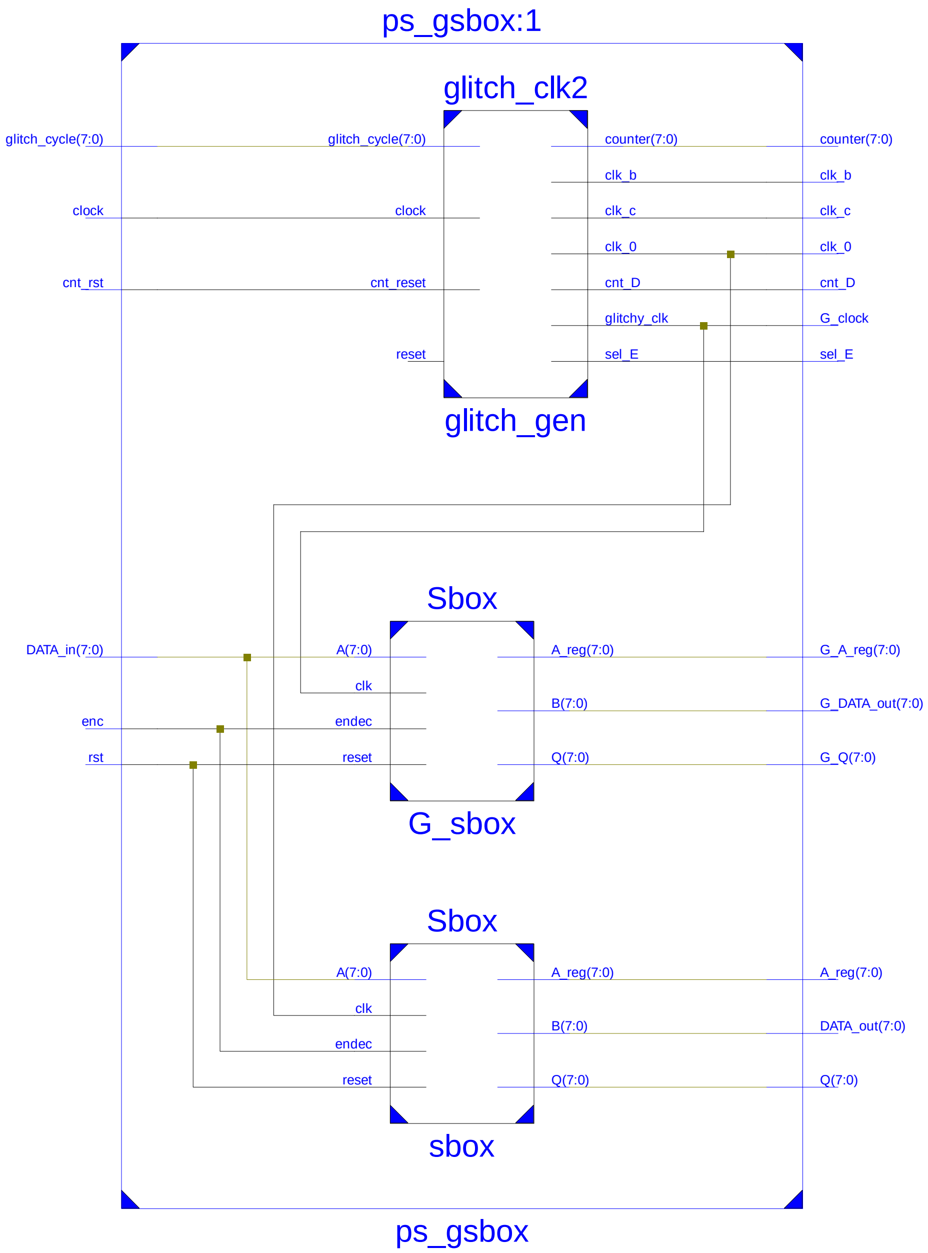


Figure 4.1: The RTL view of the Sbox design with a glitch generator.

Sbox blocks in the design: one with the normal clock and hence the correct output, and another one with the glitchy clock and the faulty output. Figure 4.1 represents the RTL view of the design and Figure 4.2 displays a typical output of the post-route timing simulation from the Xilinx ISE software. As shown in this picture, the glitch is injected in the 30-th clock cycle. The clock cycle in which the glitch is intended to be injected is an input to the system and the counter shows the clock cycle number. Data_in is the input byte which enters the input register and then undergoes the Sbox combinatorial logic. Q is the output of the Sbox function and the DATA_out is the latched value of the output. The corresponding values from the Sbox block with the glitchy clock (G_Sbox) are named by a prefix G (G_Q, G_DATA_out) and shown in the waveform. This helps us to have a better observation of the timing, keep track of the intermediate content and compare the correct and faulty values.

From the timing simulations it is observed that:

- We can generate the glitchy-cycle clock in any desired clock cycle except those during which the LOCKED signal is not activated. As explained in previous chapter and shown in Figure 3.4, before achieving this signal, the output clocks are not reliable and the timings are incorrect. Therefore, the desired precise glitch cannot be generated. Being able to inject the glitch in a known predefined cycle is a principal feature which enabled us to control the timing of the fault and inject the glitch in any encryption or decryption round of AES.

- The period of the glitch can be precisely decreased by steps of $P_{clk}/256$ where P_{clk} is the period of the main clock. We can also gradually change the glitch delay and glitch width within a constant period and therefore generate a glitch cycle with variable duty cycles. The glitch delay and glitch width are shown in Figure 3.6.

- For very small glitches, sometimes the glitch itself cannot be displayed in the waveform but its effect on the output byte is visible. As it can be seen in Figure 4.3 the ISE software cannot display the glitch but the shortened interval and its effect is obvious in the waveform. This may be due to two reasons: first the glitchy clock (G_clock) causes a time delay (visible in the picture) and second, the interval between the rising edge and the falling edge of the glitch cycle is too short.

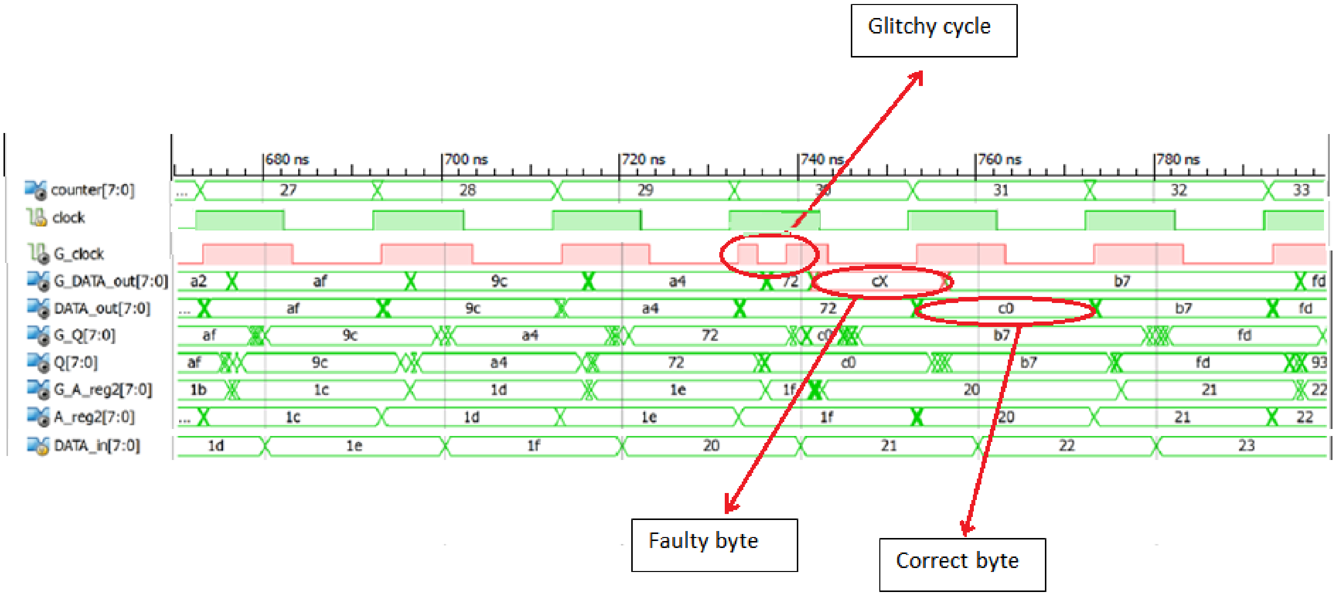


Figure 4.2: The waveform from the post-route timing simulation. It shows the outputs and the intermediate values from both normal and faulty Sbox modules.

- There is an interval for glitch period (P_g) in which the glitch causes the fault. Out of this interval two things happen: for glitch periods less than this time interval, there is neither a glitch nor a shorter cycle detected in the waveform and obviously there is no fault in the Sbox output (Figure 4.4). For P_g longer than the values in that specific interval, the glitch is generated and injected but it does not cause any fault in the output. This possibly occurs since the glitch period is likely long enough for the result of Sbox combinatorial logic to be correctly latched into registers. Figure 4.5 represents an example of this condition.

4.2 Fault injection in the last round of the AES implementation

This section examines the fault characteristics of a full AES architecture taken from AIST. Using the glitch generator block, the AES round in which the fault is going to be injected

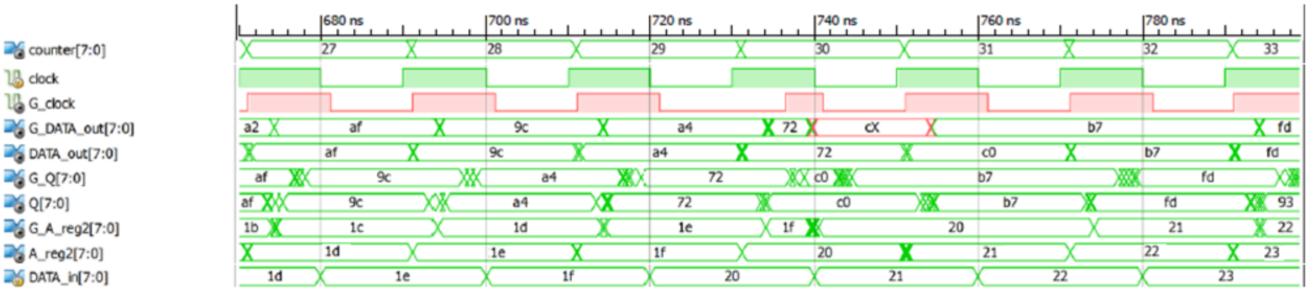


Figure 4.3: A case where the ISE software cannot display the glitch but its effect is obvious in the waveform.

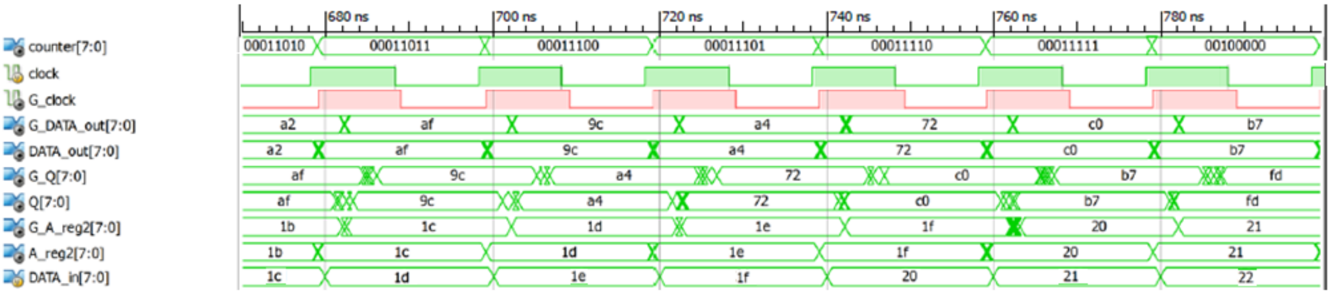


Figure 4.4: A case where the ISE software cannot recognize the glitch as a clock and no fault is generated.

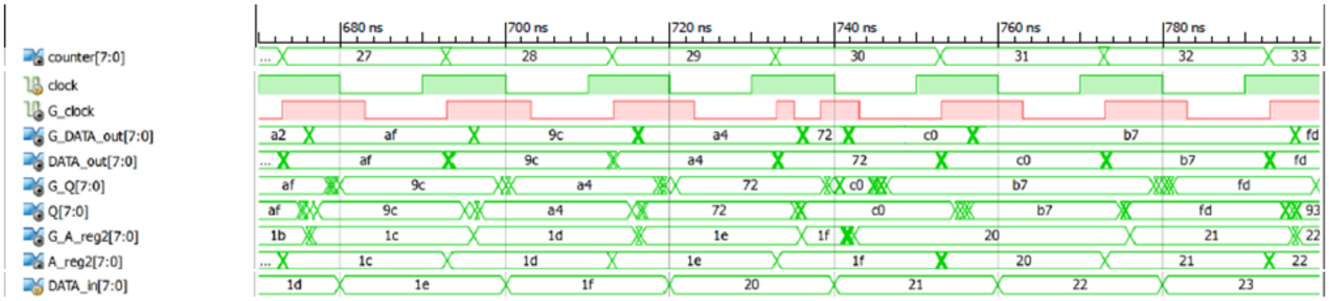


Figure 4.5: A case where the glitch does not cause a fault in the Sbox output.

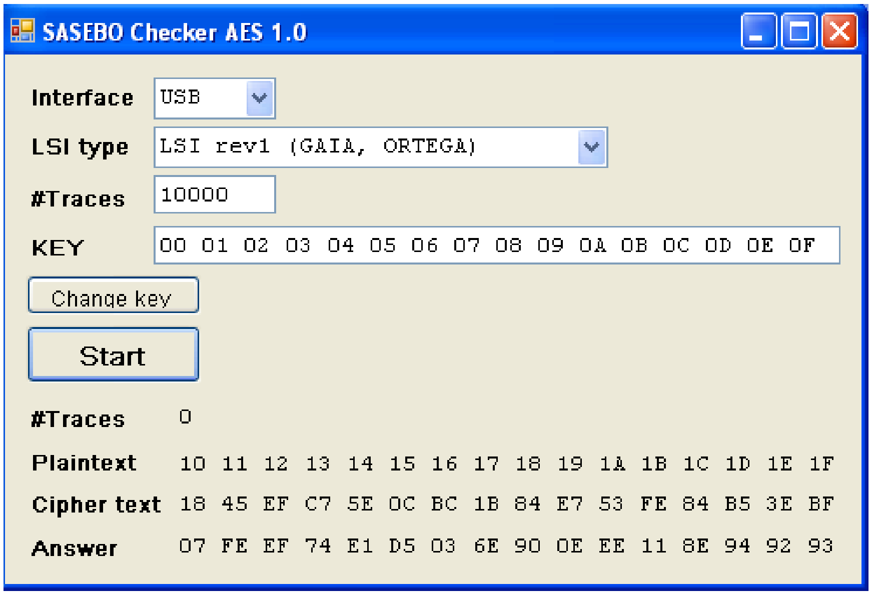
can be selected. The period of the glitch as well as its delay and width can be changed. In the current and next section, it is investigated how the glitch affects the number and position of the faulty bytes/bits in the output when the fault injection round and the glitch period is changed. There are two reasons for considering the number of bytes: first, the operations in AES are executing in a byte-wise manner. Second, many fault attacks utilize this byte-wise operation characteristics in their fault model theories. For instance, Piret's attack theory [39] uses a one-byte random fault in the intermediate state.

The glitch is induced into the last and the second last cipher rounds since these two rounds are frequently used in fault attacks and are suitable for investigating the induced faults. Our experiments are not complete attack processes but fault characterization experiments in which AES is considered as a big propagation delay source. Determining the key can be done by performing the required equations according to an attack theory based on the properly generated faults.

In the first experiment, the clock frequency is 24MHz and each AES round happens in one clock cycle. The glitch is injected in the tenth round and the glitch period (phase difference of clock and clock_b) is gradually decreased in fine steps of 162ps. The glitch delay (phase difference of clock and clk_c) is constant and equal to 3.25ns and the glitch width is changed by changing the phase difference of clock and clk_b. Each encryption is done with the same key and 1000 plaintexts. The SASEBO_AES_Checker software computes the correct answer and compares it with the output ciphertext coming from the FPGA via the USB port. Both the correct answer and the hardware ciphertext as well as the plaintext are shown in the SASEBO_AES_Checker software screen. As soon as the system recognizes a fault, it stops running and the trace number along with the correct and faulty output bytes are recorded. An example is shown in Figure 4.6.

4.2.1 The number of faulty bytes and bits

As Figure 4.7 shows, the number of faulty bytes increases by shortening the glitch period (P_g). Each dot in this figure represents the number of faulty bytes for a specific glitch period. This may be due to the setup timing violation since in the glitchy cycle the register latches much sooner than that in the normal condition. Hence, the output of the combinatorial logic (the state) is sampled prematurely. This experiment is repeatable in terms of the injected clock glitch, but the generated fault value seems to be random. The most remarkable observations are:


SASEBO Checker AES 1.0

Interface: USB

LSI type: LSI rev1 (GAIA, ORTEGA)

#Traces: 10000

KEY: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

```
#Traces      0
Plaintext    10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
Cipher text  18 45 EF C7 5E 0C BC 1B 84 E7 53 FE 84 B5 3E BF
Answer       07 FE EF 74 E1 D5 03 6E 90 0E EE 11 8E 94 92 93
```

Figure 4.6: The SASEBO_AES_Checker software screen.

- The fault happens only when the P_g is in the interval of 5.2ns and 6.4ns which is a narrow range. This means that the attacker can control the fault injection condition very finely. The glitch generator provides this feasibility of generating a glitch period in this range. In addition, the timing window in which the registers sample a wrong state is highly limited. In other words, the reliability of the registers is high.

- A comparison of correct and faulty ciphertexts reveals that the number of faulty bytes increases as P_g is decreased. At the beginning of the fault interval the error occurs in only a single bit and then the number of the faulty bits progressively increases and we see single-byte and multiple-byte faults. The number of faulty bytes varies between 1 and 5 where for $P_g \in [6ns, 6.4ns]$ there is only one output faulty byte. Note that for $P_g = 5.2ns$ the maximum fault occurs (14 bytes).

- Being able to induce a single-byte fault in the tenth round, means it is plausible that injecting such a fault in other rounds is practical. This is a desired result for the attack theories based on a single-byte fault in the output such as Piret's attack [39] and Tunstall's attack [48].

- Fault model in attacks such as Girauds one-bit attack [21] need to be able to inject only a one-bit fault in the AES state. The experimental results show that it is practical to inject one-bit errors by using such glitch generator with the capability of finely incrementing the period of the glitch. Indeed in the interval of $[6ns, 6.4ns]$ only a single-bit error occurs.

- There is no faulty output for $P_g < 5.2ns$. This is because the experimental environment cannot generate such a short clock cycle, as the glitch width is too short and the hardware cannot recognize it as a clock edge. Figure 4.8 is a plot from the oscilloscope while the algorithm is running on the FPGA. The upper signal is the clock signal from the glitch generator module. As the picture shows, there is no glitch detected in the glitchy clock signal. The signal below the clock is the BSY signal from the AES module. It is used as a trigger to represent the start and end time of the encryption procedure. As shown, at the end of the tenth round the trigger changes its value from 1 to 0.

- There is no faulty output for glitches with $P_g > 6.4ns$ (Figure 4.9). Such glitches are treated as a normal clock cycle in which the state has enough time to be correctly latched by the register. The maximum glitch period causing a fault in the output is related to the critical path delay and differs in different ciphers or implementations. It is somewhat less than the maximum path delay, hence, the maximum path delay in this experiment is more than 6.4ns which is true. According to the synthesis report from the Xilinx ISE software after simulating the code, the critical path delay is 8.295ns.

4.2.2 The position of faulty bytes and bits

To observe the positions of the faulty bytes, a glitch with the same characteristics is injected into the hardware during the encryption process of the same plaintext 20 times. From Figure 4.10, it can be seen that most of the time some particular bytes are vulnerable against the setup time violation. This shows that these bytes have a longer path delay than others. In general, each byte has a different path delay. For instance, in this

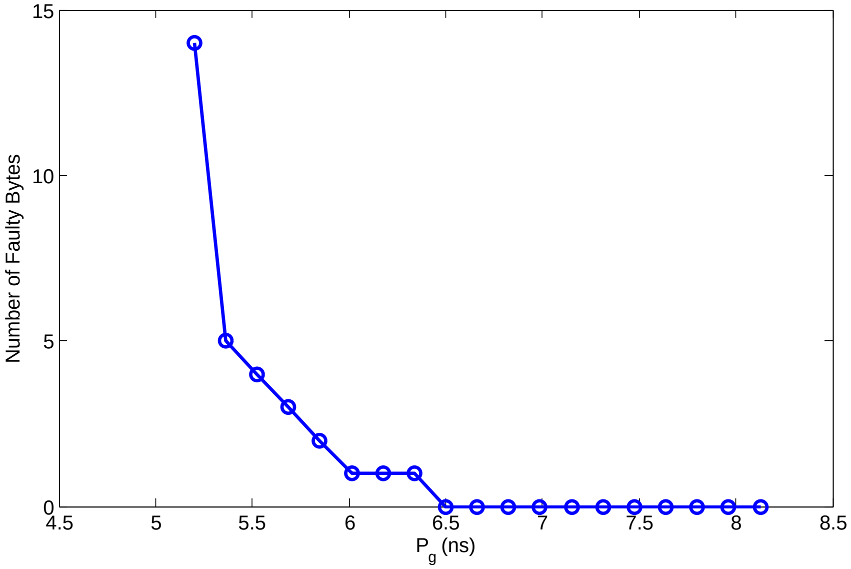
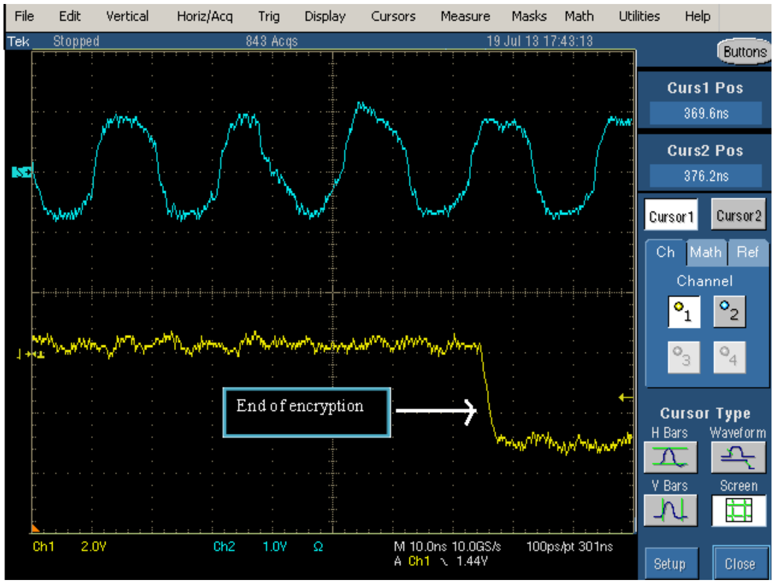


Figure 4.7: The number of faulty bytes when the glitch is injected in round 10



Tek Stopped 843 Acqs 19 Jul 13 17:43:13

Curs1 Pos 369.6ns

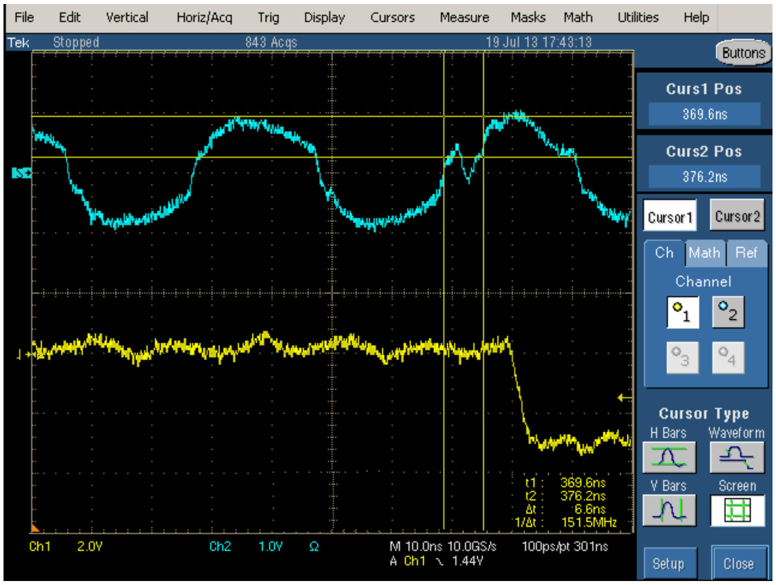
Curs2 Pos 376.2ns

Ch1 2.0V Ch2 1.0V Ω M 10.0ns 10.0GS/s 100ps/pt 301ns

A Ch1 ↘ 1.44V

End of encryption

Figure 4.8: The experiment environment cannot generate a glitch in the glitchy clock signal (the upper signal) with a very short interval. The lower signal is the BSY output which shows the end of encryption.



Tek Stopped 843 Acqs 19 Jul 13 17:43:13

Curs1 Pos 369.6ns

Curs2 Pos 376.2ns

t1 : 369.6ns
t2 : 376.2ns
Δt : 6.6ns
1/Δt : 151.5MHz

Ch1 2.0V Ch2 1.0V Ω M 10.0ns 10.0GS/s 100ps/pt 301ns

A Ch1 ↘ 1.44V

Figure 4.9: The period of the clock glitch is 6.6ns. Glitch is treated as a normal clock and there is no fault in the output. Upper: glitchy clock, lower: BSY signal.

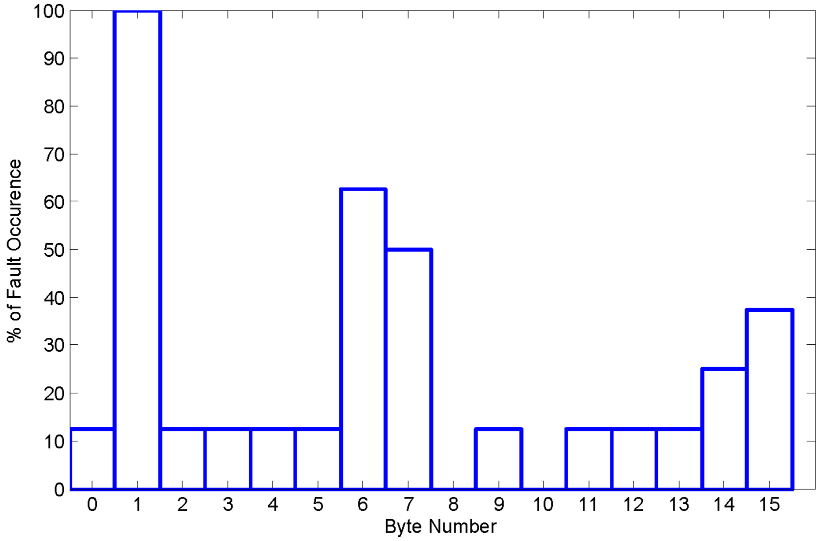


Figure 4.10: Position of faulty bytes when the glitch is injected in round 10

experiment, byte 1 is the most vulnerable one due to its longest path delay, and some bytes like bytes 8 and 10 are hard to inject a fault in .

In case of bit positions, some specific bits are more probable to become faulty. For example in this experiment, bits 12, 14, and 49 are the least robust bits against the glitch fault and mostly become faulty. This shows that the path delay is significantly variable even for the single bits. This is probably because of FPGA placement and routing implementations.

4.2.3 Repeating the experiment with a different set of plaintext

The experiment was repeated with a different set of plaintexts to verify if the same results are achieved. In the previous section, it was observed that some specific bytes/bits are more sensitive to faults due to their longer path delay. Each data bit that arrives to a register possesses its own logic path and propagation time. This highly depends on the data handled during the operation. Consequently, it is expected that changing the plaintext modifies the propagation delays and affects the probability of injecting a fault into different bits. Another possible reason to have different propagation times among bits is the type of the operation each bit undergoes. However, in AES almost all bits of the input

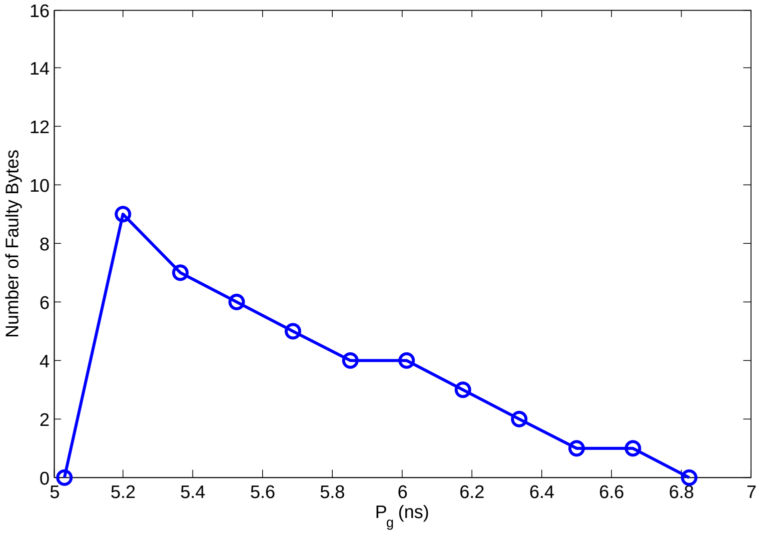


Figure 4.11: The number of faulty bytes when the glitch is injected in round 10 for a second set of plaintexts

pass the same path of functions.

Figure 4.11 shows the number of faulty bytes in the output for the new set of plaintexts. As it can be seen, the overall flow of the fault statistics is almost same for different plaintexts. For the glitch periods less than a specific amount (5.2 ns), the glitch is not recognized by the experiment environment because the glitch signal shape becomes blunt. In addition, the glitch periods in the interval of [5.2 ns, 6.6 ns] cause faulty bytes in the output, and glitch periods greater than 6.6 ns do not affect latching the intermediate values into the registers. Furthermore, the number of faulty bytes increases by decreasing P_g such that for $P_g = 5.2ns$, the ciphertext includes 8 wrong bytes and for $P_g = 6.6ns$ only one byte is faulty. Figure 4.12 represents the position of the output faulty bytes.

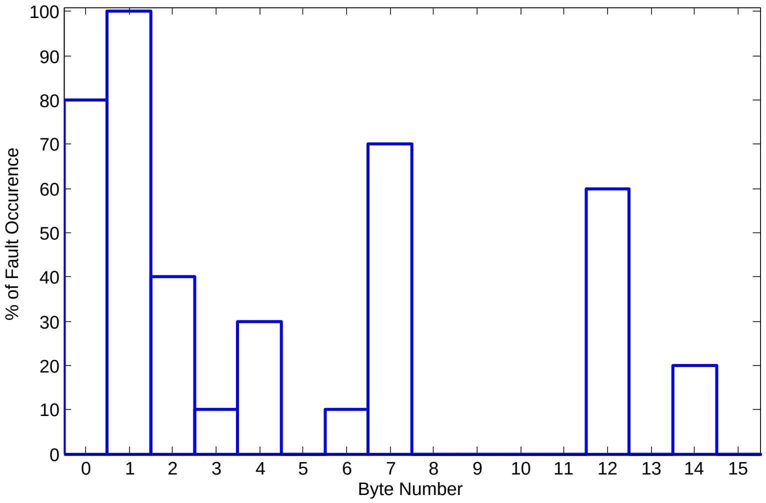


Figure 4.12: Position of faulty bytes when the glitch is injected in round 10 for a second set of plaintexts

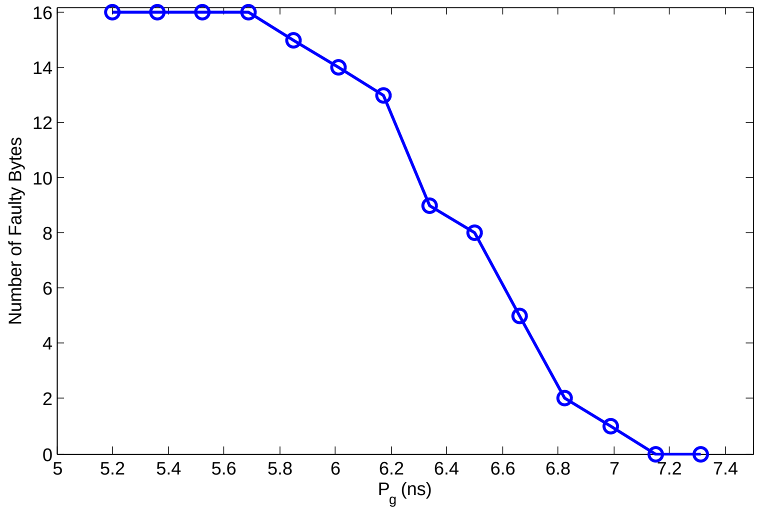


Figure 4.13: The number of faulty bytes when the glitch is injected in round 9

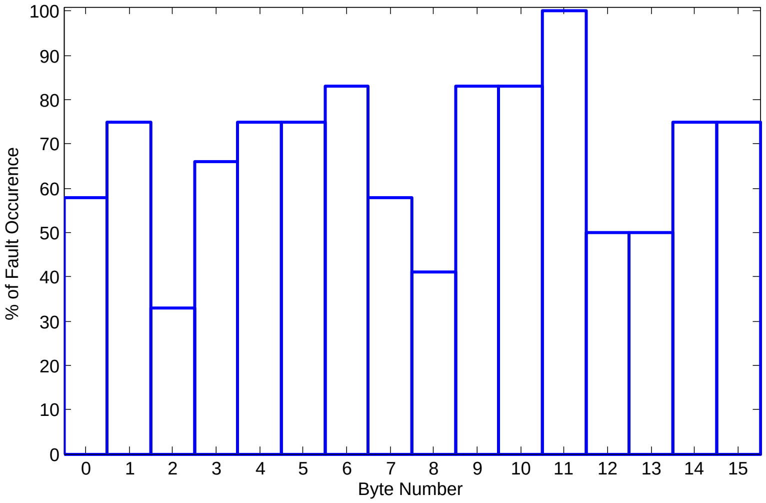


Figure 4.14: Position of faulty bytes when the glitch is injected in round 9

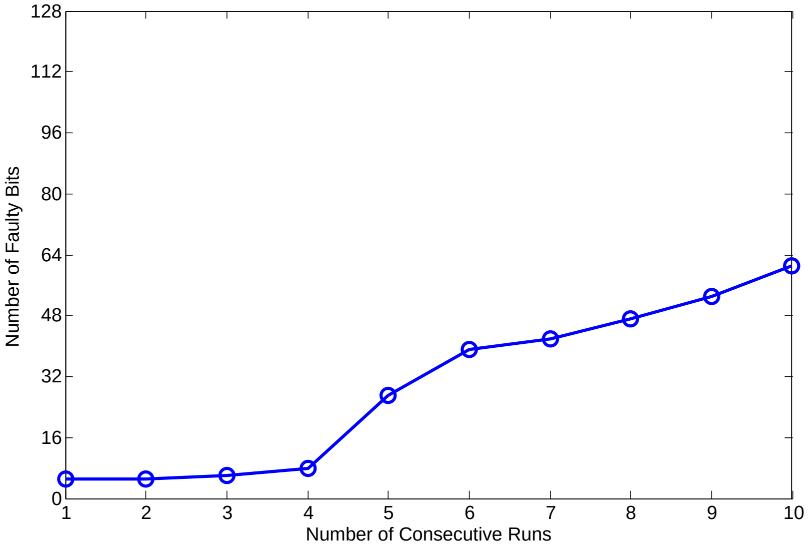


Figure 4.15: Reprogramming the FPGA increases the board temperature, and the number of output faulty bits as the result.

4.3 Fault injection in the penultimate round of the AES implementation

The experiment is repeated on round nine, i.e. the encryption process is performed with a constant key and a set of 1000 plaintexts while the clock frequency is 24MHz and each AES round occurs in a clock cycle. Then, to have a closer observation of the position of the faulty bytes, the encryption was repeated 20 times with a constant key and plaintext. Due to the diffusion properties of the AES block cipher which causes the error propagation, it is expected that the number of faults increases. The diffusion is due to the MixColumn operation in round 9 which is not executed in round 10. As shown in Figure 2.2, a single error in a round will spread over different bytes in the next round. The number of faulty bytes is displayed in Figure 4.13.

As expected, faults are present for a wider range of glitch period: $5.2ns < P_g < 7ns$. Besides, the fault is spread over a larger number of output bytes and bits such that, for $P_g < 5.7ns$ all the output bytes are faulty. Then this number decreases gradually and finally at $P_g = 6.9875ns$ the fault happens in only one byte. This is a pleasant result for implementing fault attacks with a fault model based on a single-byte error. Similar to the previous section, there are some bytes that are not easy to inject a fault in, and this confirms again that the path delay for different bytes is not exactly the same and some bytes have a considerably smaller path delay. It can be seen in Figure 4.14, which represents the fault occurrence percentage for each output byte, that bytes number 6, 9, 10, and 11 are erroneous most of the time.

4.4 The Effect of Heating the device

Generally, electronic chips only work reliable within a certain range of temperature. If the temperature outside or inside the chip is too low or too high, faults might occur. For this reason, every PC has a fan to ensure that the heat produced by the internal circuitry does not overheat the computer. There are attacks that can induce a single bit-flip fault in virtual machines in temperatures between 80 °C and and 100 °C [23]. However, the experiment also showed that if the attack conditions are not finely tuned, the attacks often cause the operating system to crash, thus requiring a complete reinstallation. Attacking a PC often requires one to open it or to disable the fan, while in portable devices, the device itself is focused by a heating source. However, it is observed that changes in memory due to heat usually affect a large area which includes many bits.

In this section we reprogram the FPGA chip in trying to increase the heat within the IC, i.e., by programming the chip and running the program repeatedly for several times. For this experiment a glitch with $P_g = 5.68ns$ is produced and the fault is induced into the tenth round of AES. The same key and plaintext are entered into the system for all acquisitions. In order to try to increase the chip temperature, the board was programmed ten times, one immediately after another. However there was no temperature sensor available to verify the temperature increment, it is assumed that the chip temperature did increase since this could be sensed by touching the chip during the process. As a result (Figure 4.15), an increase in the number of output faulty bits is observed. After the tenth run, to decrease the board temperature the waiting time between two consecutive runs was increased to 5-10 minutes. This gradually decreases the number of output faulty bits.

4.5 Summary

In this chapter, the glitch generator was added to an Sbox module and simulated in Xilinx ISE software in order to examine the theoretical fault attack hypotheses. This showed that for a glitch period in a specific interval, an error can be injected in the output registered value. Furthermore, changing the glitch characteristics changes the fault domain in a controllable manner. Then, an FPGA-based testbed was utilized to inject the glitch faults and examine the feasibility of fault attacks in practice. Based on the experimental results, there is a specific glitch period for less than which the glitches cannot be identified by the hardware and cause a fault in the output. Moreover, the clock glitches with a glitch period greater than a maximum value, which is somewhat less than the critical path delay, are treated as a normal clock cycle and hence do not produce any fault. Decreasing the glitch period in the interval between these minimum and maximum values, increases the fault probability. Considering the position of faulty bytes and bits, it is likely that bits with longer path delay are more vulnerable to glitch attacks. Finally, the effect of reprogramming the device was examined while inducing the glitch fault. This confirmed the expectation that reprogramming the FPGA continuously for several times increases the probability of successfully injecting faulty bytes in the ciphertext. This result is likely due to the generation of heat within the IC from the reprogramming.

Chapter 5

Discussion and Conclusions

This section provides a summary of the work done in this thesis, a brief discussion of limitations, and conclusion of the observed simulation and experimental results.

5.1 Summary and Discussion

In this thesis the characteristics of the glitch which generated a fault and the effect of the fault was explored. First, the clock glitch faults were injected into an Sbox architecture and the post-route timing simulation results confirmed the theoretical expectations. Then, an FPGA-based testbed was utilized to inject the glitch fault into an existing full AES. The feasibility of fault attacks in practice were examined. Experiments verify that faults can be successfully injected utilizing clock glitches generated on the FPGA. A comparison of these fault injection experimental results with timing simulation of the AES circuit was useful for verification. But unfortunately, neither the timing simulation results nor implementation information for executing it were open.

Based on the simulation approach, an Sbox based on Normal Basis arithmetic together with a glitch generator, which was able to generate a glitch with changeable period in any desired clock cycle, were designed. Various glitch characteristics were examined using the post-route timing simulation. The glitch period interval in which the fault occurred was estimated. The Sbox architecture included a glitch generator and two Sbox blocks, one with a normal clock input and the other one with the glitchy clock generated by the glitch generator block. Comparing the outputs of the two Sbox blocks in the output waveform from the Xilinx ISE software, this was observable whether or not a fault has happened in the desired clock cycle and what the fault value was.

Hardware experiments illustrate that on-chip glitch generator on a SASEBO-GII was successful in injecting faults in a complete AES implementation. Analysis of the faults illustrated the following two main characteristics:

1. The fault occurrence was finely controllable by the clock glitch details.
2. The value of the faults were random.

These experimentally observed features are used in most attacks because the fault models in those attacks assume that first, single faults, either bit-wise or byte-wise, are injected. Second, if an attack requires two faults to happen, the device must not produce the same fault in both fault generation processes.

As explained in the previous chapter, the fault happens in the output only in a narrow range of the glitch period. In other words, the timing window in which the registers sample a wrong state is highly limited. Hence, the fault occurrence is controllable by the attacker (characteristic 1). The randomness of the fault (characteristic 2) is because when shortening the clock cycle and the round calculation time, the probability of bit reversal is 0.5.

The position of the output faulty bytes likely indicates that each byte of the state has a different path delay and this is perhaps why some of them are more vulnerable to setup time violation attacks. A new place and route technique might change the path delay of the bytes and affect their robustness against clock glitch attack. Therefore, for each different implementation of the AES, testing utilizing such a fault attack testbed is important.

Investigating the effect of heating the device by reprogramming it while running a cryptographic function indicates that the temperature outside of the chip or inside the chip (due to reprogramming it for several times) may increase the number of faulty bits/bytes in the output. However, most fault attack models need a single bit or single byte fault to be injected. So, in a real attack, the attacker must watch the temperature in order to generate a desired single fault.

To the best of our knowledge, this is the first FPGA-based testbed including on-chip glitch generator to inject the glitch faults into the AES cryptosystem implemented on the FPGA. Since the glitch generator is implemented on the same FPGA as the cryptosystem, a uniform environment is provided to examine the robustness of a cipher against glitch attacks. The glitch generator proposed in [20], is applied on AES, but it uses external equipments such as two separate pulse generators and an oscilloscope and needs to be manually tuned. The pulse generators have clock signals with the same frequency but different phase shifts and the glitch happens when switching between the two clocks. Although, this mimics a real attack (because the glitch generator utilizes off-chip circuitry), this thesis utilizes an on-chip glitch generator in order to have a low-cost testbed design which is useful for investigating the viability and analysis of fault injections on various cryptographic functions in future embedded systems. The glitch generator in [19] has been applied only on RSA cryptosystem and the glitch characteristics and resulting faults have not been investigated. Furthermore, the effect of the temperature in clock glitch fault attacks have not been explored in previous works.

5.2 Conclusion and Future Work

In this work the study of fault models used in fault attack theories was analyzed by observing experimental data. Chapter 2 outlines various aspects of fault injection methods and their feasibility in practice followed by a survey and comparison of fault models introduced in literature. Chapter 3 proposes the glitch generator design method and explains how the glitch circuitry produces the fault and how the attacker can accurately control the glitch features and estimate the generated fault. Furthermore, the cryptographic board on which the AES architecture is implemented is introduced. Finally, the simulation and implementation results are presented in chapter 4.

To conclude, a hardware testbed with a glitch generator requiring only one FPGA is created which is proper for testing possible future susceptibility of fault attacks on AES and other cryptosystems. The fault can be injected reliably and finely controlled and its effect looks random. Future work would involve completing the fault injection by performing a complete analysis of the specific faults to extract the secret key.

This thesis did not examine whether or not the faults happen in the key schedule because in AES the key path delay is considerably smaller than the data path delay. However, in designs or algorithms in which there is no noticeable difference between the two path delays, it would be necessary to explore the effect of the fault injection. Other future work could explore the fault injection effect on high performance AES designs with error-detection scheme such as [44]. In addition, having a uniform hardware testbed with a glitch generator implemented on a single FPGA without external equipment, makes it suitable for testing fault injection attacks on different cryptosystems. Hence, implementing other ciphers and cryptographic functions, and injecting faults into them may be part of the future work.

In order to have a more precise observation of the temperature increment and a better control of the heating effect on the output faults, future work could use a temperature sensor during the attack or test procedure. This helps the user stop running or reprogramming the cryptographic function on the chip and hence prevent producing multiple-byte fault.

As discussed earlier, even with the same glitch characteristics the value of the resulting fault is random or at least there is no guarantee to produce the sam fault value. This is a very useful feature because if the fault model needs more than one pair of correct and faulty ciphertexts, then in order to solve the equation sets based on the correct ciphertext, faulty ciphertext, and the fault value, different fault values are needed to obtain more than one set of candidates for the secret key. Intersecting these candidate sets results in the unique secret key. As future work, it may also be interesting why the fault value produced is random and how further randomness guarantees may be possible.

References

[1] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=45989.

[2] www.xilinx.com/support/documentation/user_guides/ug190.pdf.

[3] https://www.google.ca/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&ved=0CCwQFjAA&url=http%3A%2F%2Fwww.prevailing-technology.com%2Fpublications%2FSecrets_of_the_DCM_(Part_II).ppsx&ei=lpgnUsDDA8rfqAGHs4DQAg&usg=AFQjCNEeHbJXJykUtMkEBVws8s0Q2uRFWA&sig2=KvAdz9weljSzh0yAsU5HTQ.

[4] http://www.morita-tech.co.jp/SASEBO/en/board/sasebo-g2.html.

[5] Michel Agoyan, Jean-Max Dutertre, Amir-Pasha Mirbaha, David Naccache, Anne-Lise Ribotta, and Assia Tria. How to flip a bit? In *Proceedings of the 2010 IEEE 16th International On-Line Testing Symposium*, IOLTS '10, pages 235–239, Washington, DC, USA, 2010. IEEE Computer Society.

[6] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When clocks fail: on critical paths and clock faults. In *Proceedings of the 9th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Application*, CARDIS'10, pages 182–193, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] Ross Anderson and Markus Kuhn. Low cost attacks on tamper resistant devices, 1997.

[8] C. Aumller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert. Fault attacks on rsa with crt: Concrete results and practical countermeasures, 2002.

[9] F. Bao, R. H. Deng, Y. Han, A.Jeng, A. D. Narasimhalu, and T. Ngair. Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults. In *In*, pages 115–124. Springer-Verlag.

[10] Hagai Bar-El, Hamid Choukri, David Naccache Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. 2004.

[11] Ingrid Biehl, Bernd Meyer, Volker Mller, Universitas Kristen Duta Wacana, and Jl. Dr. Wahidin. Differential fault attacks on elliptic curve cryptosystems. pages 131–146. Springer-Verlag, 2000.

[12] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems, 1997.

[13] Johannes Blömer and Jean-Pierre Seifert. Fault based cryptanalysis of the advanced encryption standard (aes). In RebeccaN. Wright, editor, *Financial Cryptography*, volume 2742 of *Lecture Notes in Computer Science*, pages 162–181. Springer Berlin Heidelberg, 2003.

[14] Dan Boneh, Richard A. Demillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. pages 37–51. Springer-Verlag, 1997.

[15] Dan Boneh, Richard A. Demillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14:101–119, 2001.

[16] José Carrijo, Rafael Tonicelli, and Anderson C. A. Nascimento. A fault analytic method against hb$^+$. *IEICE Transactions*, 94-A(2):855–859, 2011.

[17] Chien-Ning Chen and Sung-Ming Yen. Differential fault analysis on aes key schedule and some countermeasures. In Rei Safavi-Naini and Jennifer Seberry, editors, *Information Security and Privacy*, volume 2727 of *Lecture Notes in Computer Science*, pages 118–129. Springer Berlin Heidelberg, 2003.

[18] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential fault analysis on a.e.s. *IACR Cryptology ePrint Archive*, 2003:10, 2003.

[19] Sho Endo, Takeshi Sugawara, Naofumi Homma, Takafumi Aoki, and Akashi Satoh. An on-chip glitchy-clock generator for testing fault injection attacks. *Journal of Cryptographic Engineering*, 1(4):265–270, 2011.

[20] T. Fukunaga and J. Takahashi. Practical fault attack on a cryptographic lsi with iso/iec 18033-3 block ciphers. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on*, pages 84–92, 2009.

[21] Christophe Giraud. Dfa on aes. In Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa, editors, *Advanced Encryption Standard AES*, volume 3373 of *Lecture Notes in Computer Science*, pages 27–41. Springer Berlin Heidelberg, 2005.

[22] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[23] Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, pages 154–, Washington, DC, USA, 2003. IEEE Computer Society.

[24] Michael Hutter, Jörn-Marc Schmidt, and Thomas Plos. Rfid and its vulnerability to faults. In *Proceeding sof the 10th international workshop on Cryptographic Hardware and Embedded Systems*, CHES '08, pages 363–379, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[26] Chong Hee Kim. Differential fault analysis against aes-192 and aes-256 with minimal faults. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*, pages 3–9, 2010.

[27] Chong Hee Kim. Improved differential fault analysis on aes key schedule. *IEEE Transactions on Information Forensics and Security*, 7(1):41–50, 2012.

[28] Chong Hee Kim. Improved differential fault analysis on aes key schedule. *Information Forensics and Security, IEEE Transactions on*, 7(1):41–50, 2012.

[29] Chong Hee Kim and Jean-Jacques Quisquater. Faults, injection methods, and fault attacks. *IEEE Design & Test of Computers*, 24(6):544–545, 2007.

[30] ChongHee Kim and Jean-Jacques Quisquater. New differential fault analysis on aes key schedule: Two faults are enough. In Gilles Grimaud and Franois-Xavier Standaert, editors, *Smart Card Research and Advanced Applications*, volume 5189 of *Lecture Notes in Computer Science*, pages 48–60. Springer Berlin Heidelberg, 2008.

[31] Oliver Kmmerling, Markus G. Kuhn, Oliver Kmmerling, and Markus G. Kuhn. Design principles for tamper-resistant smartcard processors. pages 9–20, 1999.

[32] Donald Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[33] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. pages 104–113. Springer-Verlag, 1996.

[34] Leslie Lamport. *LATEX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[35] Yang Li, Kazuo Sakiyama, Shigeto Gomisawa, Toshinori Fukunaga, Junko Takahashi, and Kazuo Ohta. Fault sensitivity analysis. In Stefan Mangard and Franois-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 320–334. Springer Berlin Heidelberg, 2010.

[36] Amir Moradi, Mohammad T. Manzuri Shalmani, and Mahmoud Salmasizadeh. A generalized method of differential fault attack against aes cryptosystem. In *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop*, volume 4249 of *Lecture Notes in Computer Science*, pages 91–100. Springer, 2006.

[37] National and N. I. S. T. Technology. *Announcing the Advanced Encryption Standard (AES)*, 2001.

[38] Martin Otto. *Fault Attacks and Countermeasures.* PhD thesis, University of Paderborn, 2005.

[39] Gilles Piret and Jean jacques Quisquater. A differential fault attack technique against spn structures, with application to the aes. In *AES and KHAZAD. CHES 2003, LNCS 2779*, pages 77–88. Springer-Verlag, 2003.

[40] J. J Quisquater and D. Samyde. Eddy current for magnetic analysis with active sensor. 2002.

[41] Bruno Robisson and Pascal Manet. Differential behavioral analysis. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 413–426. Springer Berlin Heidelberg, 2007.

[42] Atri Rudra, Pradeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, and Pankaj Rohatgi. Efficient rijndael encryption implementation with composite field arithmetic. In etin Kaya Ko, David Naccache, and Christof Paar, editors, *CHES*, volume 2162 of *Lecture Notes in Computer Science*, pages 171–184. Springer, 2001.

[43] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A compact rijndael hardware architecture with s-box optimization. pages 239–254. Springer-Verlag, 2001.

[44] Akashi Satoh, Takeshi Sugawara, Naofumi Homma, and Takafumi Aoki. High-performance concurrent error detection scheme for aes hardware. In *Proceeding sof the 10th international workshop on Cryptographic Hardware and Embedded Systems*, CHES '08, pages 100–112, Berlin, Heidelberg, 2008. Springer-Verlag.

[45] Debdeep Mukhopadhyay Sk Subidh Ali and Michael Tunstall. Differential fault analysis of aes: Towards reaching its limits. Cryptology ePrint Archive, Report 2012/446, 2012. http://eprint.iacr.org/.

[46] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. pages 2–12. Springer-Verlag, 2002.

[47] Junko Takahashi, Toshinori Fukunaga, and Kimihiro Yamakoshi. Dfa mechanism on the aes key schedule. In *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography*, FDTC '07, pages 62–74, Washington, DC, USA, 2007. IEEE Computer Society.

[48] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential fault analysis of the advanced encryption standard using a single fault. In ClaudioA. Ardagna and Jianying Zhou, editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, volume 6633 of *Lecture Notes in Computer Science*, pages 224–233. Springer Berlin Heidelberg, 2011.

[49] Yongbin Zhou and DengGuo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing, 2005.