Incremental Fault Analysis: A New Differential Fault Attack on Block Ciphers

INCREMENTAL FAULT ANALYSIS: A NEW DIFFERENTIAL FAULT ATTACK ON BLOCK CIPHERS

BY

TREVOR E. POGUE, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

Master of Applied Science

Master of Applied Science (2018) McMaster University

(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE:	Incremental Fault Analysis: A New Differential Fault Attack on Block Ciphers
AUTHOR:	Trevor E. Pogue B.Eng. (Electrical Engineering) McMaster University, Hamilton, Ontario, Canada
SUPERVISOR:	Dr. Nicola Nicolici
NUMBER OF PAGES:	xiv, 78

To my family

Abstract

Electronic devices such as phones and computers use cryptography to achieve information security. However, while cryptographic algorithms may be strong theoretically, their physical implementations in hardware can leak unintentional side information as a byproduct of performing their computations. A device's security can be compromised from this leakage through side-channel attacks. Research in hardware security reveals how dangerous these attacks can be and provides security countermeasures. This thesis focuses on a category of side-channel attacks called fault attacks, and contributes a new fault attack method that can compromise a cryptographic device more rapidly than the previous methods when using practical fault injection techniques.

We observe that as a circuit is further overclocked, new faults are often superimposed upon previous ones. We analyze the incremental changes rather than the total sum in order to extract more secret information. Unlike many previous methods, ours does not require precise fault injection techniques and requires no knowledge of when the internal state is in a specific algorithmic stage. Results are confirmed experimentally on hardware implementations of AES-128, 192, and 256.

Acknowledgements

I would like to acknowledge all of those who contributed to the completion of this thesis. First of all, I want to express my gratitude to my M.A.Sc. supervisor, Dr. Nicola Nicolici. He puts forth a tremendous amount of effort into his students, and I am very grateful to be on the receiving end of this on a regular basis. Furthermore, I thank him for the direction and insight that made this work possible.

I thank all of the colleagues I have overlapped with during my time at the Computer-Aided Design and Test Research Group at McMaster University, both former and present members. Furthermore, I would like to thank Stefan Dumitrescu and Alex Lao for many stimulating discussions, some of which aided this work, and also allowed me to gain other technical knowledge and helped make the time throughout my degree very enjoyable. I would like to thank Dr. Pouya Taatizadeh for encouraging me to pursue graduate school during his role as my instructor for the undergraduate Digital Systems Design course at McMaster University. This inspired me to work harder and ultimately lead to my enrolment into graduate school, later providing other professional opportunities as well.

I would like to thank my parents and family for supporting me in my technical hobbies growing up. Finally, I would like to thank my wife, Michelle Pogue, for her never ending support which has played a major role in my academic success.

Glossary

Encryption: The process of transforming a readable message into a confidential unreadable form.

Decryption: The process of transforming an encrypted message back into its readable form.

Cipher: A series of transformations that perform encryption.

Inverse cipher: A series of transformations that perform decryption.

Plaintext: A decrypted message input to a cipher or output from an inverse cipher.

Ciphertext: An encrypted output from a cipher or input to an inverse cipher.

Cipher key: A large number acting like a password that is required by a cipher/inverse cipher in order to perform encryption and/or decryption.

Fault: A faulty computation occurring in a circuit due to the circuit being stressed.

Fault model: A set of characteristics required in a fault in order to satisfy a certain fault attack.

Target Fault: A fault that satisfies the fault model.

Ciphertext pair: A pair of ciphertexts resulting from encryptions of the same plaintext that differ by error propagation resulting from a fault.

Target ciphertext pair: A pair of ciphertexts resulting from encryptions of the same plaintext that differ by error propagation resulting from a target fault.

Abbreviations

AES: Advanced Encryption Standard.

DES: Data Encryption Standard.

RSA: Rivest Shamir Adleman.

SPN: Substitution Permutation Network.

IFA: Incremental Fault Analysis.

DFA: Differential Fault Analysis.

FSA: Fault Sensitivity Analysis.

DFIA: Differential Fault Intensity Analysis.

DERA: Differential Error Rate Analysis.

NUFVA: Non-Uniform Faulty Value Analysis.

SPA: Simple Power Analysis.

DPA: Differential Power Analysis.

EMA: Electromagnetic Analysis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern technology enables fast global communication, which plays a major role in the advancement of modern society. However, these interactions rely on many information security primitives, many of which are implemented through cryptography [1] [2]. For example, it is now common practice to share credit card information when shopping online, or to prove your identity when banking online. Online shopping requires the credit card information to be shared securely between two or more parties without an adversary being able to listen in on the communication and compromise the credit card information. Online banking requires both the user and bank to prove their identity before withdrawing or depositing money so that an adversary cannot compromise another's funds and a client does not deposit funds to an adversary acting as the bank. Both of these examples require a variety of information security principles to be implemented, most of which utilize cryptography.

1.1 Information Security

The applications of information security are far reaching, examples include:

1.1.1 Applications

- **Secure Internet Communication**: Security is crucial for many aspects of modern web usage such as online shopping, online banking, etc. Every website with a URL beginning with https is taking measures to ensure a secure connection between the user and website [3].
- **Secure Data Storage**: Security is crucial when passwords and other confidential data is stored on devices such as computers and phones [4].
- **Cryptocurrency**: Cryptocurrency is a digital currency system that allows for the payment of goods and services. It serves a similar purpose as physical currency except that cryptocurrency allows for decentralized, borderless, and more instantaneous transactions. In 2009 the most widely known cryptocurrency, Bitcoin, was introduced [5]. There have since been cryptocurrency schemes created that improve upon the shortcomings of Bitcoin such as transaction speed and the energy costs required to maintain its infrastructure.

1.1.2 Objectives

Information security [1] addresses the following objectives that make these applications possible:

- **Confidentiality**: To assure the security of confidential information, the information must only be readable by the intended user and otherwise unreadable

while being transmitted or stored.

- **Identity Authentication**: To engage in a secure communication, both parties need assurance that the other is who they claim to be, and that they are both currently active.
- **Message Authentication**: In secure communications, the identity of the sender must also be bound to their messages in order to continuously ensure the communication was not intercepted after identity authentication was established. This ties into data integrity as well.
- **Data Integrity**: It is important to ensure that transmitted data is not altered through interference or by malice. An error introduced during an electronic funds transfer could cause an incorrect amount of money to be credited or debited. Data integrity ensures that accidental or intentional modifications to data is detectable.
- **Non-Repudiation**: It is important to prevent an entity from denying any previous actions or commitments. For example, this would avoid a scenario in which a purchase is made and the seller later denies ever selling the item to the buyer.

1.2 Cryptography

Cryptography is a means of implementing the information security objectives listed above. The overall goal is the prevention of malicious activities. A list of primitive cryptographic building blocks are detailed below that provide the mathematical means

of achieving information security objectives.

1.2.1 Encryption

Encryption provides a means to temporarily transform a message into an unreadable representation such that the information remains confidential while being communicated or stored. This must be done in a way that the confidential representation of the message can later be transformed back to its readable representation at a later time and only by the intended user. **Decryption** refers to the process of transforming the confidential representation of the message back to its readable representation. The readable representation of the message is referred to as a **plaintext**, and the confidential representation is referred to as a **ciphertext**. A **cipher** is an algorithm that performs encryption, and an **inverse cipher** is an algorithm that performs decryption. Performing encryption or decryption requires a **key** that only the intended user will have access to. A key can be thought of like a password, and is just a very large number. There are two main categories of encryption schemes discussed below, symmetric-key and public-key.

1.2.2 Symmetric-Key Encryption

Symmetric-key encryption [6] is an encryption scheme in which the same key is used for both encryption and decryption. Symmetric-key ciphers typically have high rates of data throughput and for this reason are used for securing the bulk of a message's information. Two types of symmetric-key algorithms are described below.

Block ciphers [7] [8] [9] are one of the essential building blocks of symmetric-key encryption systems. They work by encrypting multiple characters of a message at a

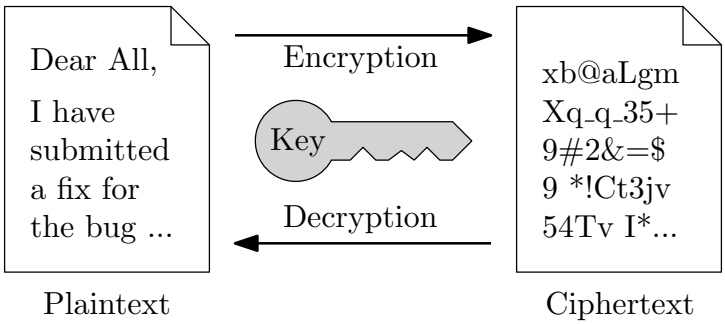

Figure 1.1: Symmetric-key encryption.

time. Block ciphers can be either symmetric-key or public-key (discussed in section 1.2.3), but are typically associated with symmetric-key encryption.

Stream ciphers [10] [11] [12] work by encrypting individual characters of a message independently. They typically have faster hardware execution speeds and lower complexity than block ciphers. They are less susceptible to error propagation which can be advantageous in scenarios in which transmission errors are more likely. They can be mandatory in some situations such as telecommunications where characters must be received and processed individually, or when the communications are sensitive to latency. Stream ciphers can be either symmetric-key or public-key (discussed in section 1.2.3), but are typically associated with symmetric-key encryption.

1.2.3 Public-Key Encryption

Public-key encryption [13] [14] [15] addresses the following issue with symmetric-key encryption: If Sally wants to send an encrypted message to Bob, how can she safely send the symmetric-key to him in order to allow him to decrypt the encrypted message? If she sends the symmetric-key along with the encrypted message, the symmetric-key could be observed by a third party and thus be used to decrypt the

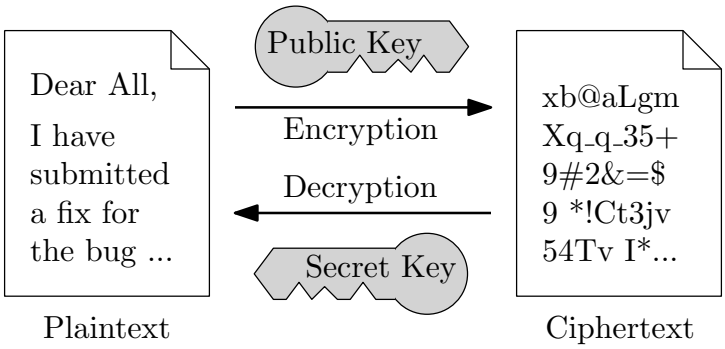

Figure 1.2: Public-key encryption.

message being sent. This is where the use of public-key encryption comes into play. In public-key encryption, each party has a unique private-key as well as a public-key. The public-keys are used to encrypt a message which can then only be decrypted by the private-key of the intended recipient. Public-key encryption typically works through more computationally demanding mathematical operations and is slower to execute than symmetric-key encryption. For this reason symmetric-key encryption is usually used for encrypting core data and public-key encryption is used for encrypting and sharing the symmetric-keys.

1.2.4 Hash Functions

A **hash function** is a function that maps a binary string of arbitrary length to a random binary string of fixed length called a **hash-value**. The likelihood of two inputs being mapped to the same hash-value must be unlikely. Furthermore, a hash function must not be invertible and the input cannot be derived based on the output. Hash-values are commonly used to act as a compact representation of their corresponding longer input strings. One common application for hash functions is for **digital signatures** [16] which are used in authentication, data integrity, and

non-repudiation.

1.3 Cryptanalysis

Cryptanalysis [17] [18] is the study of testing the strength of cryptographic algorithms and their physical implementations. Cryptosystems are tested by trying to retrieve their encrypted information without having access to the key. A. Kerckhoffs stated a cryptanalysis principle that cryptosystems are more secure if the complete details of the cryptographic algorithms used and how they are implemented are public knowledge [2]. This way, weaknesses can be found and addressed by cryptanalysts around the world. If cryptanalysts around the world eventually cannot find ways of compromising a cryptosystem, then a smaller team of adversaries will likely not be able to either.

A cryptanalytic attempt is called an **attack**. The following are some examples of different categories of cryptanalytic attacks:

- **ciphertext-only attacks**: The cryptanalyst only has access to the encrypted ciphertexts of several messages. The goal is to recover the decrypted plaintext form of the ciphertexts or recover the key in order to decrypt the ciphertexts to reveal the sensitive information [19].

- **Known-plaintext attacks**: The cryptanalyst has access to both the encrypted ciphertexts and the corresponding plaintexts of several messages. The goal in this case is to deduce the key or deduce some other method to decrypt any new messages being encrypted with the same key [20].

- **Chosen-plaintext attacks**: The cryptanalyst not only has access to both the

encrypted ciphertexts and the corresponding plaintexts of several messages, but also has control over which plaintexts are encrypted. This is more advantageous because specific plaintext combinations can be chosen that together reveal more information about the key than any random sequence of plaintexts could. The goal in this case is to deduce the key or deduce some other method to decrypt any new messages being encrypted with the same key [21].

- **Side-channel attacks**: The cryptanalyst has physical access to or is within a certain proximity of the cryptosystem and has the ability to observe physical phenomena leaked as a byproduct of the physical implementation performing its computation. This category of cryptanalysis started gaining attention in the the 1990's and is discussed more in section 1.4.

1.4 Side-Channel Attacks

While cryptographic algorithms may be strong theoretically, their physical implementations in hardware can leak unintentional side information concerning their internal state [22]. Research in hardware security reveals how dangerous this leakage is and provides security countermeasures [23]. An attack on a cryptographic system is the attempt to extract secret information that can be used to decrypt sensitive encrypted data. A side-channel attack [24] exploits secret information that is leaked as a byproduct of the physical implementation performing its computation. Examples of attacks leveraging side-channel leakage include power attacks [25] [26], electromagnetic attacks [27] [28], accoustic attacks [29], timing attacks [30] [31], cache attacks [32] [33] [34] [35], and fault attacks [36] [37] [38] [39] [40] [41] [42].

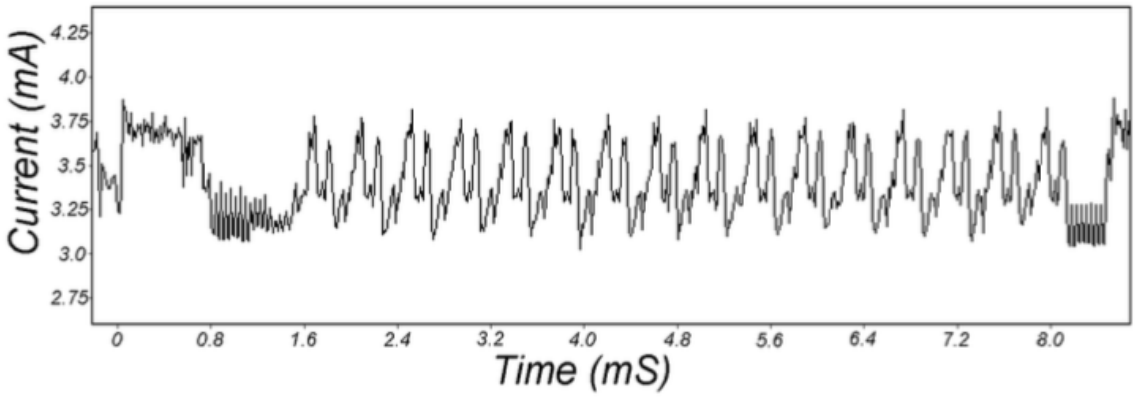

Figure 1.3: Power consumption of a cryptosystem during an encryption [25].

1.4.1 Power Attacks

The amount of power used by a digital circuit will vary depending on the data being processed. For example, the transistors that a flip-flop is composed of will draw different amounts of current depending on if the input is a logic 1 or 0. Simple Power Analysis (SPA) involves exploiting these power consumption variations in a cryptographic circuit during execution in order to gain insight on any internal secret information.

For example, Fig. 1.3 shows the power consumption of a cryptosystem during an encryption. Details of the time in which the circuit enters different algorithmic stages can be observed from variations in the power waveform.

Differential Power Analysis (DPA) [25] is the extension of SPA in which power traces recorded for multiple plaintext inputs are analyzed to recover the cryptographic key. First the adversary will require a power consumption model of the device to predict how much power it will consume based on the value of the data it is processing. For example the Hamming weight of the data is a common power consumption model used for CMOS-based devices. The power consumption for different key values for

each of the plaintext inputs is then predicted based on this model. The key value that generates predicted power consumption traces most closely correlating with the measured power traces throughout all of the different plaintext inputs is then determined to be the actual key being used in the device.

1.4.2 Electromagnetic Attacks

The electromagnetic radiation that an electronic device emits is directly correlated with its power consumption [43]. Thus, principles from power analysis can be applied also by analyzing the electromagnetic (EM) emissions. EM analysis (EMA), however, has the advantage of being completely non-invasive. This means that EM emissions from the circuit under attack can be recorded from a distance without any contact with the device. Power analysis on the other hand will require an intrusive method for measuring the device's power consumption such as inserting a resistor in series between the circuit's power supply and measuring the voltage difference across the resistor.

1.4.3 Acoustic Attacks

Computers often emit high-pitched noises while operating as a result of their electronic components vibrating. This noise is another side-channel leakage that can be exploited. Keys have been extracted from the sound generated by a computer during the decryption of chosen ciphertexts in under an hour [29]. This worked when the acoustic noises were recorded with a mobile phone placed beside the computer, or a sensitive microphone placed 10 meters away from the computer.

Figure 1.4: An electromagnetic attack using a consumer AM radio receiver placed near the targeted device and recorded by a smartphone [43].

Figure 1.5: Image of a microphone being used in an acoustic attack [29].

1.4.4 Timing Attacks

The amount of time it takes a cryptosystem to process different inputs can vary slightly. This can be due to the varying number of RAM cache hits/misses or arithmetic operations that take different amounts of time to operate for different input. This behaviour can be exploited in a timing attack. For example, the time t it takes for an algorithm to execute could be a function of the input p_1 and the key k_1. Assume that only a sub-step of the algorithm is timed in which an XOR is performed between p_1 and k_1, then t will be a function of $p_1 \oplus k_1$. The attacker will record t and also carry out experiments on an implementation in which the key value k_2 is known. The attacker then finds two values $k_2 \oplus p_2$ that take the same time to execute as $p_1 \oplus k_1$. It can then be concluded that $p_1 \oplus k_1 = p_2 \oplus k_2$ and $k_1 = p_1 \oplus p_2 \oplus k_2$.

1.4.5 Fault Attacks

A typical fault attack involves encrypting a chosen plaintext twice. One of the encryptions will be performed correctly and the other will have a faulty computation at some point during execution. Differences in the two resulting ciphertexts can be exploited to extract the key. A fault attack is considered an *active attack*, whereas the previously discussed attacks are considered *passive attacks*. Active attacks breach the security of a cryptosystem by altering its computation. Due to this, some sources will consider active attacks under a different category than side-channel attacks. For the purposes of this thesis, however, fault attacks *will* be considered under the category of side-channel attacks.

1.5 Motivation and Contributions

Physical implementations of cryptographic devices can leak unintentional information as a byproduct of them performing their computation. This information leaked through side-channels can expose clues to the device's secret information. This thesis focuses on fault-based attacks, a category of side-channel attacks, and contributes a new analysis method that improves upon the state-of-the-art in Differential Fault Analysis while relaxing the number of underlying assumptions related to fault injection. The motivation for this research is to bring to attention these discovered security flaws in embedded systems so that efforts can be initiated to mitigate these risks in the future in order to prevent potential malicious activity exploiting these vulnerabilities.

The rest of this thesis is organized as follows. Chapter 2 provides further background on fault attacks. Then the new Incremental Fault Analysis method is defined and detailed in chapter 3. This is followed by experimental validation in chapter 4. Concluding remarks are given in chapter 5.

Chapter 2

Background

The previous chapter introduced the use of cryptography in electronic devices and the security vulnerabilities introduced by their physical implementations. This chapter will provide the necessary background required to understand the Incremental Fault Analysis (IFA) fault attack method that this thesis proposes. First a literature review of fault attacks is provided as well as a description of the block cipher used to verify the effectiveness of IFA, the Advanced Encryption Standard (AES) [44].

2.1 Fault Attacks

To mount a fault attack, the adversary stresses a cryptographic circuit until one or more faults occur, and then leverages the faulty values that propagate to the output as a result. Examples of how the circuit might be stressed are by providing less power than the circuit was intended to operate with (undervolting), or increasing the internal clock frequency (overclocking). The fault can occur during undervolting or overclocking due to a *setup time violation* of one or more bits in the circuit's critical

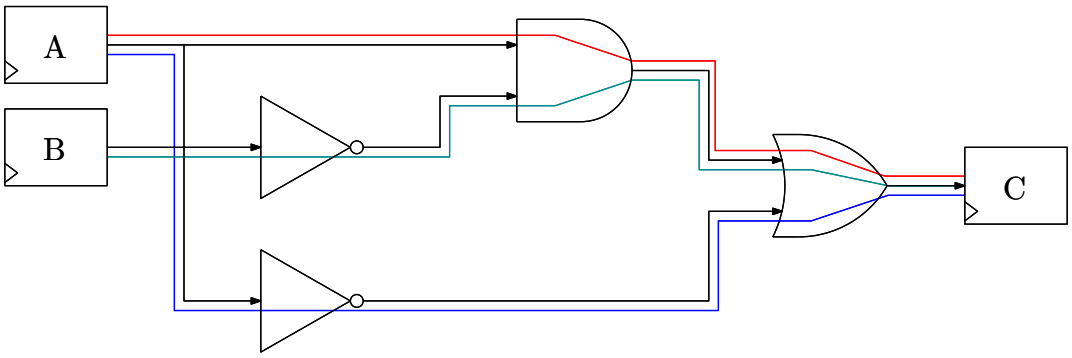

Figure 2.1: Visualization of signal propagation delay.

datapath. For example, observe Fig. 2.1 which shows a basic digital logic circuit. The output signals of registers A and B will take a certain amount of time to propagate to the input of register C. The amount of time they take to propagate must be less than the period of the clock controlling the registers. By overclocking the circuit, the signals now have a shorter window of time to propagate than they were designed for. Once the clock frequency is increased to the point in which there is no longer enough time for one or more of the signals to propagate across the datapath, a setup time violation occurs. The circuit might then produce a faulty computation called a **fault**.

In 1997, Boneh et al. [45] introduced the concept of leveraging hardware faults that occur during the execution of cryptographic devices in order to compromise their secret information. The paper describes an attack on RSA (Rivest–Shamir–Adleman), a widely used public-key algorithm. RSA is a commonly used cryptosystem and was one of the first public-key cryptography algorithms created. The security of RSA is based on the difficulty of factoring the product of two large prime numbers.

One of the uses of RSA other than for encryption is for authentication, as discussed in section 1.1.2. The purpose of authentication is related to proving the identity of a user, for example when requesting a money withdrawal from a bank. A user

can authenticate their identity through a method called RSA signing. This involves encrypting a piece of data using an entity's secret key such that the message can only be decrypted using their public key. This differs from RSA encryption, where the entity's public key is used to encrypt a message that can then only be decrypted by their private key. One portion of an RSA public key is a large number N that is the product of two prime numbers, p and q. An attacker would be able to deduce the secret key if they can factor N into these two prime factors. This is still computationally infeasible as of today, and thus RSA is still secure.

However, the authors in [45] presented a fault attack method that can be used to compromise an RSA implementation through its side-channel leakage. They showed that in order to reduce computational effort, some RSA signing implementations split up an operation that uses N into two separate operations with p and q and then join the two results. They describe a theoretical scenario where a fault corrupts only the p operation and they provide an analysis method that can leverage the resulting faulty output in order to deduce q and then p. They continue to provide more details on other variants of the attack and analysis methods targeting RSA.

The fault attack concept from [45] sparked a series of other fault attack methodologies on other cipher schemes. An attack labeled **Differential Fault Analysis** (**DFA**) [46] was introduced that targeted DES and other symmetric-key ciphers. DES is a symmetric-key algorithm developed in the early 1970's. It is now susceptible to brute-force attacks and is insecure, but was influential for the development of modern day symmetric-key algorithms such as the Advanced Encryption Standard (AES).

The term Differential Fault Analysis, also sometimes used interchangeably with **Differential Fault Attack**, has since been adopted as a general term for a fault

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])              // See Sec. 5.1.4

   for round = 1 step 1 to Nr-1
      SubBytes(state)                          // See Sec. 5.1.1
      ShiftRows(state)                         // See Sec. 5.1.2
      MixColumns(state)                        // See Sec. 5.1.3
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

Figure 2.2: Pseudo code for AES [44].

attack in which analysis is done on sets of two ciphertexts resulting from encryptions of the same plaintext where one of the plaintexts is correctly encrypted and the other is an erroneous result of a faulty computation. The two encryptions will have been executed identically up until the algorithmic stage in which the fault is injected. The differential output will then be a result of a reduced number of algorithmic stages. With this knowledge, there is also a reduced set of secret keys mathematically possible to be in use by the algorithm based on the differential output observed.

2.2 The Advanced Encryption Standard (AES)

AES is a symmetric-key block cipher that encrypts 128 bits of data at a time. The 128 bit data is visualized as a 4 by 4 byte array referred to as the **state**, shown in

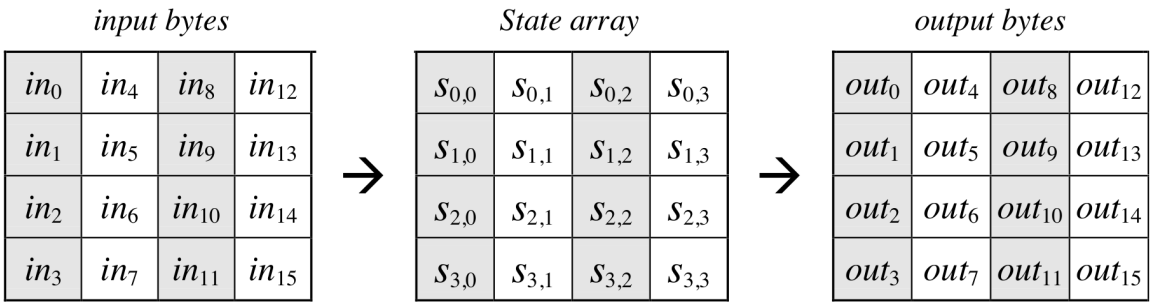

Figure 2.3: The AES state array [44].

Fig. 2.3. The state will consist of the plaintext at the start of encryption. The state will then go through various **rounds** of randomization before reaching the output as the encrypted ciphertext. Each round consists of a series of **transformations**, each described below.

- **SubBytes** (**SB**): SubBytes, shown in Fig. 2.4, is a non-linear byte substitution in which each byte of the state is mapped to a different value.
- **ShiftRows** (**SR**): ShiftRows, shown in Fig. 2.6, involves each byte of the state being cyclically shifted left by r positions, where r is the row index (starting at 0).
- **MixColumns** (**MC**): MixColumns, shown in Fig. 2.7, performs a matrix multiplication between the state and a matrix shown in Fig. 2.8.
- **AddRoundkey** (**ARK**): AddRoundKey, shown in Fig. 2.9, XORs each byte of the state with a corresponding byte of the round key.

AES [44] uses a 128, 192, or 256-bit key, called the **cipher key**. The number of 32-bit words comprising the cipher key is denoted as Nk, which will have the value 4, 6, or 8, respectively. In this thesis we consider all three AES variants. 10-14

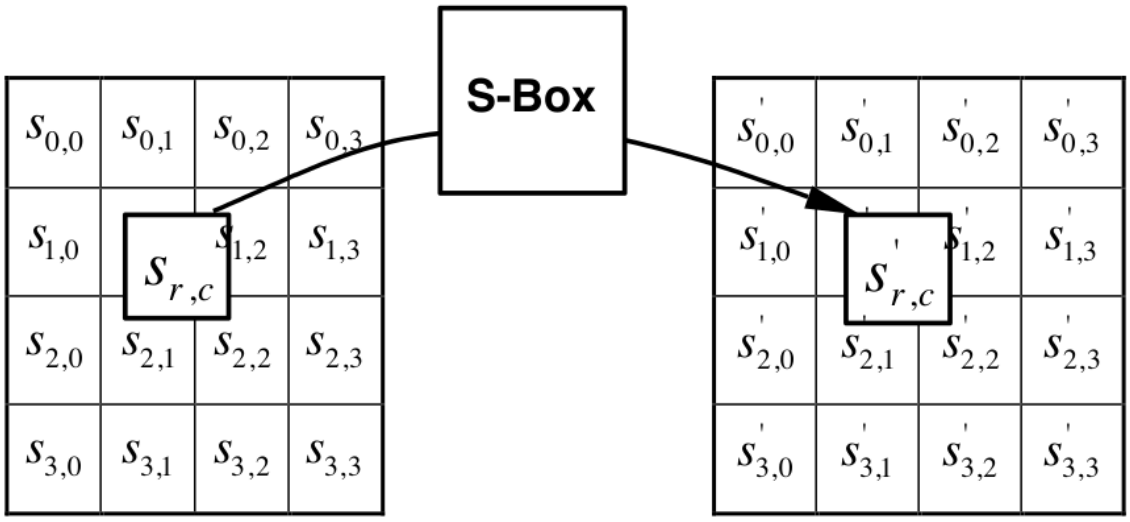

Figure 2.4: SubBytes (SB): A non-linear byte substitution that independently substitutes each byte of the state using a substitution table (S-box) [44].

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

Figure 2.5: S-box substitution values for byte xy in hexadecimal format [44].

rounds are executed depending on the value of Nk. The number of rounds executed is referred to as $\boldsymbol{Nr}$ and is equal to $Nk + 6$.

Prior to performing the cipher, AES expands the cipher key with an algorithm

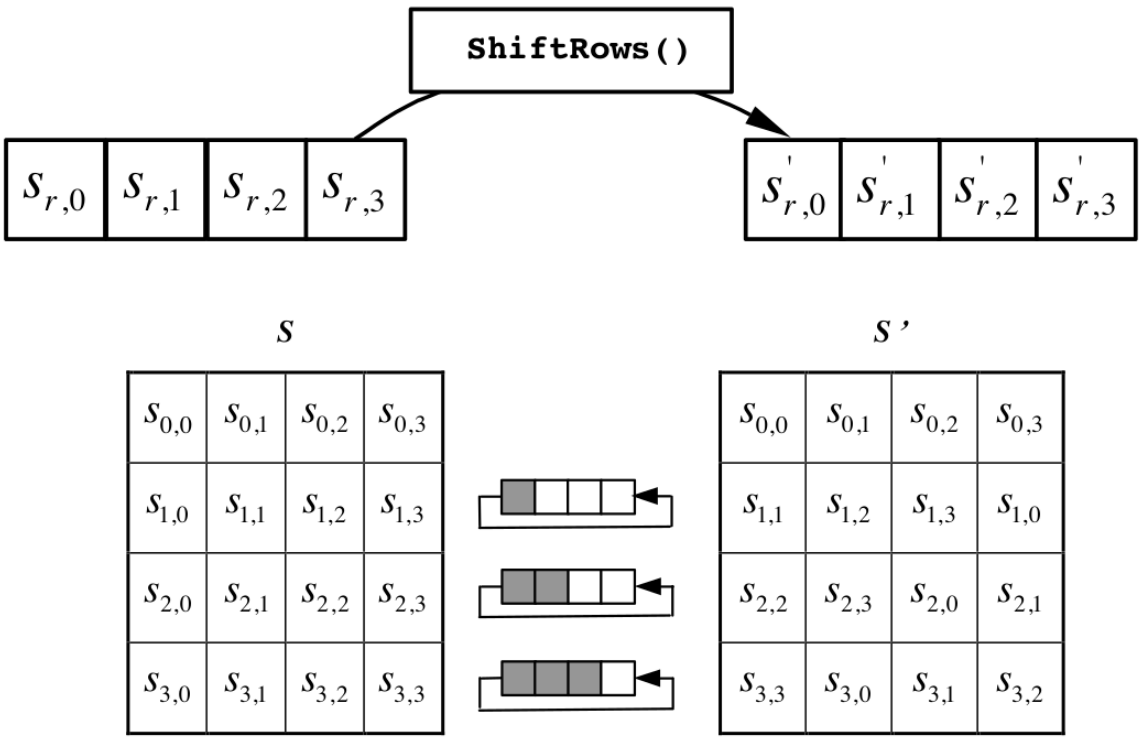

Figure 2.6: ShiftRows (SR): A cyclical shift applied to each row of the state. Each row r from 0 to 3 is shifted to the left by r bytes [44].

called **Key Expansion (KE)**, shown in Fig. 2.10. The key is expanded from $Nk * 4$ bytes to $(Nr + 1) * 16$ bytes and the expanded cipher key is referred to as the key schedule. The cipher key is initially XORed with the plaintext and then the remaining $Nr * 16$ bytes of the key schedule, w, is then divided into Nr 16-byte sets called **round keys**. A different round key is applied in each round during the AddRoundKey transformation, starting with the first round key in the first round until the last round key in the last round. The SubWord transformation in the Key Expansion algorithm is the same as the SubBytes transformation except applied to a column of the state. RotWord(i) shifts the bytes in a word by i positions. Rcon[i] is the round constant word array, containing the values [x^{i-1},{00},{00},{00}].

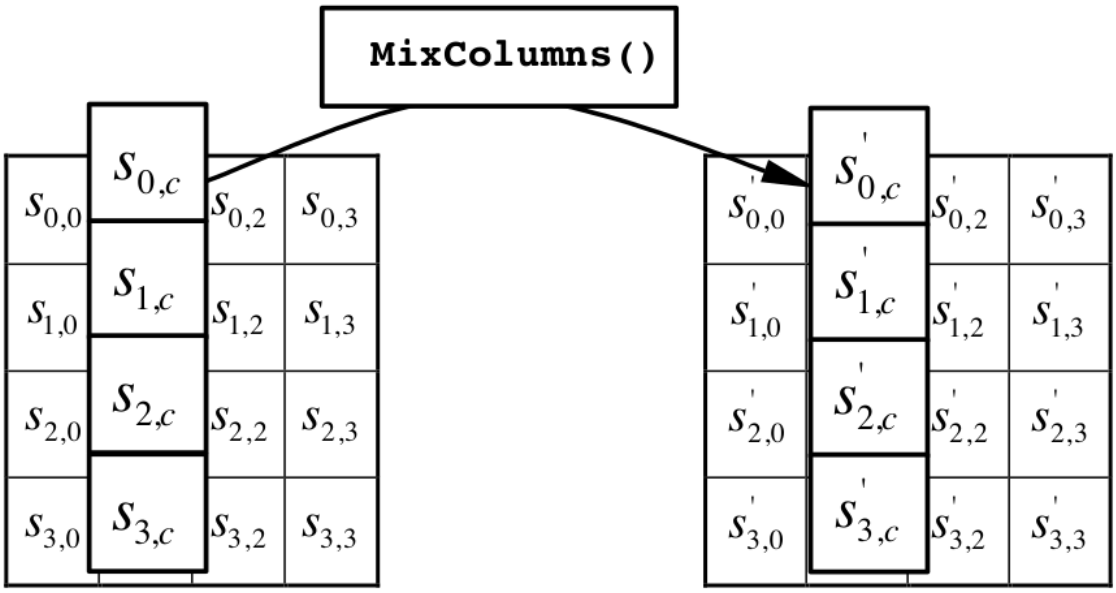

Figure 2.7: MixColumns(MC): A matrix multiplication is performed on the state where each column is considered as a 4-dimensional vector over GF(2^8) as in [44].

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \leq c < \boldsymbol{Nb}.$$

Figure 2.8: MixColumns(MC): A matrix multiplication is performed on the state where each column is considered as a 4-dimensional vector over GF(2^8) as in [44].

After Key Expansion, the state is initially XORed with the cipher key through the ARK operation, after which all Nr rounds are performed sequentially. The rounds each consist of the transforms mentioned above being applied once in the order they are listed, except for the last round which omits the MC transformation. An AES-128 cipher key can be reverse engineered from the final round key K^{Nr} by performing the

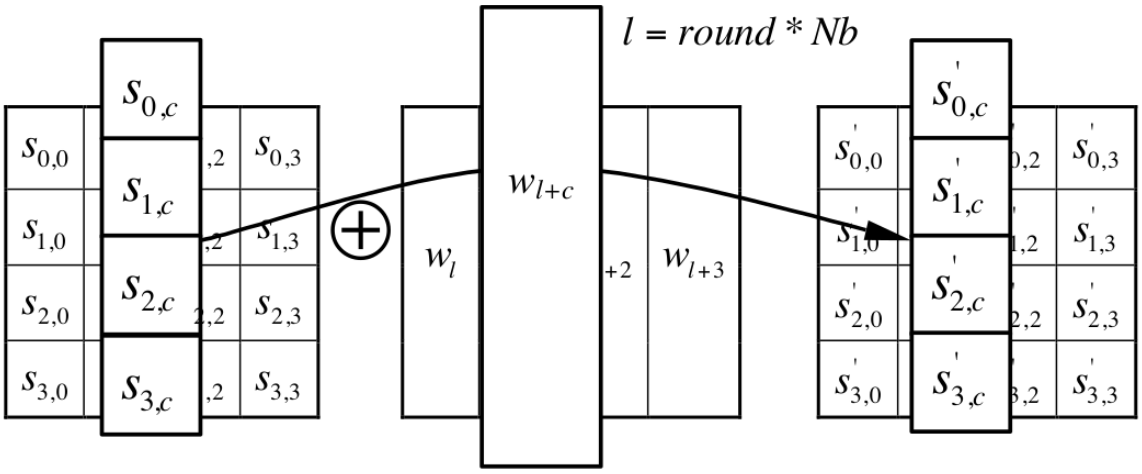

Figure 2.9: AddRoundKey (ARK): Each byte of the state is bitwise XORed with the corresponding round key. Each round key is generated from the cipher key [44].

KE in reverse. This is called **Inverse Key Expansion ($\boldsymbol{KE^{-1}}$)**, and is detailed in algorithm 1. To recover an AES-192 or AES-256 cipher key, the last two round keys are both required to perform KE^{-1}. This is because in AES-192 and AES-256, KE operates on 24 and 32-byte keys, respectively, but each round key is only 16-bytes. To perform decryption, the cipher operates similarly to encryption but transformations are performed in reverse order. The fault attacks that this thesis focuses on do not require an as in-depth understanding of decryption compared to encryption.

2.3 Fault Attacks on AES

There have been a variety of fault attacks published that target AES. In addition to exploiting faults, some also take advantage of other kinds of physical vulnerabilities observed in electronic devices. Some of these attacks are summarized below.

Fault Sensitivity Analysis (FSA) [38] exploits the fact that the level of stress

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end

Note that Nk=4, 6, and 8 do not all have to be implemented;
they are all included in the conditional statement above for
conciseness.   Specific implementation requirements for the
Cipher Key are presented in Sec. 6.1.
```

Figure 2.10: Pseudo code for Key Expansion [44].

required to cause a fault in a cryptosystem depends on the specific value of its internal state at the time of fault injection. For example, in the case of overclocking, the amount of time register outputs have to propagate to their destination register inputs is reduced by increasing the internal clock frequency of the circuit. However, the time required to allow signals to propagate across a datapath depends on the input values to the logic elements in the datapath. Therefore, the amount of overclocking

Algorithm 1 Inverse_Key_Expansion

```
    Input: Last round key
    Ouput: The AES cipher key
 1: function INVERSE_KEY_EXPANSION(word key[Nk], word w[4 * (Nr + 1)])
 2:     word tmp
 3:     for i = Nr * 4 + 3; i >= Nk; i- - do
 4:         tmp= w[4 * (i − 1)]
 5:         if i%Nk == 0 then
 6:             rot_word(tmp);
 7:             sub_word(tmp);
 8:             tmp = tmp ^ Rcon(i/Nk)
 9:         else if (Nk > 6) && (i%Nk == 4)) then
10:             sub_word(tmp)
11:         end if
12:         w[4 * (i − Nk)] = w[4 * i]^tmp
13:     end for
14:     for i = 0; i < Nk * 4; i++ do
15:         key[i] = w[i];
16:     end for
17:     Return key
18: end function
```

necessary to cause a fault will depend on the value of the internal data. FSA takes advantage of this behaviour by recording the level of overclocking required to cause a fault in a specific stage of the encryption. This is then correlated with what the key values being used internally must be to require that specific amount of overclocking to cause a fault. FSA requires an idea of the gate-level design of the cryptosystem in order to have an estimate of the design timing. It also requires that the fault is injected using a clock glitch in the last AES round. This method was successful with less than 50 fault injections and the fault value does not need to be known.

Differential Fault Intensity Analysis (DFIA) [40] takes advantage of faults being biased towards affecting only 1 - 3 bits in a byte. DFIA exploits this property by partially decrypting each faulty/correct ciphertext pair back to the stage of fault

injection using each possible sub-key hypothesis. Then the adversary records the hypothetical faults that must have occurred assuming that the hypothesis sub-key is the actual sub-key being used in the cryptosystem. The only sub-key hypotheses that are deemed possible are those which point to fault values affecting only 1-3 bits. Fault injections must be performed in the last round of AES and are injected using clock glitches. The method was successful after 7 fault injections per byte of the AES state.

Differential Error Rate Analysis (DERA) [41] uses the inherent bias of the error rates among different signals. Certain bits on the input of combinational logic will be connected to longer datapaths with more gates to pass through, and therefore will require a larger amount of time to propagate to their destination compared to other bits. This means that when a circuit is overclocked, certain bits of a datapath will tend to be corrupted more frequently than others. DERA exploits this property by partially decrypting each faulty/correct ciphertext pair back to the stage of fault injection using each possible sub-key hypothesis. Then the adversary records the hypothetical faults that must have occurred assuming that the hypothesis sub-key is the actual sub-key being used in the cryptosystem. Correct key guesses tend to return hypothetical faults with corrupted bit occurrence rates biased towards affecting certain bits and not others, whereas incorrect key guesses tend to return hypothetical faults with corrupted bit occurrence rates that are distributed more evenly across all bits. DERA uses a clock glitch fault injection method in the second-last AES round. This method was successful after 70 fault injections.

Non-Uniform Faulty Value Analysis (NUFVA) [39] assumes the value of the corrupted state after fault injection is non-uniformly distributed. Note that this is different than DFIA and DERA which assume that the differential between the faulty and non-faulty states is non-uniformly distributed. Most fault attack methods (including DFIA and DERA) assume the fault injection is modelled by an XOR, which is a linear operation. If the state values in which the faulty values are applied to are unbiased, then they will still be unbiased after a linear operation with a (biased or unbiased) faulty value. NUFVA assumes that the fault injection is modelled by a non-linear operation such as an AND or OR operation, thus assuming the corrupted state is biased after fault injection. This property has been shown to hold for some types of S-boxes under stress [47]. This method has one powerful advantage over the previously mentioned methods in which it is not a chosen-plaintext attack. This means that the same plaintext does not have to be re-applied. The method in [39] can determine the cipher key with 6 faults. However, the fault model used for this is particularly impractical to implement in practise. A more practical version of the fault model in [39] is used in [48] in which it is assumed the corrupted datapath has some less specific form of bias. Using that fault model, the cipher key is extracted from as little as 80 fault injections.

2.4 A Practical Fault Attack on AES

The attacks discussed in section 2.3 overall fall short in practicality because the adversary is required to know with high certainty that a target fault will occur during each fault injection attempt. Ensuring this means the faults must occur within a specific AES round, implying that the circuit's internal clock frequency must be briefly increased for only a certain small number of clock cycles. This introduces extra attack measures, requiring clock glitching equipment such as clock multiplexers that can briefly increase the clock frequency, and a method for probing the cryptosystem's internal state to know during which clock cycles the clock frequency must be increased.

However, in a realistic attack scenario the adversary might not have such control over the targeted device. A more flexible fault attack on AES was introduced in 2003 by Piret et al. [36]. One of the greatest advantages of this attack is that it can be used in a practical scenario in which the attacker is not required to distinguish between which faulty ciphertexts are resulting from target faults and which are resulting from non-target faults. Rather than requiring a target fault to occur during every fault injection attempt, it is only required for a small percentage of fault injection attempts to result in target faults (discussed more in section 2.4.4). The adversary can simply perform a certain number of encryptions using a cryptosystem kept at a constant stress level, then analyze all of the faulty ciphertext outputs as a whole while being otherwise completely blind to the details of the faults that are occurring internally. Since the circuit does not have to be stressed for a specific brief set of clock cycles, it is not required to have methods for probing the cryptosystem's internal state to know when it is in a target round, or to integrate clock glitching equipment into the circuit's internal clock. This practicality is something that has been overlooked in

recent work, but is essential for realistic attack scenarios.

2.4.1 Attack Procedure

The attack procedure explained in this section hinges on an understanding of AES, described in section 2.2. The attack method requires injection of randomly valued faults affecting up to a single-byte near the end of the AES cipher. Methods that expand upon this one can extract the cipher key with as little as a single fault injection [37].

First let the following notation and representations be defined:

- **SB, SR, MC, ARK**: The transformations that AES consists of, as explained in section 2.2.
- ***Nr***: The number of AES rounds, as explained in section 2.2.
- **Superscript *Nr−d***: X^{Nr-d} represents an element or transformation X in round $Nr-d$, where $d = \{0, ..., Nr-1\}$. For example, the MC transformation in round $Nr-1$ will be referred to as MC^{Nr-1}.
- **Subscript *i,j***: $X_{i,j}$ represents the byte of an element or transformation X corresponding to row i and column j of the AES state.
- ***S***: S may be used to represent the AES state at the input of a certain round and/or at a certain byte index. For example, $S^{Nr}_{1,2}$ represents the byte in row 1 and column 2 of the state at the input of round Nr.
- ***K***: The AES cipher key.

- $\boldsymbol{K^{Nr-d}}$: The $(Nr-d)^{th}$ AES round key, where $d = \{0, ..., Nr-1\}$. For example, K^{Nr} is the Nr^{th} round key and will be referred to as the last round key.
- $\boldsymbol{C}$: A ciphertext output from AES encryption.
- $\boldsymbol{C^{\lightning}}$: A faulty ciphertext output from AES encryption. This may also sometimes be written as C'.
- $\boldsymbol{\Delta}$: The **differential**, or difference, between the faulty and fault-free states throughout various stages of encryption due to a fault or fault propagation.

Now Consider Fig. 2.11 where a single-byte fault occurs at some point between MC^{Nr-2} and MC^{Nr-1}. The effect of a fault is modelled by an XOR between the state and the fault value Δ. After fault injection, the differential must initially affect 1 byte of the state. It will then spread to 4 bytes of the state after MC^{Nr-1}. Next, the SB transformation will change the value of the differential because it is a non-linear transformation and will cause the faulty and fault-free states to differ by a different amount. Lastly, the SR transformation will shuffle the positioning of the differential's bytes in the state. Note that only the MC, SB, and SR transformations change the value or positioning of the differential. The ARK does not affect the value (or positioning) of the differential because it is a linear transformation and changes the value of both the faulty and fault-free states by the same amount, thus not affecting the differential between them.

The state at the input of round Nr and the ciphertext outputs are related to each other through the transformations SB and ARK of round Nr. The state at the input of round Nr can be derived from the ciphertext by partially decrypting the ciphertext back by 1 round. For example, the fault-free state in row 0 and column 0

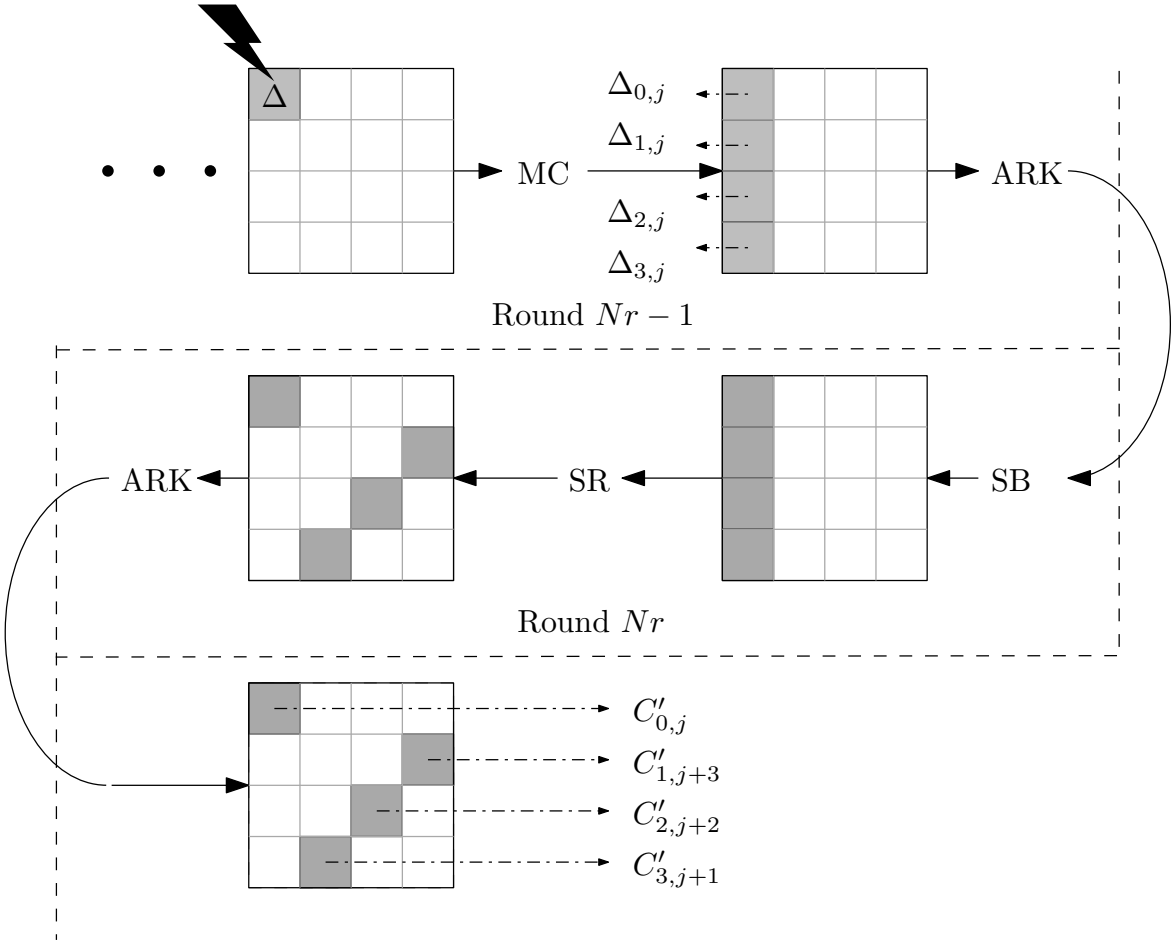

Figure 2.11: Propagation of a single-byte fault injected between MC^{Nr-2} and MC^{Nr-1}.

at the input of round Nr relates to the fault-free ciphertext byte in row 0 and column 0 as follows:

$$S_{0,0}^{Nr} = SB^{-1}(ARK^{Nr}(C_{0,0})) = SB^{-1}(C_{0,0} \oplus K_{0,0}^{Nr})$$

where SB^{-1} is the inverse of the SB transformation (the ARK inverse is not used because it is its own inverse).

In the same way, the following relationships can be derived between the 4-byte

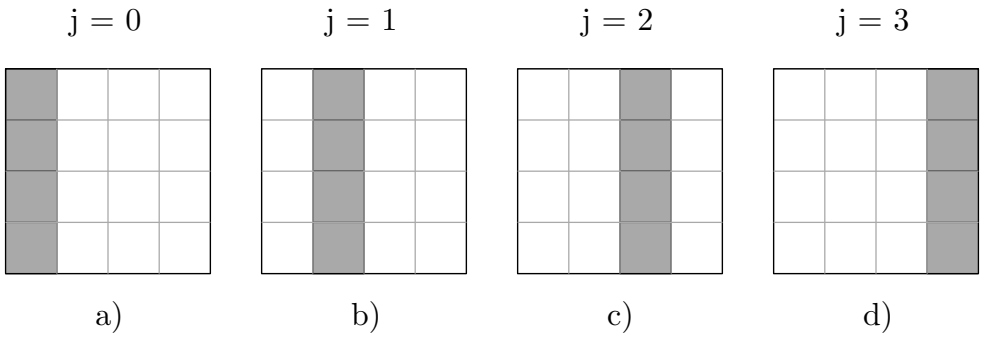

Figure 2.12: State positions for the 4 bytes composing a column j sublist of a 16-byte element or transformation depending on the value of j, where $j = \{0, 1, 2, 3\}$.

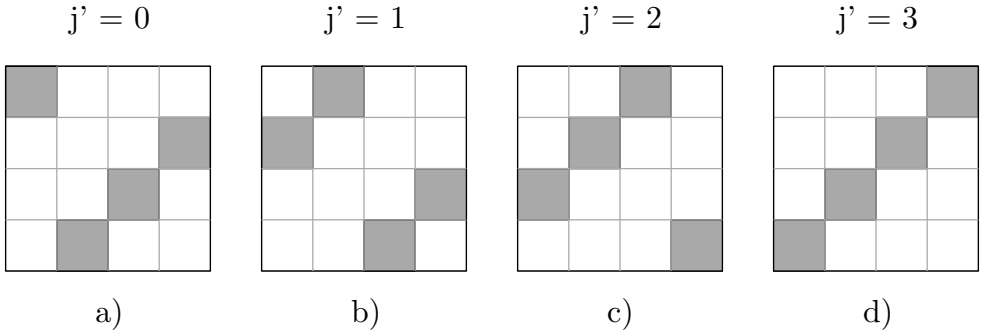

Figure 2.13: State positions for the 4 bytes composing a j' sublist of a 16-byte element or transformation depending on the value of j', where $j' = \{0, 1, 2, 3\}$.

differential at the output of MC^{Nr-1}, and the 4 output ciphertext bytes in which the differential propagates to:

$$\Delta_{0,j} = SB^{-1}(C_{0,j} \oplus K^{Nr}_{0,j}) \oplus SB^{-1}(C^{\lightning}_{0,j} \oplus K^{Nr}_{0,j}) \tag{2.1}$$

$$\Delta_{1,j} = SB^{-1}(C_{1,j+3} \oplus K^{Nr}_{1,j+3}) \oplus SB^{-1}(C^{\lightning}_{1,j+3} \oplus K^{Nr}_{1,j+3}) \tag{2.2}$$

$$\Delta_{2,j} = SB^{-1}(C_{2,j+2} \oplus K^{Nr}_{2,j+2}) \oplus SB^{-1}(C^{\lightning}_{2,j+2} \oplus K^{Nr}_{2,j+2}) \tag{2.3}$$

$$\Delta_{3,j} = SB^{-1}(C_{3,j+1} \oplus K^{Nr}_{3,j+1}) \oplus SB^{-1}(C^{\lightning}_{3,j+1} \oplus K^{Nr}_{3,j+1}) \tag{2.4}$$

In order to compress (2.1) - (2.4) into a more compact representation, let the

following notation be defined:

- **Subscript j**: X_j represents a certain 4-byte sublist of a 16-byte element or transformation X. The 4 bytes of X that are included in each j sublist correspond to the *column* of the element or transformation, as shown in Fig. 2.12 a) - d).

- **Subscript j'**: $X_{j'}$ represents a certain 4-byte sublist of a 16-byte element or transformation X. The 4 bytes of X that are included in each j' sublist are illustrated in Fig. 2.13 a) - d). The positions correspond to the bytes that a single-byte fault injected between MC^{Nr-2} and MC^{Nr-1} will propagate to on the output ciphertext depending on the column j in which the fault was injected into.

Using this notation, (2.1) - (2.4) can be compressed to the following:

$$\Delta_j = SB_{j'}^{-1}(C_{j'} \oplus K_{j'}^{Nr}) \oplus SB_{j'}^{-1}(C_{j'}^{\lightning} \oplus K_{j'}^{Nr}) \tag{2.5}$$

where:

$$\begin{aligned}
\Delta_j &= \{\Delta_{0,j}, \Delta_{1,j}, \Delta_{2,j}, \Delta_{3,j}\} \\
K_{j'}^{Nr} &= \{K_{0,j}^{Nr}, K_{1,j+3}^{Nr}, K_{2,j+2}^{Nr}, K_{3,j+1}^{Nr}\} \\
C_{j'} &= \{C_{0,j}, C_{1,j+3}, C_{2,j+2}, C_{3,j+1}\} \\
C_{j'}^{\lightning} &= \{C_{0,j}^{\lightning}, C_{1,j+3}^{\lightning}, C_{2,j+2}^{\lightning}, C_{3,j+1}^{\lightning}\} \\
j &= \{0, 1, 2, 3\}
\end{aligned}$$

Next, consider that there are 255 possible values for a single-byte differential Δ at

the point of fault injection. The possible values are 1 to 255 (0 is excluded since that would mean no fault occurred). The MC transformation operates on one column of the state at a time, and a single-byte differential could enter a MC transformation on any of 4 possible row positions. This means that there are $255 \cdot 4 = 1020$ possible unique single-byte differential inputs to a MC transformation. Therefore, even though the MC transformation spreads the single-byte differential across 4 bytes, there are only 1020 unique 4-byte differentials possible at the output of a MC that can be the result of a single-byte differential input. These 1020 possible 4-byte differentials form a list we will refer to as D. This D list must be computed before the attack and will always be the same, and can then be re-used for each application of the attack.

For example, D can be computed as follows:

$$D_0 = MC(\ \{01, 00, 00, 00\}\ ) = \{02, 01, 01, 03\}$$

$$D_1 = MC(\ \{02, 00, 00, 00\}\ ) = \{04, 02, 02, 06\}$$

$$D_2 = MC(\ \{03, 00, 00, 00\}\ ) = \{06, 03, 03, 09\}$$

...

$$D_{254} = MC(\ \{\text{ff}, 00, 00, 00\}\ ) = \{\text{e5}, \text{ff}, \text{ff}, \text{1a}\}$$

$$D_{255} = MC(\ \{00, 01, 00, 00\}\ ) = \{03, 02, 01, 01\}$$

$$D_{256} = MC(\ \{00, 02, 00, 00\}\ ) = \{06, 04, 02, 02\}$$

...

$$D_{509} = MC(\ \{00, \text{ff}, 00, 00\}\ ) = \{\text{1a}, \text{e5}, \text{ff}, \text{ff}\}$$

$$D_{510} = MC(\ \{00, 00, 01, 00\}\ ) = \{01, 03, 02, 01\}$$

$$D_{511} = MC(\ \{00, 00, 02, 00\}\ ) = \{02, 06, 04, 02\}$$

...

$$D_{764} = MC(\ \{00, 00, \text{ff}, 00\}\ ) = \{\text{ff}, \text{1a}, \text{e5}, \text{ff}\}$$

$$D_{765} = MC(\ \{00, 00, 00, 01\}\ ) = \{01, 01, 03, 02\}$$

$$D_{766} = MC(\ \{00, 00, 00, 02\}\ ) = \{02, 02, 06, 04\}$$

...

$$D_{1019} = MC(\ \{00, 00, 00, \text{ff}\}\ ) = \{\text{ff}, \text{ff}, \text{1a}, \text{e5}\}$$

The goal of the adversary is to find the value of the actual last round key sublist $K_{j'}^{Nr}$ being used to encrypt the plaintexts. However, they only have the faulty and fault-free ciphertext outputs available to them. $K_{j'}^{Nr}$ could be any of 2^{32} possible

values. However, it is known that Δ_j is an element of D, and that there are only 1020 possible values in D. To exploit this, $C_{j'}$ and $C_{j'}^{\lightning}$ are applied to (2.5) for each of the 2^{32} possible values for $K_{j'}^{Nr}$ to see which $K_{j'}^{Nr}$ values map to a Δ_j that is an element of D. On average this occurs for 1036 $K_{j'}^{Nr}$ values. The adversary has now reduced the number of possible hypotheses for $K_{j'}^{Nr}$ from 2^{32} to 1036.

The full last round key consists of 4 $K_{j'}^{Nr}$ elements, and this analysis must be done 4 times to retrieve the full last round key, once for each $j = \{0, 1, 2, 3\}$. For each j, analysis must be done on ciphertexts resulting from a single-byte fault affecting column j of the state between MC^{Nr-2} and MC^{Nr-1}. For each column j that a single-byte fault is injected into, the differential will propagate to the j' byte positions on the output ciphertext as illustrated in Fig. 2.13 a) - d).

After analysis for each $j = \{0, 1, 2, 3\}$, the adversary will have a list of on average 1036 (or $\sim 2^{10}$) hypotheses for each $K_{j'}^{Nr}$ value. The list of hypotheses for the full last round key K^{Nr} can be formed by concatenating all combinations of the 4 $K_{j'}^{Nr}$ hypotheses lists to form a list of $\sim 2^{40}$ hypotheses for the full last round key K^{Nr}. The adversary has now reduced the number of possible values for K^{Nr} from 2^{128} to $\sim 2^{40}$.

If this entire process is repeated again, then the adversary will possess 2 lists of $\sim 2^{40}$ K^{Nr} hypotheses. Each incorrect K^{Nr} hypothesis in these lists will be a random value amongst 2^{128} possible K^{Nr} values. Due to this enormous space, the likelihood of any incorrect elements in one list overlapping with the other is negligibly low, and the only overlapping element in each list will be the correct K^{Nr} value.

The overlapping last round key hypothesis K^{Nr} can be used to generate a cipher key hypothesis K by performing an inverse of the Key Expansion algorithm as described in Algorithm 1. In order to confirm if the resulting K hypothesis is the

correct one being used in the cipher, a plaintext can be encrypted using a custom AES implementation which will use the K hypothesis for encryption. If the resulting ciphertext is identical to the fault-free ciphertext from the cryptosystem targeted in the attack, then it is known that this is the correct K being used in the cryptosystem under attack.

2.4.2 Example

To better understand the relationship exploited by (2.5), consider the two encryptions in Fig. 2.14 which are identical except that the one on the left is a fault-free encryption, and the one on the right contains a fault occurring between MC^{Nr-2} and MC^{Nr-1}. The propagation of the fault is shown in detail after every transformation up until it reaches the ciphertext output.

$$\Delta_j = SB_{j'}^{-1}(C_{j'} \oplus K_{j'}^{Nr}) \oplus SB_{j'}^{-1}(C_{j'}^{\lightning} \oplus K_{j'}^{Nr}) \qquad \text{(2.5 revisited)}$$

In this example, the parameters in (2.5) (revisited above) would contain the following values:

$$\Delta_j = \{03, 09, 06, 03\}$$
$$K_{j'}^{Nr} = \{d0, 63, 0c, 89\}$$
$$C_{j'} = \{39, 6a, 85, fb\}$$
$$C_{j'}^{\lightning} = \{4b, 0d, b3, 2d\}$$
$$j = 0$$

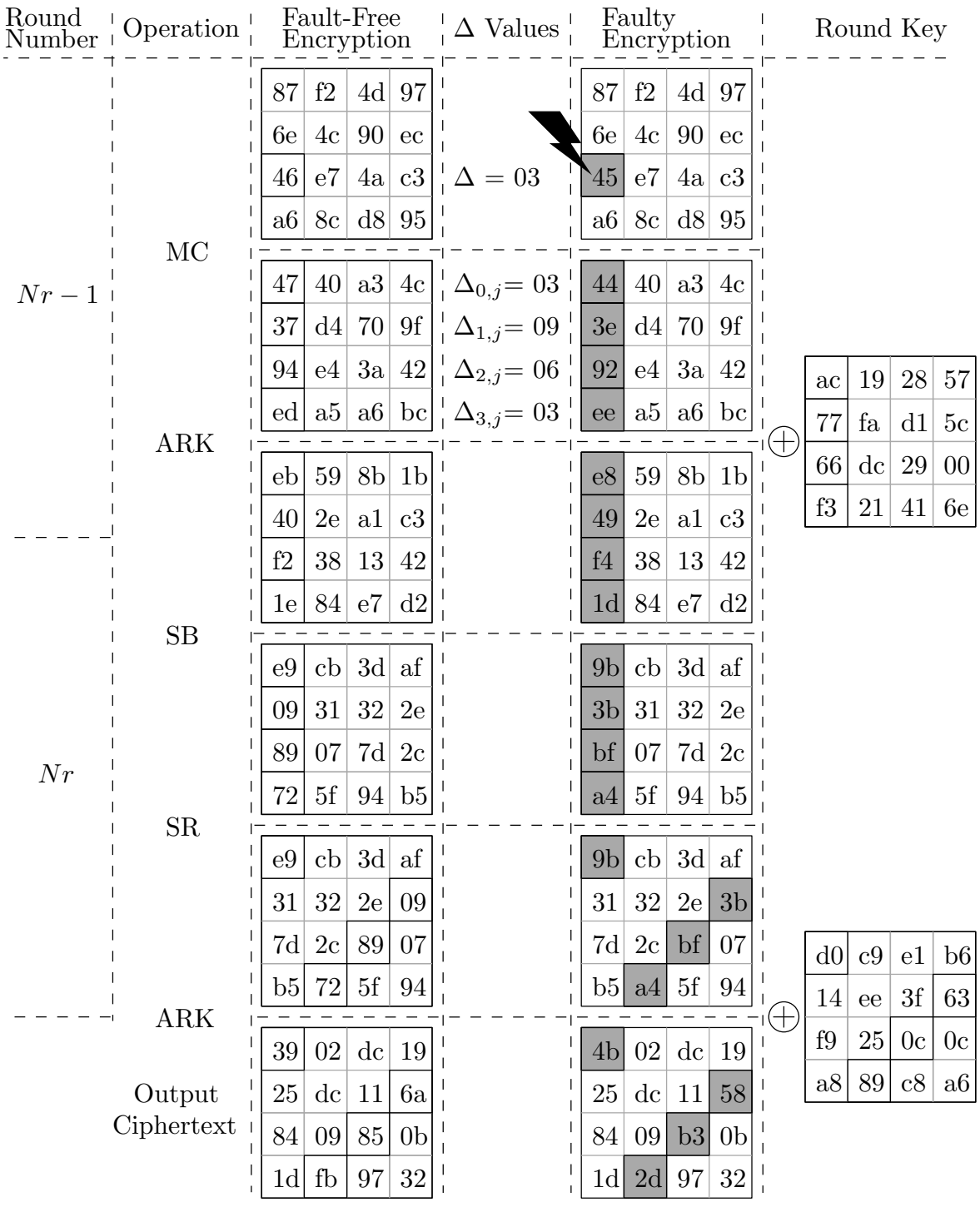

Figure 2.14: Propagation of a single-byte fault injected between MC^{Nr-2} and MC^{Nr-1}.

2.4.3 An Improvement to the Attack

The number of fault injections required to mount the attack is reduced by considering a random single-byte fault injected between MC^{Nr-3} and MC^{Nr-2} (1 round earlier than before), as shown on the right of Fig. 2.15. The encryption on the left of Fig. 2.15 shows also a single-byte fault injected between MC^{Nr-2} and MC^{Nr-1} for reference. The encryption on the right will automatically set up one single-byte differential in each column between MC^{Nr-2} and MC^{Nr-1}. Analysis can then be done for all j values from just a single faulty and fault-free ciphertext pair, reducing the number of fault injections required to just 1 or 2 in order to reduce the number of cipher key hypotheses to $\sim 2^{40}$ or 1.

Furthermore, in 2011 Tunstall et al. [37] exploited further AES relationships to show that the number of last round key hypotheses returned by this attack based on a single ciphertext pair can actually be reduced from $\sim 2^{40}$ to a mere 2^8 in the most ideal scenario.

2.4.4 Non-Target Fault Tolerance

As previously mentioned, this method can still be used even if the attacker has no way of distinguishing between which faulty ciphertexts resulted from target or non-target faults. The attack will return a solution after two or more target faults occurring between MC^{Nr-3} and MC^{Nr-2} occur, or after two or more target faults occurring between MC^{Nr-2} and MC^{Nr-1} occur for each column of the state.

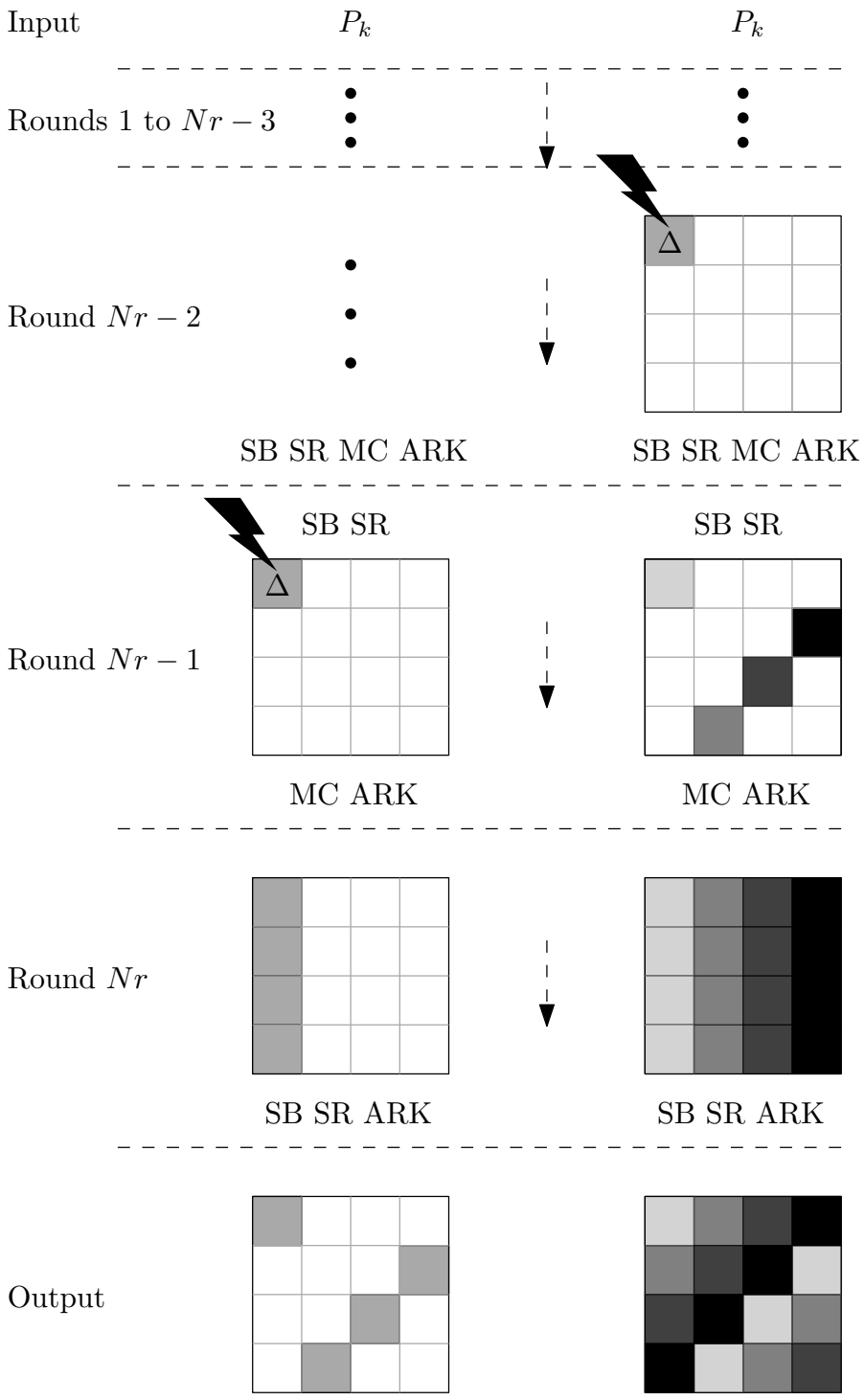

Figure 2.15: Propagation of a single-byte fault injected between MC^{Nr-2} to MC^{Nr-1} and MC^{Nr-3} to MC^{Nr-2}.

Identifying Target Faults Between MC^{Nr-2} and MC^{Nr-1}

If the adversary is targeting faults that occur between MC^{Nr-2} and MC^{Nr-1}, then non-target faults can be filtered out by only accepting faulty ciphertexts that have fault propagation on 4 bytes in the j' positions for one of $j = \{0, 1, 2, 3\}$.

To further filter the remaining faulty ciphertexts, let $\mathbb{K}_{j'}^{Nr}$ represent a list of all the ~1036 $K_{j'}^{Nr}$ values returned by analyzing a ciphertext pair for a certain $j = \{0, 1, 2, 3\}$. Each ciphertext pair analyzed will return a unique $\mathbb{K}_{j'}^{Nr}$ list. $\mathbb{K}_{j'}^{Nr}$ lists corresponding to non-target ciphertext pairs will contain ~1036 incorrect $K_{j'}^{Nr}$ hypotheses distributed sparsely across the large space of 2^{32} possible values that each element can take on. This sparsity ensures that the likelihood of one or more of the elements overlapping in two incorrect $\mathbb{K}_{j'}^{Nr}$ lists is reasonably low.

However, $\mathbb{K}_{j'}^{Nr}$ lists returned from two *target* ciphertext pairs will each contain one element that is the correct $K_{j'}^{Nr}$ hypothesis which therefore *must* overlap in the two target $\mathbb{K}_{j'}^{Nr}$ lists. Thus, the adversary can continue comparing sets of two ciphertext pairs until they happen to compare two *target* ciphertext pairs, after which a solution will be returned.

Identifying Target Faults Between MC^{Nr-3} and MC^{Nr-2}

If the adversary is targeting faults that occur between MC^{Nr-3} and MC^{Nr-2}, then non-target faults can be filtered out as follows. Let $\mathbb{K}^{Nr}$ represent a list of all the ~2^{40} K^{Nr} values returned by analyzing a ciphertext pair. Each ciphertext pair analyzed will return a unique $\mathbb{K}^{Nr}$ list. $\mathbb{K}^{Nr}$ lists corresponding to non-target ciphertext pairs will contain ~2^{40} incorrect K^{Nr} hypotheses distributed extremely sparsely across the enormous space of 2^{128} possible values that each element can take on. This sparsity

ensures that the likelihood of one or more of the elements overlapping in two incorrect $\mathbb{K}^{Nr}$ lists is negligibly low.

However, $\mathbb{K}^{Nr}$ lists returned from two *target* ciphertext pairs will each contain one element that is the correct K^{Nr} hypothesis which therefore *must* overlap in the two target $\mathbb{K}^{Nr}$ lists. Thus, the adversary can continue comparing sets of two ciphertext pairs until they happen to compare two *target* ciphertext pairs, after which a solution will be returned.

2.4.5 Extension to AES-256 and AES-192

In AES-128, the cipher key can be recovered once the last round key is known. In AES-256 and 192, however, the last *two* round keys, K^{Nr} and K^{Nr-1}, must be known in order to recover the cipher key. This is achieved by first applying the attack described above to obtain the last round key K^{Nr}. Faults are then injected in the same fashion, but one round earlier, between MC^{Nr-3} and MC^{Nr-4}. Each ciphertext pair (C, $C^{\lightning}$) can then be partially decrypted using K^{Nr} to compute a corresponding partially decrypted ciphertext pair (A, $A^{\lightning}$) equivalent to the state before the input to the last MC:

$$A = MC^{-1}(SR^{-1}(SB^{-1}(C \oplus K^{Nr}))) \tag{2.6}$$

$$A^{\lightning} = MC^{-1}(SR^{-1}(SB^{-1}(C^{\lightning} \oplus K^{Nr}))) \tag{2.7}$$

Then the same attack as above is applied except on (A, $A^{\lightning}$) rather than (C, $C^{\lightning}$) to recover $MC^{-1}(K^{Nr-1})$ as shown in (2.8) and 2.9. This can then be transformed into

K^{Nr-1} using MC as shown in (2.10).

$$B = MC^{-1}(K^{Nr-1}) \tag{2.8}$$

$$\Delta_j = SB_{j'}^{-1}(A_{j'} \oplus B_{j'}) \oplus SB_{j'}^{-1}(A_{j'}^{\lightning} \oplus B_{j'}) \tag{2.9}$$

$$K^{Nr-1} = MC(B) \tag{2.10}$$

Then K^{Nr} and K^{Nr-1} can be used by the Inverse Key Expansion algorithm to generate the cipher key similarly to how it is done for AES-128.

2.5 Summary

This chapter outlined different examples of fault attacks and summarized their strengths and weaknesses. An overview of AES was provided, and a detailed description and example were provided for a practical fault attack that targets AES [36]. The next chapter goes on to describe our improved IFA fault analysis method and demonstrates the improvements it can provide in comparison to the attack from [36].

Chapter 3

Incremental Fault Analysis

Incremental Fault Analysis (IFA) generalizes classical differential fault models from requiring a target fault to occur in isolation during an encryption to also allowing the target fault to occur *incrementally* between any two faulty encryptions. This allows the key to be computed from fewer ciphertexts and introduces a tolerance for faults occurring prior to the target fault injection round.

For the application of IFA explored in this thesis, faults are injected through overclocking. The circuit is overclocked at a steady frequency throughout the entire encryption. The frequency is then incrementally increased and the plaintext is encrypted again. This process is repeated multiple times and the resulting ciphertexts are collected after each encryption. Eventually faults will start to occur internally and propagate to the output. More faults will occur as the circuit is further overclocked. Three key observations were found experimentally when overclocking:

- Faults often affect one byte of the state at a time
- Faults occurring at clock frequency f^m will often continue to occur also at

frequencies greater than f^m

- Faults occurring at higher frequencies are often superimposed upon any fault propagation resulting from faults occurring at lower frequencies in previous algorithmic stages

Consider Fig. 3.1, where a circuit is overclocked from frequency f^0 to f^n. As outlined above, it was found based on experimental observation that a fault Δ^m appearing in a round $Nr - d$ at frequency f^m will often continue to occur for all frequencies greater than f^m as well. Now consider another fault Δ^n occurs in a later round $Nr - 1$ at frequency f^n, then the faulty ciphertext output C_k^n will contain fault propagation resulting from both Δ^m *and* Δ^n. Even though Δ^n *in isolation* is a target fault, Δ^m is not and corrupts any cipher key information that could have been extracted from the ciphertext pair (C_k^0, C_k^n).

However, IFA can still extract cipher key information in this scenario by analyzing the **incremental differential** between the two faulty ciphertexts (C_k^m, C_k^n). In the example shown in 3.1, the notation Δ^{mn} would be used to represent the **fault increment** occurring between the two encryptions producing ciphertexts C_k^m and C_k^n, and in this case is equivalent to Δ^n occurring in isolation.

3.1 Attack Procedure

In this section we explain how IFA can be applied to and enhance the practical attack method described in section 2.4 in order to reduce the number of fault injections and analysis time required to uncover the secret key. We also describe an improved algorithmic approach we developed for analyzing the data. The explanations in this

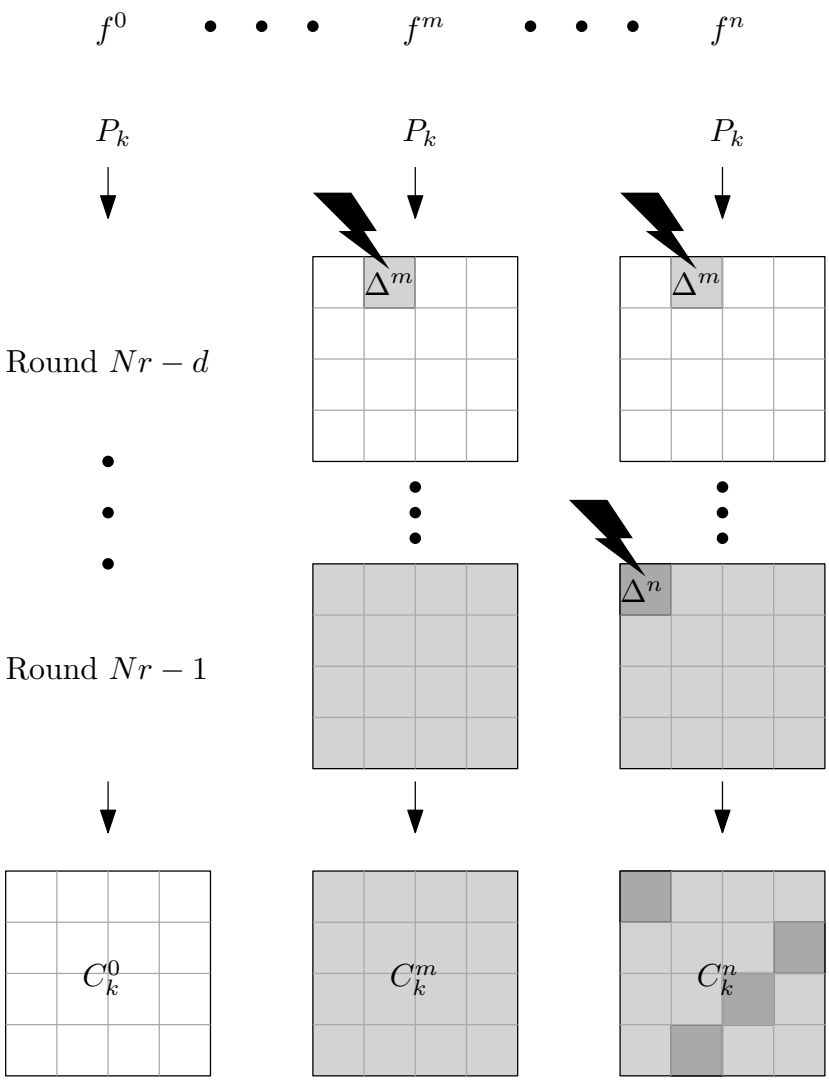

Figure 3.1: Visualizing an incremental fault.

section hinge on an in depth understanding of section 2.4, which explains the attack procedure from [36].

In Algorithm 2, each plaintext P_k is encrypted at a steady clock frequency f^w for increasing clock frequencies from f^0 to $f^{w_{max}}$. For each plaintext P_k, the attacker must have the control to reapply this same plaintext. For each plaintext, this produces a list of ciphertext outputs $\mathbb{C}$ containing ciphertexts encrypted at each frequency from

Algorithm 2 Encrypt_And_Get_Key

Ouput: The cipher key.

function ENCRYPT_AND_GET_KEY
  for $k = 0 : k_{max}$ **do**
    P_k = Random Plaintext
    for $w = 0 : w_{max}$ **do**
      $\mathbb{C}$.append(ENCRYPT(P_k, f^w)) ,
    end for
    K^{Nr} = FIND_KEY($\mathbb{C}$)
    if AES-128 **then**
      K = INV_KEY_EXPANSION(K^{Nr})
    else if AES-256 OR AES-192 **then**
      $\mathbb{C}$ = PARTIAL_DECRYPT($\mathbb{C}$, K^{Nr})
      K^{Nr-1} = FIND_KEY($\mathbb{C}$)
      K = INV_KEY_EXPANSION($K^{Nr-1,Nr}$)
    end if
    C_{test} = ENCRYPT(P_k, K)
    if $C_{test} == C^0$ **then**
      Return K
    end if
  end for
end function

f^0 to $f^{w_{max}}$. Some ciphertexts will have no incremental changes and all duplicate ciphertexts will be removed from $\mathbb{C}$. $\mathbb{C}$ is then passed to algorithm 3 to find the cipher key.

Algorithm 3 traverses through the list of ciphertexts $\mathbb{C}$, analyzing each unique ciphertext pair (C^m, C^n) in $\mathbb{C}$ as defined below:

$$\sum_{n=1}^{\mathbb{C}.size-1} \sum_{m=0}^{n-1} (C^m, C^n)$$

where C^m or C^n is ciphertext m or n in the list of ciphertexts $\mathbb{C}$. This technique

Algorithm 3 Find_Key

Input: Set $\mathbb{C}$ of ciphertexts.
Ouput: The last round key hypothesis K^{Nr}.

function FIND_KEY($\mathbb{C}$)
  $D =$ GETD
  for $n = 1 : \mathbb{C}$.size-1 **do**
    for $m = 0 : n - 1$ **do**
      for $j = 0 : 3$ **do**
        for $K_{j'}^{Nr} = 0 : 2^{32} - 1$ **do**
          if $SB_{j'}^{-1}(C_{j'}^{m} \oplus K_{j'}^{Nr}) \oplus SB_{j'}^{-1}(C_{j'}^{n} \oplus K_{j'}^{Nr}) \in D$ **then**
            $\mathbb{K}_{j'}^{Nr}$.append$(K_{j'}^{Nr})$
          end if
        end for
      end for
    end for
  end for
  for $j = 0 : 3$ **do**
    Retrieve highest occurring $K_{j'}^{Nr}$ in $\mathbb{K}_{j'}^{Nr}$ and place into K^{Nr}
  end for
  Return K^{Nr}
end function

Algorithm 4 Get$D$

Ouput: List D of all 1020 possible 4-byte MC outputs resulting from a 4-byte input where three out of four bytes are zero, and one has any non-zero value.

function GETD
  for $i = 0 : 3$ **do**
    for $B = 1 : 2^8 - 1$ **do**
      D.append(MC$(B << (8 \cdot i))$)
    end for
  end for
  Return D
end function

generalizes (2.5) to the following:

$$\Delta_{j}^{mn} = SB_{j'}^{-1}(C_{j'}^{m} \oplus K_{j'}^{Nr}) \oplus SB_{j'}^{-1}(C_{j'}^{n} \oplus K_{j'}^{Nr}) \tag{3.1}$$

(2.5) analyzes fault propagation that occurs strictly between faulty and fault-free encryptions, whereas (3.1) also analyzes any incremental fault propagation occurring between any two increasingly faulty encryptions.

Now consider lines 6 to 10 of algorithm 3. Each ciphertext pair C^m and C^n are applied to (3.1) for each of the 2^{32} possible values for $K_{j'}^{Nr}$. Only the $K_{j'}^{Nr}$ values that return a Δ_j^{mn} that is an element of D are stored into a list $\mathbb{K}_{j'}^{Nr}$ of potential $K_{j'}^{Nr}$ hypotheses. On average, 1036 $K_{j'}^{Nr}$ values applied to (3.1) will return a Δ_j^{mn} value that is an element of D. The full last round key K^{Nr} consists of 4 $K_{j'}^{Nr}$ elements, and can be retrieved after doing this analysis once for each $j = \{0, 1, 2, 3\}$, as shown on line 5 in algorithm 3. After analysis the adversary can form the K^{Nr} hypothesis by concatenating the 4 most commonly occurring $K_{j'}^{Nr}$ hypotheses in each $\mathbb{K}_{j'}^{Nr}$ list as shown on line 15 in algorithm 3.

The cipher key hypothesis K is then computed by performing Inverse Key Expansion on the K^{Nr} hypothesis as shown on lines 9 and 13 of algorithm 2. In order to confirm if the resulting K hypothesis is the correct one being used in the cipher, the plaintext P_k that was last used to generate the faulty ciphertexts is encrypted using a custom software AES implementation which will use the K hypothesis for encryption as shown on line 15 of algorithm 2. If the resulting ciphertext is identical to the fault-free ciphertext C^0 from the cryptosystem targeted in the attack, then it is known that this is the correct K being used in the cryptosystem under attack.

This application of IFA inherits the same non-target fault tolerance as the method from [36] as described in section 2.4.4, and can be used even if the attacker has no way of distinguishing between which faulty ciphertexts resulted from target or non-target faults.

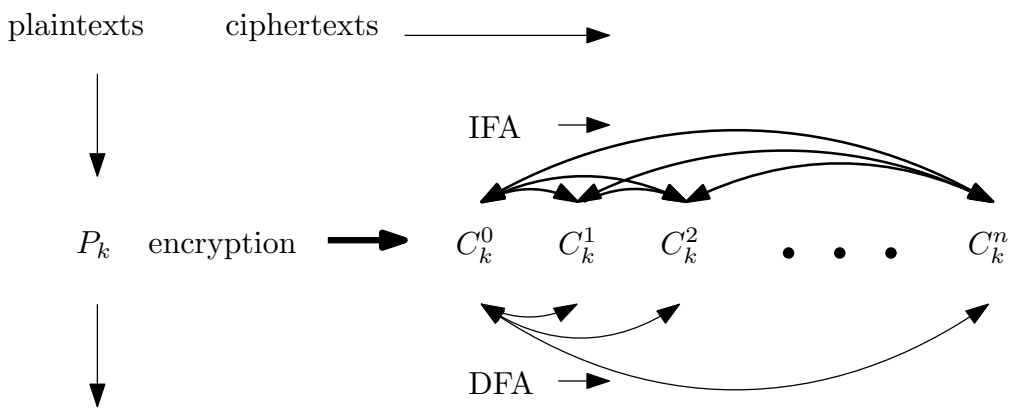

Figure 3.2: The number of ciphertext pairs extractable when using IFA compared to DFA.

The differences between IFA and classical DFA methods from [36] and [37] can be seen in Algorithm 3 on line 4. For the other methods, this for-loop would only run one iteration for the m value, with m being equal to zero, such that each faulty ciphertext is only compared to C^0 (the fault-free ciphertext). This along with Fig. 3.2 helps to illustrate how when the number of ciphertext pairs that the DFA method is able to extract is $\Theta(n)$, the number of ciphertext pairs that IFA is able to extract is proportional to $\Theta(n^2)$, where n is the number of ciphertexts analyzed.

The time complexity of the algorithm outlined in the provided pseudocode is $\Theta(n^2)$ for AES-128, 192, and 256. This is a higher time complexity than when analyzing the faults using classical DFA methods from [36] and [37], which would use $\Theta(n)$ for AES-128, 192, and 256, however, it has lower runtimes in practice because it is able to recover the key from analyzing fewer ciphertexts.

3.2 Example

As discussed above, algorithm 2 is run first to find the cipher key. In lines 3 - 6, a random plaintext is generated and then encrypted at different frequencies from frequency f^0 to $f^{w_{max}}$. Consider Fig. 3.3, showing encryptions at two of these frequencies, f^{m_a} and f^{n_a}, both of which contain a fault. The one on the left is a result of a fault occurring at a frequency m_a in an early round such as round 3. Similarly as with Δ^m in Fig. 3.1, this fault has propagated to every byte of the state by round $Nr-1$, leaving all output ciphertexts at frequencies greater than or equal to f^{m_a} useless for comparing against the fault-free ciphertext. Consider another fault occurring at a higher frequency n_a occurring between MC^{Nr-2} and MC^{Nr-1} as shown in the encryption on the right of Fig. 3.3. Even though Δ^{n_a} in isolation is a target fault, it will be superimposed upon the fault propagation resulting from Δ^{m_a}, and comparing its resulting output ciphertext against the fault-free ciphertext will not provide any cipher key information.

However, using IFA, the *fault increment* between encryptions at frequency f^{m_a} and f^{n_a} can be analyzed in the same way as the example in section 2.4.2. The ciphertext pair (C^{m_a}, C^{n_a}) from the example in Fig. 3.3 will be applied to (3.1) on lines 6 to 10 in algorithm 3 when $m = m_a$, $n = n_a$, and $j = 0$.

$$\Delta_j^{mn} = SB_{j'}^{-1}(C_{j'}^m \oplus K_{j'}^{Nr}) \oplus SB_{j'}^{-1}(C_{j'}^n \oplus K_{j'}^{Nr}) \qquad \text{(3.1 revisited)}$$

In the example from Fig. 3.3 the parameters in (3.1) (revisited above) would contain the following values:

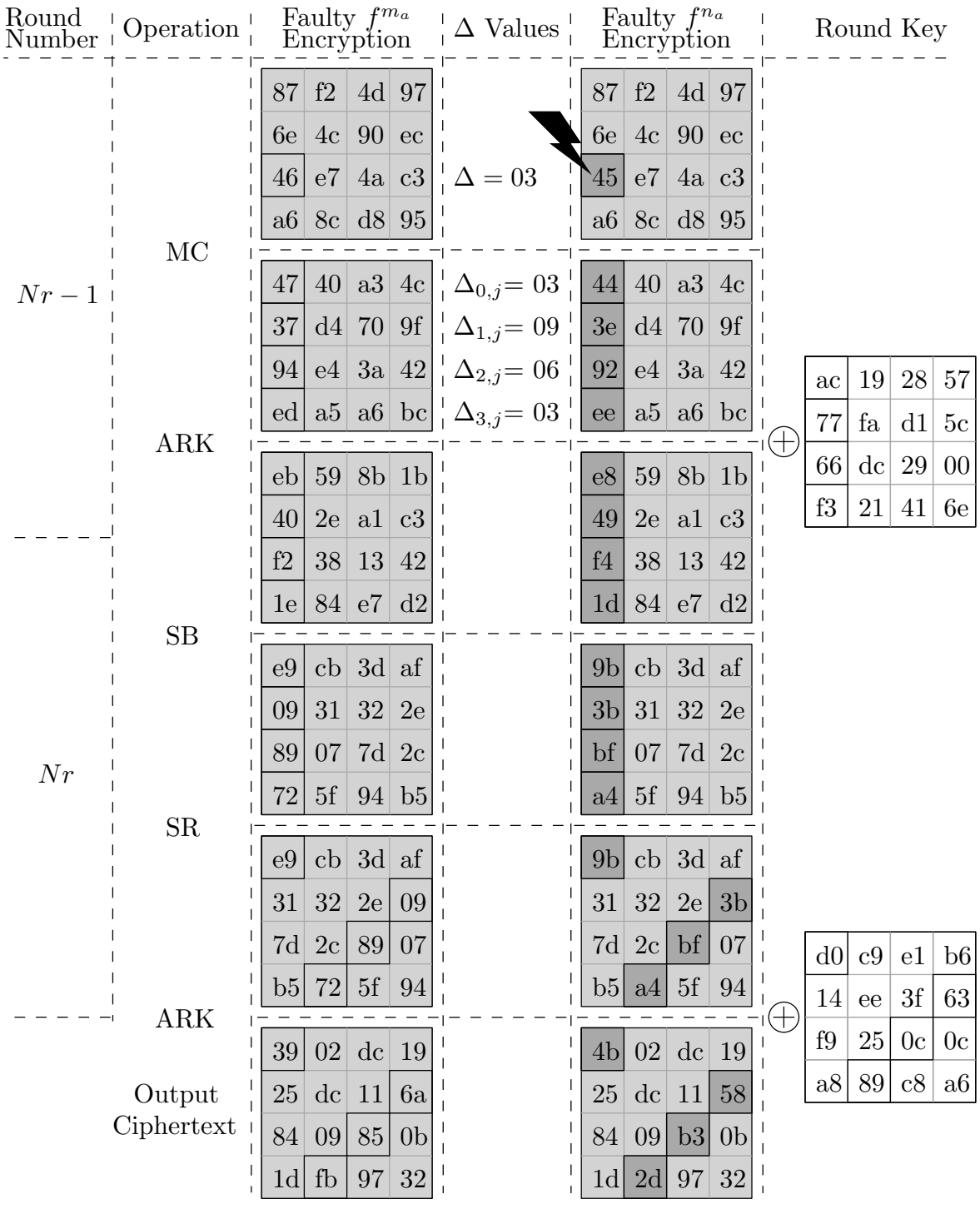

Figure 3.3: Propagation of a single-byte fault injected between MC^{Nr-2} and MC^{Nr-1}.

$$\Delta_j^{mn} = \{03, 09, 06, 03\}$$

$$K_{j'}^{Nr} = \{d0, 63, 0c, 89\}$$

$$C_{j'}^{m} = \{39, 6a, 85, fb\}$$

$$C_{j'}^{n} = \{4b, 0d, b3, 2d\}$$

$$j = 0$$

3.3 Summary

This chapter detailed Incremental Fault Analysis (IFA), our proposed fault analysis method. IFA is demonstrated by applying it to the attack from [36] and developing an algorithm to analyze the faulty ciphertexts using IFA in order to more efficiently extract the cipher key. The next chapter goes on to explain experimental results and provide quantitative measurements of how much improvement IFA provides when applied to the attack from [36].

Chapter 4

Experimental Results

To empirically justify the IFA attack method, the attack from [36] was tested both with and without IFA applied. The results shown in section 4.4 show that when using IFA, cipher keys could be deduced with over 14 times less faulty ciphertexts and after over 6.4 times less computational time. Attack success rates were 100% when using IFA, and 92% without. It is then outlined in section 4.5 how IFA applied to the attack from [36] is more practical to execute than other state-of-the-art fault attacks.

4.1 Experimental Framework for Attacks on Custom Hardware Architectures for AES

Experiments were conducted on the Intel Arria 10 SoC Development Kit [49]. Hardware implementations of the Advanced Encryption Standard (AES) [44] were tested on the Arria 10 FPGA [49].

Two different architectures were tested for both AES-128 and AES-256, and one

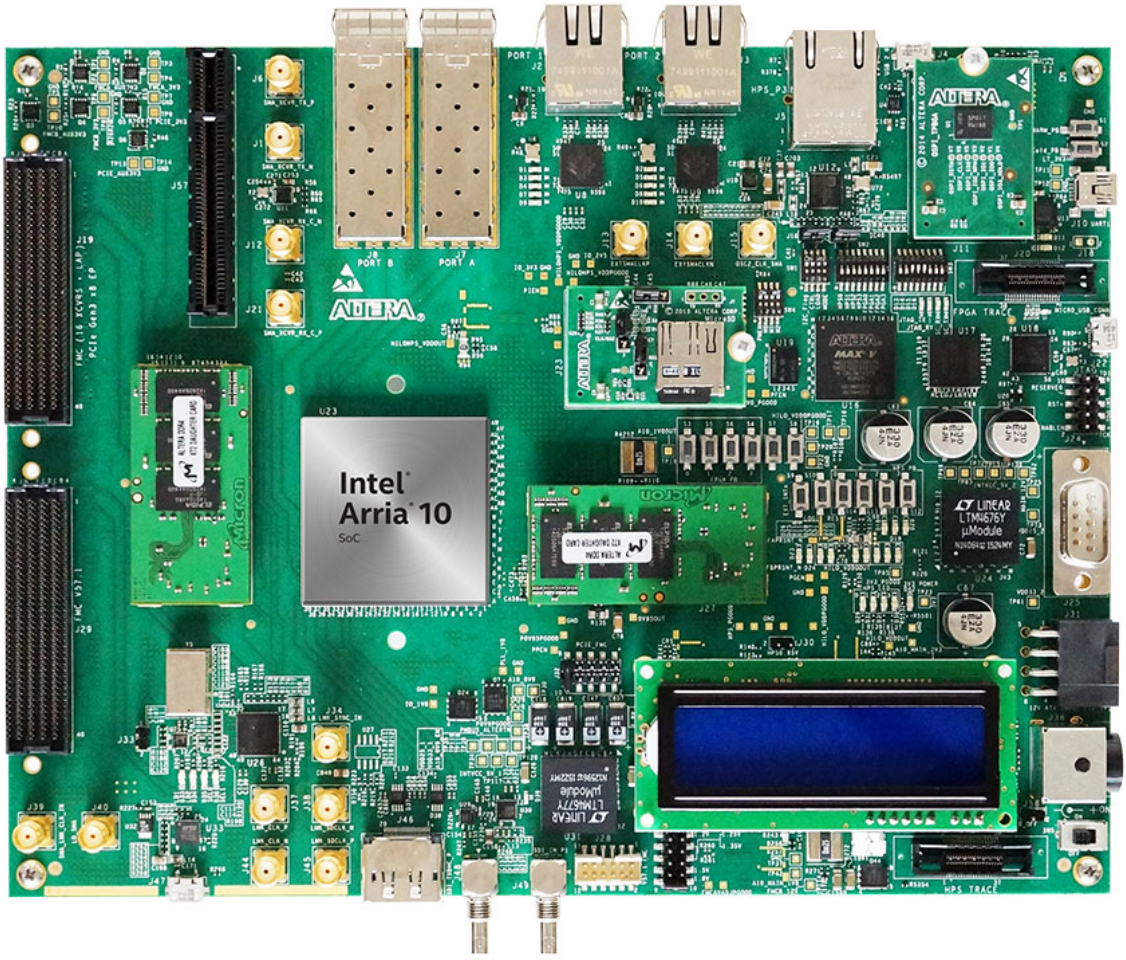

Figure 4.1: Intel Arria 10 SoC Development Kit [49].

architecture for AES-192. The first architecture used for all AES variants was self designed, and the second was an open source design [50]. We did not test an open source design for AES-192 because it is less popular and it was more difficult to find an open source implementation for.

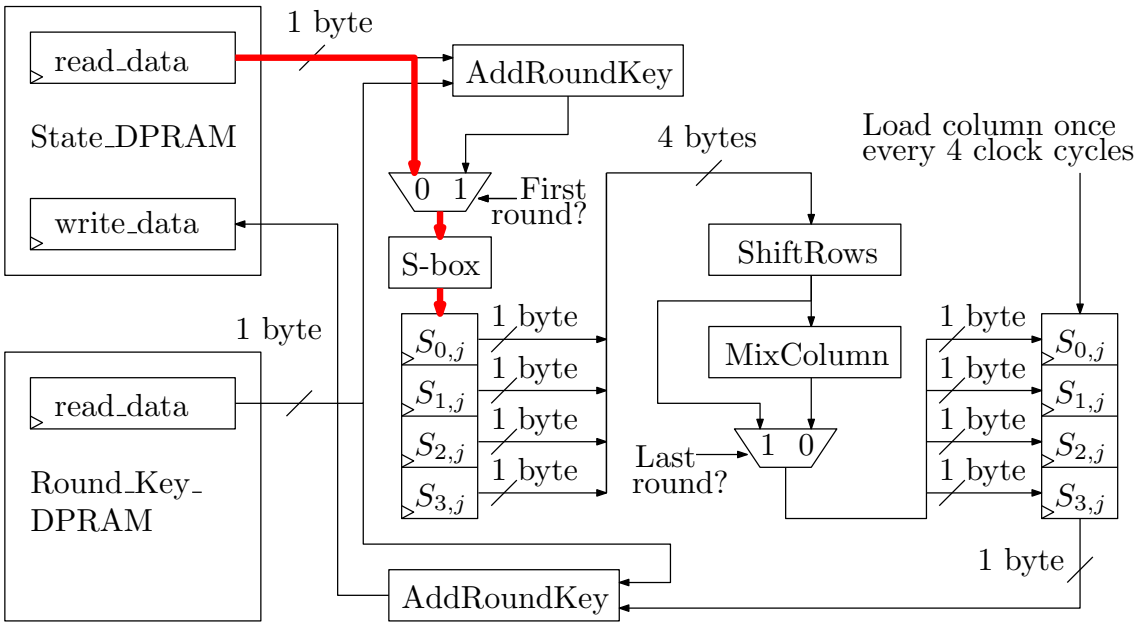

Figure 4.2: Self designed AES architecture.

4.1.1 Architecture Overview

The self designed architecture, shown in Fig. 4.2, has a throughput of encrypting one plaintext per $16 \cdot Nr$ clock cycles, with a 5 clock cycle latency, and uses minimal hardware resources. Key Expansion is performed prior to encryption and only needs to be performed once every time a new cipher key is chosen. The state and round keys are each stored in separate dual port RAMs with one byte stored per memory location. The architecture works on the state column by column, loading each byte in a column into a shift register, shown on the left of Fig. 4.2, in order to buffer a state column every 4 clock cycles. One round is performed every 16 clock cycles and the same hardware is used in every round. Prior to round 1, an initial ARK is applied between the state and cipher key as per the AES specification [44]. Additionally, MC is not applied in the last round as per the AES specification [44]. Other than these two special cases, every round is executed identically. Prior to buffering each column,

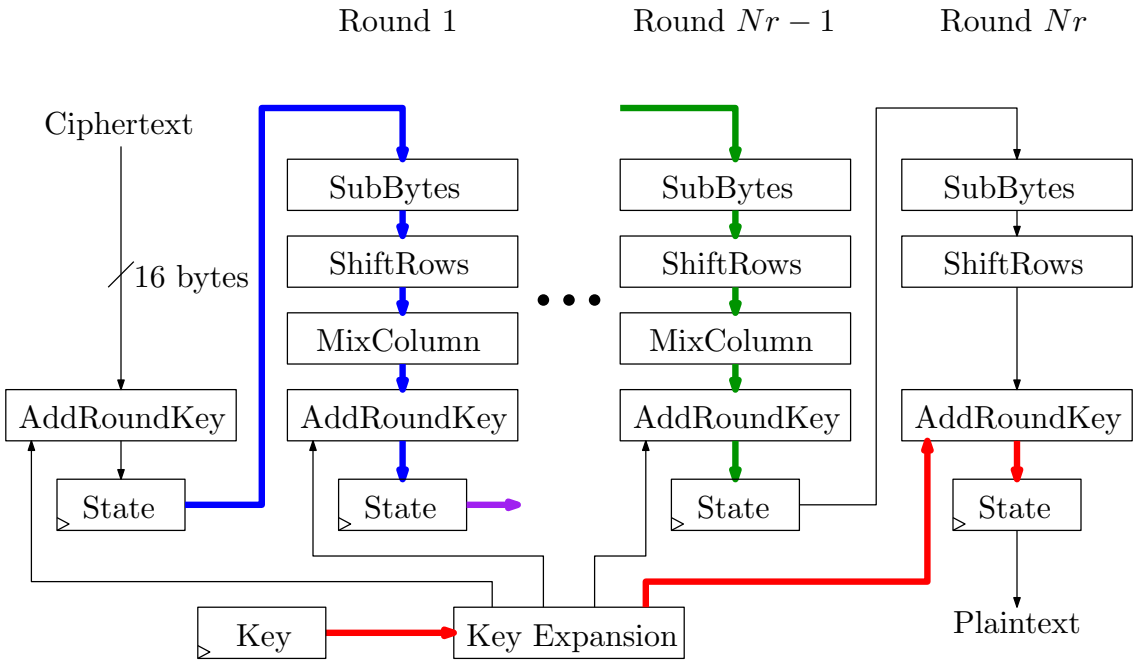

Figure 4.3: Open source AES architecture.

SB is applied individually to each byte being loaded into the shift register on the left of Fig. 4.2. Once a full column is loaded, SR and MC are applied to the column and the result is loaded into the column shift register on the right of Fig. 4.2. ARK is then applied to each byte before being stored back into the state DPRAM.

The open source architecture, shown in Fig. 4.3, utilizes significantly more hardware resources but has an impressive throughput of encrypting 1 ciphertext per clock cycle, with an Nr clock cycle latency. It has Nr instantiations of hardware modules capable of executing an AES round in one clock cycle. Thus a new plaintext can be fed to the input every clock cycle, and the corresponding ciphertexts will start appearing on the output Nr clock cycles later, with each new ciphertext appearing every clock cycle thereafter.

4.1.2 Critical Paths

The critical paths of the designs were analyzed through Intel Quartus II TimeQuest Timing Analyzer [51] and are discussed below.

For the self designed architecture, the critical path (shown in red in Fig. 4.2) in which faults are likely to occur while overclocking is between the read_data of the state_DPRAM and the first column buffer, $S_{0,j}$. The path delay is increased due to the fact that the state_DPRAM may be located further away from the rest of the encryption logic, and due to the S-box which is an 8-bit to 8-bit look up table.

In the open source architecture, Key Expansion is performed in combinational logic and the resulting round keys are fed straight into the ARK transformations without buffering them in registers first. The critical path resides between the register holding the key value, and the state register containing the output of the last round as highlighted in red in Fig. 4.3. However, in our experimental setup, this path does not get exercised because the value in the key register does not change for a sufficient number of clock cycles prior to encryption. After this, the next most critical paths lie between the state registers on the input and output of each round for rounds 1 to $Nr - 1$. The critical paths for rounds 1 and $Nr - 1$ are shown in blue and green in Fig. 4.3. These input and output registers are 128-256 bits each and the propagation delay is similar for each source bit to the corresponding destination bits. The first of these paths to cause a fault will be dependent on the plaintext input. Each path travels through SB, SR, MC, ARK. A fault will occur once one of these paths does not have enough time to propagate to up to one byte of one of the state registers containing the inputs to rounds $Nr - 2$ or $Nr - 1$. Prior to encrypting a plaintext, the plaintext input to the cipher is set to a different value for a number of clock cycles

until the transition propagates to the output. This allows faults to occur between rounds 1 to $Nr - 1$ because a fault can only occur during a round if its input state register receives a different input, and there is not enough time for the transition to propagate through SB, SR, MC, and ARK to reach the round's output state register. However, if the input state registers to each round are kept static, then faults cannot occur during the rounds because there are no transitions to be received by the output state registers.

4.2 Software/Processor Attacks

We experimented with overclocking-based fault attacks on a software/processor-based implementation using the Intel Nios II soft processor [52] on the Arria 10 FPGA. However, the attacks were less successful because the critical path was found to be in the control path, whereas the hardware implementations discussed above contained critical paths in the datapath. If the control path is corrupted with a fault then the effect on the datapath can be much more catastrophic and undetermined. For example, the processor's program counter might get corrupted, which keeps track of the current address of the instruction being executed. In this case the program would jump to a undetermined instruction or location possibly not even containing a valid instruction, and the remainder of the intended instructions may never get executed.

4.3 Analysis Software Implementation

All analysis software was efficiently implemented using C++. The C++ code was compiled using the -O3 option to turn on all compilation optimizations [53]. This

increased the execution speed by approximately 3 times.

Intel Advanced Encryption Standard (AES) New Instructions (AES-NI) [54] were used to speed up lines 10 to 14 and line 15 in algorithm 2. These instructions implement some of the complex and performance intensive steps of the AES algorithm using custom hardware in the CPU. This can accelerate the performance of an AES software implementation by 3 to 10 times.

Dense hash maps were used to implement the $\mathbb{K}_{j'}^{Nr}$ hypotheses lists that keep track of the number of occurrences for each $K_{j'}^{Nr}$ hypothesis that has been returned.

The for-loop on line 6 in Algorithm 3 appears to have time complexity of 2^{32}, but is written as such for conceptual purposes only. In the implementation this time constant was reduced to an insignificant value by analyzing $K_{j'}^{Nr}$ one byte at a time when applying it to (3.1). Each Δ_j^{mn} is then computed one byte at a time for each $K_{j'}^{Nr}$ byte, and the Δ_j^{mn} byte that each $K_{j'}^{Nr}$ byte maps to is recorded. After this is done for all $K_{j'}^{Nr}$ values, D is then traversed and each time one of its elements matches a recorded Δ_j^{mn} in which all 4 bytes were generated from the same $K_{j'}^{Nr}$ element, the corresponding $K_{j'}^{Nr}$ element is stored into the $\mathbb{K}_{j'}^{Nr}$ hypotheses list.

4.4 Experiments and Analysis

Experiments were conducted on the Intel Arria 10 SoC Development Kit [49]. The AES hardware implementations outlined above were tested on the Arria 10 FPGA. Frequency stepping was conducted using the Silicon Labs Si5338 Programmable Clock Generator [55] provided on the FPGA board. The attack from [36] was tested both

with and without IFA applied by encrypting plaintexts at a steady frequency for multiple clock frequencies, incrementing by 0.1 MHz at a time. The first 10 unique ciphertext outputs resulting from each plaintext frequency sweep were analyzed. Faults were injected into the self designed architectures by overclocking from 300 MHz to 350 MHz. Faults were injected into the open source architectures by overclocking from 200 MHz to 250 MHz.

Testing the Occurrence Rate of Target Faults

Before testing the success of the attacks, we tested how often target faults will occur while using the analysis method in [36] without IFA compared when using it with IFA. Faults were recorded across 1000 overclocked encryptions for each AES variant.

Target faults for the analysis method in [36] are single-byte faults occurring between MC^{r-3} and MC^{r-1}. To test the occurrence rate for these we created a list $\mathbb{C}_f$ of ciphertexts resulting from all possible single-byte faults of all values in all state positions in all rounds. The overclocked ciphertexts, $\mathbb{C}_k$, were then compared to $\mathbb{C}_f$ and whenever a ciphertext from $\mathbb{C}_k$ matched one from $\mathbb{C}_f$, it was concluded that this was the result of a single-byte fault occurring in the overclocked circuit. If a single-byte fault occurred between MC^{r-3} and MC^{r-1}, then this was recorded as a target fault occurrence.

Next, we compared this to the proposed IFA fault model. Target faults when using IFA are single-byte faults between MC^{r-3} and MC^{r-1} occurring *incrementally* between any two ciphertexts. To test the occurrence rate for this we compared $\mathbb{C}_f$ to the same list of ciphertexts, $\mathbb{C}_k$, except we analyzed each unique ciphertext pair

Table 4.1: Number of target faults produced.

	DFA		IFA	
	# Target Δ's	Total # Δ's	# Target Δ's	Total # Δ's
AES-128	11	900	65	4500
AES-192	13	900	65	4500
AES-256	9	900	51	4500
Average:	11	900	60	4500

(C^m, C^n) in $\mathbb{C}_k$ as defined below:

$$\sum_{n=1}^{\mathbb{C}_k.size-1} \sum_{m=0}^{n-1} (C^m, C^n)$$

where C^m or C^n is ciphertext m or n in the list of ciphertexts $\mathbb{C}_k$.

IFA applied to [36] produced 5.5 more extractable target faults on average compared to the classical Differential Fault Analysis (DFA) methods from [36]. These results are summarized in Table 4.1.

Testing the Attacks

The attack from [36] was tested both with and without IFA applied. Tables 4.2 to 4.6 show the number of ciphertexts required to extract the cipher key in each of these cases. DFA # ciphertexts is the number of ciphertexts required to extract the cipher key using the classical DFA analysis method from [36] and IFA # ciphertexts is the number of ciphertexts required to extract the cipher key using the proposed IFA method.

The results in tables 4.2 to 4.6 show that when using IFA, the AES cipher keys are deduced with over 14 times less faulty ciphertexts and after over 6.4 times less

Table 4.2: Results for self designed AES-128 architecture.

DFA		IFA	
# ciphertexts	Runtime (s)	# ciphertexts	Runtime (s)
260	0.20	10	0.17
40	0.026	10	0.0056
380	4.20	20	0.030
130	0.43	20	0.040
90	0.11	10	0.0055
130	0.30	10	0.024
110	0.095	30	0.059
90	0.074	10	0.0059
540	0.45	10	0.012
190	0.26	40	0.078
Average: 196	0.61	17	0.043

computational time compared to when using the DFA attack from [36]. Attack success rates were 100% when using IFA, and 92% without. Cipher key extraction was attempted on up to a maximum of 1000 ciphertexts before giving up when using the DFA method.

4.5 Comparison to Recent Work

Recent work such as Differential Fault Intensity Analysis (DFIA) [40], Fault Intensity Analysis (FSA) [38], Differential Error Rate Analysis (DERA) [41], and Non-Uniform Faulty Value Analysis NUFVA [39] require precise fault injection methods such as clock glitching, requiring specialized equipment. Furthermore, they require knowledge of when the internal state is in a specific round through other side-channel information such as power or time measurements in order to inject faults only in a specific stage of the encryption. More practical fault injection techniques have been explored [56] [57]

Table 4.3: Results for open source AES-128 architecture.

DFA		IFA	
# ciphertexts	Runtime (s)	# ciphertexts	Runtime (s)
30	0.0052	10	0.0021
10	0.0014	10	0.0011
40	0.0097	10	0.0021
20	0.0032	10	0.0017
40	0.0048	30	0.016
20	0.0062	10	0.0032
30	0.0045	20	0.0044
30	0.0082	20	0.0063
30	0.0088	20	0.0088
20	0.0024	10	0.0011
Average: 27	0.0054	15	0.0048

Table 4.4: Results for self designed AES-192 architecture.

DFA		IFA	
# ciphertexts	Runtime (s)	# ciphertexts	Runtime (s)
>1000	>9.31	50	0.99
600	2.94	20	0.093
730	5.63	30	0.62
180	1.17	50	0.55
490	2.41	40	0.13
760	4.45	240	0.98
380	1.86	210	0.89
290	1.90	30	0.45
>1000	>6.66	40	0.35
320	1.24	150	1.072
Average: >575	>3.76	86	0.614

Table 4.5: Results for self designed AES-256 architecture.

DFA		IFA	
# ciphertexts	Runtime (s)	# ciphertexts	Runtime (s)
630	1.91	30	0.64
340	1.17	50	0.69
570	2.43	60	0.26
>1000	>7.86	60	1.01
880	6.66	60	1.16
760	6.84	40	0.87
530	3.19	60	0.78
>1000	>7.03	60	0.84
710	3.17	10	0.048
720	4.84	40	0.91
Average: >714	>4.51	15	0.72

Table 4.6: Results for open source AES-256 architecture.

DFA		IFA	
# ciphertexts	Runtime (s)	# ciphertexts	Runtime (s)
30	0.038	10	0.026
140	0.26	50	0.087
70	0.061	50	0.11
70	0.20	40	0.070
100	0.38	10	0.027
90	0.53	40	0.073
40	0.34	10	0.029
50	0.52	10	0.023
200	0.47	140	0.28
60	0.49	30	0.055
Average: 85	0.33	39	0.079

[48] but use classical DFA analysis methods from [36], and thus require more faulty encryptions than IFA applied to [36].

In contrast, IFA allows faults to be injected using a practical overclocking-based fault injection method in which the entire encryption is overclocked at a steady frequency, after which the cipher key can be extractable without knowing any details of the internal state or faults occurring. IFA has no need for specialized clock glitching equipment or extra side-channel information to know when the cipher is in a specific round. The faults can be analyzed blindly without the attacker needing to distinguish which faulty ciphertexts are a result from target or non-target faults, which is a practicality not possible with the other methods mentioned above. This is possible due to the non-target fault tolerance in the method from [36] that this application of IFA inherits, as described in section 2.4.4. This is shown in our results by extracting the cipher key with a success rate of 100% using the first 16, 86, and 27 target/non-target ciphertexts outputted on average for AES-128, 192, and 256, respectively. This was over 14 times less ciphertexts than when using the method we based ours on in [36] without IFA. If the methods in comparison were applied using the proposed relaxed fault injection techniques, these methods would likely not be feasible to execute even with infinite ciphertexts. This is affirmed from the use of more precise fault injection methods in these papers which use clock glitching and require precise knowledge of which stages the internal state is at during encryption in order to isolate fault injection to a specific round. If injection of a target fault is guaranteed, the methods in comparison are able to extract the cipher key after 6 or more target fault injections. However, assuming the same conditions, IFA applied to [36] would be able to extract the cipher key with a single target fault injection for AES-128, or two target fault

injections for AES-192 and 256 (followed by a trivial exhaustive search from 2^8 K^{Nr} hypotheses using the analysis techniques from [37]).

Furthermore, the papers mentioned above run experiments only on AES-128. We also conducted experiments on AES-192 and AES-256. These later two are more difficult to mount attacks on because the last two round keys have to be found in two steps, where the second step is dependent on the correctness of the first.

4.6 Summary

In this chapter, experiments were conducted on different AES 128, 192, and 256 architectures on the Intel Arria 10 SoC Development Kit in order to validate our IFA attack method. The results shown in section 4.4 show that when applying IFA to [36], the AES cipher keys could be deduced from over 14 times less faulty ciphertexts and after over 6.4 times less computational time. Attack success rates were 100% when using IFA, and 92% without. It was then outlined in section 4.5 how IFA applied to the method from [36] is more practical to execute than other state-of-the-art fault attacks.

Chapter 5

Conclusion and Future Work

This thesis proposes a new fault analysis method called Incremental Fault Analysis (IFA) that allows cipher keys to be compromised more quickly and efficiently compared to existing methods when using practical fault injection methods. The effectiveness of our method comes from analyzing *incremental* fault differences between ciphertexts encrypted at increasing stress levels. Our analysis requires as few as 2 target faults to occur for successful cipher key extraction, and without need for distinguishing between which faulty ciphertexts resulted from target or non-target faults. This allows for practical fault injection methods not requiring clock glitching in which encryptions are overclocked at steady frequencies over all algorithmic stages.

To empirically justify the IFA attack method, experiments were conducted on Advanced Encryption Standard (AES) 128, 192, and 256 architectures. The attack from [36] was tested both with and without IFA applied. Our results show that when applying IFA, AES cipher keys could be deduced from over 14 times less faulty ciphertexts and after over 6.4 times less computational time. Attack success rates were 100% when using IFA, and 92% without. Furthermore, IFA applied to the method

from [36] is more practical to execute than other state-of-the-art fault attacks such as DFIA, FSA, DERA, and NUFVA.

5.1 Future Work

Topics for consideration in future work are to apply IFA with a fault injection method involving undervolting the cryptosystem under attack rather than overclocking as in [56].

IFA applied to attacks on block ciphers other than AES could be explored as in [36]. [36] describes how their attack could work theoretically on all Substitution Permutation Network (SPN) structures and mounts their attack on AES as well as the KHAZAD block cipher [58].

Counter measures for the IFA attack method could be addressed. For example, a possible countermeasure would be to design a cryptosystem in such a way that incremental faults are always triggered across multiple bytes in multiple cipher rounds. This would render the incremental fault changes useless for analysis.

IFA could also be applied to attacks other than [36] in order to improve their effectiveness. For example, consider Fig. 5.1 which shows two different faults, Δ_{01} and Δ_{10}, that were injected into a certain cipher round with fault differential values of 01 and 10. Each of these are a unique fault relatively to the fault free state shown at the bottom and the resulting faulty output from each can be analyzed relatively to the fault-free output. However, a third unique fault relationship also exists by analyzing the *incremental differential* between Δ_{01} and Δ_{10}. This differential would contain a value of 0x11 since Δ_{01} differs from Δ_{10} by a value of 0x11. While there are only 2 faults that have occurred, the information of 3 faults exists and can be

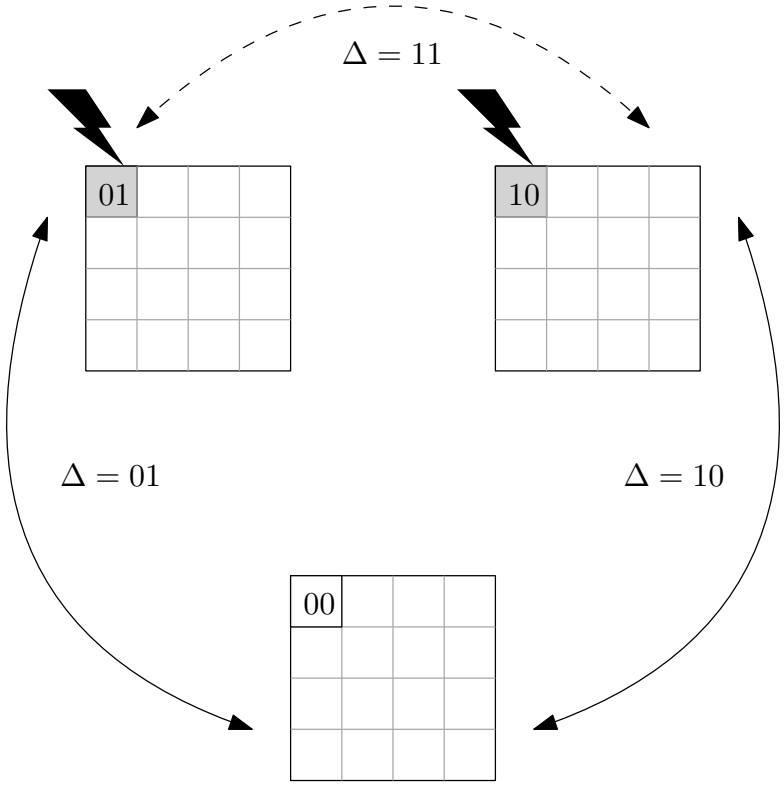

Figure 5.1: Extracting information of 3 faults from 2 using IFA.

extracted through IFA. Generally speaking, IFA allows more secret information to be extracted from the same amount of data when compared to classical Differential Fault Analysis (DFA) methods.

Bibliography

[1] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[2] Bruce Schneier. *Applied Cryptography, Second Edition: Protocols, Algorithms and Source Code in C*. Wiley, 1996.

[3] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. "Analysis of the HTTPS Certificate Ecosystem". In: *Internet measurement conference - IMC* (2013).

[4] Bala Iyer, Sharad Mehrotra, Einar Mykletun, Gene Tsudik, and Yonghua Wu. "A Framework for Efficient Storage Security in RDBMS". In: *Advances in Database Technology - EDBT*. Ed. by Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 147–164.

[5] N. Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System*. `http://bitcoin.org/bitcoin.pdf`. Accessed: 2018-12. 2009.

[6] Jerome Burke, John McDonald, and Todd Austin. "Architectural Support for Fast Symmetric-key Cryptography". In: *Proceedings of the Ninth International*

Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS IX. Cambridge, Massachusetts, USA, 2000, pp. 178–189.

[7] Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith, and Louis Wingers. "The SIMON and SPECK Lightweight Block Ciphers". In: *Design Automation Conference*. 2015, pp. 1–6.

[8] Christophe De Cannière, Orr Dunkelman, and Miroslav Knežević. "KATAN and KTANTAN — A Family of Small and Efficient Hardware-Oriented Block Ciphers". In: *Cryptographic Hardware and Embedded Systems – CHES*. Ed. by Christophe Clavier and Kris Gaj. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 272–288.

[9] Xuejia Lai. *On the Design and Security of Block Ciphers*. Hartung-Gorre, 1992.

[10] C. Ding, G. Xiao, and W. Shan. *The Stability Theory of Stream Ciphers*. Springer-Verlag, 1991.

[11] Thomas W. Cusick. *Stream Ciphers and Number Theory*. Elsevier, 2004.

[12] Alex Biryukov and Adi Shamir. "Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers". In: *Advances in Cryptology — ASIACRYPT*. Ed. by Tatsuaki Okamoto. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1–13.

[13] Arto Salomaa. *Public-Key Cryptography*. Springer, 2011.

[14] Iris Anshel, Michael Anshel, and Dorian Goldfeld. "An Algebraic Method for Public-Key Cryptography". In: *Mathematical Research Letters* 291 (1999), pp. 287–291.

[15] Sattam S. Al-Riyami and Kenneth G. Paterson. "Certificateless Public Key Cryptography". In: *Advances in Cryptology - ASIACRYPT*. Ed. by Chi-Sung Laih. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 452–473.

[16] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-key Cryptosystems". In: *Communications of the ACM* 21.2 (Feb. 1978), pp. 120–126.

[17] Eli Biham and Adi Shamir. "Differential Cryptanalysis of DES-like Cryptosystems". In: *Journal of Cryptology* 4.1 (1991), pp. 3–72.

[18] Mitsuru Matsui. "Linear Cryptanalysis Method for DES Cipher". In: *Advances in Cryptology — EUROCRYPT*. Ed. by Tor Helleseth. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 386–397.

[19] T. Siegenthaler. "Decrypting a Class of Stream Ciphers Using Ciphertext Only". In: *IEEE Transactions on Computers* 34 (1985), pp. 81–85.

[20] H. D. Phaneendra, C. Vidya Raj, and M. S. Shivakumar. "Applying Quantum Search to a Known-Plaintext Attack on Two-Key Triple Encryption". In: *Intelligent Information Processing III*. Ed. by Zhongzhi Shi, K. Shimohara, and D. Feng. Boston, MA: Springer US, 2007, pp. 171–178.

[21] Helen A. Bergen and James M. Hogan. "A Chosen Plaintext Attack on an Adaptive Arithmetic Coding Compression Algorithm". In: *Computers & Security* 12.2 (1993), pp. 157–167.

[22] Xavier Koeune. *Foundations of Security Analysis and Design III*. 2005.

[23] Marc Joye and Michael Tunstall. *Fault Analysis in Cryptography*. Springer Publishing Company, Incorporated, 2012.

[24] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology — CRYPTO*. Ed. by Neal Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113.

[25] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Advances in Cryptology — CRYPTO*. Ed. by Michael Wiener. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397.

[26] J. Longo, E. De Mulder, D. Page, and M. Tunstall. "SoC It to EM: ElectroMagnetic Side-Channel Attacks on a Complex System-on-Chip". In: *Cryptographic Hardware and Embedded Systems – CHES*. Ed. by Tim Güneysu and Helena Handschuh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 620–640.

[27] Jean-Jacques Quisquater and David Samyde. "ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards". In: *Smart Card Programming and Security*. Ed. by Isabelle Attali and Thomas Jensen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 200–210.

[28] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. "The EM Side—Channel(s)". In: *Cryptographic Hardware and Embedded Systems – CHES*. Ed. by Burton S. Kaliski, çetin K. Koç, and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 29–45.

[29] Daniel Genkin, Adi Shamir, and Eran Tromer. "RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis". In: *Advances in Cryptology – CRYPTO*. Ed. by Juan A. Garay and Rosario Gennaro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 444–461.

[30] Daniel J. Bernstein. *Cache-Timing Attacks on AES*. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`. Accessed: 2018-12. 2005.

[31] Billy Bob Brumley and Nicola Tuveri. "Remote Timing Attacks Are Still Practical". In: *Computer Security – ESORICS*. Ed. by Vijay Atluri and Claudia Diaz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 355–371.

[32] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Topics in Cryptology – CT-RSA*. Ed. by David Pointcheval. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20.

[33] Colin Percival. "Cache Missing for Fun and Profit". In: *BSDCan* (2005), pp. 1–13.

[34] Yuval Yarom and Katrina Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *USENIX Security* (2014), pp. 1–14.

[35] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: a Timing Attack on OpenSSL Constant-Time RSA". In: *Journal of Cryptographic Engineering* 7.2 (2017), pp. 99–112.

[36] Gilles Piret and Jean-Jacques Quisquater. "A Differential Fault Attack Technique Against SPN Structures, with Application to the AES and Khazad". In: *Cryptographic Hardware and Embedded Systems – CHES*. Ed. by Colin D. Walter, Çetin K. Koç, and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 77–88.

[37] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. "Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault". In: *Information Security Theory and Practice. Security and Privacy of Mobile Devices*

in Wireless Communication. Ed. by Claudio A. Ardagna and Jianying Zhou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 224–233.

[38] Yang Li, Kazuo Sakiyama, Shigeto Gomisawa, Toshinori Fukunaga, Junko Takahashi, and Kazuo Ohta. "Fault Sensitivity Analysis". In: *Cryptographic Hardware and Embedded Systems – CHES*. Ed. by Stefan Mangard and François-Xavier Standaert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 320–334.

[39] Thomas Fuhr, Eliane Jaulmes, Victor Lomne, and Adrian Thillard. "Fault Attacks on AES with Faulty Ciphertexts Only". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2013, pp. 108–118.

[40] Sikhar Patranabis, Abhishek Chakraborty, Debdeep Mukhopadhyay, and Partha Pratim Chakrabarti. "Differential Fault Intensity Analysis". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2014, pp. 49–58.

[41] Yannan Liu, Jie Zhang, Lingxiao Wei, Feng Yuan, and Qiang Xu. "DERA: Yet Another Differential Fault Attack on Cryptographic Devices Based on Error Rate Analysis". In: *Design Automation Conference*. DAC '15. San Francisco, California, 2015, 31:1–31:6.

[42] D. Karaklaji, J. Schmidt, and I. Verbauwhede. "Hardware Designer's Guide to Fault Attacks". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.12 (2013), pp. 2295–2306.

[43] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Adi Shamir, and Eran Tromer. "Physical Key Extraction Attacks on PCs". In: *Communications of the ACM* 59.6 (May 2016), pp. 70–79.

[44] National Institute of Standards and Technology. "Announcing the Advanced Encryption Standard (AES)". In: *Federal Information Processing Standards Publications* (2001).

[45] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults". In: *Advances in Cryptology — EUROCRYPT*. Ed. by Walter Fumy. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 37–51.

[46] Eli Biham and Adi Shamir. "Differential Fault Analysis of Secret Key Cryptosystems". In: *Advances in Cryptology — CRYPTO*. Ed. by Burton S. Kaliski. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 513–525.

[47] Yang Li, Yu-ichi Hayashi, Arisa Matsubara, Naofumi Homma, Takafumi Aoki, Kazuo Ohta, and Kazuo Sakiyama. "Yet Another Fault-Based Leakage in Non-Uniform Faulty Ciphertexts". In: *Foundations and Practice of Security*. Ed. by Jean Luc Danger, Mourad Debbabi, Jean-Yves Marion, Joaquin Garcia-Alfaro, and Nur Zincir Heywood. Cham: Springer International Publishing, 2014, pp. 272–287.

[48] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Victor Lomné, and Florian Mendel. "Statistical Fault Attacks on Nonce-Based Authenticated Encryption Schemes". In: *Advances in Cryptology – ASIACRYPT*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 369–395.

[49] *Intel Arria 10 SoC Development Kit*. `https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/arria-10-soc-development-kit.html`. Accessed: 2018-12.

[50] *Advanced Encryption Standard (AES) SystemVerilog Core*. https://github.com/cjdrake/AES. Accessed: 2018-12.

[51] *Intel Quartus II TimeQuest Timing Analyzer*. https://www.intel.com/content/www/us/en/programmable/support/training/course/odsw1115.html. Accessed: 2018-12.

[52] *Intel Nios II Processors for FPGAs*. https://www.intel.com/content/www/us/en/products/programmable/processor/nios-ii.html. Accessed: 2018-12.

[53] *Options That Control Optimization*. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Accessed: 2018-12.

[54] *Intel Advanced Encryption Standard (AES) New Instructions (AES-NI)*. https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni. Accessed: 2018-12.

[55] *Silicon Labs Si5338 Programmable Clock Generator*. https://www.silabs.com/products/development-tools/timing/clock/si5338-development-kit. Accessed: 2018-12.

[56] Nidhal Selmane, Sylvain Guilley, and Jean-Luc Danger. "Practical Setup Time Violation Attacks on AES". In: *Seventh European Dependable Computing Conference*. 2008, pp. 91–96.

[57] Shivam Bhasin, Nidhal Selmane, Sylvain Guilley, and Jean-Luc Danger. "Security Evaluation of Different AES Implementations Against Practical Setup Time Violation Attacks in FPGAs". In: *IEEE International Workshop on Hardware-Oriented Security and Trust*. 2009, pp. 15–21.

[58] Paulo Barreto and Vincent Rijmen. *The Khazad Legacy-Level Block Cipher*. Jan. 2000.