Technische Universität München
TUM School of Computation, Information and Technology

Fault Injection Analysis of Embedded Cryptography

Attacks and Solutions

Michael Gruber

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz:
Prof. Dr.-Ing. Hussam Amrouch

Prüfende der Dissertation:
1. Prof. Dr.-Ing. Georg Sigl
2. Prof. Jean-Max Dutertre

Die Dissertation wurde am 15.04.2024 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 04.09.2024 angenommen.

To my family, past, present, and future

Abstract

Physical attacks on cryptographic implementations have become a serious threat over the last years. The reason for this recent trend is that modern cryptography was proven to be secure against cryptanalytic attack vectors due to rigorous standardization processes which involve years of cryptanalysis.

In contrast, the security assumptions of the standardization processes only hold if an attacker has no further knowledge about the internals, i.e., in a black-box scenario. Therefore, if an attacker has access to the implementation in form of a physical device, he also has access to additional knowledge, i.e., side-channel leakage which violates the black-box scenario.

With the growing number of interconnected devices and the rise of the Internet of Things (IoT), the need for secure communications is ubiquitous. Consequently, the requirement of ubiquitous secure communication results in devices running cryptography within reach of a possible attacker.

The objective of this work is to investigate the application of state-of-the-art fault attacks, viz. Differential Fault Analysis (DFA), Persistent Fault Analysis (PFA), Algebraic Fault Analysis (AFA), and Statistical Ineffective Fault Analysis (SIFA), on state-of-the-art cryptography, i.e., candidates of the CAESAR and LWC competition, in order to evaluate the threat posed by Fault Injection Analysis (FIA).

This work's contributions are threefold: As a first main contribution we evaluated how faulty behavior can be caused on modern microcontroller by Electromagnetic Fault Injection (EMFI). As a second main contribution we propose: The DFA of the lightweight block cipher KLEIN. The PFA of the Authenticated Encryption with Associated Data (AEAD) schemes Deoxys-II, OCB, and COLM. The SIFA of the AEAD scheme Gimli. The AFA of the AEAD scheme Subterranean 2.0. As a third main contribution we evaluated how cryptography can be protected from FIA and Side-Channel Analysis (SCA). To do so we propose a novel generic solution for simultaneous protection against SCA and FIA of arbitrary order. We combine Domain-Oriented Masking (DOM) and Repetition Codes (REPs) in an orthogonal way and call this approach Domain Oriented Masking with REPetiton codes (DOMREP). The resistance against SCA and FIA can be scaled independently of each other, for the protection against higher-order SCA and the injection of multiple faults including SIFA. Furthermore, we propose a novel open-source tool called TOFU which synthesizes VCD simulation traces into power traces, with adjustable leakage models. The functionality of TOFU was verified by a CPA of an AES hardware implementation.

Kurzfassung

Physikalische Angriffe auf kryptografische Implementierungen sind in den letzten Jahren zu einer ernsthaften Bedrohung geworden. Der Grund für diesen jüngsten Trend ist die Tatsache, dass sich die moderne Kryptografie aufgrund strenger Standardisierungsprozesse, die jahrelange Kryptoanalysen beinhalten, als sicher gegenüber kryptoanalytischen Angriffsvektoren erwiesen hat.

Im Gegensatz dazu gelten die Sicherheitsannahmen des Standardisierungsprozesses nur, wenn ein Angreifer keine weiteren Kenntnisse über die Interna hat, d.h. in einem Black-Box-Szenario. Wenn also ein Angreifer Zugang zur Implementierung in Form eines physischen Geräts hat, hat er damit auch Zugang zu zusätzlichem Wissen, d.h. zu Seitenkanalinformationen, welche die Annahmen des Black-Box-Szenario verletzen.

Mit der wachsenden Zahl miteinander verbundener Geräte und dem Aufkommen des Internet of Things (IoT) ist der Bedarf an sicherer Kommunikation allgegenwärtig. Folglich führt die Anforderung einer allgegenwärtigen sicheren Kommunikation dazu, dass Geräte welche Kryptografie ausführen in die Reichweite eines möglichen Angreifers kommen.

Das Ziel dieser Arbeit ist es, die Anwendung moderner Fehlerangriffe, nämlich Differential Fault Analysis (DFA), Persistent Fault Analysis (PFA), Algebraic Fault Analysis (AFA) und Statistical Ineffective Fault Analysis (SIFA), auf moderne Kryptografie, d.h. Kandidaten des CAESAR und LWC Wettbewerbs, um die von Fault Injection Analysis (FIA) ausgehende Bedrohung zu bewerten.

Die Beiträge dieser Arbeit sind dreigeteilt: Als ersten Hauptbeitrag haben wir untersucht, wie fehlerhaftes Verhalten auf modernen Mikrocontrollern durch Electromagnetic Fault Injection (EMFI) verursacht werden kann. Als zweiten Hauptbeitrag schlagen wir vor: Die DFA der leichtgewichtigen Blockchiffre KLEIN. Die PFA der Authenticated Encryption with Associated Data (AEAD)-Schemata Deoxys-II, OCB und COLM. Die SIFA des AEAD-Schemas Gimli. Die AFA des AEAD-Schemas Subterranean 2.0. Als dritten Hauptbeitrag haben wir untersucht, wie Kryptografie vor FIA und Side-Channel Analysis (SCA) geschützt werden kann. Zu diesem Zweck schlagen wir eine neuartige generische Lösung für den gleichzeitigen Schutz gegen SCA und FIA beliebiger Ordnung vor. Dazu kombinieren wir Domain-Oriented Masking (DOM) und Repetition Codes (REPs) auf orthogonale Weise und nennen diesen Ansatz Domain Oriented Masking with REPetiton codes (DOMREP). Die Widerstandsfähigkeit gegen SCA und FIA kann unabhängig voneinander skaliert werden, um den Schutz gegen SCA höherer Ordnung und die Injektion von Mehrfachfehlern einschließlich SIFA zu gewährleisten. Darüber hinaus schlagen wir ein neuartiges Open-Source-Tool namens TOFU vor, welches VCD-Simulationsspuren in Energieverbrauchsspuren mit einstellbaren Leckagemodellen synthetisiert. Die Funktionalität von TOFU wurde durch die CPA einer AES-Hardware-Implementierung verifiziert.

Acknowledgment

First of all, I would like to thank Prof. Dr.-Ing. Georg Sigl for giving me the opportunity to pursue a Ph.D. in a supportive environment with the necessary freedom to follow my own research interests.

I would like to thank my colleagues at the Chair for Security in Information Technology for making these years a very memorable time in my life.

I would like to thank my office mates Lars Tebelmann, Thomas Schamberger, and Tim Music for the many discussions about attacks, countermeasures, and many other things.

I would like to thank Marion Zillner, Harry Olm, and Priv.-Doz. Dr.-Ing. habil. Michael Pehl for their daily support in all kinds of organizational and technical matters.

Finally, I would like to thank my family for their constant support.

List of Publications

Journal Articles

TIFS 2021 Michael Gruber et al. "DOMREP–An Orthogonal Countermeasure for Arbitrary Order Side-Channel and Fault Attack Protection". In: *IEEE Transactions on Information Forensics and Security* 16 (2021), pp. 4321–4335. DOI: 10.1109/TIFS.2021.3089875 [Gru+21]

Conference Proceedings

FDTC 2017 Oscar M. Guillen, Michael Gruber, and Fabrizio De Santis. "Low-Cost Setup for Localized Semi-invasive Optical Fault Injection Attacks". In: *Constructive Side-Channel Analysis and Secure Design*. Springer International Publishing, 2017, pp. 207–222. DOI: 10.1007/978-3-319-64647-3_13 [GGS17]

COSADE 2019 Michael Gruber and Bodo Selmke. "Differential Fault Attacks on KLEIN". in: *The Urban Book Series*. Springer Singapore, 2019, pp. 80–95. DOI: 10.1007/978-3-030-16350-1_6 [GS19]

FDTC 2019 Michael Gruber, Matthias Probst, and Michael Tempelmeier. "Persistent Fault Analysis of OCB, DEOXYS and COLM". in: *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 2019, pp. 17–24. DOI: 10.1109/FDTC.2019.00011 [GPT19]

ISVLSI 2020 Michaela Brunner et al. "Logic Locking Induced Fault Attacks". In: *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2020, pp. 114–119. DOI: 10.1109/ISVLSI49217.2020.00030 [Bru+20]

HOST 2020 M. Gruber, M. Probst, and M. Tempelmeier. "Statistical Ineffective Fault Analysis of GIMLI". in: *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2020, pp. 252–261. DOI: 10.1109/HOST45689.2020.9300260 [GPT20]

NTMS 2021 Patrick Karl and Michael Gruber. "A Survey on the Application of Fault Analysis on Lightweight Cryptography". In: *2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. 2021, pp. 1–3. DOI: 10.1109/NTMS49979.2021.9432667 [KG21]

FDTC 2021 Michael Gruber, Patrick Karl, and Georg Sigl. "Algebraic Fault Analysis of Subterranean 2.0". In: *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*. 2021, pp. 45–55. DOI: 10.1109/FDTC53659.2021.00016 [GKS21]

CHES 2022 Jonas Ruchti, Michael Gruber, and Michael Pehl. "When the Decoder Has to Look Twice: Glitching a PUF Error Correction". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.3 (2022), 26–70. DOI: 10.46586/tches.v2022.i3.26-70. URL: https://tches.iacr.org/index.php/TCHES/article/view/9694 [RGP22]

ASP-DAC 2023 Mathieu Gross et al. "FPGANeedle". In: *Proceedings of the 28th Asia and South Pacific Design Automation Conference*. ACM, 2023. DOI: 10.1145/3566097.3568352 [Gro+23]

COSADE 2023 Tobias Holl, Katharina Bogad, and Michael Gruber. "Whiteboxgrind – Automated Analysis of Whitebox Cryptography". In: *Constructive Side-Channel Analysis and Secure Design*. Springer Nature Switzerland, 2023, pp. 221–240. DOI: 10.1007/978-3-031-29497-6_11 [HBG23]

DDECS 2024 Michael Mildner et al. "Fault-Simulation-Based Flip-Flop Classification for Reverse Engineering". In: *2024 27th International Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*. 2024, pp. 53–56. DOI: 10.1109/DDECS60919.2024.10508905 [Mil+24]

HOST 2024 Matthias Probst et al. "DOMREP II". in: *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2024, pp. 112–121. DOI: 10.1109/HOST55342.2024.10545417 [Pro+24b]

FDTC 2024 Matthias Probst et al. " Switch-Glitch : Location of Fault Injection Sweet Spots by Electro-Magnetic Emanation ". In: *2024 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*. Los Alamitos, CA, USA: IEEE Computer Society, Sept. 2024, pp. 22–27. DOI: 10.1109/FDTC64268.2024.00011. URL: https://doi.ieeecomputersociety.org/10.1109/FDTC64268.2024.00011 [Pro+24a]

Miscellaneous

IACR Eprint 2022 Michael Gruber and Georg Sigl. *TOFU - Toggle Count Analysis made simple*. Cryptology ePrint Archive, Report 2022/129. https://ia.cr/2022/129. 2022 [GS22]

List of Abbreviations

AD	Associated Data
AE	Authenticated Encryption
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
AFA	Algebraic Fault Analysis
ANF	Algebraic Normal Form
CAESAR	Competition for Authenticated Encryption: Security, Applicability, and Robustness
CMOS	Complementary Metal-Oxide-Semiconductor
CNF	Conjunctive Normal Form
CPA	Correlation Power Analysis
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
DDT	Differential Distribution Table
DES	Data Encryption Standard
DFA	Differential Fault Analysis
DMR	Dual Modular Redundancy
DOM	Domain-Oriented Masking
DOMREP	Domain Oriented Masking with REPetiton codes
DPA	Differential Power Analysis
DRM	Digital Rights Managment
DUT	Device under Test
ECC	Error Correction Code
EDC	Error Detection Code
EM	Electro-Magnetic
EMFI	Electromagnetic Fault Injection
EPFA	Enhanced Persistent Fault Analysis
FA	Fault Analysis
FDT	Fault Distribution Table
FIA	Fault Injection Analysis
FPGA	Field Programmable Gate Array
GCM	Galois/Counter Mode
GHDL	G Hardware Design Language
HD	Hamming Distance
HW	Hamming Weight
IC	Integrated Circuit
IFA	Ineffective Fault Analysis

IoT	Internet of Things
LASCAR	Ledger's Advanced Side Channel Analysis Repository
LFI	Laser Fault Injection
LFSR	Linear Feedback Shift Register
LWC	Lightweight Cryptography
MAC	Message Authentification Code
MPC	Multi Party Computation
MPFA	Multiple Faults-Based Persistent Fault Analysis
NIST	National Institute of Standards and Technology
NLFSR	Nonlinear Feedback Shift Register
NMOS	N-Type Metal-Oxide-Semiconductor
PA	Power Analysis
PC	Programm Counter
PFA	Persistent Fault Analysis
PMOS	P-Type Metal-Oxide-Semiconductor
PRNG	Pseudorandom Number Generator
PRP	Pseudorandom Permutation
REP	Repetition Code
RSA	Rivest-Shamir-Adleman
SAE	Subterranean Authenticated Encryption
SCA	Side-Channel Analysis
SEFA	Statistical Effective Fault Analysis
SEI	Squared Euclidean Imbalance
SFA	Statistical Fault Analysis
SHFA	Statistical Hybrid Fault Analysis
SIFA	Statistical Ineffective Fault Analysis
SNR	Signal to Noise Ratio
SP-Box	Substitution Permutation Box
SPN	Substitution Permutation Network
TA	Template Attacks
TI	Threshold Implementation
TOFU	TOggle Foul-Up
TPM	Trusted Platform Module
TVLA	Test Vector Leakage Assesment
UART	Universal Asynchronous Receiver Transmitter
VCD	Value Change Dump
VHDL	Very High Speed Integrated Circuit Hardware Description Language
XOF	Subterranean-XOF

List of Figures

List of Tables

Contents

1. Introduction

Cryptography has evolved from its beginnings as a kind of secret science into a well-studied field of research. In the beginning there were two approaches to encipher messages: *Transposition Ciphers*, and *Substitution Ciphers*. In the first approach the cipher's output is the same as the input only the order is scrambled by a specific pattern, e.g., *Scytale*, the latter approach substitutes every input by a specific output, e.g., *Caesar Cipher*. While both approaches can be computed by pen and paper, they can also be broken by brute force.

With the broad availability of *modern cryptography* it is no longer possible to break cryptograhy by brute force. Therefore, attacks on the actual implementation of cryptographic algorithms have become an appealing alternative target.

In order to develop new cryptographic standards, cryptography is usually evaluated in rigorous standardization processes, as it was done with the Advanced Encryption Standard (AES), the successor to the Data Encryption Standard (DES). Two recent standardization processes are the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) and the Lightweight Cryptography (LWC) competition.

CAESAR The National Institute of Standards and Technology (NIST) recommends AES in Galois/Counter Mode (GCM) for Authenticated Encryption with Associated Data (AEAD) [Dwo07]. However, in recent years, there were some doubts about the security and ease of side-channel resistant implementations of AES-GCM. Consequently, the CAESAR, was announced in 2013 as an initiative by a group of international cryptographers. Its goal was to identify a portfolio of Authenticated Encryption (AE) schemes that offer advantages over AES-GCM, and are suitable for widespread adoption [Cae]. In three rounds, the number of candidates was narrowed down from over 50 submissions to six finalists; optimized for three use cases: Ascon and ACORN for lightweight applications, AEGIS-128 and OCB for high-speed applications, and Deoxys-II and COLM for applications that require additional resistance against nonce misuse scenarios. With the announcement of the finalists in February 2019, CAESAR provides the most recent, well tested ciphers for authenticated encryption with associated data. Among these finalists, AEGIS, COLM, Deoxys-II, and OCB share the same wells studied permutation $P = MixColumns \circ ShiftRows \circ SubstituteBytes$ as AES. The final CAESAR portfolio announced by NIST: Ascon and ACORN for lightweight applications; AEGIS-128 and OCB for high-performance applications; Deoxys-II and COLM for defense in depth.

LWC With the growing number of interconnected devices and the rise of the Internet of Things (IoT), the need for secure communications is ubiquitous. While today's cryptographic algorithms are well suited for high-end computers such as servers or desktops, their performance decreases dramatically when used on small, resource-constrained devices. Consequently, the NIST launched an open standardization project for lightweight cryptography in 2017 [McK+16]. In 2019, the tremendous amount of 57 round 1 submissions confirms the need and interest of lightweight cryptography. NIST focuses on small, but secure algorithms that provide both AEAD and hash

functionality. Additional features like post-quantum resistance or ease of side-channel and fault-attack resistant implementations are desirable, but not mandatory. In 20023, Ascon was chosen as winner of the LWC as it meets the needs of most use cases where lightweight cryptography is required.

Motivation Physical attacks on cryptographic implementations have become a serious threat over the last years. The reason for this recent trend can be attributed to the fact that modern cryptography has proven to be secure against cryptanalytic attack vectors as modern cryptography undergoes rigorous evaluation processes, e.g., CAESAR and LWC. Typically, the security assumption of the evaluation processes only hold in a black box scenario, where an attacker has no further knowledge about the underlying implementation's state. In contrast, when an implementation is attacked we assume a so-called gray box scenario, i.e., an attacker can observe information through possible side-channels, e.g., power, time, electromagnetic emanation, and also faulty computations. The existence of a side channel allows an attacker to mount Side-Channel Analysis (SCA).

The most commonly exploited side channel is the power side channel which is always present due to the data-dependent power consumption of CMOS logic. In its simplest form Power Analysis (PA) can be mounted by directly measuring the power consumption through a shunt resistor. By the comparison of multiple measurements with a hypothetical power consumption of intermediate values based on a certain key hypothesis, Differential Power Analysis (DPA) allows a direct retrieval of the used secret [KJJ99]. More sophisticated attacks such as Template Attacks (TA) [CRR03] include a profiling phase, where the leakage characteristic of certain values is estimated with the use of a fully controllable device. Typical countermeasures against Power Analysis include masking, which splits the processed intermediate values into several random shares making the power consumption statistically independent of the processed secret [ISW03; GMK16].

Furthermore, an attacker might also be able to disturb the correct processing of a cryptographic algorithm resulting in faulty computations. Faulty computations can also be seen as another kind of side channel that enable an attacker to gain knowledge from the faulty processing of secret data. The exploitation of faulty processed data was proposed by Boneh et al. in their seminal work [BDL00]. Fault Injection Analysis (FIA) usually aims for a modification of the processed data or the control flow during a cryptographic operation in order to reduce the underlying mathematical problem to a simpler one. The most common type of FIA is Differential Fault Analysis (DFA) which requires knowledge of multiple faulted encryptions and a correct one [BS97; BDL97]. In contrast, Statistical Ineffective Fault Analysis (SIFA), as introduced by Dobraunig et al. [Dob+18b], requires only an intermediate state with a biased distribution, i.e., a distribution deviating from the uniform distribution. Even worse, SIFA can break traditional countermeasures against FIA like detection-based or infection-based countermeasures due to its ineffective nature, as this kind of countermeasures assume effective faults.

Objective This work revolves around the FIA of candidates of the CAESAR and LWC competition from a theoretical (fault exploitation) and practical perspective (fault injection), and the protection against FIA. Consequently, this work's objective can be formulated as a set of three questions:

How to *inject* faults on modern microcontroller?

How to *exploit* the observed faulty behavior on the implementation of modern cryptography ?

How to *prevent* fault injection analysis and power analysis simultaneously?

Especially the last question should be given utmost importance, since, e.g., an implementation with power consumption independent of the processed data is still vulnerable to fault attacks.

Organization The remainder of this work is structured as follows:

Chapter 2 introduces the necessary cryptographic background.

Chapter 3 classifies attacks by attacker model, attack vector, and invasiveness.

Chapter 4 provides results of Electromagnetic Fault Injection (EMFI) on a microcontroller.

Chapter 5 introduces the necessary background of Side-Channel Analysis (SCA).

Chapter 6 introduces the necessary background of Fault Injection Analysis (FIA).

Chapter 7 introduces the Differential Fault Analysis (DFA) of KLEIN.
The results presented in this chapter are based on [GS19].

Chapter 8 introduces the Persistent Fault Analysis (PFA) of COLM, Deoxys-II, and OCB.
The results presented in this chapter are based on [GPT19].

Chapter 9 introduces the Statistical Ineffective Fault Analysis (SIFA) of GIMLI.
The results presented in this chapter are based on [GPT20].

Chapter 10 introduces the Algebraic Fault Analysis (AFA) of the Subterranean Authenticated Encryption (SAE) scheme.
The results presented in this chapter are based on [GKS21].

Chapter 12 provides a survey of the state-of-the-art countermeasures against physical attacks.

Chapter 13 introduces Domain Oriented Masking with REPetiton codes (DOMREP) a combined countermeasure for the protection of cryptographic implementations.
The results presented in this chapter are based on [Gru+21].

Chapter 14 introduces TOggle Foul-Up (TOFU) an open-source tool for the toggle analysis of cryptographic implementations.
The results presented in this chapter are based on [GS22].

Chapter 15 concludes this work followed by a brief summary of its main contributions.

2. Cryptographic Preliminaries

As this work revolves around physical attacks on cryptography we will briefly summarize the necessary cryptographic prerequisites which are required by the following chapters.

2.1. Block Ciphers

Symmetric cryptography usually consists of building blocks the smallest of which are usually block ciphers. These so-called block ciphers can also be considered as a keyed bijection with a specified security level, e.g., for AES [FIP01] either 128 bit, 128 bit, and 256 bit. Block ciphers can operate in two directions, i.e., encryption $\mathcal{E}$, and decryption $\mathcal{D}$. Therefore, in a mathematical notation where k denotes the key size, and BS the block cipher's block size block ciphers can be expressed as:

$$\mathcal{E} : \{0,1\}^k \times \{0,1\}^{BS} \rightarrow \{0,1\}^{BS}$$
$$\mathcal{D} : \{0,1\}^k \times \{0,1\}^{BS} \rightarrow \{0,1\}^{BS}$$

A graphical representation of a encryption $\mathcal{E}$ using a key $\mathcal{K}$ is shown in Fig. 2.1, as one can see a plaintext $\mathcal{P}$ is transformed into a ciphertext $\mathcal{C}$. Block ciphers are only able to achieve *confidentiality*, if *integrity* is also required it is necessary to employ AE. Furthermore, it is possible to construct AE schemes from block ciphers, e.g., AES-GCM.

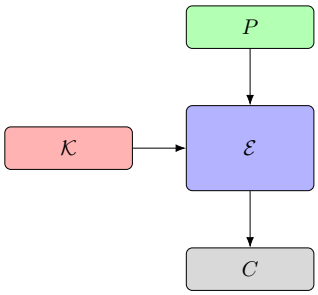


Figure 2.1.: Inputs and Outputs of a Block Cipher

2.2. Authenticated Encryption

Today's communication protocols do not only require *confidentiality*, but also *authenticity*. This can be achieved by combining an encryption scheme with a Message Authentification Code (MAC). AE combines these traditionally separated functionalities into one single algorithm. Additionally, AEAD can process Associated Data (AD), that needs to be authenticated, but must

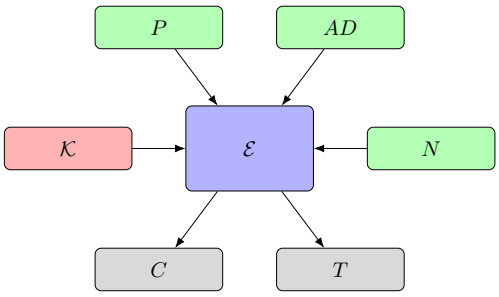


Figure 2.2.: Inputs and Outputs of a AEAD scheme

not be encrypted, e.g., header information in a network protocol. Figure 2.2 shows the inputs and the outputs of an AEAD scheme (encryption), which will be introduced in the following: Formally, let $K \in \{0,1\}^k$ denote a secret key, $N \in \{0,1\}^\nu$ a nonce, $AD \in \{0,1\}^*$ associated data, $P \in \{0,1\}^*$ a plaintext, $T \in \{0,1\}^t$ an authentication tag, and $C \in \{0,1\}^*$ a ciphertext, where $k, \nu, t \geq 1$. Therefore, AEAD is a triple $(\mathcal{K}, \mathcal{E}, \mathcal{D})$, with a key-generation procedure $\mathcal{K}$ that returns a random K, an encryption algorithm $\mathcal{E}_K(N, AD, P)$, and a decryption algorithm $\mathcal{D}_K(N, AD, C, T)$, where $\mathcal{E}$ outputs a tuple (C, T), and $\mathcal{D}$ outputs either the plaintext P or the void symbol $\perp$ if T is invalid:

$$\mathcal{E} : \{0,1\}^k \times \{0,1\}^\nu \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^t$$
$$\mathcal{D} : \{0,1\}^k \times \{0,1\}^\nu \times \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^t \to \{0,1\}^* \cup \{\perp\}$$

Consequently, AEAD is able to ensure *confidentiality*, *integrity*, and *authenticity*.

Part I.

Threats

3. Overview

In the following chapter, physical attacks on cryptography are classified according to attack model, attack vector, attack invasiveness, and attack locality.

3.1. Attack Model

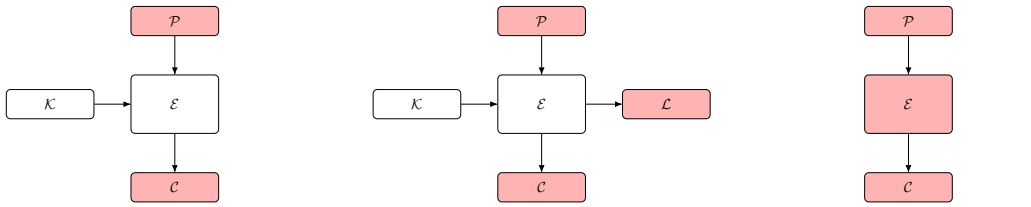
In general, there are three different attack models as shown in Fig. 3.1, where the inputs and outputs observable by an attacker are colored red.

Black Box Model The first attack model is the so-called *black box model* as shown in Fig. 3.1a. In the *black box model* an attacker is capable of observing the input respectively the output of a cryptographic algorithm, e.g., a cipher. This model is also the model which deals with a cryptanalytic attacker which tries to gain knowledge by choosing the input respectively the output of a cipher to his advantage. Common techniques to analyze a cipher are, e.g., linear cryptanalysis as introduced by Matsui et al. [Mat94], and differential cryptanalysis as introduced by Biham et al. [BS93]. Most cryptanalytic approaches share in common that the cipher is assumed to be an oracle which can be queried an arbitrary number of times in order to statistically evaluate possible weaknesses.

Gray Box Model The second attack model is the so-called *gray box model* as shown in Fig. 3.1b. In the *gray box model* an attacker is capable of observing the input respectively the output of a cryptographic algorithm, e.g., a cipher. Additionally, an attacker is able to observe some kind of side-channel information $\mathcal{L}$, which is in the general case partial information about cyrptographic algorithm's state. Side channels are in general the result of the *physical* implementation of a cipher which are not considered in the *black box model*. These side chanels can be of different nature and will be further discussed in Section 3.2. The fact that a possible attacker can access a device which runs an implementation of a cipher enables the possibility to mount physical attacks.

White Box Model The third attack model is the so-called *white box model* as shown in Fig. 3.1c. In the *white box model* an attacker is capable of observing the input respectively the output, and the internals of a cryptographic algorithm, e.g., a cipher. Even though this might sound counter-intuitive on the first-hand white box cryptography is frequently used in Digital Rights Managment (DRM) schemes if a device lacks a secure key storage, e.g., Trusted Platform Module (TPM). The first so-called white box implementation of the DES was proposed by [Cho+03a] but was as expected soon shown to be vulnerable by cryptanalytic approaches by Wyseur et al. [Wys+07]. Chow et al. also proposed a white box implementation of AES in [Cho+03b] which was also shown to be vulnerable to cryptanalytic attacks by Billet et al. [BGEC04].

Consequence Cryptographic implementations violate the assumptions of the *black box model* and must be secured against a *gray box attacker*.



(a) Black Box Model (b) Gray Box Model (c) White Box Model

Figure 3.1.: Attack Models

3.2. Attack Vector

In order to mount an attack which is based on the assumptions of a *gray box model* several attack vectors can be utilized. The most relevant for this work are either FIA or SCA which will be briefly introduced now, and extensively in the following chapters.

Side Channel Analysis According to the *gray box model* an attacker is able to observe some kind of side-channel information $\mathcal{L}$, this side-channel information can be of different nature, e.g., timing information [Koc96] power consumption [KJJ99], electromagnetic emanation [KS05], and acoustic emanation [GST14].

Fault Injection Analysis According to the *gray box model* an attacker is able to observe some kind of side-channel information $\mathcal{L}$, while usually in the literature only passive measurements are considered as a side channel, also the information gathered by faulty computations can be considered as a side channel. Boneh et al. proposed in their seminal work [BDL00] the exploitation of faulty computations.

3.3. Attack Invasiveness

The invasiveness of a physical attack can be taken into account, with three different levels of invasiveness usually being assumed in the state of the art.

Non-invasive Attacks In the non-invasive attack setting an attacker is not required to perform a modification of the Device under Test (DUT). Furthermore, the application of a non-invasive attack is non-destructive, i.e., such attacks do not cause permanent damage to the DUT.

Semi-Invasive Attacks In the semi-invasive attack setting an attacker is required to perform slight modifications of to the DUT. These slight modifications of the device can be, e.g., the removal of decoupling capacitors to mount glitch attacks or sophisticated SCA which requires partial decapsulation of the DUT.

Fully-Invasive Attacks In the fully-invasive attack setting an attacker is required to perform a significant modification of the DUT. These substantial modifications of the device can be, e.g., the removal of a DUT's package to apply probing or forcing or the permanent manipulation of the DUT's functionality in an exploitable manner.

3.4. Attack Locality

The locality of a physical attack can also be taken into account, with the state of the art usually distinguishing between two different levels of locality.

Global Attacks In the global attack setting, an attacker can either observe global behavior or manipulate the DUT globally. If the attack vector is based on SCA an example would be the PA based on a shunt resistor. Contrary, if the attack vector is based on FIA an example would be a power glitch.

Localized Attacks In the localized attack setting, an attacker can either observe local behavior or manipulate the DUT locally. If the attack vector is based on SCA an example would be an attack which utilizes a near field Electro-Magnetic (EM) emanation probe. Contrary, if the attack vector is based on FIA an example would be Laser Fault Injection (LFI).

4. Electromagnetic Fault Injection

Electro-Magnetic (EM) fields can be used to inject faults, therefore a fault injection which is based on this effect is called Electromagnetic Fault Injection (EMFI). EM fields can travel through the packaging materials and thus, removing the package of the chip is not necessarily required. However, doing so helps the attacker recognize the features in the Integrated Circuit (IC), making it easier to find the correct point to induce a fault, i.e., by partially removing the encapsulation of the chip, the attacker is able to identify areas of interest where to inject faults (like memories or buses). Moreover, removing grounded metal plates used in some packages may increase the effectiveness of the attacks as they might act as EM shields. Neve et al. [Nev+03] describe their experiments with a low-cost setup, using a camera flashgun connected to hand-made coils, to generate EM pulses capable of modifying data values in memories and the address bus. A lower cost alternative comes from Schmidt et al. [SH07], who used a spark-gap generator from a gas lighter to manually create high frequency sparks instead of magnetic fields. Due to the high charge change caused by the spark gap a strong EM burst is generated, which can be used to temporarily disrupt the device. They were able to affect the program flow as well as the memory contents (SRAM and Flash). Dehbaoui et al. [Deh+12] presented a more sophisticated EM fault injection setup, capable of producing pulses with low jitter, wide voltage ranges and high accuracy timing. Their setup is composed of a pulse generator, EM coils, and a motorized X-Y-Z table. However, specific details on the equipment used were not given. In order to showcase the capabilities of EMFI we will now introduce an exemplary EMFI setup and apply DFA to KLEIN as proposed in Chapter 7.

4.1. Electromagnetic Fault Injection Setup

An overview of the used setup is shown in Fig. 4.1. A measurement PC is the central component of the setup in charge of controlling the interaction of all the components. As we are using a fault injection technique which relies on localized effects, hence the setup uses a motorized table to position the injection probe at a specific location. The DUT is controlled via a UART interface. Furthermore, we utilize a debugger to gather additional information about the operational state of the DUT. A trigger signal is generated by the DUT and forwarded to a control unit which triggers the pulsed EM emanation. As mentioned before several EMFI setups exist, in order to compare them Toulemont et al. proposed a simple protocol for the comparision of EMFI setups [Tou+20], which we also used in this work. All results of Section 4.2 are obtained using a EMFI setup manufactured by LANGER [LAN]. The most important characteristics as specified by [Tou+20] of the used setup are summarized in Table 4.1.

4.2. Practical Evaluation

In order to demonstrate the capabilities of EMFI we will now apply DFA to KLEIN. The attack's theoretical background will be explained later in Chapter 7. As KLEIN is designed to be a lightweight cipher [GNL12] we opted for a software implementation written in `C` running on a small `ARM Cortex M0` microcontroller. The microcontroller used as DUT is a `STM32F051R8T6`

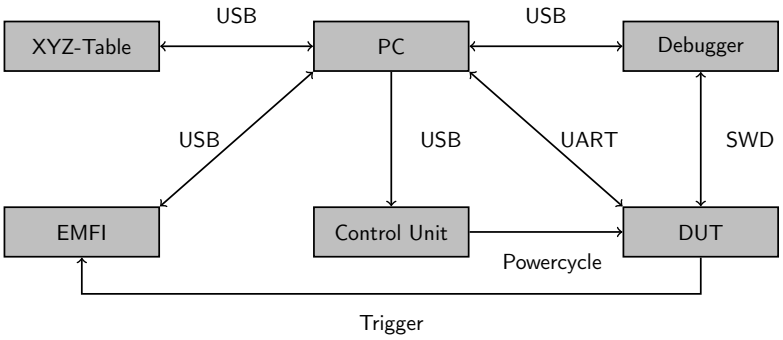


Figure 4.1.: Overview of the EMFI setup.

Dimension	Value
Probe Diameter	500 µm
Magnetic Flux Density	$\leq$ 50 mT
Pulse: Rise Time	$\leq$ 2 ns
Pulse: Rate	0.1 Hz - 20.0 kHz
Pulse: Polarity	+/−/alternating
Pulse: Voltage	$\leq$ 500 V

Table 4.1.: LANGER EMFI Specifications

which features 64 kB of flash memory and 8 kB of SRAM running at a nominal frequency of 8 MHz. In order to minimize the side effects of the induced faulty behavior we disabled peripherals which are not necessary for operation. Generally the decapsulation is optional using EMFI, but we decided to use a decapsulated chip in order to evaluate which components of the chip are prone to a faulty behavior according to our fault model. The decapsulated microcontroller is shown in Fig. 4.3 where all the components are labeled according to [OT17]. A closeup of the EMFI setup's injection coil right above the decapsulated DUT is shown in Fig. 4.2. During the operation of the EMFI setup the injector is lowered to achieve a more localized fault injection.

4.2.1. Attack Settings

During the practical evaluation of the attack we found the following settings in terms of temporal, spatial, and EMFI configuration to work best with our setup. A summary of the used settings for the DFA of KLEIN is shown in Table 4.2. Both attacks as specified in Chapter 7 are feasible with similar settings, where only the step width for the temporal injection locations differs for the attack on the state steps of 250 ns, respectively for the attack on the key schedule 50 ns. This translates into 456 injections at each location (spatial) on the chip for the attack on the state, respectively 108 injections at each location (spatial) on the chip for the attack on the key schedule.

4.2.2. Classification

After finding suitable settings for the injection we decided to classify the observed faulty behavior of the microcontroller into 3 different categories: exploitable faults are faults which occurred

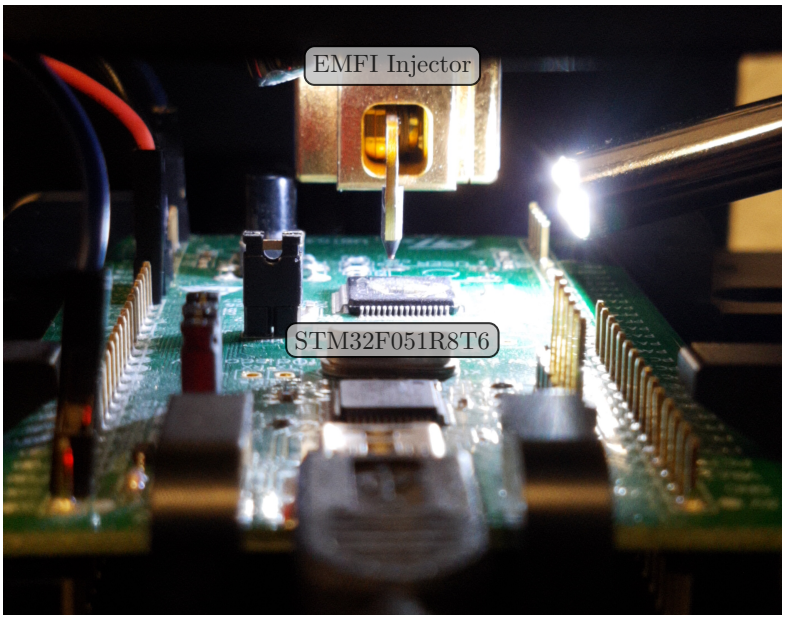


Figure 4.2.: STM32F051R8T6 EMFI Setup

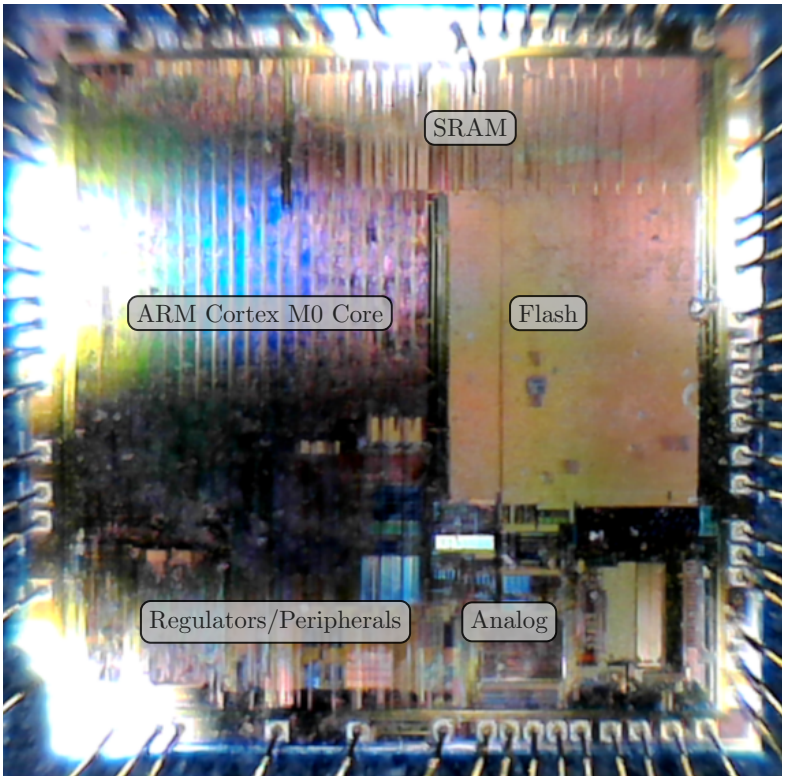


Figure 4.3.: STM32F051R8T6 Chip Layout

Dimension	Value
Time	Vary temporal location in steps of 250 ns (State) Vary temporal location in steps of 50 ns (Key Schedule)
Space	Area of 2.5 mm by 2.5 mm 0.1 mm per step (675 locations)
EMFI	Discharge voltage of 330 V Discharge duration of 10 ns

Table 4.2.: EMFI Parameters DFA KLEIN

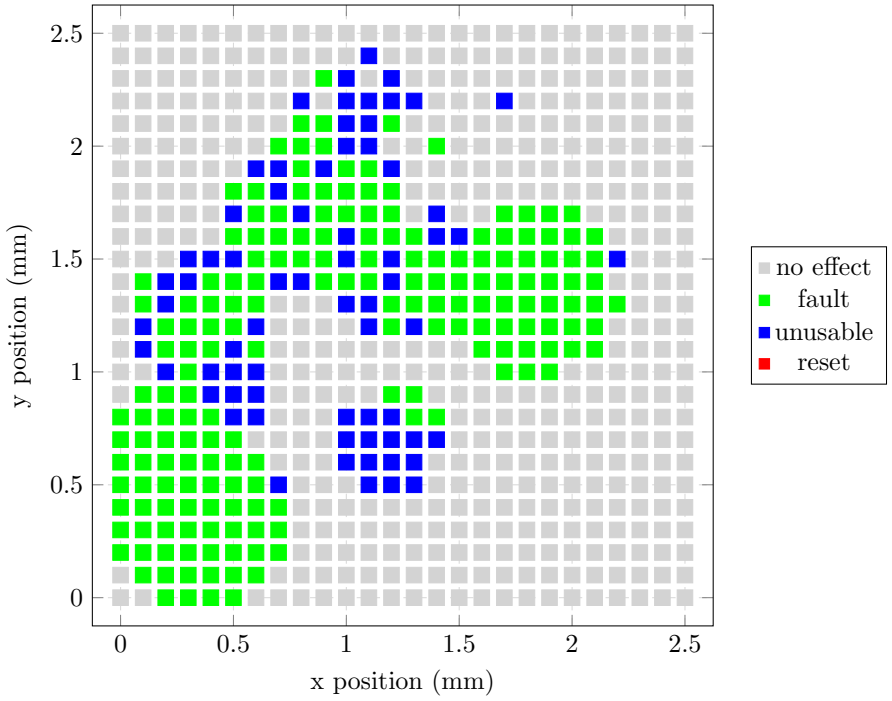
according to the fault model of the attack carried out, unusable faults are faults which do not comply with the fault model, and resets of the microcontroller which can also not be exploited.

4.2.3. Attack Results

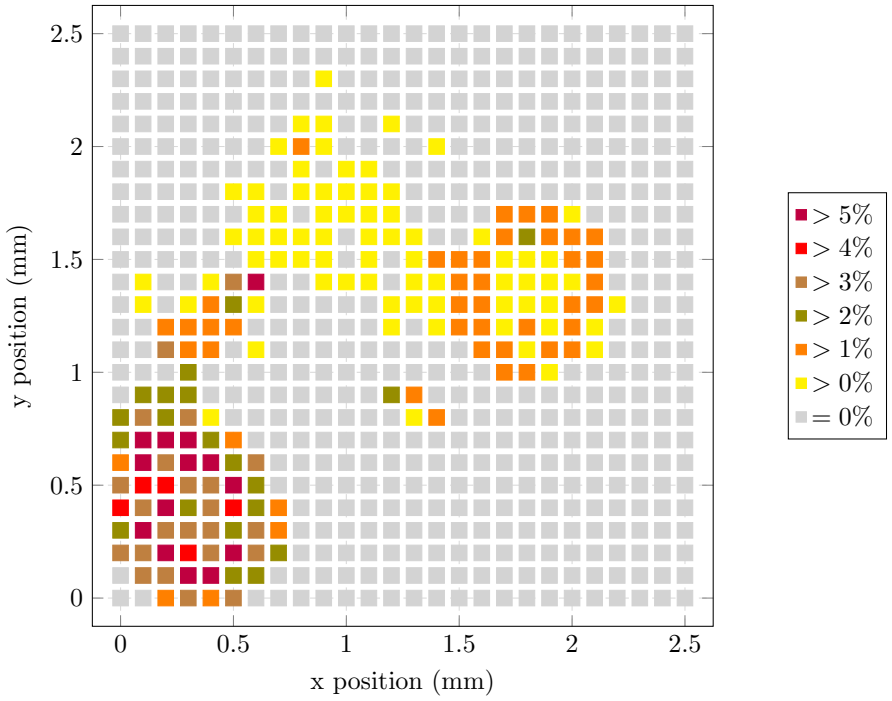
As shown in Figs. 4.4a and 4.5a we were able to conduct the attack on the state and the attack on the key schedule as described in Chapter 7, with a reasonable repeatability as shown in Figs. 4.4b and 4.5b. The fault exploitation probability as shown in Figs. 4.4b and 4.5b is calculated as the number of exploitable faults divided by the total number of injections per spatial location. Furthermore, if the decapsulated microcontroller as shown in Fig. 4.3 is compared with fault injection maps as shown in Figs. 4.4 and 4.5 it becomes clear which components of the chip cause the most faults according to the fault model, namely the ARM Cortex M0 Core, and the Regulators/Peripherals. If the faults are injected into the analog components like ADC, DAC no (observable) faults occur at all.

4.3. Summary

We showcased the capabilities of a commercial EMFI setup with the DFA of KLEIN. The EMFI setup was able to inject localized faults in an ARM Cortex M0 microcontroller namely a STM32F051R8T6, with high spatial precision and percentual repeatability of more than 5 %. The investment for such a setup is much lower than the needed for a professional laser station, and still lower than the one typically needed for side-channel analysis lab equipment. Furthermore, the rise of self build EMFI setups as proposed by [CH17; AH20; O'F19] indicates a trend towards EMFI as a fault injection mechanism. Therefore, EMFI based fault attacks should have a higher impact during risk assessments than what was previously considered reasonable.

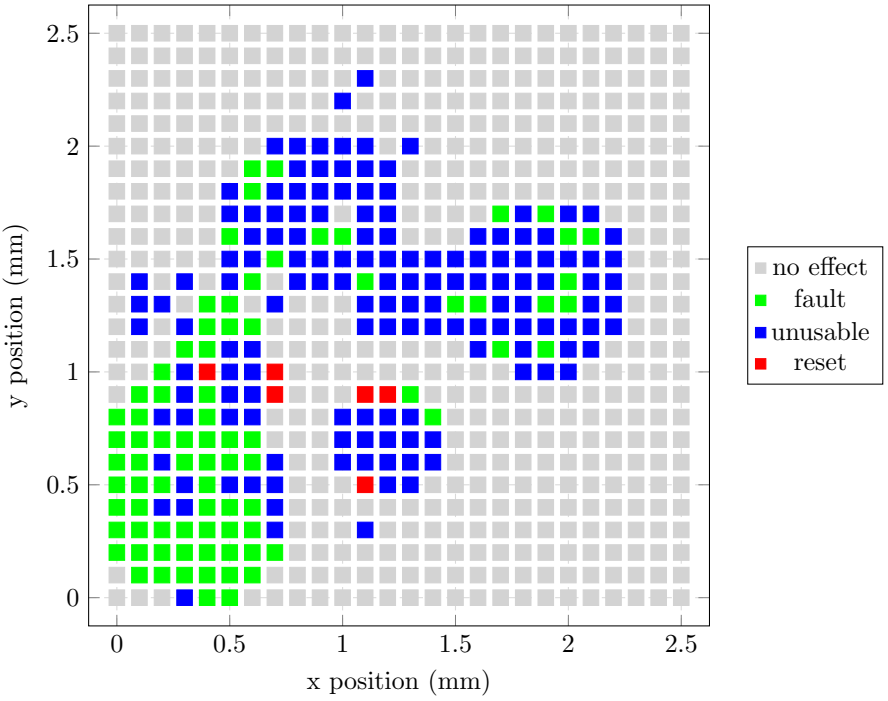


(a) Fault Classification

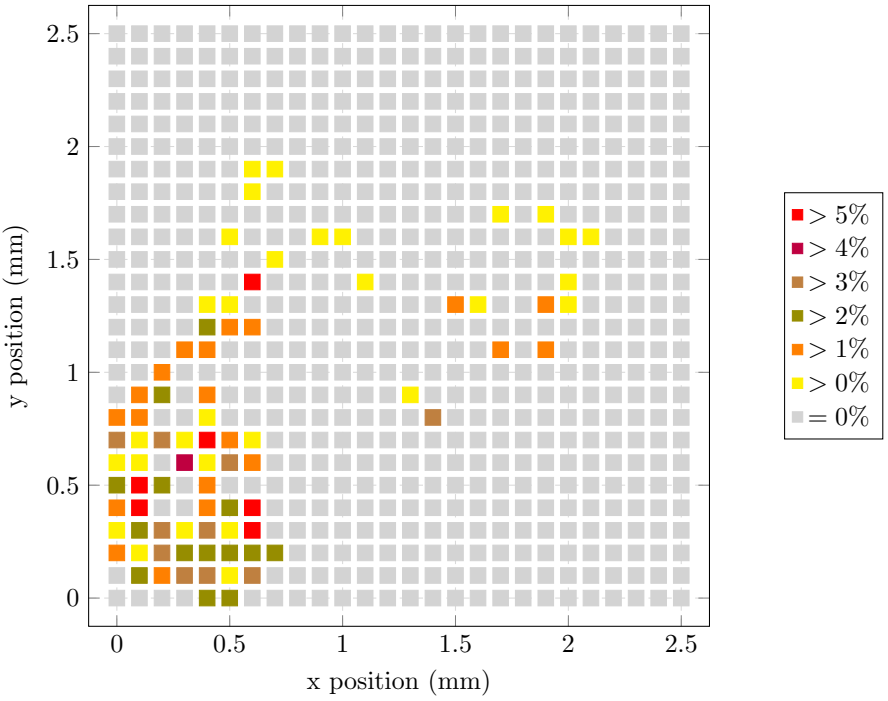


(b) Fault Exploitation Probability

Figure 4.4.: Fault Characterization Map – State



(a) Fault Classification



(b) Fault Exploitation Probability

Figure 4.5.: Fault Characterization Map – Key Schedule

5. Side Channel Analysis

This chapter introduces the necessary preliminaries of SCA, i.e., leakage models, power consumption of Complementary Metal-Oxide-Semiconductor (CMOS) circuits, Test Vector Leakage Assesment (TVLA), and Correlation Power Analysis (CPA).

5.1. CMOS Power Consumption

According to [Ins97] the power consumption P of CMOS circuits can be divided into static power consumption P_S and dynamic power consumption P_D as shown in Equation (5.1). Static leakage current occurs since N-Type Metal-Oxide-Semiconductor (NMOS) and P-Type Metal-Oxide-Semiconductor (PMOS) transistors are not perfectly switching. The static power consumption can be calculated as the product of the leakage current I_{CC} and the supply voltage V_{CC} and is therefore assumed to be independent of switching activities. The dynamic power consumption P_D can be divided into transient power consumption P_T and capacitive-load power consumption P_L which is proportional to the output's capacitive load C_L. The dynamic power dissipation capacitance C_{pd} is defined as the equivalent internal capacitance of a device calculated, by measuring the operating current without load capacitance. Transient power consumption occurs when the input to the gate changes which is indicated by f_I. Switching the value of the gate output leads to the charging or discharging of the output capacitance which is indicated by f_O. Furthermore, $N_{switches}$ denotes the number of switching inputs or outputs. Although static power consumption is significantly less than dynamic power consumption, static power consumption can also cause side-channel leakage, as shown by Khairallah et al. [Kha+18].

$$\begin{aligned} P &= P_S + P_D \\ P_S &= V_{CC} \times I_{CC} \\ P_D &= P_T + P_L \\ P_T &= C_{pd} \times {V_{CC}}^2 \times f_I \times N_{switches} \\ P_L &= C_L \times {V_{CC}}^2 \times f_O \times N_{switches} \end{aligned} \tag{5.1}$$

5.2. Leakage Models

A simplification of the power consumption calculation outlined in Section 5.1 are leakage models [BCO04]. These leakage models directly translate processed data into a hypothetical power consumption, the most common ones are the Hamming Weight (HW), and the Hamming Distance (HD). The calculation of the HW, and the HD for processing the data sequence $\{D_1, D_2, ..., D_m\}$ with n-bit data words $D_i = d_1^{(i)} d_2^{(i)} ... d_n^{(i)}$ is shown in Eq. (5.2), where every $d_n^{(i)}$ represents a single bit.

$$\begin{aligned} HW(D_i) &= \sum_{j=1}^{n} d_j^{(i)} \\ HD(D_i, D_{i+1}) &= HW(D_i \oplus D_{i+1}) \end{aligned} \tag{5.2}$$

The most common leakage model used is the HW as it requires no knowledge about previously processed data in contrast to the HD cf. Eq. (5.2).

5.3. Correlation Power Analysis

DPA is a type of side-channel attack proposed in 1999 by Kocher et al. [KJJ99] and is based on partitioning. A similar attack called CPA which uses correlation as a distinguisher was proposed by Brier et al. in 2004 [BCO04]. For a CPA attack, the power consumption of a cryptographic operation is measured multiple times, e.g., different plaintexts with the same key for AES. Next, an intermediate value depending on known values, as well as an unknown key byte is chosen. A possible intermediate value t during the encryption of AES is the output of the first round's S-box, i.e., $t = S(k_0^{(0)} \oplus p_0)$. For all key hypotheses of the key byte $k_0^{(0)}$, the intermediate value is calculated. By the application of a leakage model to the intermediate value, the correlation of all key hypotheses with the measured power traces can be calculated. The correct key byte is then identified by the hypothesis which results in the highest absolute correlation.

5.4. Test Vector Leakage Assessment

In order to examine the resistance of an implementation against Side-Channel Analysis (SCA) the presence of possible side-channel information, also called "leakage", has to be evaluated. Methods based on calculating the Signal to Noise Ratio (SNR) [Man04] or the correlation [DS16] of intermediate variables require assumptions about exploitable intermediate states or a specific hypothesis model. Another commonly used approach is Test Vector Leakage Assesment (TVLA) a methodology that uses Welch's t-test to statistically evaluate the presence of side-channel leakage does not require prior knowledge about the investigated implementation or intermediate values [Goo+11; SM15]. Welch's t-test evaluates whether the distributions of two sets $\mathcal{Q}_0$ and $\mathcal{Q}_1$ with respective means μ_0 and μ_1 and variances s_0 and s_1 differ significantly from each other. The resulting *t-value* is calculated as

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}}, \tag{5.3}$$

where n_0 and n_1 denote the respective cardinalities of the sets. A high t-value corresponds to a low probability to accept the null hypothesis that both sets were drawn from the same distribution. Usually a threshold of $|t| > 4.5$ is defined to reject the null hypothesis with a confidence of > 0.99999 and is taken as an indicator for possible side-channel leakage.

For leakage evaluations the non-specific or fixed-vs.-random t-test can be applied: The evaluator measures the power consumption of multiple algorithm executions with a fixed secret, while randomly choosing the input data of subsequent measurements to have a fixed or random value. Measurements are then split into a set $\mathcal{Q}_0$ with fixed input data and a set $\mathcal{Q}_1$ with random input data. Finally, the t-value according to Eq. (5.3) is calculated for each point in time. If the result indicates that both sets can be distinguished ($|t| > 4.5$), the implementation exhibits

side-channel leakage, which can potentially be used to mount an SCA attack. Otherwise, the implementation can be considered secured for first-order univariate attacks with the evaluated amount of traces. As the t-value is computed for each measurement sample individually it allows for identifying points of interest for an attack.

In order to evaluate univariate higher-order leakage and therefore the side-channel resistance against higher-order attacks, Eq. (5.3) can be extended by data preprocessing. For hardware implementations, where data is processed in parallel, mean-free squared measurements $X' = (X - \mu)^2$ are a suitable preprocessing to reveal univariate second-order side-channel leakage. In addition, it is also possible to directly compute the higher-order statistical moments as described in [SM15].

Part II.

Attacks

6. Overview

The origin of the research field FIA is based on the observation of May and Woods that radioactive radiation produced by the elements that a chip's package is made from causes undesirable behavior [MW78]. The deliberate exploitation of such faults was proposed by Boneh et al. in their seminal work [BDL97]. This approach was later refined by Boneh et al. and thus the first fault attack was applied to the Rivest-Shamir-Adleman (RSA) asymmetric cryptosystem which uses the Chinese Remainder Theorem (CRT) for faster computation [BDL00]. In this attack, a single faulty signature is sufficient to recover the entire private key. The fact that a single faulty signature is enough to break RSA implementations was an early indicator that FIA is a powerful type of attack. In the following, the most important types of FIA are presented in short form, each of which will be applied to a cipher in the remainder of this work.

6.1. Differential Fault Analysis

The most common type of FIA is the so-called Differential Fault Analysis (DFA), which is based on the approach of evaluating tuples of correct and faulty encryptions. This type of fault attack was first applied to DES by Biham and Shamir [BS97]. The proposed attack was able to recover the whole DES key using between 50 and 200 tuples of correct and faulty encryptions. A similar attack was conducted by Piret and Quisquater on AES, where the complete key can be recovered after only two tuples of correct and faulty encryption [PQ03]. Furthermore, Tunstall et al. were even able to recover a complete AES key with a single fault [TMA11].

The simplest representation of the last round of a Substitution Permutation Network (SPN) based cipher is shown in Fig. 6.1, similar to the final round of AES, where the last *AddRoundKey* takes place after *SubBytes*. In order to recover the unknown value X an attacker can exploit knowledge about the substitution layer, i.e., S-box. To do so an attacker performs the same encryption twice one time without the influence of a fault which results in X, and a second time under the influence of fault prior to the S-box which results in a faulty value $\overline{X}$ as shown in Eq. (6.1).

$$\begin{aligned} X &= S^{-1}(Y) + K \\ \overline{X} &= S^{-1}(\overline{Y}) + K \end{aligned} \tag{6.1}$$

The calculation of the output differential ΔY (sum on GF(2)) does not require any prior knowledge nor hypotheses as it only depends on a correct ciphertext C and a faulty one $\overline{C}$ as shown in Eq. (6.2).

$$\Delta Y = \Delta C = C + \overline{C} \tag{6.2}$$

Subsequently, the attacker computes the difference ΔX of a correct X and a faulty encryption $\overline{X}$ as shown in Eq. (6.3), as one can see the key K cancels itself out as addition is self-inverse.

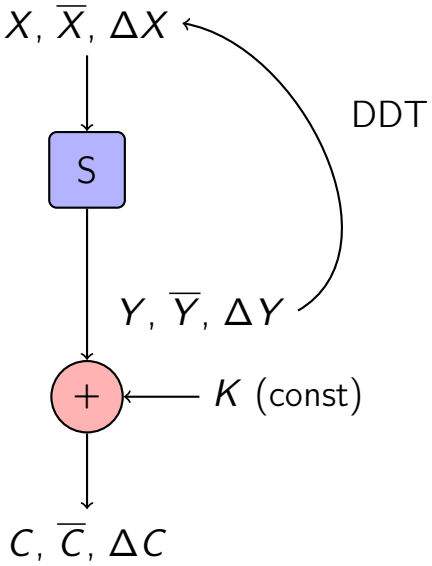


Figure 6.1.: Substitution Layer DFA

$$\begin{aligned} \Delta X &= X + \overline{X} \\ \Delta X &= S^{-1}(Y) + S^{-1}(\overline{Y}) \\ \Delta X &= S^{-1}(Y) + S^{-1}(Y + \Delta Y) \end{aligned} \tag{6.3}$$

Finally, the information based on the Differential Distribution Table (DDT) is exploited to sort out impossible intermediate states, i.e., discard hypotheses of X which result in impossible tuples of differentials ΔX and ΔY. By the repetition of this approach an attacker ends up with a unique intermediate state which is in fact the correct intermediate state. The correct intermediate state can then be used to calculate the target partial subkey, i.e., the last round key.

6.2. Persistent Fault Analysis

Another type of FIA is Persistent Fault Analysis (PFA) which was introduced by Zhang et al. as an approach to attack block ciphers that require repeated access to stored constants [Zha+18]. In contrast to DFA where each execution of a cryptographic function (e.g. each encryption) requires a fault injection, for Persistent Fault Analysis (PFA) it is sufficient to inject a single, persistent fault. While differential fault attacks alter a computed, intermediate value, PFA aims to alter a (stored) constant, e.g., a S-box entry, or to manipulate the generation of a constant, e.g., the computation of a S-box. This has the advantage of not requiring multiple fault injections during runtime, but only one during the generation of the constant which now is under the influence of a persistent fault. A simplified substitution layer of a SPN based cipher is shown in Fig. 6.2. The input of the b-bit wide S-box S is x_j, i.e., the j^{th} word of the state. The output of the S-box is y_j. The involved round key and the resulting ciphertext are called k_j and c_j, respectively. This can also be expressed as $c_j = k_j + y_j$, where $y_j = S[x_j]$. Furthermore, the distribution probability of the S-box's output and the ciphertext is denoted by $P(y_j)$ and

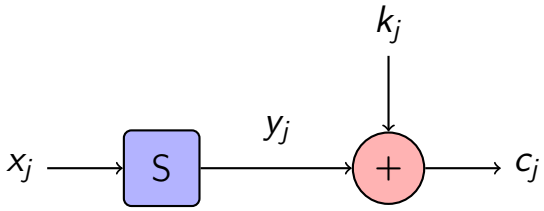


Figure 6.2.: Substitution Layer PFA

$P(c_j)$, respectively. For a well-designed cipher we can assume: $P(y_j = i) = 2^{-b}\ \forall\ i \in [0, (2^b - 1)]$. If one element of the S-box is faulted, e.g., $S[0]$ the former correct value $S[0] = v$ is changed to $S^*[0] = v^*$ where $v \neq v^*$. For the remainder of this section we assume the attacked S-box is the one used by AES where $b = 8$. Consequently, the probability $P(y_j = v)$ is zero, and the probability $P(y_j = v^*)$ is doubled to $2^{1-b} = \frac{2}{256}$. All other values y_j still have the same probability $P(y_j) = 2^{-b} = \frac{1}{256}$. Since the key k_j is fixed, the distribution of y_j and c_j are related, which is expressed in Eq. (6.4).

$$P(c_j) = P(y_j + k_j). \tag{6.4}$$

Let $t \in [0, (2^b - 1)]$ denote each possible value of c_j. Then, the value of t occurring least frequent and most frequent are denoted t_{min} and t_{max}, respectively.
There are three possible attack strategies:

1. As t_{min} can be determined and the original value of the S-box v is known, the key can be extracted with the following equation:

$$k_j = v + t_{min} \tag{6.5}$$

2. The values of $t \neq t_{min}$ can be used to eliminate impossible key candidates, under the assumption v is known:

$$k_j \neq v + t \tag{6.6}$$

3. If v^* is known, t_{max} can be used to determine k_j:

$$k_j = v^* + t_{max} \tag{6.7}$$

The first and the second strategy are analytical approaches. In contrast, strategy three is a statistical approach, which requires enough faulty ciphertexts to ensure t_{max} converges 2^{b-1}. An example where the probabilities of all possible values t in relation to the number of encryptions for a faulty S-box S^* (with one faulty entry) is shown in Fig. 6.3. The red line corresponds to the original value v. The blue line corresponds to v^*. The gray lines represent all other values. It can be seen, that the relative frequency of v^* approaches asymptotically the calculated probability $\frac{2}{256}$ and v is constant zero. All other values stabilize at $\frac{1}{256}$. It must be noted, that all strategies require the structure depicted in Fig. 6.2 for the last round. If there is an additional permutation layer in the last round, appropriate measures must be taken to counteract the effect of the permutation layer as we will show in Chapter 7. In summary PFA is able to extract the round key k which is added right after the substitution layer (during the last round), as shown in Fig. 6.2. If the addition of the key is followed by the addition of a variable, PFA is only able to recover the sum of the variable and the key, we denote this by R. However, we will show how to further process R to recover the actual key in Chapter 8.

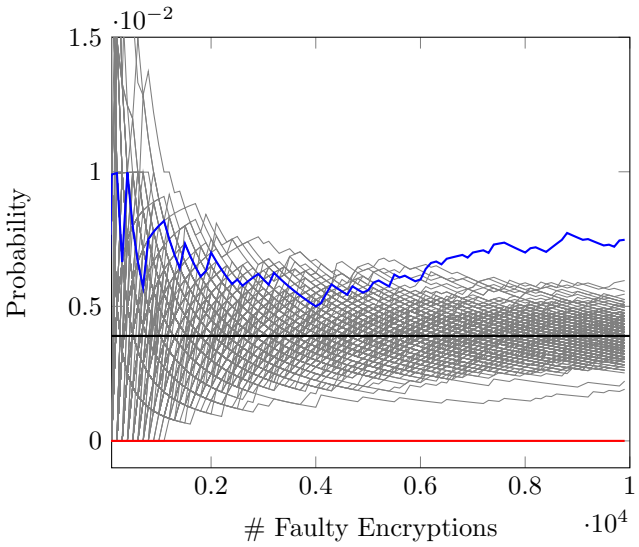


Figure 6.3.: Distribution of a faulty S-box S^*

6.3. Statistical Ineffective Fault Analysis

Another type of FIA is Statistical Ineffective Fault Analysis (SIFA) which was proposed by Dobraunig et al., originally intended to attack symmetric cryptography [Dob+18b]. SIFA can be seen as the combination of Ineffective Fault Analysis (IFA), as proposed by Clavier and Christophe [Cla07], and Statistical Fault Analysis (SFA) as proposed by Fuhr et al. [Fuh+13]. The basics of SIFA are explained in detail below.

Background As SIFA combines the advantages of its predecessors IFA and SFA we will now briefly compare them in terms of the required fault model, key recovery, and the ability to overcome countermeasures.

Fault Model Both attacks differ in their assumed fault model. The required fault model of IFA, is rather specific, e.g., Clavier et al. proposed a fault model where the computation of an XOR results always in a zero value [Cla07]. By forcing the output of an operation to a specific value, the attacker can distinguish, if the output of the faulty operation is equal to the fault free output. Subsequently, if the faulted output equals the correct output, an ineffective fault occurred. Figure 6.4 (IFA) shows this behavior, where the computation of an XOR always returns zero as in [Cla07]. The downside of the assumed fault model is the fact that the required fault model is difficult to achieve in practice, especially with low-cost equipment cf. Guillen et al. [GGS17]. In contrast, the assumptions of the required fault model for a SFA are loose as the only requirement for a successful SFA is a biased intermediate state as shown by Fuhr et al. [Fuh+13]. As shown in Fig. 6.4 (SFA) an attacker injects a fault after the computation of f_1 and before f_2, where $f_i \mid i \in \{1, 2\}$ denotes parts of the cryptographic operation.

Key Recovery Both attacks differ significantly in how they recover the key. In IFA the recov-

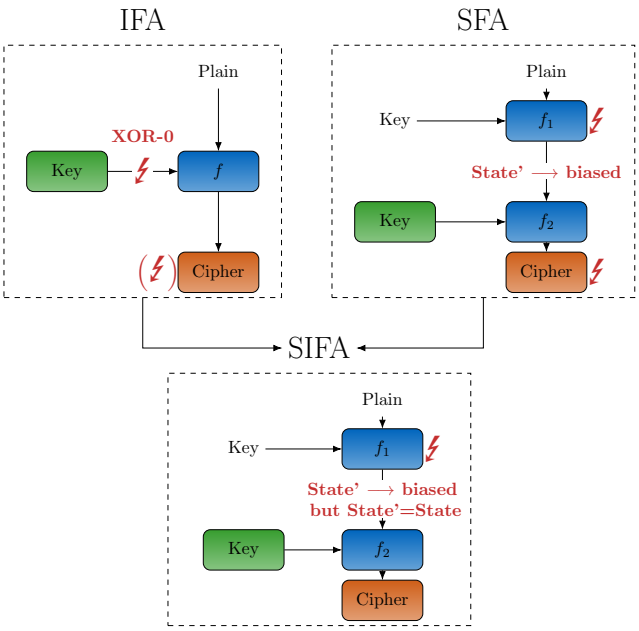


Figure 6.4.: SIFA Background

ery of the correct target partial sub key is strictly analytical. Contrary, in SFA a statistical approach is used where the deviation from the uniform distribution is used as metric to distinguish the correct target partial subkey. One of the main benefits of the statistical approach in SFA, is the immunity to noisy faults, i.e., injections that do not comply with the required fault model.

Countermeasures Generally, there are two possible approaches to the design of countermeasures against FIA: detection-based countermeasures and infection-based countermeasures [PCM15; Zha+16b]. The most common form of detection-based countermeasures is temporal redundancy, where an encryption or decryption is performed twice. If the results do not match, a fault occurred and appropriate measures like a system-reset can be taken. The infection-based countermeasure applies additional operations in order to increase the fault propagation to a level where an attack is no longer feasible. As a result, IFA is able to overcome the most common countermeasures against fault attacks as the fault does not alter the result of the computation. In contrast, SFA can be thwarted by countermeasures based on either detection or infection as the injected fault does alter the result of the computation.

Foundations of Statistical Ineffective Fault Analysis SIFA evaluates the statistical distribution of intermediate values and identifies appropriate key candidates with a statistical distinguisher. We assume that a fault only corrupts a part s of an intermediate state. The partial intermediate state after a fault injection is denoted by s'. The alteration of $s \rightarrow s' \mid s \neq s'$ results in a faulty computation and will affect the outcome of the cryptographic algorithm. However, an alteration $s \rightarrow s' \mid s' = s$ does not affect the outcome of the cryptographic algorithm. This type

of alteration is referred to as an ineffective fault. Such ineffective faults can be exploited, if they cause a biased distribution in the intermediate value.

We now introduce the reasoning behind a biased distribution of an intermediate value. If we assume a single bit intermediate value denoted by s prior to a fault injection and consequently s' the same intermediate value after a fault injection. The deviation from the uniform distribution depends on how the fault is coupled into the intermediate value. For our example we assume a coupling based on a random logical `OR`, i.e., $S' = S \vee F$. Furthermore, we can assume that the cryptographic algorithm under attack is a Pseudorandom Permutation (PRP), so we can also assume that Eq. (6.8) holds, i.e., the probability of the intermediate value s and the fault f being either zero or one is equal.

$$\begin{aligned} P(S=0) &= P(S=1) &&= 0.5 \\ P(F=0) &= P(F=1) &&= 0.5 \end{aligned} \tag{6.8}$$

Consequently, the only case where $s' = 0$ holds is the specific case where $s = 0$ and $f = 0$, in all other cases $s' = 1$. The according probabilities for all the cases are shown in Eq. (6.9).

$$P(S'=0) = P(S=0, F=0) = 0.25$$

$$\begin{aligned} P(S'=1) &= 1 - P(S'=0) \\ P(S'=1) &= P(S=0 \wedge F=1) + P(S=1 \wedge F=0) + P(S=1 \wedge F=1) \\ P(S'=1) &= 0.75 \end{aligned} \tag{6.9}$$

As one can see the probability for the intermediate value after a fault injection s' to be zero is lower by a factor of three compared for the probability of the state to equal one. If this reasoning is extended to multiple bit states the transition probabilities can be visualized with Fault Distribution Tables (FDTs).

The six exemplary FDTs in Table 6.1 show the transition probability of two-bit intermediate values for six typical fault models. The FDTs shown in Table 6.1 are generated under the assumption of a *Random Or* coupling as introduced above, a *Random And* coupling where $S' = S \wedge F$, a *Stuck at Zero* where $S' = 0$, a *Probabilistic Bit Flip* coupling where $S' = S + F$, a *Random Fault* coupling where $S' = F$, and a *Bit Flip* coupling where $S' = S + 1$.

In order to apply SIFA, the diagonal of such a table (marked in blue) must differ from the uniform distribution. This holds true for Tables 6.1a to 6.1d. The table's entries which are not colored blue denote the probability of effective faults.

By injecting a fault in an operation an intermediate value of n-bit is affected. The intermediate value is represented by the two random variables S and S', before and after the injected fault. This means the intermediate value is denoted by the random variable S when no faults are present and S' otherwise. Both random variables can take values $s \in \mathcal{S} = \{0, ..., 2^n - 1\}$. The probabilities of the individual entries of the FDT are calculated as shown in Eq. (6.10), where s corresponds to the values in the rows and s' to the values in the columns of Table 6.1.

$$p_s(s') := P(S' = s' \mid S = s). \tag{6.10}$$

The elements of the diagonal of an FDT correspond to the probabilities for different ineffective faults as shown in Eq. (6.11).

$$p_{s'}(s') := P(S' = s' \mid S = s'). \tag{6.11}$$

We assume the FDT is not known to the attacker. Nevertheless, it is still possible to exploit the diagonal's deviation from the uniform distribution. Since we assume the presence of a (detection-based) countermeasure we will state the statistical model explicitly for this scenario. Here, the

(a) Random Or

s \ s'	00	01	10	11
00	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
01	0	$\frac{1}{2}$	0	$\frac{1}{2}$
10	0	0	$\frac{1}{2}$	$\frac{1}{2}$
11	0	0	0	1

(b) Random And

s \ s'	00	01	10	11
00	1	0	0	0
01	$\frac{1}{2}$	$\frac{1}{2}$	0	0
10	$\frac{1}{2}$	0	$\frac{1}{2}$	0
11	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$

(c) Stuck at Zero

s \ s'	00	01	10	11
00	1	0	0	0
01	1	0	0	0
10	1	0	0	0
11	1	0	0	0

(d) Probabilistic Bit Flip

s \ s'	00	01	10	11
00	$\frac{4}{9}$	$\frac{2}{9}$	$\frac{2}{9}$	$\frac{1}{9}$
01	$\frac{4}{9}$	$\frac{2}{9}$	$\frac{2}{9}$	$\frac{1}{9}$
10	$\frac{4}{9}$	$\frac{2}{9}$	$\frac{2}{9}$	$\frac{1}{9}$
11	$\frac{4}{9}$	$\frac{2}{9}$	$\frac{2}{9}$	$\frac{1}{9}$

(e) Random Fault

s \ s'	00	01	10	11
00	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
01	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
10	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
11	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$

(f) Bit Flip

s \ s'	00	01	10	11
00	0	0	0	1
01	0	0	1	0
10	0	1	0	0
11	1	0	0	0

Table 6.1.: FDTs of 2-bit variables

attacker has only access to samples where the intermediate values under attack fulfill $S \equiv S'$. Under the assumption, that S is uniformly distributed with $\mathrm{P}(S = s) = 2^{-n} = \frac{1}{|\mathcal{S}|}$, the rate of ineffective faults r_{ineff} can be calculated as shown in Eq. (6.12).

$$r_{\text{ineff}} = P(S' \equiv S) = \sum_{s' \in \mathcal{S}} \frac{p_{s'}(s')}{|\mathcal{S}|} \tag{6.12}$$

The diagonal of the FDT can be expressed as conditional distribution as shown in Eq. (6.13). This distribution is later estimated by an attacker, as neither the diagonal nor S' can be observed.

$$p_{\text{ineff}}(s') = P(S' = s' \mid S' \equiv S) = \frac{p_{s'}(s')}{|\mathcal{S}| \cdot r_{\text{ineff}}} \tag{6.13}$$

However, it is possible to calculate the hypothetical distribution p_H of S'_H under the assumption of a fixed key hypothesis k_H. By using the correct key guess k_H the correct distribution $p_{\text{ineff}}(s') = p_{H=\text{correct}}(s')$ is observed. In order to distinguish it from incorrect key guesses, we assume that all distributions of incorrect key hypotheses $k_{H=\text{wrong}}$ are close to the uniform distribution, i.e., $p_{H=\text{wrong}}(s') \approx \theta(s') = 2^{-n}$. The distinguisher $D(p_H)$ is used to rank the key candidates according to their distance to $\theta(s')$. The chi-squared (χ^2) test as introduced by Pearson [Pea00], is used to calculate a metric for the difference of two probability distribution

function a and b, both with values $x \in \mathcal{X}$ and N samples as shown in Eq. (6.14)

$$\chi^2(a,b) = N \sum_{x \in \mathcal{X}} \frac{(a(x) - b(x))^2}{b(x)}. \tag{6.14}$$

Since incorrect key guesses follow the uniform distribution, we can use the χ^2 metric to distinguish distributions resulting from the correct key hypothesis $p_{H=\text{correct}}$ and a uniform distribution θ caused by an incorrect hypothesis. This leads to the χ^2-distinguisher as shown in Eq. (6.15).

$$D(p_{H_i}) = \text{CHI}(p_{H_i}) := \chi^2(p_{H_i}, \theta) \tag{6.15}$$

Alternatively, a scaled version of CHI, the Squared Euclidean Imbalance (SEI) [Riv09] as shown in Eq. (6.16), can be used.

$$D(p_{H_i}) = \text{SEI}(p_{H_i}) \ = \frac{1}{|\mathcal{S}| \cdot N} \cdot \text{CHI}(p_{H_i}) \tag{6.16}$$

Where N denotes the number of observed decryptions under the influence of ineffective faults.

SIFA Fault Models In general, SIFA's fault model can be divided into two versions, viz. SIFA-1 and SIFA-2 [Sah+20]. Both versions of SIFA differ in their assumption of how and where a fault is injected:

The SIFA-1 fault model assumes that a fault injection causes a statistical bias in n shares of a masking scheme which affects the state variable, without affecting all shares of a bit simultaneously. Masking schemes are frequently used as a countermeasure against SIFA-1 cf. Section 12.3.

In contrast, the SIFA-2 fault model assumes a random fault during the computation of a cryptographic algorithm's sub-function, e.g., the computation of an S-box which means that all shares of a masked implementation are affected.

6.4. Algebraic Fault Analysis

Algebraic cryptanalysis is based on the idea to express a cipher as an equation system, and solve it which is the equivalent of breaking a cipher. Back in 2002, Courtois and Pieprzyk were able to show, that the conversion of a cipher into an overdefined system of algebraic equations poses a threat to cryptographic systems [CP02]. In fact, an overdefined equation system is easier to solve and might lead to a sub-exponential increase in security with a growing number of rounds of an iterated cipher. As a consequence of [CP02], increasing the round function's complexity or the number of rounds is recommended to ensure that a cipher cannot be broken by brute-force within a feasible amount of time. Consequently, additional information is required to reduce the complexity of the equation system to solve it in a reasonable amount of time.

In Algebraic Fault Analysis (AFA) this additional information in the form of equations is gathered from knowledge based on fault injections which characterize the fault injection and fault propagation. If the additional equations are combined with the equations gathered from the cipher representation, the additional equations ensure that the whole equation system can be solved in a feasible time, this approach is called AFA.

In order to automatically solve the algebraic representation of a cipher a combination of tools can be used: *BOSPHORUS* [Cho+18] to convert the Algebraic Normal Form (ANF)-representation into a Conjunctive Normal Form (CNF)-representation while simultaneously optimizing the equation system, and *CryptoMiniSat* [SNC09] to solve the optimized system.

7. Differential Fault Analysis of KLEIN

Exchange of information in computer networks often requires the use of cryptography, to ensure the integrity of messages, the confidentiality of the message or the authenticity of the communication partner. However, the computational effort caused by cryptographic algorithms can be prohibitive for resource constrained devices. A typical example for this class of devices are IoT devices. These are often low-power sensor nodes, which are deployed over a large area and submit measurement data to some back-end system. Each node is battery powered and is thus very limited in its energy consumption. For these applications *lightweight block ciphers* were developed in recent years. The idea is to offer a symmetric block cipher (since asymmetric ciphers are always more costly in terms of performance), with a security level that does not have many reserves, but at a much smaller computational overhead. The most prominent example for lightweight block ciphers is PRESENT [Bog+] but many other proposals have been developed, e.g., KLEIN as proposed by Gong et al. [GNL12]. In this chapter we apply DFA to the lightweight block cipher KLEIN.

State of the art To the best of our knowledge, there are no publications about the DFA of KLEIN which targets either an intermediate state or an intermediate round key[1]. In contrast, Yoshikawa et al. developed a generic attack based on the manipulation of the control flow [Yos+], where an attacker aims to increase the number of rounds artificially using a fault injection. As shown by Yoshikawa et al. this attack requires one faulty ciphertext and one correct ciphertext to recover the last round key of KLEIN with a key size of 64 bit.

Contribution We introduce two different DFAs on the lightweight cipher KLEIN. The first fault attack requires an attacker to inject faults into the state of the encryption process. This attack method works on all variants of KLEIN. Furthermore, we present a second attack on the key schedule, which works only on the variant of KLEIN using a 64 bit key. This attack enables the attacker to determine the key with 4 fault injections. For both attacks we prove the according efficiency by means of simulations.

Organization Chapter 7 is structured as follows: In Section 7.1 the basic working principle of the KLEIN cipher is described. Section 7.2 explains the attack based on the fault injection into the encryption, whereas Section 7.3 explains the attack on the key schedule of KLEIN-64. A discussion of the performance of both attack strategies is given in Section 7.4. Section 7.5 concludes the DFA of KLEIN.

[1] There are two additional publications in chinese: A DFA by Wang et al. [WRZ16] and a DFA by Cunyan et al. [CYX15]. However, the latter obviously uses the generic approach of injecting single-bit faults before the last S-box operation to exploit the differential distribution table (cf. the appendix of the original KLEIN publication [GNL12]) and discard key hypotheses which lead to impossible differentials.

7.1. KLEIN

KLEIN [GNL12] is a SPN-based cipher similar to other state-of-the-art block ciphers (e.g. AES or PRESENT) and features three different security levels with according key sizes of 64, 80 and 96 bit. For all three variants a block size of 64 bit is used, only the number of rounds performed and the key schedule differs. However, in contrast to AES, KLEIN is not operating on bytes, but on 4 bit wide *nibbles*. In the following we will give a very brief description of the general structure and the individual round functions.

7.1.1. The Round Function

The cipher is composed from $R \in \{12, 16, 20\}$ executions of the round function, depending on the key size of 64 bit, 80 bit or 96 bit. Each round i utilizes a round key sk^i, which is derived from the previous round key through the *KeySchedule* function. Basic building blocks of each round are the functions *AddRoundKey*, *SubNibbles*, *RotateNibbles* and *MixNibbles*. Algorithm 1 shows the general structure of the KLEIN cipher.

Algorithm 1 The structure of the KLEIN cipher.

$sk^1 \leftarrow KEY$
$STATE \leftarrow PLAINTEXT$
for $i = 1$ *to* R **do**
  $AddRoundKey(STATE, sk^i)$
  $SubNibbles(STATE)$
  $RotateNibbles(STATE)$
  $MixNibbles(STATE)$
  $sk^{i+1} \leftarrow KeySchedule(sk^i, i)$
end for
$CIPHERTEXT \leftarrow AddRoundKey(STATE, sk^{R+1})$

7.1.2. SubNibbles

The *SubNibbles* function is the nonlinear permutation step of KLEIN which ensures nibble-wide diffusion. A notable property of the used 4 bit S-box function S is the fact that it is involutive, i.e., $S(x) = S^{-1}(x)\ \forall\ x\ \in \{0, ..., 15\}$. This saves the costs for the implementation of an inverse S-box. The S-box is given in Table 7.1.

Table 7.1.: The 4 bit S-box of KLEIN.

Input:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Output:	7	4	A	9	1	F	B	0	C	3	2	6	8	E	D	5

7.1.3. RotateNibbles

The *RotateNibbles* function rotates the full 16 Nibbles wide input $[n_0, n_1, \ldots, n_{15}]$ by two bytes (4 Nibbles) to the left:

$$[n_0, n_1, \ldots, n_{15}] \rightarrow [n_4, n_5, \ldots, n_{15}, n_0, n_1, n_2, n_3, n_4]$$

7.1.4. MixNibbles

The *MixNibbles* function is a linear mapping that ensures state-wide diffusion. It subdivides the input state into two arrays[2] of 4 bytes (8 nibbles $[n_0, \ldots, n_7]$ and $[n_8, \ldots, n_{15}]$) which are interpreted as polynomials in $\mathbb{F}_2^8$. The multiplication with the permutation matrix is calculated modulo the reduction polynomial $x^4 + 1$. MixNibbles uses thereby the exact same 4×4 bytes permutation matrix that is used in the AES:

$$\begin{bmatrix} n_0^{i+1}||n_1^{i+1} & n_8^{i+1}||n_9^{i+1} \\ n_2^{i+1}||n_3^{i+1} & n_{10}^{i+1}||n_{11}^{i+1} \\ n_4^{i+1}||n_5^{i+1} & n_{12}^{i+1}||n_{13}^{i+1} \\ n_6^{i+1}||n_7^{i+1} & n_{14}^{i+1}||n_{15}^{i+1} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} n_0^i||n_1^i & n_8^i||n_9^i \\ n_2^i||n_3^i & n_{10}^i||n_{11}^i \\ n_4^i||n_5^i & n_{12}^i||n_{13}^i \\ n_6^i||n_7^i & n_{14}^i||n_{15}^i \end{bmatrix}$$

7.1.5. Key Schedule

KLEIN's key schedule reuses the *SubNibbles* functions from the round function. The key schedule is composed from a cyclic left shift by two nibbles (one byte), followed by a Feistel network. Subsequently, four nibbles (two bytes) are substituted by the *SubNibbles* function and the *round constant* i is added to the fifth and sixth nibble (third byte). Figure 7.2 depicts the structure of the key schedule for 3 iterations. In contrast to the round function the key schedule works in a byte oriented way, as all operations are performed on a multiple of two nibbles.

7.1.6. Modified Representation

To simplify the explanation of our attack and to deal with the application of the last round's *MixNibbles*, it is necessary to slightly change the representation of the last round function, since one nibble of the resulting ciphertext is influenced by several nibbles of the last round key. Unlike AES, KLEIN does not omit the *MixNibbles* operation in the last round [GNL12]. The effects of omitting the last *MixColumns* in the AES were extensively studied by Dunkelman et al. in [DK10]. The last *AddRoundKey* step and the previous *MixNibbles* step are therefore swapped, the modified representation of KLEIN is shown in Algorithm 2. Since *MixNibbles* is a linear function, it holds that $MixNibbles(a + b) = MixNibbles(a) + MixNibbles(b)$, the same reasoning can also be applied to the *AddroundKey* function. Therefore, we can exchange the *AddRoundKey* and the *MixNibbles* step, if we substitute the added round key sk^i with $MixNibbles(sk^i)$. As a result of the exchanged order of *AddRoundKey* and *MixNibbles*, it is also necessary to apply the inverse *MixNibbles* function to the last round key RK^{R+1} prior to the addition to the state RB^R. Furthermore, for the sake of simplicity from now on we will represent KLEIN in a byte-oriented view, in contrast to the originally proposed nibble-oriented view. In fact *SubNibbles* is the only function, which actually operates on nibbles. However, the application of a 4 bit S-box on two nibbles can be replaced by a compound 8 bit S-box without loss of generality, if no particular properties of the 4 bit S-box are considered. Therefore, in the following we represent the KLEIN's state as an array of bytes, which is transformed by the functions *SubBytes*, *MixBytes* and *RotateBytes*.

7.1.7. Notation

Throughout the remaining sections we will use the following notation. An intermediate state of KLEIN is named according to the abbreviation of the function which was applied to the intermediate state last, i.e., ARK^x is the state after the application of *AddRoundKey* during

[2] || denotes a concatenation

Algorithm 2 The structure of the modified KLEIN cipher.

$sk^1 \leftarrow KEY$
$STATE \leftarrow PLAINTEXT$
for $i = 1$ *to* $R - 1$ **do**
  $AddRoundKey(STATE, sk^i)$
  $SubNibbles(STATE)$
  $RotateNibbles(STATE)$
  $MixNibbles(STATE)$
  $sk^{i+1} \leftarrow KeySchedule(sk^i, i)$
end for
$AddRoundKey(STATE, sk^R)$
$SubNibbles(STATE)$
$RotateNibbles(STATE)$
$sk^{R+1} \leftarrow KeySchedule(sk^R, R)$
$AddRoundKey(STATE, invMixNibbles(sk^{R+1}))$
$CIPHERTEXT \leftarrow MixNibbles(STATE)$

round x. A subscript refers to a specific byte of the state. The states of the key schedule are abbreviated as RK^x. A multiplication of two bytes is done as defined in the AES [FIP01]. Faulted values are indicated by an overline (eg. the faulty byte $\overline{ARK}_0^{R-1}$).

7.2. Attack on the Encryption

The proposed attack strategy is quite similar to those formerly published on AES by Piret et al. [PQ03], Tunstall et al. [TMA11] and Mukhopadhyay et al. [Muk09]. Unlike those, this attack is split into two separate parts, each revealing 32 bits of the corresponding round key. As mentioned in Section 7.1.6, it is also possible to compute all operations of KLEIN on byte level, contrary to Algorithm 2, we will use this alternative representation to simplify the description of our proposed DFA. The attack on the encryption works on all three variants of KLEIN. A random single-byte fault is injected into the state between MB^{R-2} and RB^{R-1}. We opted for a random single-byte fault model, as KLEIN should be a lightweight cipher [GNL12] which are often implemented on 8-bit platforms. As a result of using a 8-bit platform random, single-byte faults can be achieved easily, e.g., due to an instruction skip [GGS17]. This fault will lead to a completely corrupted ciphertext, affecting all 8 bytes. Figure 7.1 shows the propagation for two different faults, injected either into the left half cf. Fig. 7.1a or the right half cf. Fig. 7.1b of the state. The faulted byte is indicated by f. We will now outline the attack for a fault injection which affects the left half of the state MB^{R-1}. After the application of *MixBytes* in round $R-1$, the former single-byte fault has spread over all four bytes of the left half. Since *MixBytes* is a linear function (cf. Section 7.1.6), the resulting fault can be described as a byte-wise multiple of f: Depending on which byte position before *MixBytes* was faulted, the individual bytes after *MixBytes* inherit an additive fault with the values f, $2f$ or $3f$. These multiples of the same value f can be exploited to formulate a set of equations. An attacker cannot obtain f directly by reverse calculating from the ciphertext, as he is only able to observe the transformed version of f, i.e., $F_i,\ i \in \{0, 1, 2, 3\}$ after passing through the four S-boxes. But the attacker can describe an implicit relationship, of the S-box's input and output fault, the following equations demonstrate

this for the case depicted in Fig. 7.1a:

$$\begin{aligned} &SubBytes(ARK_6^{R+1} + RK_6^{R+1}) + SubBytes(\overline{ARK}_6^{R+1} + RK_6^{R+1}) \\ &= 2 \cdot \Big(SubBytes(ARK_7^{R+1} + RK_7^{R+1}) + SubBytes(\overline{ARK}_7^{R+1} + RK_7^{R+1})\Big) \end{aligned} \tag{7.1}$$

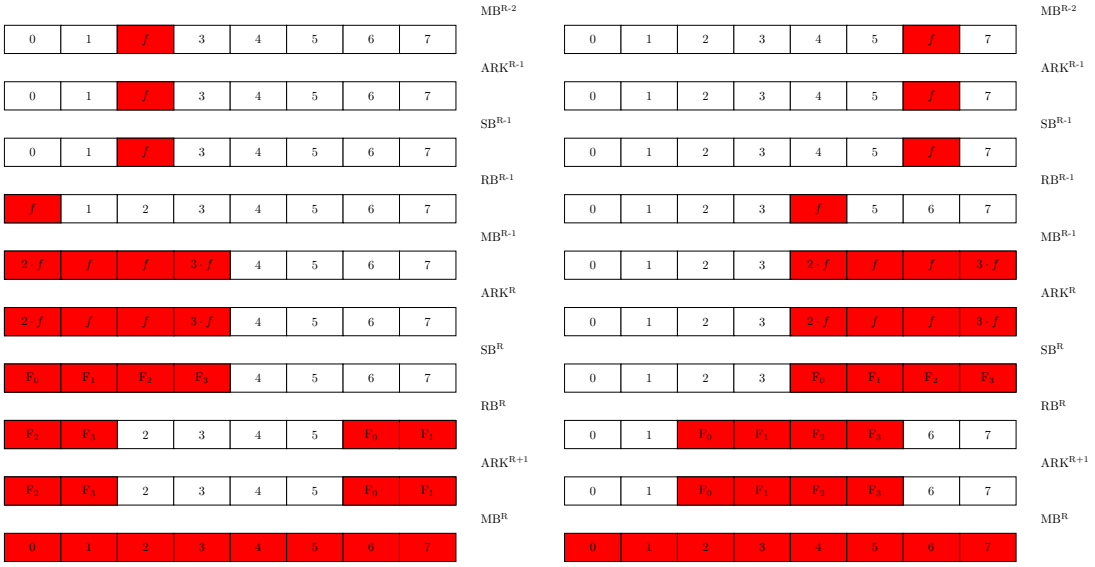
$$\begin{aligned} &SubBytes(ARK_7^{R+1} + RK_7^{R+1}) + SubBytes(\overline{ARK}_7^{R+1} + RK_7^{R+1}) \\ &= SubBytes(ARK_0^{R+1} + RK_0^{R+1}) + SubBytes(\overline{ARK}_0^{R+1} + RK_0^{R+1}) \end{aligned} \tag{7.2}$$

$$\begin{aligned} &SubBytes(ARK_1^{R+1} + RK_1^{R+1}) + SubBytes(\overline{ARK}_1^{R+1} + RK_1^{R+1}) \\ &= 3 \cdot \Big(SubBytes(ARK_7^{R+1} + RK_7^{R+1}) + SubBytes(\overline{ARK}_7^{R+1} + RK_7^{R+1})\Big) \end{aligned} \tag{7.3}$$

Each equation combines two different bytes and therefore uses a hypothesis over two different key-bytes $(RK_i^{R+1}, RK_j^{R+1}) \; \forall \; (i,j) \in \{(6,7),(0,7),(1,7)\}$. Thus, the attacker has a set of three equations depending on four different key-bytes. Since this set of equations is under-determined, there is no unique solution. However, the attacker can use these equations to discard all those 4-byte key-hypotheses, which do not solve this set of equations. Therefore, all possible keys are stored in a set of hypotheses. Using the result of an additional fault injection with a different fault f, the attacker can discard those keys in the set of hypotheses, which do not fulfill the new equations. The computational complexity of this step can be significantly reduced from 2^{32} by testing only those 4-byte key hypotheses, where the individual four key bytes were tested as valid. Since the attacker usually does not know which of the four possible byte positions were faulted, all four options have to be tested. However, the resulting increase in complexity of a factor of 4 does not present a problem. The question of which half of the state was faulted (a single example for both cases is depicted in Fig. 7.1a and Fig. 7.1b), can be determined by applying the inverse *MixBytes* function to the observable fault at the output, i.e., the addition of the correct and faulty ciphertext. In order to reveal the full 64-bit round key, the attacker can choose to inject another fault at a different position to run the attack on both halves of the key, or to determine the missing 32 bit part with the brute-force approach.

7.3. Attack on the Key Schedule

In our proposed attack on the key schedule, the attacker is expected to induce a random byte fault Δ into the key state RK^{R-2}, which corrupts the byte RK_5^{R-2} as shown in Fig. 7.2. The proposed fault model and location was chosen to be within a path through the key schedule free of nonlinear functions, which results in a partial cancellation of the fault during round R. Choosing a fault injection location that results in partial cancellation of the injected fault is advantageous to keep complexity during the attack low and to limit the fault in the key schedule state to a single byte. The fault injection into the key schedule empowers a fault propagation into both halves of the state simultaneously, due to the Feistel-like structure of the key schedule where one half of the key state is added to the other half. One has to keep in mind after a fault injection into the key schedule the fault spreads throughout the key schedule and after an *AddRoundKey* operation also in the state. The approach of the attack can be divided as usual into three parts *fault propagation*, *fault exploitation* and *state recovery*.



(a) Fault in Byte 2

(b) Fault in Byte 6

Figure 7.1.: Fault propagation for a single-byte fault injected between MB^{R-2} and MB^{R-1}.

7.3.1. Fault Propagation

The propagation of the faulty byte is based on two parts, the propagation through the key schedule and the propagation through the state of KLEIN.

Key schedule Under the assumption of a random byte fault model, the byte RK_5^{R-2} is perturbed with a fault Δ, as shown in Fig. 7.2. After the fault injection, both halves are rotated byte-wise to the left by 1. During the Feistel step, the faulted byte spreads to both halves of the key state RK^{R-1}. As one can see the path chosen avoids nonlinear functions, therefore the bytes RK_0^{R-1}, RK_4^{R-1} are both under the influence of the same fault Δ. Due to the Feistel structure of the key schedule the fault Δ of byte RK_7^R is canceled out as a result of the addition of both halves ($\Delta + \Delta = 0$). During the last iteration of the key schedule from round R to $R+1$ the single-byte fault Δ passes through one S-box, as a result the byte RK_6^{R+1} is the only byte in the key schedule's state with a fault different from Δ. In total there are four distinct locations during the last three iterations of the key schedule where the faults are fed into the state of KLEIN, the four locations are indicated by a lighting symbol under each faulted byte, as shown in Fig. 7.2.

State During the round key addition in round $R-1$ the state ARK^{R-1} is perturbed at first, with the fault Δ at indices 0 and 4 as shown in Fig. 7.3. After passing through the S-boxes the fault Δ is transformed into the faults f_1 and f_2. Applying a shift to the state does not change the faults. As a result of passing one faulty byte on each half (i.e. RB^{R-1}) through the *MixBytes* operation, each byte of the state MB^{R-1} is now influenced by multiples of the fault either $\{1, 2, 3\}$ times the original fault f_1, f_2. After the addition with the round key RK^R, the byte ARK_3^R is now faulted with $f_1 + \Delta$. After the application of *SubBytes* the faults are transformed into the faults $F_i,\ i \in \{0, 1, 2, 3M, 4, 5, 6, 7\}$. The suffix M indicated that the fault is masked by another fault introduced from subsequent *AddRoundKey* applications. As a result of the last *AddRoundKey* operation, the whole right half of the state ARK^{R+1} is perturbed again

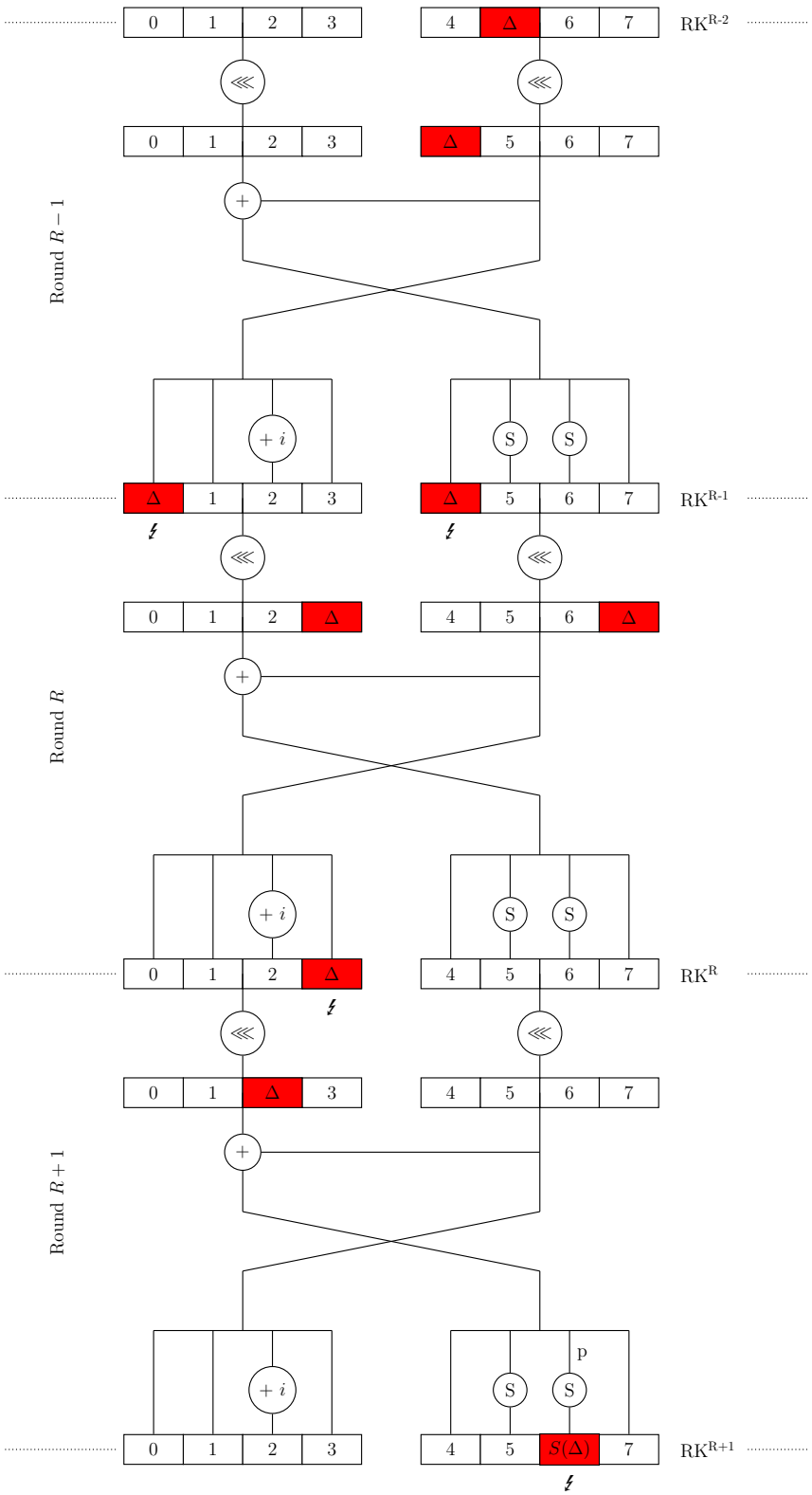


Figure 7.2.: Fault propagation of a single-byte fault in round 10 of the KLEIN-64 key schedule.

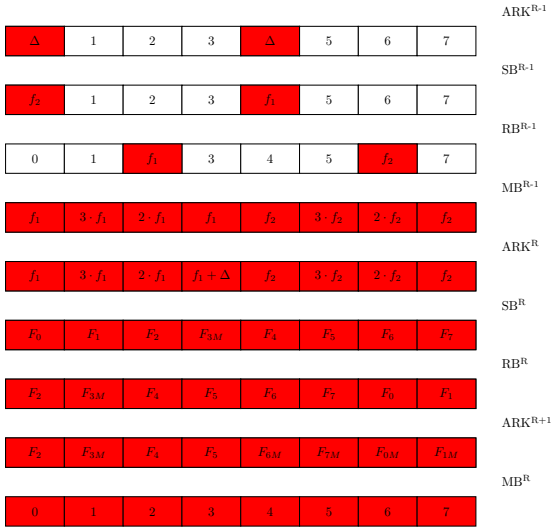


Figure 7.3.: Fault propagation through the state of KLEIN.

with a fault coming from the key schedule. After the application of *AddRoundKey* the faults are transformed into the faults which are observable $F_i,\ i \in \{0M, 1M, 2, 3M, 4, 5, 6M, 7M\}$. The disturbance of the whole right half of the state occurs due to the modified variant of KLEIN where the *AddRoundKey* and *MixBytes* operation are swapped. Swapping both functions requires an application of *MixBytes* to the round key RK^{R+1}, prior to the execution of *AddRoundKey* in round $R+1$.

7.3.2. Fault Exploitation

In order to recover the last round key which is in the case of KLEIN-64 also the master key as the key schedule is invertible, we have to recover the state ARK^R which is KLEIN's state before the *SubBytes* function in round R. After the recovery of ARK^R we can calculate the last round key using the ciphertext C as shown in Eq. (7.4).

$$\begin{aligned} RK^{R+1} &= RB^R + ARK^{R+1} \\ RK^{R+1} &= RB^R + inverseMixBytes(C) \end{aligned} \tag{7.4}$$

Since the S-boxes are the only nonlinear elements of KLEIN this will provide us a filtering mechanism for wrong state hypotheses. As an example of how to derive the required equations we will demonstrate this for the fault F_0 in detail. The fault F_0 is defined as the result of adding the bytes SB_0^R and $\overline{SB}_0^R$. We can express SB_0^R as an application of *SubBytes* to ARK_0^R, respectively $ARK_0^R + f_1$ in the faulted case as shown in Eq. (7.5).

$$\begin{aligned} F_0 &= SB_0^R + \overline{SB}_0^R \\ F_0 &= SubBytes(ARK_0^R) + SubBytes(\overline{ARK}_0^R) \\ F_0 &= SubBytes(ARK_0^R) + SubBytes(ARK_0^R + f_1) \end{aligned} \tag{7.5}$$

Analogous to F_0 we express F_1 as shown in Eq. (7.6), both equations for either F_0 or F_1 depend on the fault f_1, which is not observable.

$$F_1 = SubBytes(ARK_1^R) + SubBytes(ARK_1^R + 3 \cdot f_1) \tag{7.6}$$

The equations for F_2 and F_3 are constructed similarly, but this time the equation for F_3 depends also on the injected fault Δ as shown in Eq. (7.7).

$$\begin{aligned} F_2 &= SubBytes(ARK_2^R) + SubBytes(ARK_2^R + 2 \cdot f_1) \\ F_3 &= SubBytes(ARK_3^R) + SubBytes(ARK_3^R + f_1 + \Delta) \end{aligned} \tag{7.7}$$

Equation (7.8) is a special case, because it is the only equation set where the faults F_4 and F_5 are not overlaid with another fault coming from the key schedule (self cancellation of Δ) therefore this equation set will be the starting point for the state recovery.

$$\begin{aligned} F_4 &= SubBytes(ARK_4^R) + SubBytes(ARK_4^R + f_2) \\ F_5 &= SubBytes(ARK_5^R) + SubBytes(ARK_5^R + 3 \cdot f_2) \end{aligned} \tag{7.8}$$

Equation (7.9) is constructed similarly to Eq. (7.8), this time the equations are influenced by the fault f_2 and a multiple of f_2. The faults F_6 and F_7 can not be observed due to the addition of the faulty round key RK^{R+1}, the same holds also for the faults F_0 and F_1.

$$\begin{aligned} F_6 &= SubBytes(ARK_6^R) + SubBytes(ARK_6^R + 2 \cdot f_2) \\ F_7 &= SubBytes(ARK_7^R) + SubBytes(ARK_7^R + f_2) \end{aligned} \tag{7.9}$$

As a result of not being able to observe the output faults F_i where $i \in \{0, 1, 6, 7\}$, we introduce helper variables F_{iM}, which are observable at the output as shown in Eq. (7.10). These helper variables are composed from the unobservable faults F_i and the addition of the *MixBytes* transformed fault which passed through the S-box. Also, we introduce another helper variable p which represents the actual value of RK_6^{R+1} before the application of the S-box as shown in Fig. 7.2.

$$\begin{aligned} F_{0M} &= F_0 + E \cdot (SubBytes(p) + SubBytes(p + \Delta)) \\ F_{1M} &= F_1 + 9 \cdot (SubBytes(p) + SubBytes(p + \Delta)) \\ F_{6M} &= F_6 + D \cdot (SubBytes(p) + SubBytes(p + \Delta)) \\ F_{7M} &= F_7 + B \cdot (SubBytes(p) + SubBytes(p + \Delta)) \end{aligned} \tag{7.10}$$

Additionally, the relationships between the injected fault and the transformed faults are shown in Eq. (7.11), which describes the relationships between f_1, f_2 ,Δ and two state bytes from the round $R - 1$. These equations aim to eliminate wrong hypotheses for the injected fault Δ, the actual values of ARK_0^{R-1} and ARK_4^{R-1} are not of interest.

$$\begin{aligned} f_1 &= SubBytes(ARK_4^{R-1}) + SubBytes(ARK_4^{R-1} + \Delta) \\ f_2 &= SubBytes(ARK_0^{R-1}) + SubBytes(ARK_0^{R-1} + \Delta) \end{aligned} \tag{7.11}$$

Having formulated an equation for every byte of the state ARK^R as shown in Eqs. (7.5) to (7.10), and the relationships between the injected fault and intermediate faults as shown in Eq. (7.11), we will now provide a description of how to recover the state ARK^{R-1} in several steps.

7.3.3. State Recovery

To recover the state ARK^R the attacker has to solve several sets of equations. At first, the attacker calculates the fault state F which is composed of the values:
$F_i,\ i \in \{0M, 1M, 2, 3M, 4, 5, 6M, 7M\}$. To do so an addition of the correct ciphertext C and the faulty ciphertext $\overline{C}$ is transformed with the inverse *MixBytes* operation as shown in Eq. (7.12). The position of the faults F_i throughout the state of KLEIN is also shown in Fig. 7.3.

$$F = inverseMixBytes(C) + inverseMixBytes(\overline{C}) \tag{7.12}$$

Throughout the attack's description we use a shorthand notation for the addition of a correct S-box with a faulty one, $filter(x, f) = SubByte(x) + SubByte(x + f)$. The $SubByte$ function refers to the substitution of a single-byte using KLEIN's S-box. As we are unable to recover the whole state at once we apply a divide and conquer strategy. Therefore, as a preliminary step, we define the set $P = \{0, ..., 255\}$ with $p \in P$ which represents all possible values of an eight bit variable. Furthermore, we assume that each byte of the state ARK^R is initialized with P, i.e., $\mathcal{ARK}_i^R = P\ \forall\ i\ \in \{0, ..., 7\}$. While one faulty encryption is processed there will also be sets containing hypotheses for the faults f_1, f_2 and Δ, in contrast to the sets of state bytes these sets are only valid while processing one faulty encryption, as the next encryption is probably under the influence of another fault. The following steps are repeated for all faulty encryptions, in order to decrease the number of hypotheses in the sets. The attacker starts with the recovery of $\mathcal{ARK}_4^R$, $\mathcal{ARK}_5^R$, using Eq. (7.8). As one can see the equation set depends on five different variables, the known value of the faults F_4, F_5, the unknown values $\mathcal{ARK}_4^R$, $\mathcal{ARK}_5^R$ and the unknown fault f_2. The system of equations is then used to reduce the solution space for $\mathcal{ARK}_4^R$, $\mathcal{ARK}_5^R$ and f_2 using an exhaustive search with all unknown variables as search space. The first step of the attack is shown in Eq. (7.13). Hypotheses that satisfy both conditions are kept as valid hypotheses.

$$\begin{aligned} T_{poss} &= \{\mathcal{ARK}_4^R \times \mathcal{ARK}_5^R \times \{1, ..., 255\}\} \\ T_{valid} &= \{(x, y, f) \in T_{poss} \mid F_4 \equiv filter(x, f) \wedge F_5 \equiv filter(y, 3 \cdot f)\} \\ \mathcal{ARK}_4^R &= \{\ x \mid (x, y, f) \in T_{valid}\ \} \\ \mathcal{ARK}_5^R &= \{\ y \mid (x, y, f) \in T_{valid}\ \} \\ f_2 &= \{\ f \mid (x, y, f) \in T_{valid}\ \} \end{aligned} \tag{7.13}$$

Now that the attacker has gained knowledge of the fault f_2, he can process the second equation of Eq. (7.11). He iterates through all possible values of $\mathcal{ARK}_0^{R-1} \in \{0, ..., 255\}$ and stores the valid hypotheses for Δ. The description of the second part of the attack is shown in Eq. (7.14).

$$\begin{aligned} T_{poss} &= \{f_2 \times \mathcal{ARK}_0^{R-1} \times \{1, ..., 255\}\} \\ T_{valid} &= \{(f, x, \delta) \in T_{poss} \mid f \equiv filter(x, \delta)\} \\ \Delta &= \{\ \delta \mid (f, x, \delta) \in T_{valid}\ \} \end{aligned} \tag{7.14}$$

As a result of having knowledge of Δ the attacker aims now to recover the state bytes $\mathcal{ARK}_2^R$, $\mathcal{ARK}_3^R$ to do so the attacker has to solve Eq. (7.7) in the same manner as in the first step, as a result the attacker gains additional knowledge of the faults f_1. The third part of the attack is

shown in Eq. (7.15).

$$\begin{aligned} T_{poss} &= \{\mathcal{ARK}_2^R \times \mathcal{ARK}_3^R \times \Delta \times \{1, ..., 255\}\} \\ T_{valid} &= \{(x, y, \delta, f) \in T_{poss} \mid F_2 \equiv filter(x, 2 \cdot f) \wedge F_3 \equiv filter(y, f + \delta)\} \\ \mathcal{ARK}_2^R &= \{\ x \mid (x, y, \delta, f) \in T_{valid}\ \} \\ \mathcal{ARK}_3^R &= \{\ y \mid (x, y, \delta, f) \in T_{valid}\ \} \\ f_1 &= \{\ f \mid (x, y, \delta, f) \in T_{valid}\ \} \end{aligned} \tag{7.15}$$

Now that the attacker has also knowledge of the fault f_1 he can process the first equation of Eq. (7.11) using another exhaustive search to shrink the number of possible hypotheses for Δ, under the assumption of $\mathcal{ARK}_4^{R-1} \in \{0, ..., 255\}$. The algorithmic description of the fourth part of the attack is shown in Eq. (7.16).

$$\begin{aligned} T_{poss} &= \{f_1 \times \mathcal{ARK}_4^{R-1} \times \Delta\} \\ T_{valid} &= \{(f, x, \delta) \in T_{poss} \mid f \equiv filter(x, \delta)\} \\ \Delta &= \{\ \delta \mid (f, x, \delta) \in T_{valid}\ \} \end{aligned} \tag{7.16}$$

During the recovery of $\mathcal{ARK}_0^R$,$\mathcal{ARK}_1^R$, $\mathcal{ARK}_6^R$ and $\mathcal{ARK}_7^R$ the attacker faces the problem that the faults $F_i,\ i \in \{0, ..., 7\} \setminus \{2, 4, 5\}$ do not take into account that the observable faults from the output are composed from several faults coming from the key schedule and the state. Also, the faulted byte p (the *MixBytes* transformed fault) from the key schedule, influences the right half. Therefore, the attacker needs to apply the correction from Eq. (7.10) and combine the equations with Eq. (7.5) and Eq. (7.9). The algorithmic description for the recovery of $\mathcal{ARK}_0^R$,$\mathcal{ARK}_1^R$ is shown in Eq. (7.17).

$$\begin{aligned} T_{poss} &= \{\mathcal{ARK}_0^R \times \mathcal{ARK}_1^R \times P \times \Delta \times f_1\} \\ T_{valid} &= \{(x, y, p, \delta, f) \in T_{poss} \mid F_{0M} + E \cdot filter(p, \delta) \equiv filter(x, f) \wedge \\ &\qquad F_{1M} + 9 \cdot filter(p, \delta) \equiv filter(y, f)\} \\ \mathcal{ARK}_2^R &= \{\ x \mid (x, y, p, \delta, f) \in T_{valid}\ \} \\ \mathcal{ARK}_3^R &= \{\ y \mid (x, y, p, \delta, f) \in T_{valid}\ \} \\ P &= \{\ p \mid (x, y, p, \delta, f) \in T_{valid}\ \} \end{aligned} \tag{7.17}$$

The recovery of $\mathcal{ARK}_6^R$ and $\mathcal{ARK}_7^R$ as shown in Eq. (7.18) is similar to the bytes $\mathcal{ARK}_0^R$ and $\mathcal{ARK}_1^R$.

$$\begin{aligned} T_{poss} &= \{\mathcal{ARK}_6^R \times \mathcal{ARK}_7^R \times P \times \Delta \times f_2\} \\ T_{valid} &= \{(x, y, p, \delta, f) \in T_{poss} \mid F_{6M} + D \cdot filter(p, \delta) \equiv filter(x, 2 \cdot f) \wedge \\ &\qquad F_{7M} + B \cdot filter(p, \delta) \equiv filter(y, f)\} \\ \mathcal{ARK}_6^R &= \{\ x \mid (x, y, p, \delta, f) \in T_{valid}\ \} \\ \mathcal{ARK}_7^R &= \{\ y \mid (x, y, p, \delta, f) \in T_{valid}\ \} \\ P &= \{\ p \mid (x, y, p, \delta, f) \in T_{valid}\ \} \end{aligned} \tag{7.18}$$

An attacker has to repeat all the steps mentioned above for each faulted encryption, in order to reduce the number of hypotheses (state bytes) until it becomes feasible to brute force the remaining complexity (key space), which will be discussed in the next section.

7.4. Simulation and Discussion

We will now discuss the performance of the attacks. To do, so we will determine the number of required ciphertexts to reduce the remaining complexity (keyspace) to a certain threshold. We opted to implement the simulation of the faulty ciphers in *Python*, and the attacks as `C` extension for *Python*. The attacks were performed on an `Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz` based desktop computer. The amount of RAM required during the attack is negligible, on average the attack takes five minutes on the computer we used depending on the injected fault.

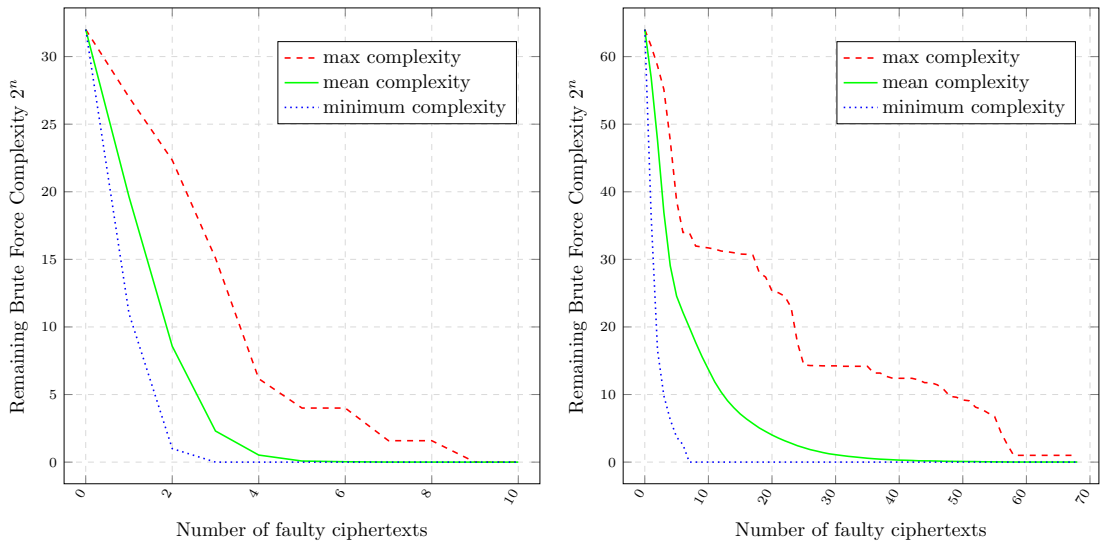
7.4.1. Simulation

To evaluate the performance of each attack we performed several simulations, using the following approach. For every iteration of the simulation we generated 100 faulty ciphertexts using one random plaintext, 100 was found to be a reliable upper bound for the maximum number of required faulty ciphertexts. Afterwards we launched the attacks and stored the complexity of the key space in bits for every processed faulty ciphertext during the attack. The remaining brute force complexity was defined as power of two of the product of the cardinality of the state byte sets[3]. We then repeated the previous step 500 times to get significant data. Additionally, we have addressed the issue of faults, which do not comply with the required fault model. This results in an empty set of remaining key candidates for both attacks. To overcome this issue an attacker can partition the set of faulty ciphertexts and test the subsets separately until he will find a set containing only ciphertexts according to the fault model. The result of the attack's simulation is shown in Fig. 7.4, where one can see the number of faulty ciphertexts on the x-axis, and the remaining brute force complexity on the y-axis. The three different plots in each subfigure are either the maximum, mean or minimum complexity. As a result of the simulation it was found out that the attack on the state requires five faulted encryptions on average to reduce the brute-force complexity of the last round key from 2^{64} to 2^{32} on average, as shown in Fig. 7.4a (we opted to attack only on one half of the state therefore the maximum complexity starts at 2^{32}). For the attack on the key schedule it was found out that four faulted encryptions are required on average to reduce the remaining brute-force complexity of the last round key from 2^{64} to 2^{32} on average as shown in Fig. 7.4b.

7.4.2. Discussion

As the structure of KLEIN is similar to the AES [FIP01] we will compare the attack on the state with the attacks of [PQ03; Muk09; TMA11], and the attack on the key schedule with the attacks of [CY03; AM11; Kim12]. Our attack on the state of KLEIN performs worse in terms of required faulty encryptions than the attack on the state of AES by [PQ03; Muk09; TMA11]. This can be attributed to the structure of KLEIN's round function where a fault injected into one half of the state does not spread to the other half, in contrast to the AES. But still only four faulted encryptions (on the same half) are required to deduce the last round key of KLEIN as shown in Fig. 7.4a. Our attack on the key schedule performs worse in terms of required faulty encryptions if compared to the attacks of [CY03; AM11; Kim12]. This can be attributed to the key schedule of KLEIN which is based on a Feistel network where a fault does not create a sufficient avalanche effect which results in an immediate corruption of a whole half of the key schedule. After only one faulty encryption the complexity of the key space was decreased on average to $2^{56.9}$. If we assume a complexity of 2^{32} to be the upper bound for a brute force attack as in [AM11], this

[3] i.e. for the attack on the key schedule, $complexity = 2^{\prod_{i=0}^{7} |\mathcal{ARK}_i^R|}$



(a) Attack on the encryption (half key).

(b) Attack on the key schedule.

Figure 7.4.: Remaining brute force complexity (64-bit) key length).

results in four faulty encryptions with a complexity of $2^{29.0}$ on average as shown in Fig. 7.4b. Additionally, one noteworthy detail of Fig. 7.4b is the maximum complexity which remains 1 bit, even if we evaluate the simulation up to 100 faulted encryptions, but an increased number of key hypothesis by a factor of two can be neglected. To justify our upper bound of complexity of 2^{32} for both attacks we will focus on the scenario where KLEIN-64 is used to generate a MAC [GNL12]. The attacker aims to forge a message within a limited amount of time and resources. Therefore, we evaluated how long it takes to perform 2^{32} encryptions, using a `C` implementation of KLEIN as a result it was found out this takes 4.6 h on average, which seems to be a reasonable tradeoff between complexity and required faulty encryptions.

7.5. Summary

In this chapter we demonstrated the DFA of KLEIN's round function, and the first key schedule based DFA of KLEIN-64. Furthermore, we validated the performance of our attacks by simulation, and evaluated the remaining brute force complexity, with respect to the number of faulted encryptions. As a result it was found out, an attacker is able to reduce the key space from 64 bit to 32 bit, with only five fault injections into the round function, and with four faults injected into a specific byte in the key schedule. Both attacks can be conducted without knowing the actual plaintext. It is sufficient to know that the same plaintext was processed.

8. Persistent Fault Analysis of COLM, Deoxys-II, and OCB

The most common type of FIA is DFA which exploits differences of correct and faulty encryptions. Unfortunately the exploitation of differences is also the major drawback of DFA as this requires an attacker to obtain tuples of correct and faulty encryptions. Furthermore, it is necessary to inject precise faults which adhere to the algorithm-specific fault model at runtime for every encryption. In contrast to DFA, PFA as introduced by Zhang et al. only requires a fault injection with a persistent effect, i.e., the fault is present at runtime [Zha+18]. Due to the persistent nature of the fault, it is not necessary to perform fault injections at runtime. Since no faults need to be injected at runtime, this type of attack is particularly suitable for scenarios where countermeasures prevent frequent fault injection.

State of the art Prior to this work, PFA was only applied to AES-128 by Zhang et al. [Zha+18]. They were also able to successfully attack a fault hardened implementation based on the Dual Modular Redundancy (DMR) countermeasure, which is a common countermeasure against fault attacks. Furthermore, Xu et al. were even able to enhance the performance of PFA with their Enhanced Persistent Fault Analysis (EPFA) approach by the exploitation of leakage in deeper rounds with the objective to lower the number of required ciphertexts [Xu+21]. Additionally, Tang and Liu extended PFA to Multiple Faults-Based Persistent Fault Analysis (MPFA) to deal with the fact that low cost fault injection techniques cf. Guillen et al. [GGS17] usually are imprecise in terms of achievable fault model [TL22]. MPFA enables an attacker to exploit persistent faults even several occur at once. Multiple persistent faults at once also lower the computational complexity and the number of required ciphertexts.

Contribution We present the PFA of three AEAD schemes selected for the final portfolio of the CAESAR competition. The schemes in the final portfolio which we analyzed for their susceptibility to PFA are COLM, Deoxys-II, and OCB, which use AES or derivatives as underlying cryptographic primitive. As COLM, Deoxys-II, and OCB are AEAD schemes we especially aimed to find possible locations to mount PFA in AEAD schemes. Additionally, we evaluate how effectively PFA can be applied to COLM, Deoxys-II, and OCB by means of simulation.

Organization The rest of this chapter is structured as follows: Section 8.1 introduces COLM and the application of PFA. Section 8.2 introduces Deoxys-II and the application of PFA. Section 8.3 introduces OCB and the application of PFA. Section 8.4 summarizes the results for the PFA of the AEAD schemes COLM, Deoxys-II, and OCB which were obtained by means of simulation. Section 8.5 provides a summary of the results for the proposed PFA of COLM, Deoxys-II, and OCB.

8.1. COLM

COLM was introduced by Andreeva et al. in 2016 [And+16]. It is an encrypt-linear-mix-encrypt mode of AES. There are two variants of COLM: COLM_{127} primarily designed as a high-speed cipher, and COLM_0 primarily designed as a defense in depth cipher. As COLM was selected to be part of the final CAESAR portfolio as a defense in depth cipher, we focus on COLM_0 and further simply refer to it as COLM. However, the attack can also be mounted on COLM_{127}. Operations in COLM are performed in $GF(2^n)$. Addition is thereby achieved by bitwise XOR which is denoted by $+$. Multiplication of two polynomials $a(x), b(x) \in GF(2^n)$ is defined as $a(x) \cdot b(x) \bmod f(x)$. The result of the polynomial multiplication is reduced with reduction polynomial $f(x) = x^{128} + x^7 + x^2 + x + 1$, in $GF(2^{128})$.

8.1.1. Structure

The encryption of COLM is shown in Fig. 8.1. E_K denotes the AES encryption of a 128 bit data block with a 128 bit key K. The initialization vector IV is calculated as the sum of all AD

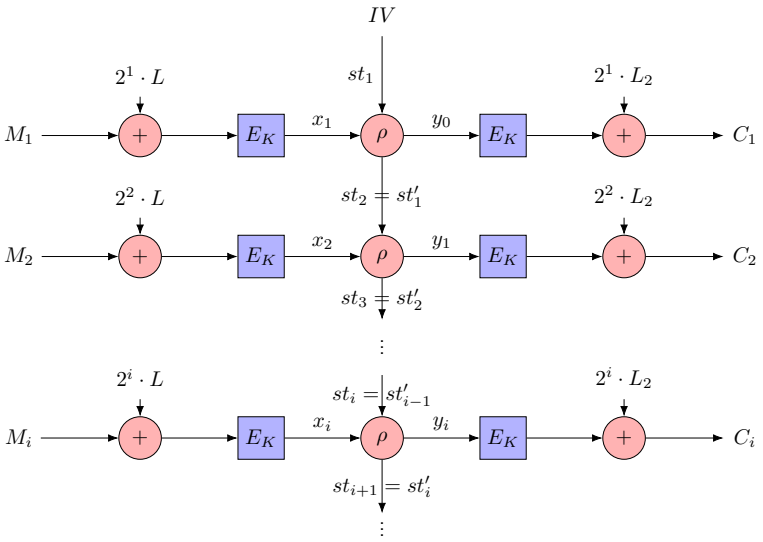


Figure 8.1.: COLM_0 Encryption of intermediate message blocks

blocks, where every AD block is first added to a multiple of a mask and then encrypted with E_K. However, the concrete value is not needed for the attack. The masks involved in the encryption process are defined as $L = E_K(0)$ and $L_2 = 3^2 \cdot L$. The linear mixing function ρ transforms x and st to y and st' as shown in Eq. (8.1).

$$\begin{aligned} y &= x + 3 \cdot st \\ st' &= x + 2 \cdot st. \end{aligned} \tag{8.1}$$

During the encryption the following steps are performed for each message block M_i where $i \in [1, l-1]$: First, a mask $2^i \cdot L$ is added to each message block M_i. Then, an AES-encryption

E_K with key K is performed on the result. Afterwards, the linear mixing function ρ is applied, and another AES-encryption is performed. Finally, the output mask $2^i \cdot L_2$ is added. The padded encryption and masking of the last message block is not relevant to the PFA and thus not described in this work.

8.1.2. PFA of COLM

As COLM is using AES as the underlying block cipher, the principles of PFA can be applied. Our attack targets the encryption stage of COLM. Since COLM adds a mask to the AES output, PFA can only extract the sum of the last round key k and the current mask $2^i \cdot L_2$. This sum is denoted by R_i as shown in Eq. (8.2)

$$R_i = k + 2^i \cdot L_2 \tag{8.2}$$

To overcome this problem two different values of R_i are required: $R_l = k + 2^l \cdot L_2$ and $R_r = k + 2^r \cdot L_2$, where $l \neq r$ and $l, r \in [1, m+1]$. The addition of these two values leads to Eq. (8.3), which can be solved for L_2 as shown in Eq. (8.4), as additions $+$ and multiplications $\cdot$ are distributive in $GF(2^n)$.

$$R_l + R_r = (k + 2^l \cdot L_2) + (k + 2^r \cdot L_2) = (2^l + 2^r) \cdot L_2 \tag{8.3}$$

$$L_2 = (2^l + 2^r)^{-1} \cdot (R_l + R_r) \tag{8.4}$$

As L_2 is known, the last round key k can be calculated as shown in Eq. (8.5).

$$k = R_i + 2^i \cdot L_2 \tag{8.5}$$

Finally, the inverted key schedule of AES is used to calculate the master key K.

Example We use the first and second message block to obtain the key: First, the mask L_2 is calculated as shown in Eqs. (8.6) and (8.7)

$$R_1 + R_2 = (k + 2^1 \cdot L_2) + (k + 2^2 \cdot L_2) = 6 \cdot L_2 \tag{8.6}$$

$$L_2 = 6^{-1} \cdot (R_1 + R_2) \tag{8.7}$$

Next, the multiplicative inverse 6^{-1} in $GF(2^{128})$ is calculated. Finally, the last round key of E_K can be computed as shown in Eq. (8.8).

$$k = R_1 + 2^1 \cdot L_2 \tag{8.8}$$

The PFA-based attack, which was applied to COLM_0, can also be applied to COLM_{127} with minor modifications, during the processing of the masks.

8.2. Deoxys-II

Deoxys-II is a two pass, nonce misuse resistant, tweakable AEAD scheme proposed by Jean et al. [Jea+16] it provides 128 bit security with respect to both privacy and authenticity. The attack we present on Deoxys-II only applies to Deoxys-II-128-128, which has a key and tweak size of 128 bit and uses the underlying block cipher Deoxys-BC-256. For the rest of this work we simply refer to them as Deoxys-II and Deoxys-BC.

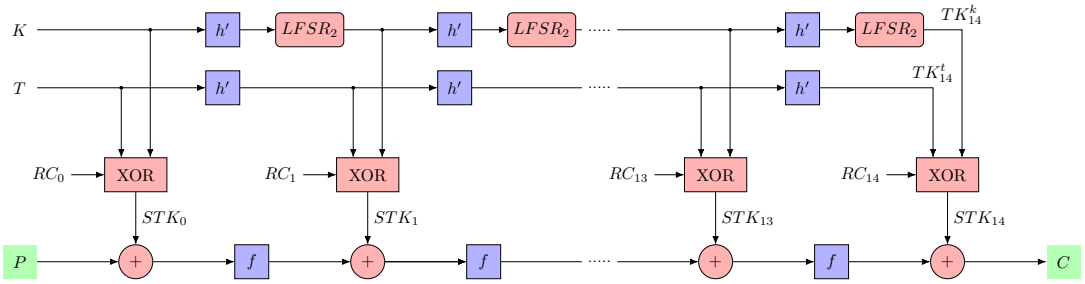


Figure 8.2.: Deoxys-BC-256 Encryption

8.2.1. Structure

Fig. 8.2 shows Deoxys-BC which is denoted as $E_K^T(P)$, where a single encryption is invoked on the plaintext P with a key K and a tweak T. The round function f of Deoxys-BC is similar to the round function of AES. A single round of Deoxys-BC is composed of the following operations: AddRoundTweakey (ATK), the addition of the sub tweakey; SubBytes (SB), a non-linear S-box; ShiftRows (SR), a row-wise shift; and MixBytes (MB), a matrix multiplication. Consequently, a single round of Deoxys-BC is defined as $f = MB \circ SR \circ SB \circ ATK$. The most significant difference to AES is the key schedule which is replaced by a tweakable round key addition based on the TWEAKEY framework [JNP14]. During each round of Deoxys-BC, the key K and the tweak T are updated by a permutation h' and a Linear Feedback Shift Register (LFSR), specifically $LFSR_2$ as shown in Fig. 8.2. The definition of the permutation h' is shown in Eq. (8.9) where each byte of the state is replaced by another one according to their indices. The definition of the LFSR $LFSR_2$ is shown in Eq. (8.10) where a single byte x gets permuted on a bit-level where $x_i \; \forall \; i \; \in \{0, ..., 7\}$ denotes the i-th bit. Furthermore, a round constant RC_i is added to the state during each round of the tweakey schedule as shown in Fig. 8.2.

$$h' : \begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix} \mapsto \begin{bmatrix} 1 & 5 & 9 & 13 \\ 6 & 10 & 14 & 2 \\ 11 & 15 & 3 & 7 \\ 12 & 0 & 4 & 8 \end{bmatrix} \tag{8.9}$$

$$\begin{gathered} LFSR_2 : \\ (x_7 \parallel x_6 \parallel x_5 \parallel x_4 \parallel x_3 \parallel x_2 \parallel x_1 \parallel x_0) \\ \downarrow \\ (x_6 \parallel x_5 \parallel x_4 \parallel x_3 \parallel x_2 \parallel x_1 \parallel x_0 \parallel x_7 + x_5) \end{gathered} \tag{8.10}$$

In the first pass of Deoxys-II the authentication is performed. In the second pass the generated authentication tag is then used as part of the tweak for the encryption.

Tag Generation Figure 8.3 shows the tag generation. The function `int`() returns the input as a unsigned integer representation, i.e., a bit string. First, each message block M_i, $i \in [1, l]$ is encrypted with key K and tweak $T = 0000||int(i)$. Then, all encrypted blocks are added together. We omit the authentication of the AD, as it only adds the additional value `Auth` to the

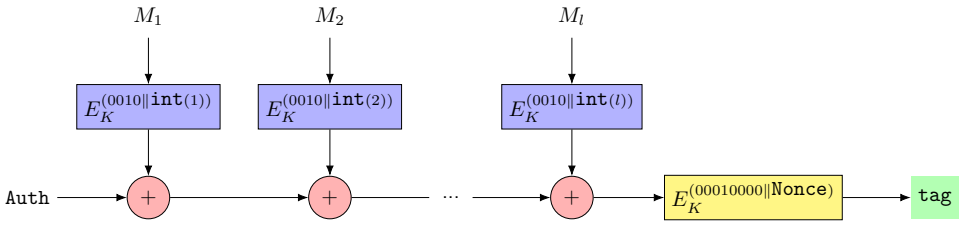


Figure 8.3.: Deoxys-II Tag Generation

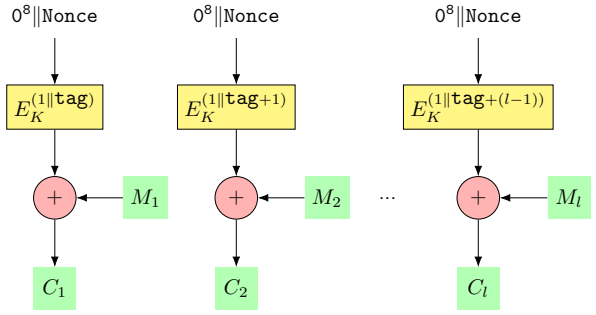


Figure 8.4.: Deoxys-II Message Encryption

XOR-tree. Finally, the result is encrypted once more with the nonce being part of the tweak. The output of the encryption is the `tag`.

Message Encryption Figure 8.4 shows the encryption. In contrast to the authentication not the message blocks are encrypted using E_K^T, but a zero-padded nonce is encrypted. While the key K stays the same for all blocks, the tweak T depends on the previously generated tag and the current block number $i \in [1, l]$. The so encrypted nonce is then added to the message block M_i to generate the cipher block C_i. This leads to Eq. (8.11).

$$C_i \;=\; M_i + E_K^{1\|\texttt{tag}+\text{int}(i-1)}(0^8 \parallel N) \tag{8.11}$$

8.2.2. PFA of Deoxys-II

In the following section we will outline how to apply PFA on either the generation of the tag or the message encryption.

Preliminaries In order to mount a PFA successfully, it is necessary to have access to the substitution layer's output, i.e., the S-box, as shown by the Zhang et al. [Zha+18]. This is not the case in Deoxys-II, as it performs an additional ATK with STK_{14} after the last round function f_{13} for key whitening. To overcome this problem, it is necessary to apply minor modifications to

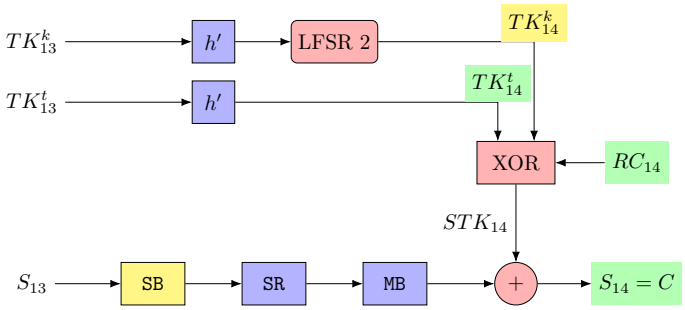


Figure 8.5.: PFA on the last round of Deoxys-BC-256.

the standard PFA approach: Fig. 8.5 shows the last steps of the Deoxys-BC. As it can be seen, they do not match the steps in Fig. 6.2. Therefore, we swap SR and SB notional, such that $f' = MB \circ SB \circ SR \circ ATK$. As both functions only operate on bytes, f' is equivalent to f. Still, the output of the substitution layer of Deoxys-BC is not directly accessible, because the linear MB operation is not omitted in the last round, like the `MixColumns` operation is in AES. Since MB and ATK are linear functions with respect to $GF(2)$, the order of execution can also be notionally swapped cf. Section 7.1.6. Therefore, the structure is essentially the same as shown in Fig. 6.2. However, now we must not add STK_{14} but the $\texttt{inverseMB}(STK_{14})$ to counteract the effect of the shifted `MB` on the key. Algorithm 3 shows the modified last round. It is behaviorally equivalent to the original last round. Furthermore, the effects of the tweakey schedule

Algorithm 3 Modified last round Deoxys-BC

$ATK(State, STK_{13})$
$SR(State)$
$SB(State)$
$STK_{14} \leftarrow TweaKeySchedule$
$ATK(State, inverseMB(STK_{14})$
$Cipher \leftarrow MB(State)$

must be taken into account: the result returned by the PFA equals $\texttt{inverseMB}(STK_{14})$. To reveal the last round key STK_{14}, `MB` must be applied again. When performing the PFA, not the ciphertext, but the $\texttt{inverseMB}(C)$ is used as the input to the PFA. Accordingly, not STK_{14} but $\texttt{inverseMB}(STK_{14})$ is revealed, which can easily tansformed to STK_{14} by applying the regular `MB`. To recalculate the key K, the sub tweakey must be split into the round tweak TK^t_{14} and the round key TK^k_{14}. The round tweak TK^t_{14} can be computed from the publicly known tweak T_0 by applying h' and $LFSR_2$ 14 times as shown in Fig. 8.2. Now that STK_{14}, TK^t_{14}, and RC_{14} are known, the last round tweak TK^k_{14} can be calculated as shown in Eq. (8.12).

$$TK^k_{14} = TK^t_{14} + RC_{14} + STK_{14} \tag{8.12}$$

This is also visualized in Fig. 8.5: the green colored values are known, the yellow values are under attack. There are two possible targets in Deoxys-II to mount a PFA: either the tag generation or the message encryption.

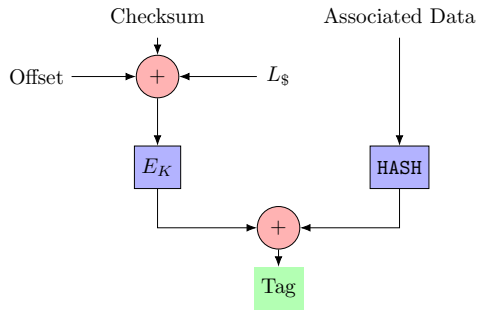


Figure 8.6.: OCB Tag Generation

Attack on the Message Encryption As depicted in Fig. 8.4, each message block can be encrypted separately. Thus, a PFA can be mounted on any block, as long as `tag`, M_i and C_i are known. In order to access the output of the encryption E_T^K, it is necessary to add the message block M_i to the corresponding cipher block C_i. Thus, in contrast to the original PFA, the PFA on Deoxys-II is not a ciphertext only attack, as it requires a ciphertext-plaintext pair. After mounting a PFA, that takes the effects of `MB` in the last round into account, the key K_0 can be recalculated by applying the inverted h' and $LFSR_2$ functions 14 times.

Attack on the Tag Generation: Alternatively, the last encryption E_K^T, of the tag generation, marked in yellow in Fig. 8.3 can be attacked. It calculates the tag as shown in Eq. (8.13).

$$\texttt{tag} \leftarrow E_K(0001 \parallel 0^4 \parallel N, \texttt{intermediateTag}) \tag{8.13}$$

In this case an attacker only requires knowledge about the `Nonce` and the `tag` in order to mount the PFA. Therefore, making the PFA on Deoxys-II a tag-only attack again.

8.3. OCB

OCB is a mode of operation for AES proposed by Krovetz and Rogaway [KR16]. For the rest of this work we refer to the version of OCB which uses AES with a key length of 128 bit by OCB.

8.3.1. Structure

The Structure of OCB can be decomposed into two building blocks: the authentication of the AD and the message encryption.

Message Authentication Figure 8.6 shows the tag generation. The associated data is authenticated by a `HASH` function that returns either the hash over the AD, or the zero string 0^{128} for empty AD. The message is authenticated by a standard AES encryption E_K: First, all (padded) message blocks are added to form the `Checksum`. Then, two additional masks (Offset_l and $L_\$$) are added. Next, the result is encrypted by a standard AES encryption E_K. Finally, both outputs are added again to form the tag.

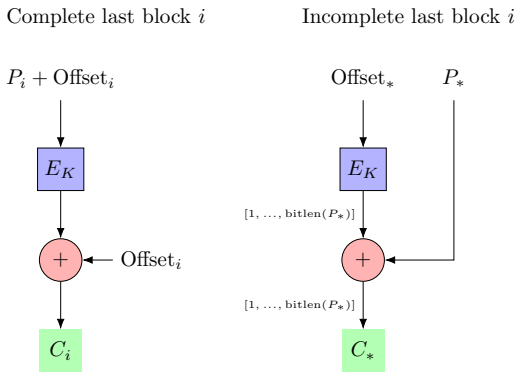


Figure 8.7.: OCB Message Encryption

Message Encryption Figure 8.7 shows the encryption of both complete and partial message blocks. The encryption of a complete message block M_i is based on the addition of a mask Offset_i prior to the application of E_K. After the encryption of the message block with E_K, the same mask Offset_i is added again to the intermediate result which then forms the ciphertext. The encryption of a partial message block M_* differs significantly: For a partial message block, the ciphertext C_* is calculated as the addition of the last plaintext block P_* with the encrypted mask Offset_* using E_K.

8.3.2. PFA of OCB

PFA can be applied either to the encryption of the last incomplete message block or on the calculation of the tag, when no associated data is used. Both attacks reduce the attack to the standard PFA of AES, as proposed by [Zha+18]. Consequently, the input of E_K is not needed for the attack.

Attack on the message encryption Similar to the attack of Dobraunig et al. [Dob+16b], we also attack the encryption of the last incomplete message block. Since no mask is used when encrypting the last incomplete message block, this is a convenient attack target. The processing of the last message block M_* is shown in Fig. 8.8. The values, that are used for the attack, are colored in yellow. First, the offset Offset_* is encrypted with AES. Then, the result of the encryption is xored with the last plaintext block. If the last plaintext-block and the resulting cipher is known, the output of the AES is known up to the length of the last plaintext-block. The original PFA on AES leads to a partial key recovery, where the length depends on the length of the last message block. The number of remaining key candidates $\#K$ can be calculated as shown in Eq. (8.14)

$$\#K = 2^{(128 - bitlen(P_*))} \tag{8.14}$$

By choosing a large last block, the brute force effort can be minimized. During our simulations we assumed $bitlen(P_*) = 120$, therefore 256 key candidates remain.

Attack on the tag generation The PFA can also be applied to the tag generation, if no AD is processed by OCB. An empty AD results in adding 0^{128} to the result of an AES encryption E_K

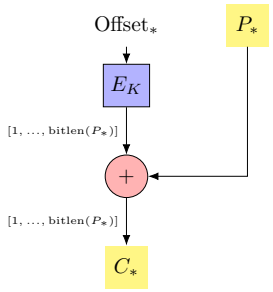


Figure 8.8.: PFA on OCB's last incomplete message block

Table 8.1.: Applicability of PFA

Cipher-Family	Version	Applicable
Deoxys-II	Deoxys-II-128-128	✓
	Deoxys-II-256-128	✗
OCB	with AES-128, all tag-sizes*	✓
	with AES-192, all tag-sizes	✗
	with AES-256, all tag-sizes	✗
COLM	COLM_0	✓
	COLM_{127}	✓

* Attack on Tag-generation: Brute force effort $2^{128-\text{Taglen}}$

in the last step of the tag generation. Therefore, E_K can be attacked with the standard PFA.

8.4. Results

In the following section we will discuss the results of applying PFA to Deoxys-II, OCB and COLM. We verified the efficiency of the attacks by means of simulation. Due to the reason that Deoxys-II, OCB and COLM use AES or a slightly modified version of AES as underlying block cipher, the results are similar to the results of Zhang et al. [Zha+18]. The biggest challenge during the application of PFA to AEAD schemes is to map the prerequisites of PFA onto the AEAD scheme under attack. Table 8.1 shows which versions of Deoxys-II, OCB and COLM are vulnerable to PFA. As there are several functions within an AEAD scheme that can be attacked, i.e., the tag generation and message encryption, the attacks target different functions. Table 8.2 summarizes the different strategies. We will now outline the results of our simulations at the example of Deoxys-II, for simplicity reasons we used the analysis strategy where t_{min} can be determined and the original value of the S-box v is known, Zhang et al. refered to his approach as *Strategy I* [Zha+18]. Furthermore, we assume a single altered S-box entry. Fig. 8.9 shows the average (1000 attacks) number of candidates for a key byte, for a PFA applied to the `tag` generation of Deoxys-II. As one can see in Fig. 8.9 approximately 1600 faulty tags are required

Table 8.2.: Requirements for each Attack Strategy

Cipher	Attacked Function	Requirements
Deoxys-II	Tag-Generation	Faulty tags, nonce
	Message-Encryption	Tags, faulty cipher and plain-texts
OCB	Tag with AD empty	Faulty tags
	Incomplete Message-Block	Incomplete cipher- and corresponding plain-texts
COLM	Message-Encryption	Faulty cipher-texts

Table 8.3.: Number of needed encryptions.

Cipher	Attack Strategy	n_{avg}
Deoxys-II	Tag-Generation	2139.00
	Message-Encryption	2274.83
OCB	Tag without AD	2270,58
	Last incomplete Message-Block	2248.38
COLM	Message-Encryption	2078.28

on average to reveal the correct ($N_K = 1$) key byte, also it becomes clear that there is barely a difference between the average of 1000 attacks and one particular attack. When attacking the underlying encryption scheme, which usually processes more than one message block, we can extract more information about the key per authenticated encryption compared to an attack on the tag. As an example if ten messages blocks are processed per authenticated encryption, only a tenth of the number of faulty tags, i.e., 160 is required to determine the correct key byte. This is shown in Fig. 8.10. The results of our simulations for an attack on the whole key are summarized in Table 8.3 where one can see the average number of required encryptions or tag generations n_{avg}, with respect to the attack strategy. In order to compare the attack on the tag generation with the attack on the encryption (which usually encrypts more than one message block) we encrypted only a single block of data, i.e., 128 bit per authenticated encryption.

8.5. Summary

The fact that PFA can be applied to AEAD schemes with minor modifications should be considered a threat. This is mainly due to the fact that the fault model assumed by PFA is rather simple to achieve, i.e., a faulted constant which is processed during a cryptographic operation. Especially when we consider resource constraint devices where it may be necessary to generate S-boxes dynamically instead of a lookup table. Furthermore, one of the main benefits when working with PFA is the fact that no fault injections are required at runtime, which again reflects

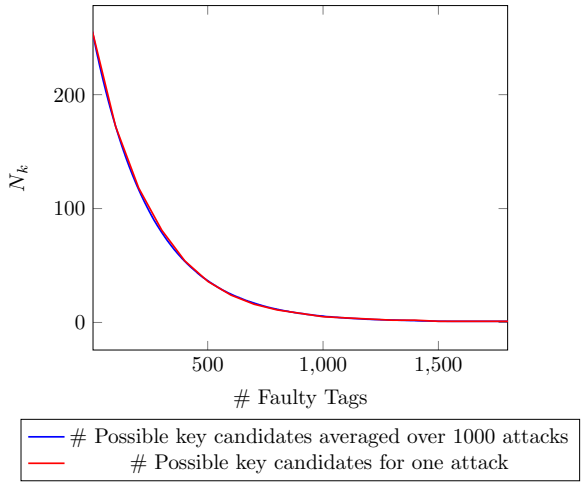


Figure 8.9.: PFA on tag generation Deoxys-II - single byte

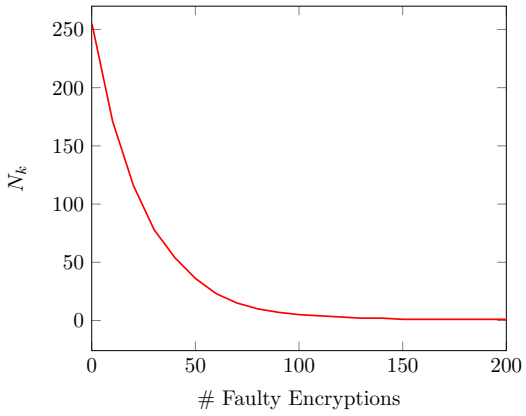


Figure 8.10.: PFA on encryption Deoxys-II - single byte

the persistent nature of the fault model introduced by the authors of [Zha+18].

9. Statistical Ineffective Fault Analysis of Gimli

Statistical Ineffective Fault Analysis (SIFA) was proposed by Dobraunig et al. [Dob+18b]. The main advantage of SIFA is the fact that the only requirement is an intermediate state with a biased distribution, i.e., a distribution that deviates from the uniform distribution. Also, in contrast to DFA it is not necessary to have tuples of faulty and correct encryptions. SIFA is based on the idea to combine two different kinds of FIA in order to benefit from the advantages of both approaches. The two FIAs in question are the so called Ineffective Fault Analysis (IFA) as proposed by Clavier [Cla07], and Statistical Fault Analysis (SFA) as proposed by Fuhr et al. [Fuh+13]. The main benefit of IFA is the ability to overcome countermeasures due to the ineffective nature of the assumed fault model. The main benefit of SFA is the ability to deal with noisy fault injections, i.e., fault injections which do not adhere to the specified fault model. Consequently, SIFA can break traditional countermeasures against fault attacks like detection-based or infection-based countermeasures due to the ineffective nature of the faults required by SIFA. Also, the robustness against noisy fault injection makes SIFA especially suitable for scenarios where no precise fault model can be achieved, e.g, faults generated by a low-cost fault injection setup cf. Guillen et al. [GGS17].

An improvement to SIFA was proposed by Vafaei et al. in order to increase the number of exploitable faults the bias caused by effective faults is evaluated which is therefore called Statistical Effective Fault Analysis (SEFA) [Vaf+22]. Furthermore, Vafaei et al. proposed the combination of SIFA and SEFA which is referred to be Statistical Hybrid Fault Analysis (SHFA) [Vaf+22]. SHFA can be regarded as adaptive attack strategy which automatically chooses either SIFA or SEFA according to the achievable injected faults.

State of the art So far the priciples of SIFA were applied to a variety of cryptographic algorithms ranging from block ciphers as AES [Dob+18b] to authenticated encryption schemes like Ketje, Keyak [Dob+18a] and Ascon [RAD19]. Therefore, SIFA was proven to be a versatile type of FIA that can be mounted on block ciphers and AEAD schemes as well.

Contributions We present the first SIFA of Gimli [Ber+17]. Gimli was a second round candidate of the LWC standardization process initiated by the National Institute of Standards and Technology (NIST) [NIS17]. Additionally, we verify the efficiency of our attacks by means of simulation. In addition, we evaluate the influence of the fault model on the rate of ineffective faults.

Organization Chapter 9 is structured as follows: Section 9.1 presents the NIST-LWC candidate Gimli. Section 9.2 introduces our SIFA on Gimli. Section 9.3 provides the results of our proposed attack on Gimli which were obtained by means of simulation. Section 9.4 provides a summary of the conducted SIFA on the NIST-LWC candidate.

9.1. Gimli

Gimli is a suite of cryptographic primitives based on the Gimli-permutation proposed by Bernstein et al. [Ber+17]. It participates in the NIST lightweight cryptographic project for authenticated encryption and hash. In this chapter, we focus on the Gimli-Cipher, a family of AEAD schemes.

9.1.1. Gimli-Permutation

Gimli's permutation is based on a 384-bit-state. As shown in Eq. (9.1), the state is defined as a 3×4 matrix: the rows are denoted by a, b, c; the columns are denoted by 0, 1, 2, 3; the round is denoted by r. For example a_1^{11} denotes to the second 32 bit word before the execution of the 11th round.

$$\text{State} := \begin{pmatrix} a_0^r & a_1^r & a_2^r & a_3^r \\ b_0^r & b_1^r & b_2^r & b_3^r \\ c_0^r & c_1^r & c_2^r & c_3^r \end{pmatrix} \tag{9.1}$$

Algorithm 4 describes how this state is permuted during 24 consecutive rounds. The rounds are enumerated in reverse order, i.e, the permutation starts with round 24 and ends with round 1. During each round, the state is first substituted and permuted (SP-Box). Every second round, the state is mixed linearly (alternating between either a small swap or a big swap). Finally, every fourth round, a constant is added.

Algorithm 4 Gimli Permutation

```
function Permute(a, b, c)                                          ▷ Input State
    for r = 24 downto 1 do
        for j = 0 to 3 do
            t_a ← a_j ⋘ 24                                         ▷ SP-Box
            t_b ← b_j ⋘ 9
            t_c ← c_j

            a_j ← t_c ⊕ t_b ⊕ ((t_a & t_b) ≪ 3)
            b_j ← t_a ⊕ t_b ⊕ ((t_a | t_c) ≪ 1)
            c_j ← t_a ⊕ (t_c ≪ 1) ⊕ ((t_b & t_c) ≪ 3)
        end for

        if r mod 4 = 0 then
            a_0||a_1||a_2||a_3 ← a_1||a_0||a_3||a_2                 ▷ Small Swap
        else if r mod 4 = 2 then
            a_0||a_1||a_2||a_3 ← a_2||a_3||a_0||a_1                 ▷ Big Swap
        end if

        if r mod 4 = 0 then
            a_0 ← a_0 ⊕ 0x9e377900 ⊕ r                             ▷ Constant Addition
        end if
    end for
    return (a, b, c)                                               ▷ Output State
end function
```

9.1.2. Gimli-AEAD

Gimli-Cipher is a sponge based AEAD scheme with a rate of 128 bit and a capacity of 256 bit. The rate matches a, the capacity matches the concatenation of b and c. Fig. 9.1 depicts the four phases during an AEAD. As the exploited fault is ineffective in respect to the output, the exact behavior of the AEAD scheme is secondary and a brief description of the four phases is sufficient:

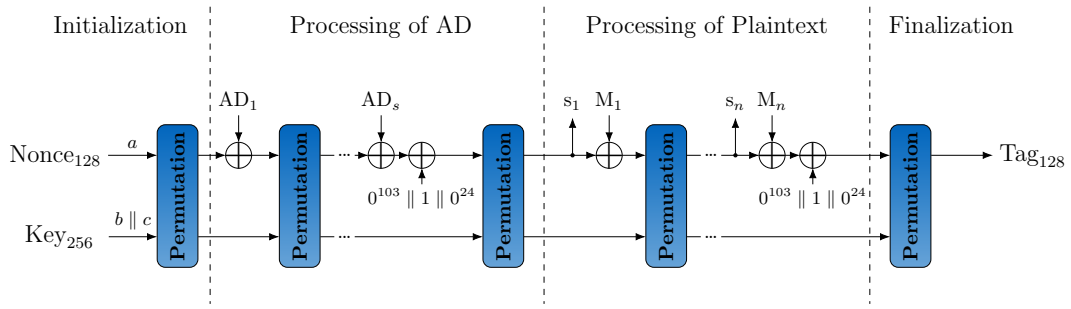


Figure 9.1.: Gimli Sponge Construction

First, the state is initialized with a 128 bit nonce, and the 256 bit key as shown in Eq. (9.2).

$$\begin{aligned} &\text{Nonce:} && a_0^{24} \ldots a_3^{24} \leftarrow n_0\, n_1\, n_2\, n_3 \\ &\text{Key:} && b_0^{24} \ldots b_3^{24} || c_0^{24} \ldots c_3^{24} \leftarrow k_0\, k_1\, \ldots\, k_7 \end{aligned} \tag{9.2}$$

Second, the associated data blocks AD_i are absorbed in chunks of 128 bit. Subsequently, the message block key s_i is squeezed. Depending on the mode, either the ciphertext $C_i = M_i \oplus s_i$ or the plaintext $M_i = C_i \oplus s_i$ is generated. In any case, next, the plaintext M_i is absorbed. Finally, the tag is calculated. Between all phases and absorbed blocks the Gimli-permutation is invoked. Incomplete blocks are padded. Additionally, there is a domain separation between the processing of AD, the processing of plaintext, and finalization. For decryption, the tag is not output, but compared to the received tag. If both tags match, the plaintext is released, otherwise, the empty string is output.

9.2. SIFA of Gimli

We target the decryption of Gimli because AEAD schemes only release the plaintext if the computed tag matches the original tag. This behavior can be exploited to distinguish between effective and ineffective faults, as an effective fault results in a tag mismatch.

9.2.1. Fault Injection Location

For the SIFA of Gimli we evaluated different locations where an induced ineffective fault can be exploited. Similar to the attacks on Ketje and Keyak [Dob+18a], we use the nonce and a hypothesis of the target partial subkey k_H to calculate an intermediate value of Gimli. In general, this is possible for any intermediate value during the decryption of Gimli. However, in order to reduce the number of involved key-bits of the intermediate value and the number of hypotheses N_H, it is desirable to attack the early rounds of the first Gimli-Permutation. Fig. 9.2 shows an attack mounted during the initialization phase. When we target the first Gimli-Permutation, we can choose from one of the 24 rounds. The Substitution Permutation Box (SP-Box) of the Gimli-Permutation poses the best attack target, due to the involved non-linearity. A possible position to inject an ineffective fault into the SP-Box is colored red in Fig. 9.3. We mount the

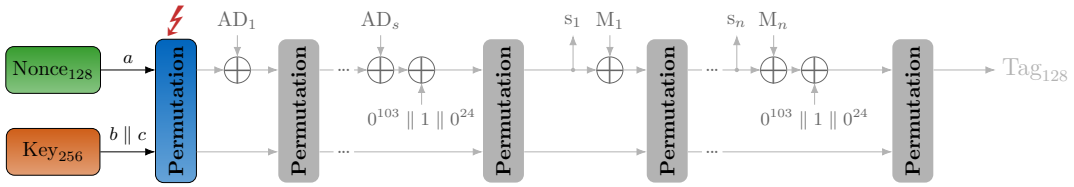


Figure 9.2.: Fault Injection Location Gimli

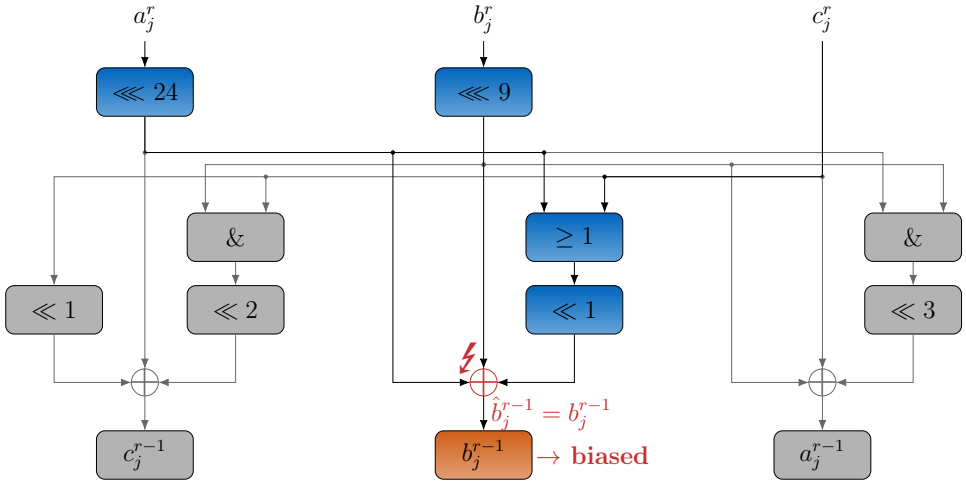


Figure 9.3.: Fault Injection Location SP-Box

attack on a biased b_j^r, however a similar reasoning can be applied for a biased a_j^r or c_j^r. The attacked round is a trade-off between the number of recoverable key-bits n_{keybits} and number of possible key hypotheses N_H. The earlier the attack, the simpler are the equations for the intermediate values, but also fewer key bits can be revealed. The later the attack, the higher is the number of involved key bits n_{keybits} and thus, more hypotheses N_H must be checked. The number of hypotheses N_H grows exponentially with the number of involved key bits n_{keybits} as shown in Eq. (9.3).

$$N_H \sim 2^{n_{\text{keybits}}} \tag{9.3}$$

In order to determine the exact number of involved key bits the dependencies of an intermediate value must be traced back to the initialization phase where the state gets initialized using the known nonce and the unknown key.

9.2.2. Calculation of Intermediate Values

The dependencies of an intermediate value under attack are related to the fault injection location. We will demonstrate an attack of bit $b_{0,7}^r$. Hereby $b_{0,7}^r$ denotes the seventh bit of the word b_0^r during round r. The resulting dependencies of $b_{0,7}^r$ with respect to the according injection location are shown in the second row of Table 9.1.

r	n_{keybits}	Dependencies of $b_{0,7}^{r}$
23	2	$b_{0,7}^{r=23} = k_{0,31} \oplus n_{0,15} \oplus (n_{0,14} \mid k_{4,6})$
22	11	$b_{0,7}^{r=22} = k_{0,21} \oplus n_{0,6} \oplus (n_{0,5} \mid k_{4,29}) \oplus k_{5,15} \oplus k_{1,6} \oplus$ $\oplus (n_{1,20} \;\&\; k_{1,3}) \oplus c_{15} \oplus [(k_{5,14} \oplus k_{1,5} \oplus (n_{1,19} \;\&$ $\&\; k_{1,2}) \oplus c_{14}) \mid (n_{0,14} \oplus k_{4,5} \oplus (k_{4,4} \;\&\; k_{0,27}))]$
21	37	cf. Eq. (9.4)

Table 9.1.: Dependencies of $b_{0,7}^{r}$ for different injection locations

$$
\begin{aligned}
b_{0,7}^{r=21} &= b_{0,30}^{r=22} \oplus a_{0,15}^{r=22} \oplus (a_{0,14}^{r=22} \mid c_{0,6}^{r=22}) \\
\\
b_{0,30}^{r=22} &= b_{0,21}^{r=23} \oplus a_{0,15}^{r=23} \oplus (a_{0,5}^{r=23} \mid c_{0,29}^{r=23}) \\
a_{0,15}^{r=22} &= c_{0,15}^{r=23} \oplus b_{0,6}^{r=23} \oplus (a_{0,20}^{r=23} \;\&\; b_{0,3}^{r=23}) \\
a_{0,14}^{r=22} &= c_{0,14}^{r=23} \oplus b_{0,5}^{r=23} \oplus (a_{0,19}^{r=23} \;\&\; b_{0,2}^{r=23}) \\
c_{0,6}^{r=22} &= a_{0,14}^{r=23} \oplus c_{0,5}^{r=23} \oplus (c_{0,4}^{r=23} \;\&\; b_{0,27}^{r=23}) \\
\\
b_{0,21}^{r=23} &= k_{0,12} \oplus n_{0,29} \oplus (n_{0,28} \mid k_{4,20}) \\
a_{0,6}^{r=23} &= k_{5,6} \oplus k_{1,29} \oplus (n_{1,11} \;\&\; k_{1,26}) \oplus c_{6} \\
a_{0,5}^{r=23} &= k_{5,5} \oplus k_{1,28} \oplus (n_{1,10} \;\&\; k_{1,25}) \oplus c_{5} \\
c_{0,29}^{r=23} &= n_{0,5} \oplus k_{4,28} \oplus (k_{4,27} \;\&\; k_{0,18}) \\
\\
c_{0,15}^{r=23} &= n_{0,23} \oplus k_{4,14} \oplus (k_{4,13} \;\&\; k_{0,4}) \\
b_{0,06}^{r=23} &= k_{0,29} \oplus n_{0,14} \oplus (n_{0,13} \mid k_{4,5}) \\
a_{0,20}^{r=23} &= k_{5,20} \oplus k_{1,11} \oplus (n_{1,25} \;\&\; k_{1,8}) \oplus c_{20} \\
b_{0,03}^{r=23} &= k_{0,26} \oplus n_{0,11} \oplus (n_{0,10} \mid k_{4,2}) \\
\\
c_{0,14}^{r=23} &= n_{0,22} \oplus k_{4,13} \oplus (k_{4,12} \;\&\; k_{0,3}) \\
b_{0,05}^{r=23} &= k_{0,28} \oplus n_{0,13} \oplus (n_{0,12} \mid k_{4,4}) \\
a_{0,19}^{r=23} &= k_{5,19} \oplus k_{1,10} \oplus (n_{1,24} \;\&\; k_{1,7}) \oplus c_{19} \\
b_{0,2}^{r=23} &= k_{0,25} \oplus n_{0,10} \oplus (n_{0,9} \mid k_{4,1}) \\
\\
a_{0,14}^{r=23} &= k_{5,14} \oplus k_{1,5} \oplus (n_{1,19} \;\&\; k_{1,2}) \oplus c_{14} \\
c_{0,05}^{r=23} &= n_{0,13} \oplus k_{4,4} \oplus (k_{4,3} \;\&\; k_{0,26}) \\
c_{0,04}^{r=23} &= n_{0,12} \oplus k_{4,3} \oplus (k_{4,2} \;\&\; k_{0,25}) \\
b_{0,27}^{r=23} &= k_{0,18} \oplus n_{0,3} \oplus (n_{0,2} \mid k_{4,26})
\end{aligned}
\tag{9.4}
$$

A fault injection in the very first round, i.e., after round 24, to attack the bit $b_{0,7}^{23}$ only affects

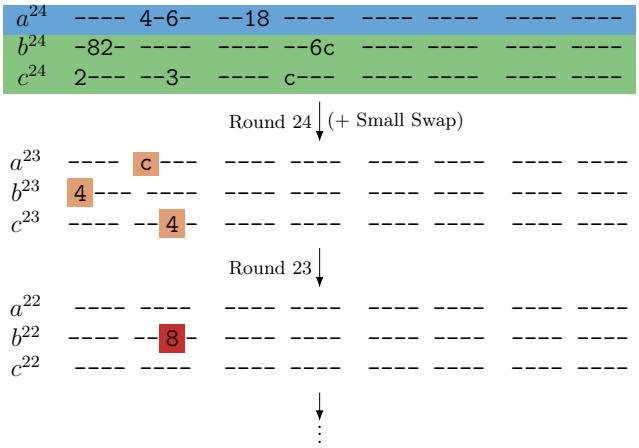


Figure 9.4.: Dependencies of $b_{0,7}^{22}$

two key bits and therefore, does not offer a big advantage in terms of recoverable key bit this is shown in the first row of Table 9.1. If the fault is injected one round later, i.e., after round 23, eleven key bits are involved in the computation of the bit $b_{0,7}^{22}$ this is shown in the second row of Table 9.1. By biasing the intermediate bit $b_{0,7}^{21}$ again a round later, an attacker can utilize 37 involved key-bits, some of the involved key-bits can only be recovered in the form of a sum denoted in blue. Involved key-bits lead to a dependency due to the path along which they properagate through the Gimli-Permutation. The bit wise dependencies after each Gimli-round are visualized similarly to Dobraunig et al. [DEM15]. Involved bits, i.e., dependencies are represented by a 1 independent bits are represented by '0' or '-', e.g., `c=1100` means that only bit 3 and 2 of this nibble are involved in a computation. The position of the key is colored in green and the nonce in blue. Fig. 9.4 shows, which bits are involved in the computation of the intermediate bit $b_{0,7}^{22}$ which is colored in red. Even though 11 bits of the key are involved in the calculation of the intermediate value $b_{0,7}^{22}$, not all of them can be identified distinctively due to linear dependencies of the involved key bits. Eq. (9.5) shows these linear dependencies. Each of the three sums are affected by three different key bits. Therefore, only the sums k_{s1}, k_{s2} and k_{s3}, but not the individual key bits can be recovered.

$$\begin{aligned} k_{s1} &= k_{0,21} \oplus k_{5,15} \oplus k_{1,6} \\ k_{s2} &= k_{5,14} \oplus k_{1,5} \\ k_{s3} &= k_{4,5} \ \oplus (k_{4,4} \ \& \ k_{0,27}) \end{aligned} \tag{9.5}$$

Consequently, an attack on the intermediate value $b_{0,7}^{22}$ reveals only the key bits $k_{4,29}$, $k_{1,3}$ and $k_{1,2}$. However, the key sums can also be used to build hypotheses. This results in an advantage of 2^6 compared to brute-forcing each individual bit of the involved key bits. An attack on the intermediate state $b_{0,7}^{21}$ already involves 37 key bits. Taking linear dependencies into account, the number of hypotheses is 2^{22}. A graphical representation of the dependencies of $b_{0,7}^{21}$ is shown in Fig. 9.5 Going one round further ($b_{0,7}^{20}$) increase the number of involved key bits to 168. However, testing 2^{168} hypotheses is not feasible in a reasonable amount of time. Thus, the attack on the intermediate states in rounds 22 and 21 offer a reasonable trade-off between the number of hypotheses and recoverable key bits.

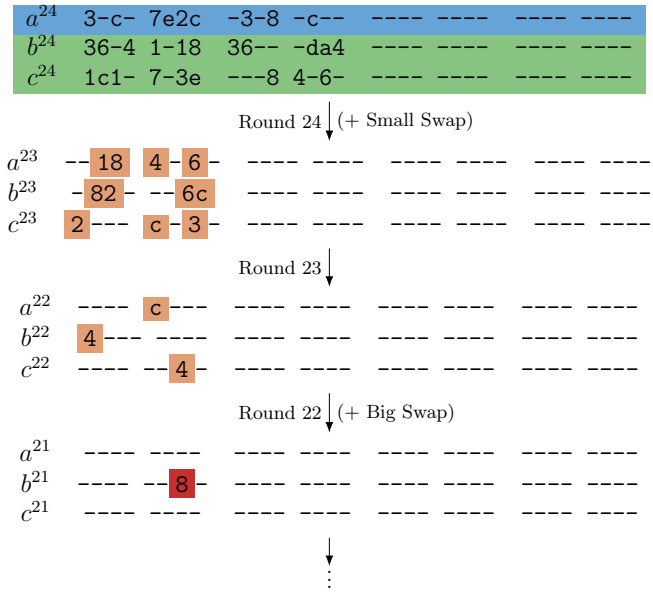


Figure 9.5.: Dependencies of $b^{21}_{0,7}$

9.2.3. Fault Model

In Section 6.3 the influence of some typical fault models onto the FDT's was shown in Table 6.1 at the example of a two bit intermediate state. However, the fault models are not limited to 2-bit but can be applied to words with variable width w. Since we cannot choose the word-width of the implementation of Gimli but still want to evaluate the distribution of a single bit, it is important to evaluate, if a byte based fault model also biases each bit separately. For example a fault of width $w = 8$ is the equivalent to a byte based fault model. We simulated faults with $w = 8$ according to the probabilistic bit flip fault model where a flip from $1 \rightarrow 0$ occurs with probability $P_{1\rightarrow 0} = \frac{2}{3}$ and a flip $0 \rightarrow 1$ with probability $P_{0\rightarrow 1} = \frac{1}{3}$. This biased bit flip probabilities for a one bit intermediate value b result in the histogram shown in Fig. 9.6 This behavior is the same as the FDT shown in Table 6.1d which depicts the two dimensional case. The bias of the 8-bit intermediate value $b^{22}_{0,0-7}$ caused by an ineffective fault is shown in Fig. 9.7. The nearly normal distributed values without any fault are colored in green whereas all values leading to ineffective faults are colored in blue. If one compares the histogram as shown in Fig. 9.7 with the previously introduced FDTs as shown in Table 6.1, it becomes clear that this distribution can be attacked due to the deviation from the uniform distribution which is directly recognizable. Based on the simulations as shown in Fig. 9.7 we decided to use a byte based fault model, i.e., $w = 8$ during the explanation of the attack strategy.

9.2.4. Attack Strategy

For the attack it is necessary to generate decryptions under the influence of an ineffective fault. As a result of Gimli being a AEAD scheme the collected decryptions are all under the influence of an ineffective fault otherwise there would be no output due to a tag mismatch. After a sufficient number of decryptions N_d is obtained we calculate the hypothetical intermediate bit

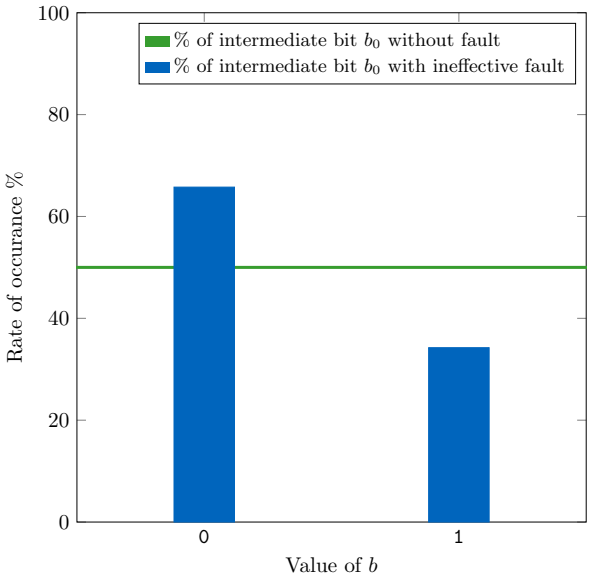


Figure 9.6.: Histogram of intermediate values b

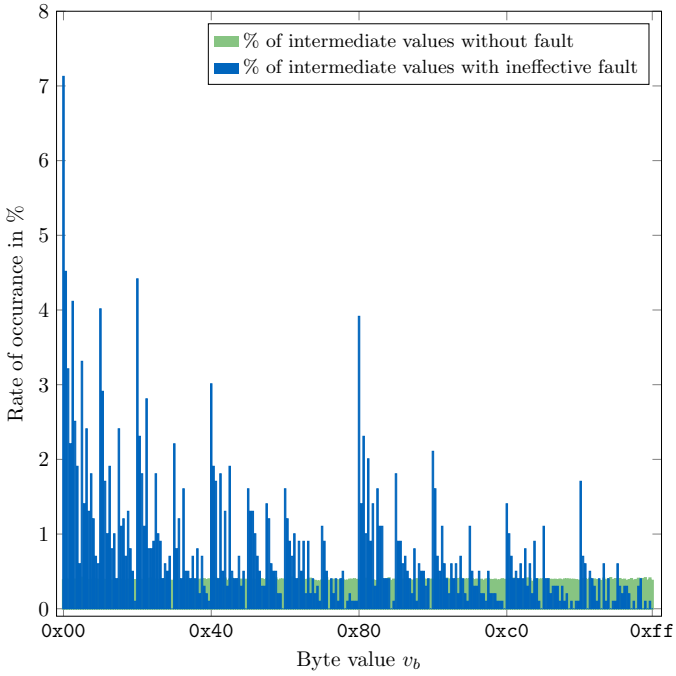


Figure 9.7.: Histogram of intermediate values $b^{22}_{0,0-7}$

inter. The calculation of the intermediate bit is done with respect to the possible key hypotheses and all obtained nonces n. The distribution of *inter* is then ranked by according to the SEI. Now that each hypothetical distribution has been assigned an SEI, the correct key hypothesis can be identified as the one with the largest SEI. The algorithmic representation of the attack is shown in Algorithm 5. First the hypothetical intermediate values are calculated for all possible keys. Then for each key hypothesis the SEI of the intermediate distribution is calculated and stored. If a new SEI is greater than or equal to the old one the corresponding key hypothesis is used as the new correct hypothesis. After all hypotheses have been processed, the algorithm terminates. In our attack strategy, it is necessary to choose an appropriate intermediate value

Algorithm 5 SIFA of Gimli

```
N_H ← 2^{n_keybits}
n[N_d] ← loadNonces()
maxSEI ← 0
corrHypo ← 0
for i = 0 to N_H do
    for j = 0 to N_d do
        inter[i][j] ← calcIntermediateBit(i, n[j])
    end for

    SEI[i] ← calcSEI(inter[i][*])

    if SEI[i] ≥ maxSEI then
        maxSEI ← SEI[i]
        corrHypo ← i
    end if
end for
return corrHypo
```

to attack as the involved key bits which can be recovered for each intermediate value differ. As introduced in Section 9.2.2, it is possible to calculate some key bits directly and some key bits only in the form of a sum. The computation of the intermediate state $b_{0,7}^{22}$ involves 11 key bits as shown in the dependency equations in Table 9.1. However, 8 key bits influence $b_{0,7}^{22}$ only in the form of a sum as shown in Eq. (9.5). The computation of $b_{0,7}^{22}$ involves $n_{\text{keybits}} = 11$ but only hypotheses on 6 key bits are required as the remaining key bits only appear in the form of a sum. Therefore, the number of hypotheses N_H shrinks from 2^{11} to 2^6. As a result three key-bits $k_{4,29}$, $k_{1,3}$ and $k_{1,2}$ can be determined uniquely. The same effect also occurs when we target the intermediate value at the same position one round later, i.e., $b_{0,7}^{21}$. The computation of the intermediate state $b_{0,7}^{21}$ involves 37 key bits as shown in the dependency equations in Table 9.1. Therefore, 15 key-bits $k_{4,26}$, $k_{4,20}$, $k_{4,5}$, $k_{4,4}$, $k_{4,3}$, $k_{4,2}$, $k_{4,1}$, $k_{1,26}$, $k_{1,25}$, $k_{1,8}$, $k_{1,7}$, $k_{1,2}$, $k_{0,26}$, $k_{0,25}$ and $k_{0,18}$ can be determined uniquely. Furthermore, the sum of 22 key bits in the form of 7 sum can be determined. From the 37 involved key bits we are able to obtain an advantage of 22 bits compared to the brute force effort over all involved key bits. Due to some ambiguity in the large equation for $b_{0,7}^{21}$ the SIFA reveals three candidates with the same SEI after 340 decryptions under the influence of ineffective faults. The ambiguity is caused by some nonce bits that do not differ when they cause an ineffective fault. Although the described ambiguity is present, there is always the correct key-hypothesis among those three candidates. By the injection of 8-bit ineffective fault, hypotheses can be build on 8 intermediate bits that can be evaluated simultaneously with almost no extra computational effort. With this we get an advantage of at most $8 \cdot 6 = 48$ bits when attacking round 22 and at most $8 \cdot 22 = 176$ bits when attacking round 21. In order to obtain the complete key which is loaded during the initialization phase of Gimli it is necessary to repeat the proposed attacks with varying intermediate states under attack until all key bits are recovered.

Due to the fact that we are also able to recover the sums of certain key-bits the real effectiveness

of our attack is higher than the stated values which provide the worst case estimation. The required number of ineffective faults to recover the whole key involves building up an equation system, which also exploits the knowledge gained from the sums of key bits. As a result of the complicated estimation for the full key recovery, we opted to use the more intuitive variant (worst case estimation). The worst case estimation is calculated as the division of the size of the full key by the number of recoverable key bits.

9.3. Results

Now that we have clarified the prerequisites for the attack, we will present the results. First we will evaluate the influence of the fault width w on the ineffectiveness rate of the injected faults. Second the obtained results for the attack on $b_{0,7}^{22}$ and $b_{0,7}^{21}$ are presented. Both attacks exploit the bias of an ineffective fault with fault width $w = 8$ bit injected after round 23 respectively 22 under the assumption of a probabilistic bit flip fault model.

9.3.1. Influence of fault width on ineffectiveness rate

SIFA exploits the bias present in an intermediate state independently of the assumed fault model. In practice, it is usually assumed that an attacker has no information about the FDT which is caused by the ineffective fault injection. Nevertheless, the only prerequisite for a successful attack is that an intermediate value follows a biased distribution. The biased distribution is indicated by the diagonal of the FDT which follows a non-uniform distribution as shown in Table 6.1. Furthermore, we typically cannot choose the target architecture where GIMLI-AEAD is run on, which can either be a software implementation running on a microcontroller or a hardware implementation running on a FPGA or ASIC. If we consider the case of a software implementation of GIMLI-AEAD, then the fault width w will be the same as the word with of the micro-controller. If we consider a hardware implementation of GIMLI-AEAD the fault width w is usually dependent of the implementation. In the following we use the typical fault models *Random And*, *Stuck at Zero*, and *Probabilistic Bit Flip* exemplary to simulate a fault on a software implementation, the same reasoning can also be applied to hardware implementations. Faults are injected during round 23 of the first GIMLI-Permutation on state b_0^{22}. The width w of the fault ranges from 1 to 32 bit with $w \in \{1, 4, 8, 16, 32\}$. By calculating the number of ineffective faults n_{ineff} divided by the number of total encryptions N we obtain the ineffectiveness rate as shown in Eq. (9.6).

$$r_{\text{ineff}} = \frac{n_{\text{ineff}}}{N}. \tag{9.6}$$

The ineffectiveness rate r_{ineff} with respect to the fault width w is shown in Fig. 9.8. As one can see the ineffectiveness rate r_{ineff} decreases almost linearly with the assumed fault width w. The linear decrease of the ineffectiveness rate occurs independently of the three fault models. The ineffectiveness rate of the *Probabilistic Bit Flip* fault model is dependent on the assumed bit flip probabilities we decided to use this model to provide a worst case estimation. In practice this means that attacking a 32-bit software implementation of GIMLI, should be feasible according to Fig. 9.8. Especially the *Random And* model offers a significant rate of ineffective faults at $w = 32$ bit. However, due to the very low ineffectiveness rate for the other fault models, a number of more than 10^9 total encryptions is required for the attack. For the sake of simplicity further simulations where done with a fault width $w = 8$ bit in order to minimize the computational effort of generating ineffective faults. The computational effort is not based on the size of the underlying word, but the rate of ineffective faults with respect to the assumed fault width and the round under attack.

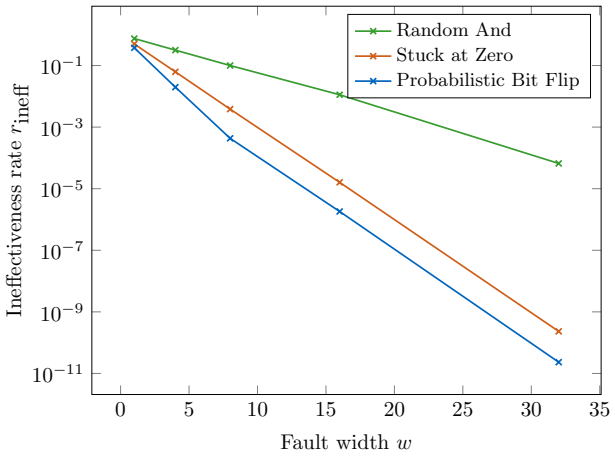


Figure 9.8.: Ineffectiveness rate r_{ineff} of different fault models

9.3.2. Attack on $b_{0,7}^{22}$

The attack on the intermediate state $b_{0,7}^{22}$ is able to retrieve the involved key bits correctly after approximately 180 ineffective faults. The number of required encryptions under the influence of an ineffective fault is shown in Fig. 9.10 where we used the SEI as statistical metric. In Fig. 9.10 the best wrong hpyothesis is colored red and the correct hypothesis in blue. Furthermore, it is important to notice, that after the point both are crossing line, the correct hypothesis keeps a significantly higher value. Figure 9.9 shows the advantage over brute forcing when increasing the number of decryptions with ineffective faults. The maximum advantage is defined as the number of unique definable parameters when attacking the single bit $b_{0,7}^{22}$, i.e., the three key-bits and the three sum-values therefore the maximal advantage of the attack on round 21 can be 6 bit. The unstable advantage at the beginning is caused by multiple key hypotheses with similar SEI values, which leads to frequent change of the key hypothesis having the current maximum SEI. Although the correct key hypothesis is retrieved after 180 ineffective faults, some bits of a wrong hypothesis still are equal to the corresponding bits in the correct guess leading to an advantage of less than 6 bits. An attack on $b_{0,7}^{22}$ is able to recover three key-bits uniquely which equals $\frac{3}{256} \approx 1\%$ of the whole key.

9.3.3. Attack on $b_{0,7}^{21}$

The attack on the intermediate state $b_{0,7}^{21}$ is able to retrieve the involved key bits correctly after approximately 340 ineffective faults. Again, the number of required encryptions under the influence of an ineffective fault is shown in Fig. 9.12 where we used the SEI as statistical metric. In Fig. 9.12 the best wrong hpyothesis is colored red and the correct hypothesis in blue. Fig. 9.11 shows the advantage over brute forcing when increasing the number of decryptions with ineffective faults. The possible advantage when attacking $b_{0,7}^{21}$ is 22-bits at max. Although the hypothesis with highest SEI changes frequently when using less than 340 ineffective faults, the correct key-guess has the maximal SEI after obtaining it. An attack on $b_{0,7}^{21}$ is able to recover 15 key-bits uniquely which equals $\frac{15}{256} \approx 6\%$ of the whole key.

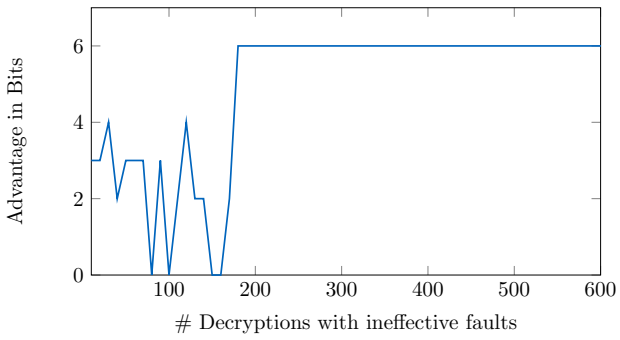


Figure 9.9.: Advantage - Attack on round 22

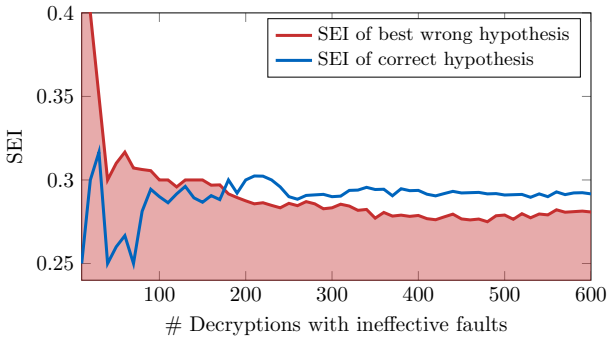


Figure 9.10.: SEI - Hypotheses, round 22

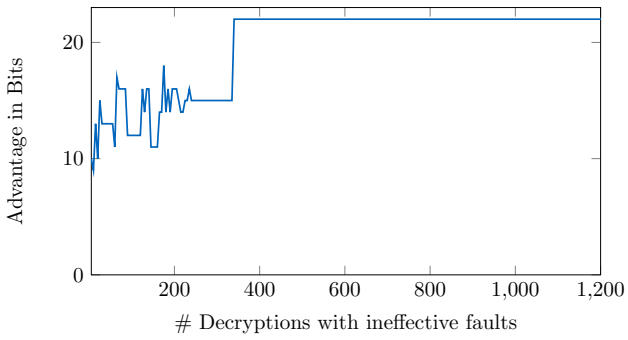


Figure 9.11.: Advantage - Attack on round 21

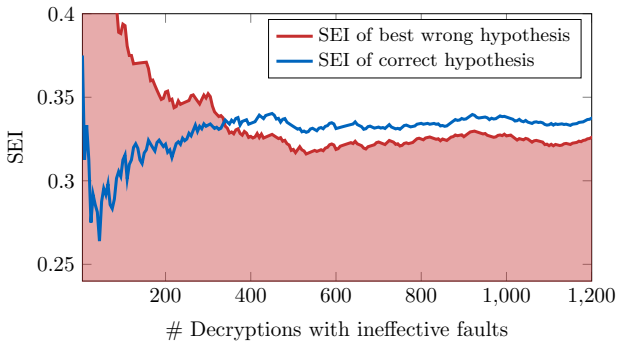


Figure 9.12.: SEI - Hypotheses, round 21

9.4. Summary

In this chapter we presented the SIFA of the AEAD scheme Gimli (Gimli-24-Cipher). Furthermore, we investigated the influence of the fault width w on the rate of ineffective faults r_{ineff}. The fact that SIFA can be applied to the AEAD scheme Gimli should be considered a threat. This is mainly due to the fact that the fault model assumed by SIFA is rather simple to achieve, i.e., a biased intermediate value which is processed during a cryptographic operation. Due to the ineffective characteristic common countermeasures against fault attacks can be circumvented by SIFA.

10. Algebraic Fault Analysis of SAE

Algebraic Fault Analysis (AFA) as introduced by Courtois et al. [CJW10] can be seen as a combination of algebraic cryptanalysis and DFA. A necessary prerequisite for AFA is to describe the cipher as an equation system, where additional equations describe the effects of the fault injection which then allow to solve the equation system. This transformation is most effective when applied to sparse systems of equations of low algebraic degree. As a consequence, ciphers operating on small state sizes and round functions of low algebraic degree are attractive targets for AFA. A class of ciphers which fulfills the above requirements is referred to as lightweight cryptography, which typically features small state sizes and operations with low algebraic degree allowing for area and energy efficient implementations. As the IoT advances, the demand for such primitives increases as more and more small and low-cost devices need to be able to communicate securely. To promote this requirement, the NIST initiated the Lightweight Cryptography (LWC) competition for lightweight authenticated encryption and hash applications back in 2018. This competition aimed to evaluate lightweight ciphers based on their performance, area, energy, and power requirements in order to define a new standard for LWC. Consequently, in 2023 the NIST announced the decision to standardize Ascon, as proposed by Dobraunig et al. [Dob+16a]. Ascon was chosen for standardization because it meets the needs of most use cases where lightweight cryptography is required. Another promising candidate which made it into the second round of the LWC competition is the Subterranean 2.0 cipher suite, as proposed by Daemen et al. [DMR19].

State of the art Courtois et al. [CJW10] proposed AFA and demonstrated the capabilities, at the example of DES. Zhang et al. successfully applied AFA to DES using a single fault [Zha+13a]. Due to their small states, lightweight block ciphers are frequently attacked using AFA. For instance LED was attacked in [JKP12; Zha+12; Zha+13b; Zha+13a]. Other block ciphers which were also attacked by AFA are GOST [Zha+14], Katan [Que14], and Piccolo [Zha+13a]. However, AFA is not limited to block ciphers. Stream ciphers are also susceptible to AFA, e.g., Trivium was attacked by Mohamed et al. [MBB11]. Furthermore, hash functions can also be attacked, as Luo et al. demonstrated with their AFA of SHA-3 [Luo+17].

Contribution In this chapter we apply AFA to the Subterranean Authenticated Encryption (SAE) scheme in order to extract the secret key, and evaluate how different fault models affect the performance of our attack, we verify our claims by means of simulation. Furthermore, we extend the framework proposed by Zhang et al. [Zha+16a] in a hardware-centric manner.

Organisation The rest of this chapter is structured as follows: Section 10.1 introduces the SAE scheme. Section 10.2 explains the concrete realization of the attack. Section 10.3 explains how the required fault equations are generated. Section 10.4 explains how the faulty outputs are obtained. Section 10.5 provides the results of the proposed AFA on Subterranean 2.0. Section 10.6 provides a summary of the AFA applied to SAE.

10.1. Subterranean 2.0

The Subterranean 2.0 cipher suite was introduced by Daemen et al. [DMR19] which is suitable for authenticated encryption, hashing, stream encryption, and MAC computation. The Subterranean permutation is optimized for power- and resource efficient hardware implementations without compromising efficiency for software implementations. Two primitives of the Subterranean 2.0 cipher suite were candidates in the second round of the NIST LWC competition. The Subterranean-XOF (XOF) cipher was a candidate for the hashing category, while the Subterranean Authenticated Encryption (SAE) scheme was a candidate in the AEAD category. The SAE scheme is presented here and will then serve as the target for our attack. For simplicity reasons, we will omit specification details that are not required to understand the SAE scheme.

10.1.1. Subterranean Permutation

The Subterranean 2.0 cipher suite is based on a 257-bit state and a single-round permutation. Every bit s_i of the state is denoted by an index $i \in [0, 256]$. In the following we will use a shorthand notation for the state indices $s_i \equiv s_{i \ mod \ 257}$, i.e., from a mathematical point of view the last bit of the state is consecutive to the first bit of the state. The permutation consists only of additions and multiplications in GF(2). Therefore, the operations are directly realized by `XOR`, and `AND` gates. The round function R is composed of four different layers, viz. a non-linear layer χ, an inversion layer ι, a mixing layer θ, and a transposition layer π, as shown in Eq. (10.1).

$$R = \pi \circ \theta \circ \iota \circ \chi \tag{10.1}$$

The different layers are shown in Eq. (10.2) are described in detail next.

$$\begin{aligned} \chi &: s_i \leftarrow s_i + (s_{i+1} + 1)s_{i+2} \\ \iota &: s_i \leftarrow s_i + \delta_i \\ \theta &: s_i \leftarrow s_i + s_{i+3} + s_{i+8} \\ \pi &: s_i \leftarrow s_{12i} \end{aligned} \tag{10.2}$$

Non-linear Layer The non-linear χ layer is designed to be a sparse, shift-invariant mapping of algebraic degree 2. Sparse means, it generates each output bit by using only 3 (in this case) consecutive state bits. The low algebraic degree is beneficial when implementing countermeasures like Boolean masking, as they tend to become more complex with increasing algebraic degree [GMK16].

Inversion Layer The inversion layer ι introduces asymmetry into the permutation, by inverting only the 0-th bit of the state. δ_i refers to the Kronecker delta such that $\delta_i = 1$ if $i = 0$ and $\delta_i = 0$ otherwise.

Mixing Layer The mixing layer θ is calculated as the sum ,i.e., `XOR` of three state bits having an offset of 0, 3 and 8.

Transposition Layer The transposition layer π places bits that are 12 positions away from each other next to each other using the mapping $s_i \leftarrow s_{12i}$. In addition to that, neighboring bits are moved 150 positions apart from each other. As $150 \cdot 12 \ mod \ 257 = 1$ it follows $s_{150j} \leftarrow s_j$.

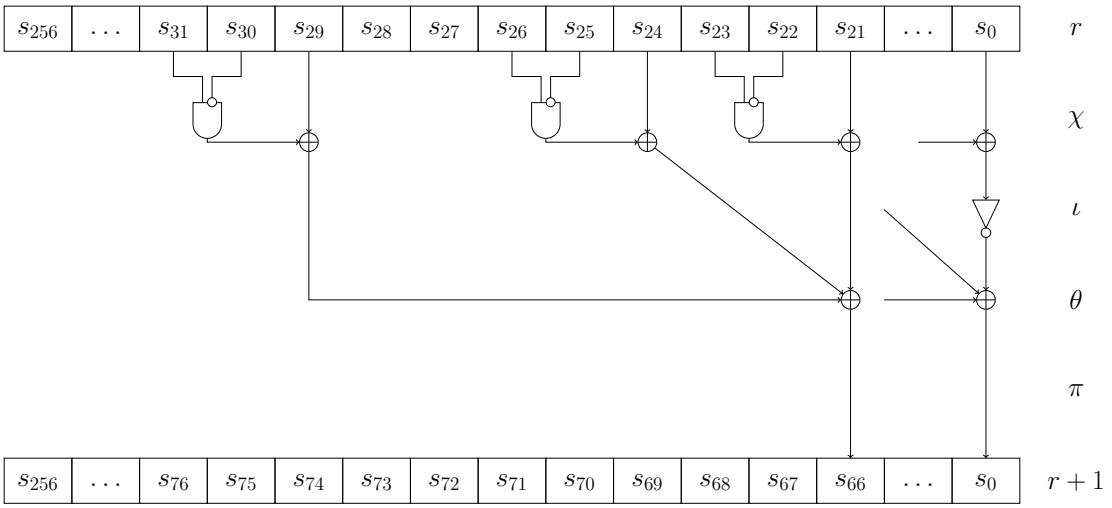


Figure 10.1.: Subterranean 2.0 round function.

Round Function The structure of the resulting round function R is shown in Fig. 10.1. It can be seen that, state bit s_{66} at round $r+1$ depends on 9 bits of the previous state at round r. In the χ-layer, there are three `AND` gates with one inverted input whose output is added to one unmodified state bit. The results of all three additions are then summed up in the θ step. In the ι-layer, the 0-th state bit features an additional inverter before summing it up. The π layer is then realized by permutating the summed output of the θ layer. The permutation of SAE consists of a single invocation of R. Therefore, the round function is also the permutation. For the sake of simplicity, not all gates of each layer between the rounds are shown in Fig. 10.1.

10.1.2. Subterranean Authenticated Encryption

Subterranean operates in the so-called *duplex*-mode [Ber+12a], i.e., the round function is applied and afterwards, a 33 −bit input σ is injected into the state. This σ string consist of input data up to a length of 32 bits which is 10^*-padded to 33 bits. As a consequence the injection rate is 33 −bit. Between each duplex call, 64 bits are extracted from the state. These 64 bits are split into 32 pairs. Summing up the bits of each pair then forms a 32 −bit output z, which is used as key stream. As introduced in Section 2.2 AEAD schemes process a secret key k, a nonce n, associated data ad and some plaintext pt as input, which is then processed into a ciphertext ct and a corresponding tag t. For the SAE scheme, the key k, nonce n and tag t all have the same length of 128 −bit. The associated data ad and plaintext pt or ciphertext ct have variable length which implies the necessity of padding. It is also possible to process empty messages, i.e., messages where either and or the length of ad and pt/ct equals zero. The pseudocode representation of the SAE scheme is given in Algorithm 6, where in the case of encryption the input x is the plaintext and the output y is the ciphertext, and vice versa in the case of decryption.
The SAE scheme can be divided into 3 different phases, *Initialization*, *Processing*, and *Finalization*, which are described in detail below:

Initialization The state absorbs the key k and nonce n, and an initial permutation is applied. Calling *absorb(a,op)* splits the input a into 32 −bit words and performs consecutive duplex calls for each such word, until the input a is completely absorbed, if *op* equals *None* data is only

Algorithm 6 Subterranean SAE

```
function SUBTERRANEAN SAE(op, k, n, ad, x, t'), op ∈ {enc, dec}
    s ← 0^257                                              ▷ Initialization
    s.absorb(k, None)
    s.absorb(n, None)
    s.blank(8)
    s.absorb(ad, None)                                     ▷ Processing
    y ← s.absorb(x, op)
    s.blank(8)                                             ▷ Finalization
    t ← s.squeeze(4)
    if op == dec && t ≠ t' then
        (y, t) = (ϵ, ϵ)
    end if
    return (y, t)
end function
```

absorbed into the state and not extracted. After absorbing the last word of a, an empty duplex call is performed in case the last input word is not strictly shorter 32 bits. The key and nonce are 128 −bit both and thus, require 4 duplex calls plus one empty call each. This results in $2 \times (4 + 1)$ duplex calls. Finally, the *blank(8)* call performs another 8 empty duplex calls for initial permutation.

Processing The associated data *ad* is absorbed first. Consecutive duplex calls absorb the data in 32 −bit words before an optional duplex call required for padding is performed. Afterwards, when encrypting or decrypting the input x, the output y is obtained as $y = x \oplus z$, where x is either plaintext *pt* or ciphertext *ct*, and z is the key stream. In addition to that, the input is absorbed into the state, whereas *absorb(x, op)* makes sure, that the bits being written into the state correspond to their *ct* representation. Again, the absorption finishes with an optional duplex call for padding.

Finalization In order to ensure a strong dependence of the state from the processed input bits, *blank(8)* performs another 8 empty duplex calls. Finally, the *squeeze(a)* function generates the according tag by performing empty duplex calls and taking bits of the keystream z until enough are obtained to form the tag t. As the tag is 128 −bit, squeezing it requires 4 duplex calls. Depending on the operation, i.e, encryption or decryption, the cipher then either outputs the message with corresponding tag t, or respectively verifies the tag t' and only outputs the message if the tags t and t' match.

10.2. AFA of Subterranean SAE

To mount AFA it is necessary to represent Subterranean as an equation system. Consequently, we will show, how to derive the equations for both the cipher and the fault injections. In this section we will derive an ANF representation for the 4 phases of SAE, i.e., the *initialization*, the *data absorption*, *output generation* and *tag generation* phase. State variables are denoted as s_i^x, where $i \in [0, 256]$ refers to the state index and x refers to the point in time, which can be either a clock cycle p or a variable t denoting a temporal location.

Initialization During the first phase of the SAE scheme the state s is initialized to zero. This results in 257 equations introducing the initial 257 state variables.

$$s_i^0 = 0 \qquad , \forall i \in [0, 256] \tag{10.3}$$

Data Absorption The duplex element is the core of the Subterranean 2.0 cipher suite and performs the round function and σ-injection. As shown in Eqs. (10.1) and (10.2), the round function only consists of logical `AND`- and `XOR`-operations. The construction of an equivalent ANF representation is therefore straightforward. Only the χ-layer equation must be expanded to get rid of the parentheses.

For SAT-solving, it is beneficial to keep the equations rather sparse. Splitting up equations with many terms into multiple equations with fewer terms accelerates the solving process. Therefore, the round function is divided into two equations. The χ- and ι-steps make up the first equation, the θ- and π-steps the second one. This is shown in Eqs. (10.4) and (10.5), where the state variables during round p are processed forming two new sets of intermediate state variables:

$$s_i^{t_0} = s_i^p + s_{i+1}^p s_{i+2}^p + s_{i+2}^p + \delta_i \qquad , \forall i \in [0, 256] \tag{10.4}$$

$$s_i^{t_1} = s_{12i}^{t_0} + s_{12i+3}^{t_0} + s_{12i+8}^{t_0} \qquad , \forall i \in [0, 256] \tag{10.5}$$

This results in 2 additional equations and variables per state bit, i.e., 514 equations and variables in total.

The σ-injection is then modeled by the addition of the input bits σ_i to the previously generated variables of $s_i^{t_1}$. One has to keep in mind that, not all bits of the 33 −bit input string σ necessarily correspond to the input message. The last input word per message segment is strictly shorter than 32 −bit and is 10^*-padded to 33 −bit. Adding the 0-padding to the state does not change the value and therefore, does not result in additional equations. As a result, only the bits up to (and including) the 1-padded bit require additional equations. For simplicity, let σ^{valid} be the set of all the bits σ_i that require these additional equations. The σ-injection is then formulated as an equation system as shown in Eq. (10.6).

$$s_i^{p+1} = \begin{cases} s_i^{t_1} + \sigma_i & \forall i \in \{\text{indices of faulted bits}\} \wedge \sigma_i \in \sigma^{valid} \\ s_i^{t_1} & \text{, otherwise} \end{cases} \tag{10.6}$$

With that, every such equation representing a valid input bit then introduces 2 new variables: one for the new state bit s_i^{p+1} and one variable representing the value of the input bit $\sigma_i \in \sigma^{valid}$. Note, that in the case of plaintext injection, a variable representing σ_i is introduced in the *output generation* phase. Similarly, for the 1-padded bit the value is directly assigned such that no extra variable for σ_i is introduced. Therefore, for every $\sigma_i \in \sigma^{valid}$, one equation with either 1 or 2 new variables is introduced. Note that the case $s_i^{p+1} = s_i^{t_1}$ does not result in an additional equation, as the last updated state variable is s^{t_1} anyway.

Output Generation The output generation refers to the generation of the ciphertext ct or plaintext pt in case of an encryption respectively decryption. During the *Data Absorption*, the input x is split into distinct 32 −bit words, such that one input word of up to 32 bits can be processed at a time cf. Algorithm 6. Every output bit y_i is then the sum of the input bit x_i and the key stream bit, which itself is the sum of two state bits as shown in Eq. (10.7).

$$y_i = x_i + s_{12^{4i}}^p + s_{-12^{4i}}^p \qquad , \forall x_i \in x[n] \tag{10.7}$$

Every processed bit then results in one such equation. For every equation, two new variables are introduced: one representing y_i and one representing x_i. This allows to abstract the ANF generation from the actual input message values.

Tag Generation In order to generate the 128 −bit tag, four consecutive 32 −bit words from the key stream must be extracted. Every 32 −bit keystream word is obtained as shown in Eq. (10.8)

$$t_i = s^p_{12^{4i}} + s^p_{-12^{4i}} \qquad , \forall i \in [0, 31] \tag{10.8}$$

Therefore, every bit of the tag adds one additional equation with one new variable for each bit of the tag t_i. This results in 128 new equations and variables in total. After the extraction of every word, a duplex call is required which adds more equations and variables. Although the SAE scheme defines a final duplex call after extracting the last part of the tag, there is no reason to generate equations for that as they do not provide additional knowledge.

Summary of the Equation based Representation The number of equations and variables representing the cipher are summarized in Table 10.1. In addition to the previously described phases, the number of equations required for assigning the variables of the known input and output is added. Every bit of the input and output actually results in one equation setting the corresponding variable to either 0 or 1. This overhead will be eliminated during the preprocessing step when solving the equation system.

Phase	Equation	# Equations	# Variables
Initialization	Eq. (10.3)	257	257
Output Generation	Eq. (10.7)[1]	$\|x\|$	$2 \times \|x\|$
Data Absorbtion (per cycle)	Eqs. (10.4) and (10.5)	2×257	2×257
	Eq. (10.6) [2]	$\|\sigma^{valid}\|$	$2 \times \|\sigma^{valid}\| - 1$
Tag Squeezing	Eq. (10.8)	128	128
Assignments	-	$\|n\| + \|pt\| + \|ct\| + \|tag\|$	-

[1] $|x|$ size of cipher input, i.e., of plaintext or ciphertext
[2] $|\sigma^{valid}|$ number of valid bits σ_i absorbed in corresponding duplex calls

Table 10.1.: Equation system overhead for cipher description

10.3. Generation of Fault Equations

Modeling the fault injection adds additional equations and variables. To specify the injected fault, we will introduce the following parameters:

The fault model $fault\_model$ which can either be a *Guaranteed Bit Flip* or a *Random Bit Flip*. The temporal fault position p, which refers to the clock cycle in which the fault is injected. The spatial fault location l which refers to the bit position in the state register where the fault is injected, such that $0 \leq l < 257$. The fault width w is the number of adjacent bits affected by the fault injection. In addition to that, it is further distinguished whether l is known or unknown.

The fault injection can then be seen as an addition ($GF(2)$) of a w−bit wide fault mask to the state at the bit location l during clock cycle p. According to the $fault\_model$, the w−bit vector is either filled with all 1 in the case of a *Guaranteed Bit Flip*, or randomly chosen bit values in the case of a *Random Bit Flip*.

Zhang et al. introduced a generalized framework to apply AFA on block ciphers [Zha+16a]. They derived fault equations independent of the underlying cipher and it's operations. In this

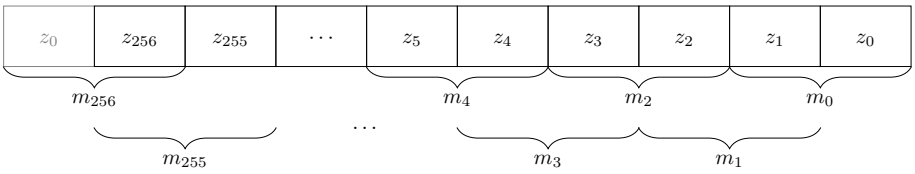


Figure 10.2.: Fault differential z_i split into multiple blocks m_i.

chapter, their way of expressing the fault injection is extended to comply not only with the *Random Bit Flip*, but also with the *Guaranteed Bit Flip* model. In addition to that, this framework is modified to model a hardware-centric approach such that it allows for overlapping fault locations over consecutive fault injections.

10.3.1. Intermediate State Differential

AFA is designed to exploit one correct and multiple faulty outputs in order to deduce the intermediate state at the fault injection's location (temporal, spatial). To do so, the difference induced by the fault must be modeled. If we let s_i denote the correct state bits and s'_i the corrupted state bits after the fault injection. The difference between the correct and the faulty bit is then denoted as z_i. With that in mind, the fault differential can be described as:

$$z_i = s_i + {s'}_i \qquad , \forall i \in [0, 256] \tag{10.9}$$

This adds one equation with 2 new variables each in order to represent z_i and s'_i, i.e., 257 equations and 514 variables in total. The variables for s_i are already introduced during the generation of the equation system for the correct computation.

In the next step, the state differential $z = z_{256}|\ldots|z_0$ is divided into 257 blocks m_i of size w, which overlap for $w > 1$. Every block m_i can be seen as a possible location for the fault injection. For software implementations, the processed blocks are usually aligned, as a CPU only operates on a specific (native) word size. When injecting a fault in a certain clock cycle, only the processed word might be faulted. However, the Subterranean 2.0 primitives are especially tailored towards hardware implementations. Thus, it is assumed that all the state operations occur in the same clock cycle. As a consequence, the fault injection can occur at every possible state slice which means that the blocks overlap. Figure 10.2 illustrates this division of z into the blocks m_i for the case where $w = 2$. For blocks that contain bits z_i with $i > 256$, the index is flipped such that it continues with z_0, z_1 etc. In the following, blocks which are not affected by the fault injections are indexed as m_j, whereas a block affected by the fault is referred to as m_l.

10.3.2. Known Fault Location

To make use of the fault differential z, some information on its characteristic is required. If the fault location l is known, we can deduce for all correct blocks m_j, the corresponding bits $z_i \in m_j$ are zero. Assuming a fault model based on a *Guaranteed Bit Flip*, the faulty bits $z_i \in m_l$ are all 1. This is shown in Eqs. (10.10) and (10.11), respectively.

In the case of a *Random Bit Flip*, however, only the fact that a fault occurred in a certain block m_l is known, but not which bits actually flipped. Therefore, a variable u_l is introduced, that indicates that block m_l is faulty. As shown in Eq. (10.12), u_l is generated by multiplying all inverted bits $z_i \in m_l$ that correspond to the faulty block m_l. An inverted z_i is zero, if the

state bit s_i flipped, i.e., $s_i \neq s'_i$. Multiplying all inverted z_i and setting the result u_l to zero then implies that at least one bit is faulty.

$$z_i = 0 \qquad , \forall z_i \in m_j \tag{10.10}$$

$$z_i = 1 \qquad , \forall z_i \in m_l, \text{ Guaranteed Bit Flip} \tag{10.11}$$

$$u_l = \prod_{z_i \in m_l} (1 + z_i) = 0 \qquad , \text{Random Bit Flip} \tag{10.12}$$

10.3.3. Unknown Fault Location

For unknown fault locations, none of the z_i can be assigned a specific value directly, as it is unclear which bits of the state differential z are affected by the fault. Again, the equations for the two fault models are slightly different.

Guaranteed Bit Flip As the fault cannot be located, for every block m_i a variable u_i is introduced indicating whether block m_i is faulty or not. This is achieved by the multiplication of all variables z_k corresponding to the same block m_i, as shown in Eq. (10.13). The faulty block m_l is the only block exclusively containing variables with $z_k = 1$. As a consequence, it follows $u_l = 1$, while for all the fault-free blocks m_j it holds $u_j = 0$. The later is because every fault-free block m_j contains at least one variable $z_k = 0$. In Eq. (10.14), all the variables u_i are inverted and multiplied. Inverting means, that for the faulty block m_l it holds $(1+u_l) = 0$. For all the correct blocks m_j it holds $(1 + u_j) = 1$. Multiplying all these variables and setting the result to zero then states that there must be *at least* one faulty block. Finally, generating all possible pairs of $u_i u_k$ and setting the result to zero also states that there is *at most* one faulty block, as always one unfaulty block m_j with $u_j = 0$ is part of the multiplication. This is calculated as shown in Eq. (10.15).

$$u_i = \prod_{z_k \in m_i} z_k \qquad , \forall i \in [0, 256] \tag{10.13}$$

$$0 = \prod_{i=0}^{256} (1 + u_i) \tag{10.14}$$

$$0 = u_i u_k \qquad , \forall i, k \ , \ 0 \leq i < k \leq 256 \tag{10.15}$$

Random Bit Flip Similar to the *Guaranteed Bit Flip* model, variables u_i are introduced to indicate whether block m_i is faulty or not. However, for the random fault, one cannot assume that all bits in the fault mask are set. Simply multiplying all z_k of a block m_i would therefore quite likely result in all u_i including u_l of the faulted block m_l being zero. The only exception is a fault injection where every bit in m_l is faulted, which occurs only with probability 2^{-w}. To cope with that behavior, the bits z_k are first inverted and then multiplied to generate u_i, as shown in Eq. (10.16). As a result, it is $u_i = 0$ as soon as one bit in m_i is flipped, whereas for all unaffected blocks it is $u_i = 1$. With that, the fact that there must be *at least* one faulty block can be expressed by multiplying all block variables u_i and setting the result to zero. This is expressed in Eq. (10.17).

The fact that $u_i = 0$ as soon as one bit of a block m_i is flipped also implies, that for overlapping blocks, up to $2w - 1$ different blocks are faulted. As a consequence, it is not possible to identify at most *one* faulty block. To express this behavior, it is necessary to multiply $2w$ inverted block variables and to set the result to zero. This ensures that for every combination of $2w$ blocks, at

least one block m_j is fault-free such that $(1 + u_j) = 0$. Equation (10.18) then expresses the fact that there are *at most* $2w - 1$ faulty blocks.

$$u_i = \prod_{z_k \in m_i} (1 + z_k) \qquad , \forall i \in [0, 256] \tag{10.16}$$

$$0 = \prod_{i=0}^{256} u_i \tag{10.17}$$

$$0 = (1 + u_i) \dots (1 + u_k) \qquad , \forall i, k \mid 0 \leq i < k \leq 256 \tag{10.18}$$

As a consequence of Eq. (10.18), a fault injection with different fault models and information on the location can be fully described. Comparing Eqs. (10.13) to (10.15) with Eqs. (10.16) to (10.18) shows, that the structure of the equations is quite similar. A significant difference is the number of multiplied block variables in order to limit the number of faulty blocks.

10.3.4. Summary of the Number of Fault Equations

The number of equations and variables introduced by the corresponding fault model is summarized in Table 10.2. Let N denote the number of tuples $u_i u_k$ as described in Eq. (10.15) (respectively $(1 + u_i)...(1 + u_k)$ as described in Eq. (10.18)). N is calculated as shown in Eq. (10.19). In our specific case of Subterranean, $n = 257$ is the number of blocks and k the number of u_i variables that are multiplied. For the *Guaranteed Bit Flip* model, this would lead to $k = 2$ independent of the fault width w, because there is only one single faulty block. For the *Random Bit Flip* model however, $2w - 1$ blocks can be faulty and therefore, $k = 2w$.

$$N = \binom{n}{k} = \frac{n!}{k!\ (n-k)!} \qquad , 0 \leq k < n \tag{10.19}$$

The cipher equations modeling Subterranean 2.0 are of a maximum degree of 2, but generating fault equations can result in equations of degree up to 257 (e.g. Eqs. (10.14) and (10.17)). As introduced in Section 6.4, obtaining the solution of an equation system benefits most from sparse and low-degree equations. The function $\texttt{red}(n, d)$ returns the number of equations required to substitute an expression of degree n with multiple expressions of degree d. For $d = 2$, every equation with a degree greater than 2 is reduced, i.e., the equation is split into multiple equations of max degree 2. Pairs of variables representing the result of the degree 2 equations are then multiplied again. The iterative application of this reduction step ensures that the correctness of the equations is retained while ensuring a low-degree equations. In this chapter, the function $\texttt{red}(n, 2)$ is applied, to ensure a maximum degree of 2.

10.4. Obtaining Faulty Outputs

The application of AFA makes it necessary to generate an equation system which represents the cipher- and fault equations. In addition to that, an attacker is required to obtain a single correct and several faulty outputs for the same input. To obtain faulty outputs, the SAE Python reference implementation [DMR19] was incorporated into a simulation framework which allows to simulate the fault injection with different values for p, l, and w. We will now discuss reasonable choices of these parameters.

Bit Flip Model	Equation	# Equations	# Variables
Both	Eq. (10.9)	257	514
Known Fault Location l			
Guaranteed	Eqs. (10.10) and (10.11)	257	-
Random	Eq. (10.10)	$257 - w$	-
	Eq. (10.12)	$w + \text{red}(w, 2) + 1$	$w + \text{red}(w, 2)$
Unknown Fault Location l			
Guaranteed	Eq. (10.13)	$257 \times \text{red}(w, 2)$	$257 \times \text{red}(w, 2)$
	Eq. (10.14)	$257 + 265$	$257 + 264$
	Eq. (10.15)	$2N$	N
Random	Eq. (10.16)	$257 \times (1 + \text{red}(w, 2))$	$257 \times (1 + \text{red}(w, 2))$
	Eq. (10.17)	265	264
	Eq. (10.18)	$257 + N(\text{red}(2w, 2) + 1)$	$257 + N \times \text{red}(2w, 2)$

Table 10.2.: Equation system overhead for fault description

10.4.1. The temporal fault position p

The overall number of clock cycles required for the computation of the SAE scheme is data dependent. It depends on the size of the associated data and the message to either encrypt or decrypt. In Table 10.3, the data dependent number of clock cycles is shown for an encryption with respect to the number of processed words. During the first cycle 0, the state is initialized to all-zero. The key k is then absorbed in 32 −bit words k_i in four consecutive cycles. In the 5-th cycle, an empty duplex call (padding) is performed, as every message segment must include one final non-full input word. The same then applies for the nonce n, followed by 8 duplex calls with empty input which ensure enough diffusion.

Afterwards, starting at cycle $p = 19$, the associated data is absorbed. It takes at least one clock cycle for the last, non-full (eventually empty) input word. For every full 32 −bit input word, an additional clock cycle is required. This is denoted by x, which is the sum of the 19-th clock cycle and the number of full AD-words. In the same way, the number of cycles required by the plaintext absorption is calculated and denoted by y. It starts at cycle $x + 1$, requires at least one cycle for the final non-full word and adds an additional clock cycle for every full 32 −bit plaintext word.

After another 8 empty duplex calls, the tag is squeezed in 4 consecutive cycles. Note, that for a decryption, the same amount of cycles is required, however the input/output sections of the plaintext/ciphertext are exchanged. Therefore, 33 cycles are required to compute the tag for a single, empty input. The earlier the fault is injected, the stronger is the fault propagation and more bits of the generated output tag are affected. However, this also results in a larger equation system, because more clock cycles are modeled. Therefore, we opted for a fault injection into the last cycle before the tag is generated ($y + 8$ in Table 10.3), which in this case would be $p = 27$.

10.4.2. The spatial fault location l

The fault location l refers to the state indices at which the faults are injected with $l \in [0, 256]$. A multi-bit fault affects w consecutive bits, i.e., state bits s_l to s_{l+w-1}. To choose a suitable value

Cycle p	Function	Input	Output
0	init()	-	-
$1 \dots 5$	absorb(K)	$k_0, \dots, k_3$	-
$6 \dots 10$	absorb(N)	$n_0, \dots, n_3$	-
$11 \dots 18$	blank(8)	-	-
$19 \dots x$	absorb(AD)	$ad_0, \dots, ad_n$	-
$x+1 \dots y$	absorb(PT)	$pt_0, \dots, pt_n$	$ct_0, \dots, ct_n$
$y+1 \dots y+8$	blank(8)	-	-
$y+9 \dots y+12$	squeeze(4)	-	$t_0, \dots, t_3$
$x = 19 + \#full\_AD\_words$, $y = x + 1 + \#full\_PT\_words$			

Table 10.3.: SAE cycle count for arbitrary AD and PT segment lengths.

for l, two strategies are considered: l can be chosen either randomly or in an ordered fashion. As the name suggests, for the random fault location a value $l \in [0, 256]$ is randomly chosen. In case of the ordered location, the first computation is faulted at location 0, the second at location 1, the third at location 2 and so on. Both strategies differ in their effect on the tag if they are injected in one of the last few cycles as the effect of a fault also depends on the value of the current state. The difference between the two different fault injection strategies originates from the structure of the Subterranean 2.0 round function itself. As shown in Section 10.1.1, every state bit is computed from 9 bits of the previous state. Because these bits are partially neighbored, two fault injections at adjacent locations l will have an intersection in the set of fault-affected bits. Two faults with a large difference in their location l will affect disjunct state bits and therefore, propagate faster. This might also cause more linear dependencies and result in more efficient equation systems, in terms of resources required for finding the solution. Therefore, it is expected that the random strategy for choosing l benefits the solving process compared to the ordered strategy.

10.4.3. The fault width w

The choice of the fault width is closely related to the attacker's capabilities. Not every fault injection method is capable of injecting faults of arbitrary precision. Therefore, we distinguish between *local* and *global* injection. A local fault injection refers to a fault injection, that target single bits at defined locations in the state. In [SA03] for example, it was shown how single SRAM cells can be flipped using laser fault injection. In contrast to local fault injection, global fault injection may affect many bits with a single fault injection. In the context of the Subterranean 2.0 ciphers, a clock glitch during the computation can be seen as the injection of a $257-$bit random fault. This of course yields less information on the exact fault effect, however is easily achievable in terms of assumed fault model.

10.5. Results

In the following we will evaluate the performance of our proposed attack on the Subterranean SAE scheme with respect, to the underlying fault model. The objective of our attack is twofold. At first we aim to recover Subterranean's state and after the recovery of the state compute the secret key based on the internal state. This is achieved by analyzing the difference between the

cipher's correct and faulty outputs. For the sake of comparability/simplicity, the cipher processes an empty message during every experiment, if not explicitly stated otherwise. The choice for an empty message has two reasons: First, an empty message results in the smallest possible equation system when considering the whole SAE scheme. Processing such a message requires 33 clock cycles. Second, the 128 −bit tag output is the output an attacker is always able to obtain from the SAE scheme. As a consequence, at least 128 bits of the keystream are available independent of the processed data.

For every result outlined in the next subsections, 20 instances of CryptoMiniSat were run on a workstation with 2 Intel Xeon E5-2670 v3 processors running at 2.30 GHz base frequency. If no solution of the equation system is found within 12 h, the experiment was aborted and the according equation system was labeled as unsolvable.

10.5.1. Fault Model

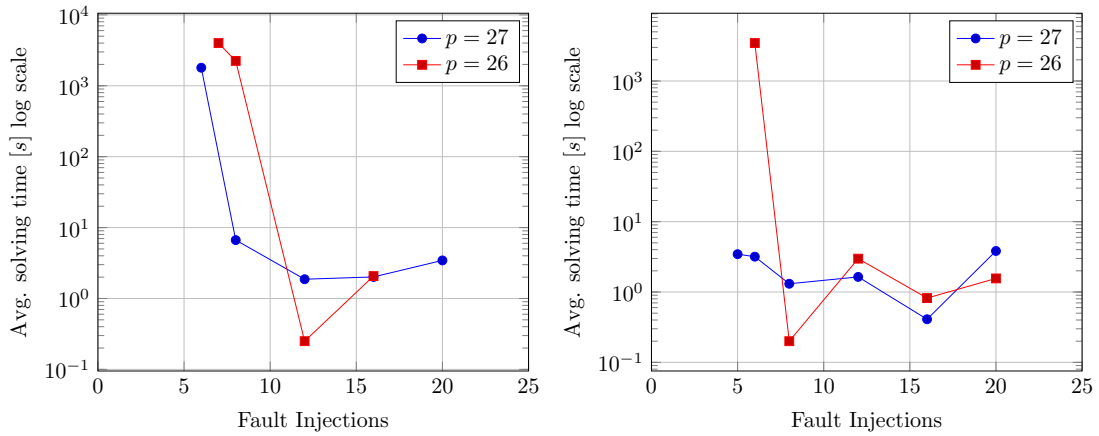
The assumed fault model is crucial for a successful attack and can be adjusted by several parameters like fault location l, fault position p and the fault width w. The influence of each is evaluated in the following.

Fault Location l The fault location l is assumed to be known to the attacker which essentially models a rather powerful attacker that is capable of injecting multiple faults at predefined parts of the state.

There are two options for deciding where to inject the faults, i.e., the ordered- and random location. Ordered location means, that the first fault is injected at state bit s_0, the second at s_1 etc. The random strategy, however, randomly chooses values for l.

Several experiments were conducted evaluating both strategies for a fault width $w = 1$ at position $p = 27$. The choice of p ensures that the fault affects the generation of the whole tag. Simulation-based results of this scenario are shown in Fig. 10.3, where Fig. 10.3a depicts the average solving time depending on the number of fault injections for the ordered fault locations, and Fig. 10.3b for the random fault locations, respectively. It can be observed, that for more than 10 fault injections, the solving times only differ slightly. However, with fewer fault injections the random strategy clearly outperforms the ordered strategy. The solving time for the ordered strategy increases rapidly with 6 fault injections. With fewer, the system becomes unsolvable within the predefined 12 h. In contrast to that, the random strategy, does only slightly increase for these numbers of samples and is even solvable within seconds for 5 fault injections. As indicated by Fig. 10.3a, the solving time increases exponentially when decreasing the amount of output samples. Therefore, even for the random location strategy, the system is not solvable within 12 h for less than 5 samples.

When injecting faults at earlier rounds, the fault propagates further and thus, the number of wrong bits in the resulting tag increases for both strategies. Nevertheless, the intermediate state differences still behave differently, as ordered fault injections have an intersecting set of bits affected by the fault. That this effect still dominates is also shown in Fig. 10.3 for $p = 26$. It clearly shows that for both strategies, the solving time increases with fewer samples. In addition to that, a minimum of 6 fault injections are required for the random strategy and 7 for the ordered strategy. This shows, that independent of the position p, the random strategy yields better results. When comparing the subplots of Fig. 10.3, it is also observable that the position p of the fault injection has an influence on the system's solving time. The scale for the average solving time increases when injecting the faults at earlier rounds. The effect of the position p will be subject in the following experiments.



(a) Ordered Strategy

(b) Random strategy

Figure 10.3.: Comparison of ordered and random fault locations for $p = 26$, $p = 27$

Fault Position p The choice of the fault position p correlates with the amount of equations added by each fault injection. The earlier a fault is injected, the more cycles are computed until the tag is squeezed. Injecting the fault too late has the effect that parts of the tag are already computed and thus, do not provide any new information. As a result, it seems most beneficial to inject a fault one cycle prior to generating the first word of the tag, denoted by $p = 27$. To verify this assumption, the influence of the fault position p is evaluated for $p = 26$ to $p = 29$. The early fault injection at $p = 26$ increases the equation system and the effect of the propagated fault. Nevertheless, the exploitable output and thus, amount of keystream bits stays constant. Injecting faults at $p = 28$ means, the first 32-bit word of the tag is the sum of the correctly generated tag word and the injected fault. For the later $p = 29$, the first word of the tag is correct as it was extracted before the fault injection and the second word is again, the fault added to the correctly computed word. With respect to the results in the previous experiments, the location l is randomly chosen. The results are shown in Table 10.4. It clearly verifies the assumption that the optimal cycle for the fault injection is $p = 27$, i.e., one cycle before the tag generation starts. Injecting the fault at an earlier cycle then requires one additional output sample to solve the problem and also increases the solving time. Injecting the fault one cycle later, i.e., at $p = 28$ does not require more faulted outputs but significantly increases the solving time. However, injecting the fault even later at $p = 29$ clearly amplifies the negative effects.

Despite the increasing effort required to find the solution, injecting faults at early positions would have additional benefits: If the fault is injected before the nonce, or before the nonce's last word is absorbed, an attacker could satisfy the unique nonce requirement of the SAE scheme by mimicking nonce effects by a fault injection. In addition to that, it would allow the exploitation of more differences in the keystream bit when injecting the fault before data encryption or decryption.

Fault Width w The definition of the fault width depends on the attackers capabilities and resources. Flipping a single bit requires a precise laser injection, whereas a 257 −bit fault might be achieved by glitching the whole circuit during computation. This, of course, is less precise and can easily be reproduced if an attacker knows at which point in time the computation starts.

In the following experiments, the effect of the fault width w for both the *Guaranteed Bit Flip*

Fault Injections	Fault Positions			
	$p = 26$	$p = 27$	$p = 28$	$p = 29$
4	-	-	-	-
5	-	3.45	3336.91	-
6	3462.58	3.18	245.03	-
8	0.20	1.31	1.48	-
12	2.98	1.64	11.80	-
16	0.82	0.41	1.36	-
20	1.56	3.83	2.31	11391.08

Table 10.4.: Average solving time [s] for different fault positions

Fault Injections	Guaranteed Bit Flip			Random Bit Flip	
	$w = 1$	$w = 4$	$w = 8$	$w = 4$	$w = 8$
4	-	-	-	-	-
5	3.45	1.63	1.70	6391.27	-
6	3.18	4.81	8.98	396.39	1538.26
8	1.31	3.45	3.12	5.44	313.42
12	1.64	4.78	4.00	2.11	3.99
16	0.41	2.69	1.46	1.14	8.46
20	3.83	8.99	2.40	11.28	0.80

Table 10.5.: Average solving time [s] for different fault widths

and the *Random Bit Flip* fault model was examined. The fault location l was randomly chosen and the position p was set to $p = 27$. Table 10.5 lists the results for a 1 −bit fault, a 4 −bit and an 8 −bit fault. Note, that for $w = 1$, both fault models are equal, as there can be only one bit set in the fault mask.

It can be observed, that the 1-bit fault yields to the best results, i.e., 5 fault injections are sufficient to solve the equation system within seconds. For the *Guaranteed Bit Flip*, 5 fault injections still suffice to extract the key for larger fault widths. However, for the *Random Bit Flip* model the time increases significantly when reducing the number of fault injections. For $w = 8$, at least 6 fault injections are required to solve the problem within 12 h. That is, because an increasing width w means that there are more possible realizations of the fault mask. In fact, there are $2^w - 1$ different fault masks which might have occurred. Thus, guessing the correct one requires more effort with increasing w.

Similar to previous results, the tables of the *Random Bit Flip* model are shifted towards an increasing number of fault injections when modifying w. However, for the *Guaranteed Bit Flip* model, all results are in the range of a few seconds and differ only slightly. This clearly shows the benefit of the a priori knowledge on the fault mask of the *Guaranteed Bit Flip*. Due to the fact that the value of the fault is known, the solver does not need to evaluate different fault masks. From an analytical point of view, the equation for the *Random Bit Flip* block variable u_l as shown in Eq. (10.12) is of degree w, i.e., if w increases, the equations degree also does.

Fault Injections	Avg. time [s]
4	-
5	3.45
6	3.18
8	1.31
12	1.64
16	0.41
20	3.83

(a) Location l known

Fault Injections	Avg. time [s]
19	-
20	14,925.87
21	15,683.94
22	16,664.07
23	1,285.66
24	1,953.17
25	6,584.75

(b) Location l unknown

Table 10.6.: Comparison of known and unknown fault location

Unknown Fault Location l The fault model can be further relaxed by assuming that the location l is unknown, under the assumption that only the position p and width w are known to the attacker. Respecting the previous results, they are set to $p = 27$ and $w = 1$, l is randomly chosen. As described in Section 10.3.3, an unknown fault location has a significant overhead compared to the known location. The main reason for that is due to Eq. (10.18). It states, that there are at least $2w - 1$ faulty blocks, which results in $32,896$ additional expressions for $w = 1$.

A direct comparison of the known and unknown fault location is shown in Table 10.6. The results for the known fault location as obtained by previous measurements are stated in Table 10.6a, whereas the results for the unknown fault location are shown in Table 10.6b.

It can be observed, that the unknown fault location yields worse results: Not only the solving time increases, but also number of fault injections that are required to find the solution within 12 h. In addition to that, the solving time varies significantly. Whereas the solving times for the known location differ only in a few seconds, it increases up to thousands of seconds for the unknown location. Due to computational constraints, it is not feasible to evaluate more than 25 fault injections as the number of variables in the resulting equation system would exceed the capabilities of `BOSPHORUS` [Cho+19].

Fault Evaluation Summary The above evaluation shows, that the fault parameters have a significant influence on the complexity of the attack. The optimal fault parameter set turned out to be a *Guaranteed Bit Flip* model in conjunction with a known but randomly chosen fault location l. Furthermore, the fault is applied at the location l, just one cycle before the tag generation, where the fault width w should be as small as possible, i.e., $w = 1$. Injecting the faults one cycle before the output is generated, results in the least amount of unusable overhead.

10.5.2. Non-Empty Message

The previous results focused on the evaluation of the fault model when considering the computation of an empty message. It has been shown, that injecting the fault at earlier clock cycles negatively affects the performance of the attack. Another question that might come up is, how the overall number of clock cycles influences the results. To evaluate this case, an Ethernet frame with 14 bytes of associated data and 50 bytes of plaintext is used as an example. According to Table 10.3, the Ethernet frame requires 15 additional duplex calls compared to an empty message. The faults of width $w = 1$ are injected at random but known locations. The fault positions are $p = 42$ for the Ethernet frame message and $p = 27$ for the empty message. Both positions correspond to the last clock cycle before the tag is generated. A comparison of the Ethernet

	Avg. solving time [s]	
Fault Injections	Empty Message	Ethernet Frame
4	-	-
5	3.45	2,640.9
6	3.18	1,133.43
8	1.31	368.2
12	1.64	485.32
16	0.41	656.25
20	3.83	212.29

Table 10.7.: Comparison of empty message and Ethernet frame encryption

message and the empty message is shown in Table 10.7. The results show, that the solving time increases for the Ethernet frame. As the equation system becomes larger with every additional duplex call, this result has been expected. In addition to that, the amount of output bits an attacker obtains does not necessarily improve the attacks' performance. For the Ethernet frame, 50 bytes have been encrypted, which corresponds to 400 bits of available keystream material. Nevertheless, these bits are equal for both, the correct and faulted encryptions. That is, because the fault is injected at a later position. As a result, there are no dependencies between the keystream bits of the faulted and correct computation introduced by the plaintext encryption. The actual exploitable output remains the 128 −bit authentication tag.

10.5.3. Comparison with Trivium

In the following, this chapter's results are compared with AFA results of Trivium [MBB11]. Trivium's design rational is similar to the one of Subterranean 2.0, i.e., a lightweight, hardware tailored stream cipher. Table 10.8 shows a comparison of Trivium and Subterranean 2.0 and the results of the attacks. Although Subterranean's internal state is smaller, it features a larger key size than Trivium. For the results of both ciphers, single bit faults at known positions and locations are assumed. It shows, that the attack on Trivium required less fault injections but more keystream bits. Depending on this number of bits, the key was recovered between a fraction of a second and slightly more than two minutes. For the attack on Subterranean 2.0, it was assumed that only the keystream bits of the 128 bit tag are available to an attacker. Nevertheless, the key was also recoverable within a few seconds.

Despite the fact that Trivium and Subterranean 2.0 are both lightweight and hardware tailored stream ciphers, a quantitative comparison of the AFA results shown above must be taken with care. That is, because not only their internal state or key sizes differ, but also the calculation of the state update differs. In fact, the Subterranean 2.0 core element updates the whole state in every clock cycle, where each new state bit depends on 9 previous state bits. However, Trivium consists of 3 Nonlinear Feedback Shift Registers (NLFSRs). In each clock cycle, one bit per register is updated by taking 5 previous state bits into account. The remaining bits in the registers are simply shifted by one position. The difference in the internal structure significantly affects the number of equations required to describe the cipher. Besides that, the tools used to conduct the attack differ.

	Trivium [MBB11]	**Subterranean 2.0**
State size [bit]	288	257
Key size [bit]	80	128
ANF/CNF conversion	ANF2CNF [AS10]	BOSPHORUS
SAT-Solver	MiniSat	CryptoMiniSat
	AFA Results	
No. Faults	2	5
No. Key bits	$420 - 800$	128
Solving time [s]	$0.127 - 138.653$	3.45

Table 10.8.: Comparison of Trivium and Subterranean 2.0

10.6. Summary

In this chapter we presented the AFA of the SAE scheme which is part of the Subterranean 2.0 cipher suite. To conduct the proposed attack the SAE scheme was transformed into an equation system. Furthermore, the equation system was then augmented with additional equations which model the effect and propagation of the injected fault. Under the assumption of the optimal fault model, the proposed attack successfully recovers the secret key of Subterranean 2.0 in less than four seconds using only five fault injections. By analyzing the effects of different fault parameters, it was shown that the accuracy of the injected error is crucial for the complexity of the resulting equation system. Precise fault injections allow an optimized system of equations to be solved in seconds, requiring only a few fault injections. In contrast, when relaxing the fault model, significantly more time and fault injections are required to find the solution. As we considered only the exploitation of the tag, evaluating the influence on the number of additional keystream bits might be of interest for further research.

11. FIA Strategy Comparison

We will now compare the Fault Injection Analysis (FIA) strategies used in Part II based on their key characteristics: The underlying *Evaluation Strategy* which can either be analytical or statistical, usually statistical strategies require more samples than analytical ones. The *Fault Model* defines the degree to which the attacked algorithm can deviate from its specified behavior. The *Robustness* against noisy fault injection, i.e., fault injections which do not adhere to the specified *Fault Model*. The inherent capability of the FIA type to overcome *Countermeasures*. A comparison of the key characteristics with respect to the FIA strategy is shown in Table 11.1.

Differential Fault Analysis (DFA) A type of FIA which exploits information gained from the difference between a correct and faulty encryption is DFA. The *Evaluation Strategy* of DFA is purely analytical and therefore prone to failure if the physical fault injection mechanism may cause noisy fault injections. Consequently, the *Robustness* of DFA is rather low. Furthermore, the *Fault Model* is rather strict and usually specific to the algorithm under attack, e.g., the processing of AES is byte-wise. Consequently, most DFAs of AES assume a single byte fault or a fault of multiple bytes as *Fault Model*.

Persistent Fault Analysis (PFA) A type of FIA which exploits information gained from the manipulation of constants which are used during an encryption is PFA. The *Evaluation Strategy* of PFA is either analytical or statistical and therefore more resistant against failure if the physical fault injection mechanism may cause noisy fault injections. Consequently, the *Robustness* of PFA is higher than the one of DFA. Furthermore, the *Fault Model* is rather strict and as usual specific to the algorithm under attack but only a single fault injection with a persistent fault is required. Also, the persistent nature of the *Fault Model* enables an attacker to overcome countermeasures which are based on, e.g., either infection or detection.

Statistical Ineffective Fault Analysis (SIFA) A type of FIA which exploits information gained from ineffective faults, i.e., faults that cause a behavior which does not deviate from a correct encryption is SIFA. The *Evaluation Strategy* of SIFA is purely statistical and therefore robust to failure if the physical fault injection mechanism may cause noisy fault injections. Consequently, the *Robustness* of SIFA is very high. Furthermore, the *Fault Model* is rather easy to achieve as the only assumption is that the distribution of an intermediate value is not uniform. Also, the ineffective nature of the assumed faults enables an attacker to overcome many countermeasures which were proposed prior to the introduction of SIFA.

Algebraic Fault Analysis (AFA) A type of FIA which exploits information gained from correct and faulty encryption is AFA, the approach is similar to DFA, but the effects of a fault injection are exploited by suitable tooling. The *Evaluation Strategy* of AFA is purely analytical and therefore prone to failure if the physical fault injection mechanism may cause noisy fault injections. Consequently, the *Robustness* of AFA is rather low. The biggest advantage of AFA is the way how the key is recovered as this is done automatically by specialized tools.

Table 11.1.: FIA Strategy Comparison

	Characteristic			
Analysis	Evaluation Strategy	Fault Model	Robustness	Overcome Countermeasures
DFA	analytical	–	–	–
PFA	analytical/statistical	+	+	+
SIFA	statistical	++	++	++
AFA	analytical	0	0	–

Part III.

Solutions

12. Overview

The protection of an implementation against potential attacks makes it necessary to implement appropriate countermeasures. These countermeasures can be categorized into three types: Countermeasures against SCA, countermeasures against FIA, and combined countermeasures which provide protection against SCA and FIA simultaneously. In the following we will provide an overview of the three different types of countermeasures.

12.1. Side-Channel Analysis Countermeasures

Resistance against SCA is typically achieved using masking schemes, a common approach to prevent SCA is boolean masking [ISW03] or Threshold Implementation (TI) [NRR06].

A rather versatile masking approach is Domain-Oriented Masking (DOM) as introduced by Gross et al. [GMK16]. DOM is designed to be a scalable, secure masking scheme which is also efficient in terms of required randomness [GMK16]. In addition to that, it prevents leakage caused by glitches by inserting additional registers. In DOM, an arbitrary value x is split into a number of shares each corresponding to a different share domain. Share domains are independent of each other, e.g., shares from a domain A are independent of shares of a domain B. Therefore, for a two share implementation Eq. (12.1) holds.

$$x = A_x + B_x \quad \text{and} \quad y = A_y + B_y \tag{12.1}$$

As for conventional masking schemes, d-th order security is achieved by $d + 1$ shares. In DOM values are split into $d + 1$ share domains. As linear functions only process a single share, they are typically easy to implement. However, non-linear functions require multiple shares and therefore cross the borders of different share domains, i.e., a value is calculated based on values from different domains. Having two shared values x and y as shown in Eq. (12.1), a multiplication of x and y results in the terms as shown in Eq. (12.2).

$$\begin{aligned} z &= x \times y \\ &= A_x A_y + A_x B_y + B_x A_y + B_x B_y \end{aligned} \tag{12.2}$$

All terms in Eq. (12.2) are uncritical as the terms either contain only shares of a single domain or shares of different domains but from the different values x and y. However, considering the case where $x = y$, the cross domains terms $A_x B_x$ and $B_x A_x$ are identical which results in a violation of the independence property, i.e., the cross domain terms cancel each other out. Therefore, a random term R is added to terms which cross domains as shown in Fig. 12.1. The additional register stage after the share addition is required to prevent glitch based leakage.

A reason that makes DOM very efficient in terms of randomness is that in case the inputs x and y are independently shared, the same random share can be used to refresh both cross-domain terms $A_x B_y$ and $B_x A_y$.

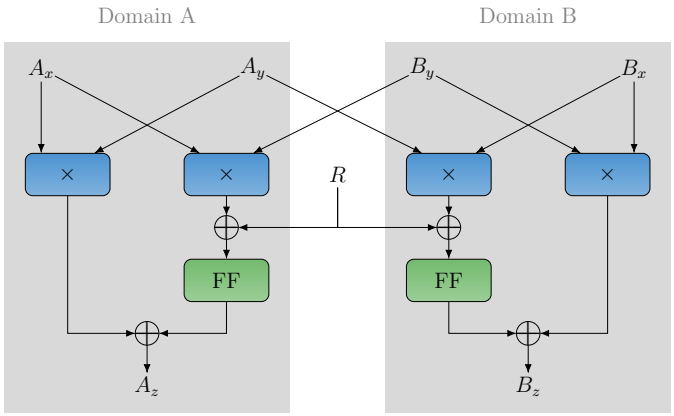


Figure 12.1.: DOM-*indep* multiplier GF(2)

12.2. Fault Injection Analysis Countermeasures

Being able to systematically fault the computation and collect the corrupted outputs of a cipher is a mandatory precondition to apply FIA. As a consequence, a cipher can be protected against FIA, by ensuring that an attacker has no access to faulted outputs. Such countermeasures are traditionally divided into three approaches: *Detection*, *Infection*, and *Error-Correction*.

Detection This kind of countermeasure tries to detect faults occurring during the computation. This is usually accomplished by implementing some kind of redundancy mechanisms. For instance, when computing a result twice and comparing the outputs, a fault in one of the computations would be detected because the results differ. If two computations are equally faulted, the attack cannot be detected. In that case, at least three redundant computations are required to detect the attack. In fact, a result must be computed $k + 1$ times in order to detect up to k faulted computations. In [BE+04], Bar-El et al. provide an overview of some mechanisms to implement detection-based countermeasures. If a fault is actually detected, the output can be discarded and appropriate steps like a system reset can be initiated.

Infection This kind of countermeasure typically "infects", i.e., randomizes the output if a fault occurs. The randomization ensures that the faulted output is useless for an attacker. An advantage of an infection-based approach is that no comparison between different computation results is required. In addition to that, it overcomes the problem that faulting multiple redundant computations will pass the final check before outputting the result. For example, Gierlichs et al. showed a mechanism based on dummy rounds to propagate the fault and infect the cipher's state in case a fault occurred [GST12].

Error-Correction This kind of countermeasure is typically implemented on a bit-level, i.e., every bit of a cipher's internal state is encoded into a codeword. As a result, every fault will be corrected as long as the fault injection does not exceed the code's error correction capabilities. The simplest form of error correction based countermeasure is a Error Correction Code (ECC) built from a repetition code, where each bit of the state is repeated $2k + 1$ times to form a codeword. This

code is then capable of correcting up to k faults per bit. As a result, an attacker can only obtain correct outputs and thus, cannot exploit faulty outputs nor distinguish between effective and ineffective faults. Consequently, a countermeasure based on ECC to defend against SIFA was proposed by Breier et al. [Bre+19a].

12.3. Combined Countermeasures

To this date, several combined countermeasures have been proposed that aim to provide resistance against both SCA and FIA. Most of them combine a masking scheme against SCA with either a detection or an infection mechanism against FIA. In the following, a brief description on state-of-the-art countermeasures is provided.

De Cnudde et al. [DN16] extended the Private Circuits-II scheme of Ishai et al. [Ish+06], we will refer to this countermeasure as *PC-II+*. The main idea behind this countermeasure is to use a threshold implementation for side-channel resistance in an encoded form. That is, every shared value is encoded using a Manchester encoding, i.e., a shared value x will be encoded to a two-bit codeword $(x, \overline{x})$, such that there is only two valid codewords, i.e., $(1, 0)$ and $(0, 1)$. A flipped bit caused by an attacker would result in an invalid codeword which then would be detected. Additional error cascading gates can be implemented to invalidate the output such that an attacker does not obtain information on the faulted ciphertext.

A similar approach was proposed in the hardware-tailored combined countermeasure *ParTI* introduced by Schneider et al. [SMG16]. *ParTI* uses threshold implementations against SCA and Error Detection Codes (EDCs) for fault detection. As the name indicates, the parity part of an input is generated in a first step. Successively, the expected output of the target function is calculated in a so-called predictor. This predictor is designed to be identical to the target function, however with a parity de-/encoder before and afterwards. The output of the predictor is then transformed to an error vector in a *check-and-combine* step to detect a potential fault.

Reparaz et al. proposed *CAPA* in 2018, a countermeasure against physical attacks stemming from the domain of Multi Party Computation (MPC) [Rep+18]. According to the *CAPA* countermeasure, every sensitive value x is accompanied by an information theoretic MAC tag τ_x. Both, the sensitive value x and the corresponding MAC tag τ_x are transformed into a shared representation using boolean masking. In addition to that, the whole circuit is split up into different parts, so-called tiles where each tile stores and processes only one share. However, some operations require multiple shares. Therefore, a protocol is implemented that broadcasts locally randomized versions of the shares such that every tile can operate on all the shares. Due to this broadcasting, each tile can check the correctness of the MAC tag and thus, errors are detected as long as at least one tile is not faulted. *CAPA* provides provable security, however at a large cost of required randomness. That is because the operations (*beaver*-operations) require auxiliary data, so-called *beaver*-tuples.

Another countermeasure which uses information theoretic MAC tags is *Masks and Macs* introduced by De Meyer et al. in 2019 [De +19]. The countermeasure combines boolean masking against SCA with information theoretic MAC tag checking and infection against Fault Analysis (FA). Every sensitive value x is accompanied by an information theoretic MAC tag τ_x, both present in shared form. All the operations in the cipher are performed on both the shared data and MAC tag. That allows to perform an error check on the computed MAC tag and the desired MAC tag. If they differ, an error occurred and a random mask is added to infect the cipher output. Due to the nature of the MAC tags, it's difficult to infect both the data and the MAC tag such that both correspond to each other.

Furthermore, the countermeasure introduced by Breier et al. [Bre+19b] which directly ad-

dresses SIFA can be seen as a more general approach of countering SIFA. The basic principle is to use ECCs such that an attacker cannot distinguish between correct and faulted ciphertexts. This prevents to collect a set of biased ciphertexts. Although error correction codes are inherently limited in the number of faults they can correct, the authors state that this is not a big concern against SIFA. Injecting multiple faults in the same byte would lower the ineffectivity rate and thus automatically throttle the performance of SIFA.

Table 12.1 summarizes the state-of-the-art countermeasures and states the underlying principle and the resistance against different types of attacks, i.e., SCA, SIFA-1, and SIFA-2.

Countermeasure	Principle	Protection against
PC-II+ [DN16]	Masking combined with encoding of masked shares. Invalid codewords invalidates output.	SCA (masking) SIFA-1 (masking) Effective Faults (Encoding)
ParTI [SMG16]	Threshold implementation combined with Error-Detection-Code (EDC).	SCA (masking) SIFA-1 (masking) Effective Faults (EDC)
CAPA [Rep+18]	Masking is combined with MAC tags in a MPC related manner. The circuit is split in different tiles where each tile works on a single share of sensitive value and tag. Intermediate shares are broadcasted and every tile checks correctness.	SCA (masking) SIFA-I (masking) Effective Faults
Masks & Macs [De +19]	Masking combined with information theoretic MACs for infection (Harder to inject faults that result in valid MAC corresponding to faulted ciphertext). Can be extended to include detection mechanism.	SCA (masking) SIFA-1 (masking) Effective Faults (MAC tag)
Encode [Bre+19b]	Error correcting gates to prevent effectiv, ineffective faults.	SIFA-1 (ECC Gate) SIFA-2 (ECC Gate) Effective Faults (ECC Gate)
Toffoli [Dae+19]	Use invertible building blocks, based on Toffoli gates where a fault during the computation does not cancel out.	SCA (masking) SIFA-1 (masking) SIFA-2 (Toffoli Gates) Effective Faults (Toffoli Gates)
Transform-and-Encode [Sah+20]	Transform state into another independent state (can be realized by masking). Apply error correction on transformed state.	SCA (masking) SIFA-1 (masking) SIFA-2 (ECC) Effective Faults (ECC)
RS-Mask [RAD20]	Transform state into another independent state (can be realized by masking). Apply infection mechanism on transformed state.	SCA (masking) SIFA-1 (masking) SIFA-2 (infection) Effective Faults (infection)

Table 12.1.: Overview of Countermeasures and their Resistance against SIFA, and SCA.

12.3.1. Comparison of Combined Countermeasures

PC-II+ [DN16] makes use of masking and encodes the shares to detect invalid codewords. Assuming a secure masking scheme, protection against SCA and SIFA-1 is provided. In addition to that, effective faults changing the ciphers output are detected by the encoding. However, SIFA-2 is expected to be applicable, as the corruption of functions is not specifically covered by *PC-II+*.

ParTI [SMG16] follows a similar approach, i.e., using TI in combination with EDC, the same

arguments as stated for *PC-II+* apply.

The authors of *CAPA* [Rep+18] made use of masking in combination with information theoretic MAC tags to make the infection algorithm more robust to fault attacks. Again, a secure masking scheme (not leaking information on sensitive values) prevents SCA and SIFA-1. The fact that the difference in actual MAC tag and desired MAC tag is used for infection, protects also against effective faults. However, comparing the faulted, randomized output with a correct output still allows collecting biased ciphertexts under the influence of SIFA-2 faults.

As the combination of masking and information theoretic MAC tags is also chosen in *Masks and Macs*, the same assumptions on protection capabilities hold.

The approach of Breier et al. [Bre+19b] is based on the idea of *error-correcting-gates* which allow the correction of effective faults. Due to that, an attacker does not really know whether the fault was effective or ineffective. This makes this classification superfluous and an attacker cannot collect biased ciphertexts. This inherently protects against both types of SIFA. However, an attacker is not limited to the ECC capabilities of correcting a certain amount of errors, as mentioned in [Sah+20]. As a result injecting multiple faults in the same codeword would lower the ineffectivity rate and thus automatically throttle the performance of SIFA.

12.3.2. Countermeasures against SIFA-2

All the countermeasures discussed so far do not provide protection against SIFA under the assumption of the SIFA-2 fault model. As the main focus of our proposed countermeasure lies on the resistance against SCA, SIFA-1, and SIFA-2 we will now introduce the state-of-the-art countermeasures which fulfill these requirements, i.e., the last three entries of Table 12.1.

The first countermeasure proposed by Daemen et al. [Dae+19] introduces two methods to protect a cipher against SIFA. Daemen et al. showed that although a masked implementation is used, an ineffective fault in a single share can depend on a native value. This dependency is caused by one share affecting multiple non-linear gates. They show one method to counter SIFA by using building blocks based on invertible gates so called *Toffoli* gates, such that faults manipulate only a single operation or share. Therefore, faults can either be detected or otherwise an incomplete part of shares is influenced so an attacker can not get information from an ineffective fault. The second method proposed by Daemen et al. is based on the idea to detect faults in shares affecting multiple non-linear gates. As a result, resistance against m-fault injections SIFA, when using $m + 1$ shares can be achieved with higher implementation cost.

The second scheme which fulfills these requirements is *Transform-and-Encode* as proposed by Saha et al. [Sah+20]. The idea is to transform the internal state into a computing domain such that a bias injected by faults does not affect the original state. This transformation can be realized by classical masking schemes. The approach has the benefit of providing side-channel resistance as well. However, if a fault is injected directly in an operation and not in the state, this transformation is not sufficient. Therefore, *Transform-and-Encode* makes use of error correction in order to prevent an attacker from distinguishing between correct and ineffectively faulted cipher outputs.

The third scheme which fulfills these requirements is called *RS-Mask* introduced by Ramezanpour et al. [RAD20]. The main idea of this protection scheme is to map intermediate states to a so-called random space (RS). Computations are then carried out on the random space which provides inherent protection against SCA as the underlying principle of RS-Masks is essentially masking. After the computation, the intermediate state is transformed back into the former state space. The random space transformation also takes care of the randomization of data which is also used as countermeasure against SIFA.

Both approaches have in common that to overcome SCA and SIFA it is necessary to combine a

countermeasure against SCA with a mechanism to overcome SIFA, in both schemes masking also hinders SIFA-1. *Transform-and-Encode* [Sah+20] hinders SIFA-2 using error correction based on duplication codes, in contrast *RS-Mask* [RAD20] uses intrinsic redundancy as an infection based countermeasure which randomizes the output of the computation.

A summary of the countermeasures, and their underlying principles is shown in Table 12.1. Furthermore, a comparison to our approach is shown in Table 13.1. From now on we will only focus on the last two rows of Table 12.1, as they are the only ones which fulfill the requirement to protect against SCA, SIFA-1, SIFA-2, and provide numbers[1] for a protected implementation which we can compare our proposed countermeasure with. One noteworthy detail of Table 12.1 is that SIFA countermeasures which hinder SIFA-1 by masking schemes, are inherently protected against SCA as well.

Every countermeasure shown in Table 12.1 which uses some kind of masking scheme can be seen as a possible instantiation of the *Transform* step of *Transform-and-Encode*. Furthermore, every countermeasure which uses some kind of ECC can be seen as a possible instantiation of the *Encode* step of *Transform-and-Encode*. Therefore, *Transform-and-Encode* is the abstract generalization of the design rationales behind every countermeasure which should provide simultaneous protection against SCA, SIFA-1, SIFA-2, and effective faults.

Consequently, we will also use the principles of *Transform-and-Encode* as a starting point for the design rationale of DOMREP cf. Section 13.1.

[1] Unfortunately [Dae+19] doesn't provide concrete numbers for an implementation.

13. DOMREP a Combined Countermeasure against FIA and SCA

As previously introduced in Chapter 12 the protection of cryptographic implementations requires suitable countermeasures. Subsequently, combined countermeasures have become a necessity for protection of an implementation against both kind of attacks. Recently, the NIST performed a standardization of LWC. One of the second round candidates of this standardization process was Gimli. Therefore, we selected Gimli as a proof of concept for our proposed combined countermeasure Domain Oriented Masking with REPetiton codes (DOMREP).

Contributions: We propose DOMREP, a combined countermeasure against SCA and FIA. In contrast to state-of-the-art combined countermeasures DOMREP provides independently scalable arbitrary-order protection against SCA and FIA including SIFA. We present a secured hardware architecture of Gimli protected by DOMREP. We evaluate the overhead created by DOMREP for Gimli with parameters of protection for different orders of side-channel and fault attacks and compare the principles to the state of the art. Furthermore, we verify the resistance of our Gimli implementation against SCA using TVLA, and the resistance against SIFA by fault emulation. In addition, we provide a guideline of how to perform error correction in the presence of an attacker. In DOMREP we take this problem into account by adding redundancy to the error correction step itself. Also, we consider three different attacker models which require different error correction strategies. Furthermore, we provide guidelines how and where to apply DOMREP.

Organization: The rest of this chapter is structured as follows: Section 13.1 explains our design rationals for the construction of DOMREP. In Section 13.2 we introduce the AEAD scheme Gimli briefly. Section 13.3 provides the results of the application of our proposed countermeasure on Gimli. Finally, Section 13.4 provides a summary of our proposed countermeasure DOMREP.

13.1. DOMREP Design Rationales

Subsequently, to the discussion of state-of-the-art countermeasures which also provide resistance against SIFA in Section 12.3.2, we will outline the design rationales of our proposed countermeasure Domain Oriented Masking with REPetiton codes (DOMREP). We will also introduce the concept of orthogonal protection which combines countermeasures against SCA and FIA simultaneously. The main reasoning behind a combined countermeasure which protects simultaneously against SCA and SIFA can usually be reduced to the general approach to combine a masking scheme with a suitable error correction scheme [Sah+20; RAD20]. Therefore, we opted for a similar approach, i.e., Domain-Oriented Masking (DOM) in conjunction with Repetition Codes (REPs).

13.1.1. Orthogonal Protection

As we aim to create a combined countermeasure which provides simultaneous protection against SCA and FIA we designed DOMREP orthogonally. By orthogonal protection we refer to the property that the security assumptions against SCA and FIA can be tuned independent of each other, i.e., orthogonally. The DOMREP protected domain is composed of two nested domains, i.e., the side-channel protected DOM domain nested into the fault attack protected REP domain. The DOM domain does not provide exploitable side-channel leakage, and the REP domain cannot be manipulated by fault injections. We can therefore deduce that the DOMREP domain is protected against SCA and FIA. Furthermore, as the DOM domain and the REP domain are separated from each other we can tune the resistance against attacks independently. The principle of encapsulated domains is generic and can therefore be applied to a variety of algorithms. The only prerequisite is that the different domains can be separated and each domain can be protected using a suitable principle. The main reasoning behind the choice to embed the DOM-protected domain into the REP-protected domain is the fact that it is the natural approach of avoid leakage first and then ensure there is no corruption, even though the reverse order would also be possible. In order to implement an orthogonal arbitrary order protection scheme it is necessary to fulfill two conditions: The different protection domains must be encapsulated and interconnected securely. The order of protection of the separate domains can be scaled independently.
In the following we will outline the details of how both domains can be encapsulated securely with each other.

13.1.2. DOMREP Fundamentals

For protection against SCA we utilize DOM as a masking scheme. As DOM is a generic approach of protecting multipliers against side-channel leakage it scales well for higher orders of protection, only the architecture of the multipliers is adapted accordingly. Implementations protected against higher-order attacks require more multiplications and therefore more randomness. The order of protection with DOM against SCA is denoted as $d_{DOM} \in \mathbb{N}_0$, where $d_{DOM} = 0$ equals a configuration unprotected against SCA. As a result, $n = d_{DOM} + 1$ domains are required to achieve the degree of protection.

In order to protect against SIFA respectively FIA in general, we opted for a repetition code as error correction mechanism to prevent the collection of effective and ineffective faulted encryptions. One of the reason for that is again the generic scalability of a repetition code in terms of redundancy and the straightforward assumptions of the attacker model. The order of protection against SIFA using a REP is denoted with $d_{REP} \in \mathbb{N}_0$, where $d_{REP} = 0$ equals a configuration unprotected against SIFA. A codeword of order d_{REP} has a length $k = 2\ d_{REP} + 1$ and provides protection against d_{REP} faults. As a result, the representation of a single bit is based on n domains, each share repeated k times. For example the instance of DOMREP with $d_{\mathrm{DOM}} = 1$ and $d_{\mathrm{REP}} = 1$ where every processed bit i can be represented as a 2×3-matrix as shown in Eq. (13.1).

$$i := \begin{pmatrix} A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \end{pmatrix}, \tag{13.1}$$

The domains are oriented column-wise and the codewords are oriented row-wise, where the original value of bit i equals the majority vote of the sum of all domains $A_m \oplus B_m$ where $m \in \{0, 1, 2\}$. In the non-faulty case, each element of a row holds the same value. As an example, we consider the protected multiplication of two bits $\alpha := \beta \times \gamma$. The structural representation of the multiplication which provides resistance against first-order SCA and FIA

is shown in Fig. 13.1. The actual multiplication (blue block) is performed three times and the result of the multiplication in different domains is then feed into the error correction block, i.e., majority vote (red block). For example the input of the first DOM protected multiplier consists from β_{A_0}, β_{B_0}, γ_{A_0}, γ_{B_0} the index zero refers to the first redundant codeword, furthermore A or B refers to the DOM domain. The input to the remaining other two multipliers is specified analogously. Due to the structure of DOMREP a manipulation of a multiplier gets corrected during the ECC, i.e., the majority vote which provides the resistance against SIFA or in general FIA. That means that the possible corrupted outputs of the multipliers $(\hat{\alpha}_{A_m}, \hat{\alpha}_{A_m})$ respectively $(\hat{\alpha}_{B_m}, \hat{\alpha}_{B_m})$ with $m \in \{0, 1, 2\}$ are getting corrected that $\alpha_{A_m} = \text{ECC}(\hat{\alpha}_{A_0}, \hat{\alpha}_{A_1}, \hat{\alpha}_{A_2})$ for all $m \in \{0, 1, 2\}$. How to ensure the correctness of the output of the ECC even if the ECC is under attack will be discussed in Section 13.1.4.

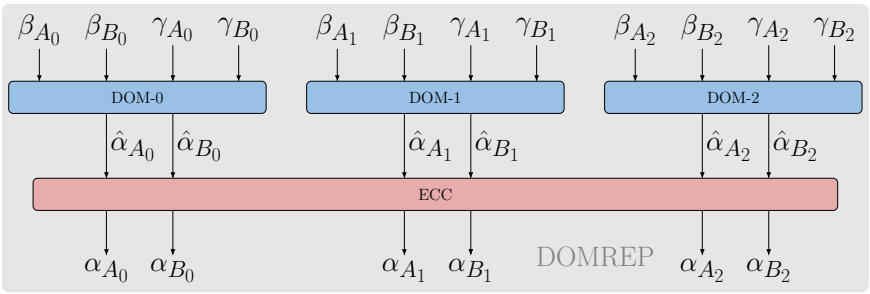


Figure 13.1.: First Order DOMREP Protected Multiplier

In contrast to the protection of a non-linear gate, the protection of a linear operation, i.e., an addition ($GF(2)$) which translates to a logical exclusive or is straight forward as this operation can be applied to all the domains separately, followed by an error correction step as described in the following. For example if one would like to calculate the addition of two variables $\alpha := \beta + \gamma$, this is done via the addition of the variables domains.

13.1.3. Resistance against SCA

The resistance of DOMREP against SCA is purely based on the assumptions of the underlying masking scheme. Therefore, the SCA resistance of DOMREP is based on the assumptions of DOM as introduced by Gross et al. [GMK16]. The ECC replicates the masked data using the repetition code. In our proposed variant of DOMREP this does not exhibit leakage as the share domains are still separated. Only the codewords of a single domain are corrected mutually.

13.1.4. Resistance against SIFA

The reason why we opted for the use of repetition codes to protect against SIFA is based on the assumptions of the SIFA-1 respectively SIFA-2 fault model. Due to the principles of the encapsulated domains it is not necessary that the implementation of the underlying DOM multiplier is resistant against faults, as if one computation fails, the corrupted computation will be corrected using the majority vote after the redundant multiplication.

To achieve protection against SIFA-1 based faults it is necessary to ensure that no intermediate bit (value) can be biased. If we recall the example representation of a single bit in DOMREP as shown in Eq. (13.1) one bit is split up into different domains, e.g., A, B. Furthermore, if we

neglect the repetition code for the following proof of the resistance against SIFA-1 one bit v can be calculated as

$$v = A + B + \cdots \ . \tag{13.2}$$

For resistance against SIFA-1 it is necessary to ensure that no value v is biased. The following proof for the absence of a bias in a bit v is based on Matsui's pilling up lemma [Mat94]. The probability for one share domain to equal one is

$$P(A = 1) = \frac{1}{2} + \epsilon_A \ , \tag{13.3}$$

where ϵ_A is the influence of a possible bias. According to Matsui's pilling up lemma [Mat94], the probability of the sum of all domains can be calculated as

$$P(V = 0) = P(A + B + \cdots = 0) = \frac{1}{2} + \epsilon_V \ , \tag{13.4}$$

where the bias ϵ_V of the resulting value v is calculated as

$$\epsilon_V = 2^{n-1} \prod_{i=1}^{n} \epsilon_i \ . \tag{13.5}$$

If we now consider the structure of Eq. (13.5) it becomes clear that the bias ϵ_V of the addition equals zero as long as a single share domain has a zero bias $\epsilon_i = 0$. Therefore, DOM can be used as a countermeasure against an attacker under the SIFA-1 fault model, following the same reasoning as in [Sah+20; RAD20].

In DOMREP the resistance against the SIFA-2 fault model is based on two principles, the redundancy of data, and the redundancy of functions.

The redundancy of the data can be achieved straight forward if the state (of Gimli) is replicated according to the length of the repetition code k. Therefore, to achieve first-order protection $d_{\mathrm{REP}=1}$ against an attacker which attacks under the assumption of a SIFA-2 fault model it is necessary to replicate a single bit v three times, for second-order protection it is necessary to replicate v five times, in general v must be replicated $2d_{\mathrm{REP}} + 1$ times, where d_{REP} equals the order of protection. The repetition code is applied on bit level, i.e., every bit is represented as a codeword of size $k = 2d_{\mathrm{REP}} + 1$ bit. Therefore, the error correction capability $n_{correctablefaults}$ can be calculated as shown in Eq. (13.6)

$$n_{correctablefaults} = \left\lfloor \frac{k-1}{2} \right\rfloor \tag{13.6}$$

The reason why we opted for the usage of a repetition code is based on the fact that all the redundancy provided by a repetition code is basically a full copy of the original data. As a result it is possible to not only detect and correct errors caused by effective fault injections, but it becomes also possible to apply functions redundantly.

The redundancy of the underlying data, i.e., the state of Gimli is a necessary precondition to apply all operations of the Gimli permutation in a redundant manner. To achieve protection against SIFA-2 based faults it is necessary to protect the sub functions of the Gimli permutation. This means it is necessary to compute all the sub functions redundantly according to the length of the codeword k.

Using repetition codes to protect against SIFA has several benefits and also one major drawback. One of the main benefits of the repetition code is the scalability, i.e., repetition codes can be seen as a generic approach of protection against manipulations under the SIFA-1 and SIFA-2

fault model. The biggest drawback is obviously the increased resource requirement with increasing order of protection. For generic higher-order SIFA-secured implementations, it is sufficient to increase the codeword length of DOMREP. A similar approach was used by Breier et al. using a 3-Repetition-Hamming-Code [Bre+19b] but without the goal to protect against SCA as well.

Due to the inherent structure of DOMREP and the according redundancy of data and functions DOMREP provides resistance against SIFA. However, it is still possible to inject a fault into the computation of the majority vote. To overcome the problem of having a single majority vote under attack it is necessary to decrease the attack surface of the majority vote as well. In order to do so every bit of the codeword is computed by one majority vote from all the bits in the codeword. Therefore, we use k majority votes per codeword each updating one bit of the codeword, the mutual update process is shown in Fig. 13.2.

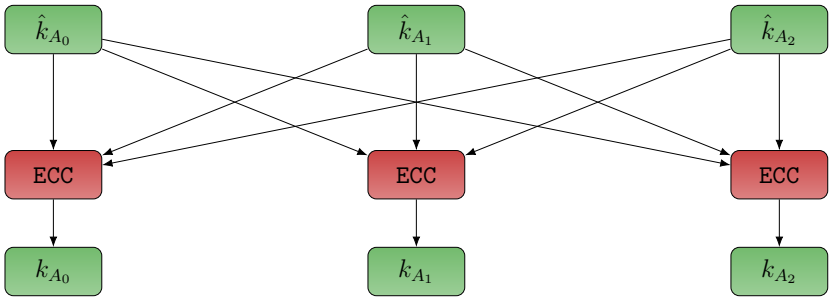
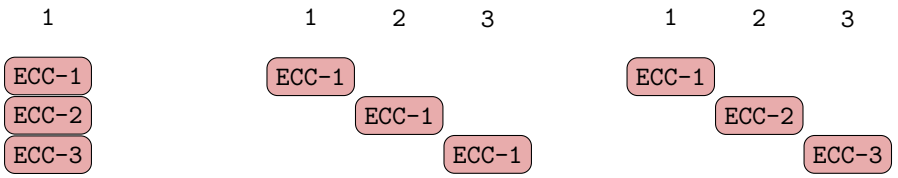


Figure 13.2.: Majority Vote Mutual Update Step

The whole structure of a protected multiplier is shown in Fig. 13.1 where the ECC part of the whole structure is shown in detail in Fig. 13.2. At the example of a single domain one can see the probably corrupted outputs of the multipliers $\hat{\alpha}_{A_0}$, $\hat{\alpha}_{A_1}$ and $\hat{\alpha}_{A_2}$ for all $m \in \{0, 1, 2\}$ are fed into the error correction where the mutual updates $k_{A_m} = \mathrm{ECC}(\hat{\alpha}_{A_0}, \hat{\alpha}_{A_1}, \hat{\alpha}_{A_2})$ for all $m \in \{0, 1, 2\}$ are calculated.

However, it is still necessary to consider how the mutual update is implemented, the three different possibilities are shown in Fig. 13.3 where $d_{\mathrm{REP}} = 1$. The first possibility as shown in Fig. 13.3a is the straightforward approach where at a single point in time three different instances of the majority vote are executed in parallel, this has the benefit of the smallest overhead in time. From a performance point of view the approach shown in Figure 13.3a is the most suitable one for hardware implementations. In contrast, in Fig. 13.3b one instance of the majority vote is executed in three different points in time, which has the benefit of having less resource requirements but a larger overhead in time. From an implementation point of view the approach shown in Figure 13.3b is the most suitable one for software implementations. The last configuration, depicted in Fig. 13.3c, is the most robust one as the three different instances of the majority vote are utilized at three different points in time. The choice of the configurations is depending on the threat model, i.e., if we assume an attacker who will use a fault attack based on a global effect like glitching it is necessary to spread the computation in time like in Figs. 13.3b and 13.3c. On the other hand if we assume an attacker who uses laser fault injection it becomes necessary to spread the majority vote in space like in Figs. 13.3a and 13.3c where three different instances of the majority vote are used for the mutual error correction step. The majority vote based on the architecture shown in Fig. 13.3c provides the robust security assumptions in terms of time and space, but also requires the most resources in time and space.



(a) Variant one (b) Variant two (c) Variant three

Figure 13.3.: Protected Majority Vote

13.1.5. DOMREP Summary

The resistance against a passive attacker is build upon the underlying masking scheme DOM and can be scaled to arbitrary order by an increasing the number of domains. The resistance against an active attacker is based on several properties. The first line of defense against an attacker under the assumption of a SIFA-1 based fault model is established by the masking scheme. The second line of defense against an attacker under the assumption of a SIFA-2 based fault model is based on the approach to calculate redundant functions on redundant data. To ensure the integrity of the underlying data the error correction is also performed redundantly according to the threat model.

13.1.6. DOMREP Comparison

Followed by the introduction of DOMREP we will now compare our proposed countermeasure with the state-of-the-art countermeasures as introduced in Section 12.3.2. The main benefit of DOMREP is that the resistance against passive SCA and active FIA, and SIFA can be tuned independently according to the assumed attacker.

Orthogonality One of the main design goals of DOMREP was the ability to scale the level of protection against possible threats, in order to ensure this kind of ability we designed DOMREP orthogonal, i.e., scale the resistance against SCA and FIA independent of each other. The ability to scale is ensured by the orthogonal composition of DOM and REP which enables, e.g., changing the order of protection in our proposed Gimli design by changing a parameter prior to synthesis. In contrast, Saha et al. [Sah+20] and Ramezanpour et al. [RAD20] et al. do not mention the importance of free scalability.

SCA - Resistance The general approach to protect against SCA is the use of a suitable masking scheme. Saha et al. [Sah+20] mention that most masking schemes fulfill the properties required by their *Transform* operation. Similarly, Ramezanpour et al. [RAD20] designed their countermeasure to be used with most masking schemes. In contrast, we designed DOMREP to be explicitly used with DOM, as this masking scheme was especially designed to be free from glitches and requires less randomness compared to other masking schemes [GMK16]. Furthermore, we evaluated the resistance of DOMREP against SCA on an implementation of Gimli, in contrast to Saha et al. [Sah+20] (Present) and Ramezanpour et al. [RAD20] (AES), as we would like to show that DOMREP is also suitable for sponge-based cryptography.

SIFA-1 - Resistance The resistance against SIFA under the assumption of a SIFA-1 based fault model is achieved by the utilization of masking this approach is used by the schemes of Saha et al. [Sah+20] , Ramezanpour et al. [RAD20], and our work. Saha et al. describe masking as a possible implementation of their *Transform* operation. Similarly, Ramezanpour et al. use masking as a possible implementation of their random space transformation.

SIFA-2 - Resistance Compared with Saha et al. [Sah+20] where the resistance against SIFA-2 is based on their *Encode* operations which is essentially an error correction step. Ramezanpour et al. [RAD20] hinders SIFA-2 using the intrinsic redundancy introduced by their *RS-Mask* as error detection in combination with an infection based mechanism. In contrast, we follow a slightly different approach against SIFA-2 based attacks as we use the combination of redundant computation on redundant data and mutual error correction this provides protection even if the error correction itself is under attack.

A comparison of the state-of-the-art countermeasures functional principles with the proposed one is shown in Table 13.1.

	Protection against		
Scheme	SCA	SIFA-1	SIFA-2
TaE [Sah+20]	masking	masking	ECC
RS-Mask [RAD20]	masking	masking	Infection
This work	masking	masking	Redundancy of Data and Functions

Table 13.1.: Summary of combined Countermeasures

13.2. Gimli

We will now briefly justify why we have chosen Gimli as a proof of concept, for a detailed description of Gimli cf. Section 9.1. The main reason to showcase DOMREP at the example of Gimli is twofold:

First, the structure of Gimli allows the implementation of the domain-independent variant of the multiplier proposed by Gross et al. [GMK16] as the inputs of the multipliers are independent and therefore independently shared. As a result, less randomness is required in comparison to other masking schemes [GMK16].

Secondly Gimli's permutation only operates on $GF(2)$ therefore a multiplication is the equivalent of a logical `AND`. In order to protect Gimli against SCA and SIFA it is necessary to protect the core component of Gimli, i.e., the permutation. A closer look at the permutation reveals that Gimli's permutation consists only of logical `AND`, `OR` and `XOR`. The logical `OR` can be transformed into its `AND` representation by the application of De Morgan's law. The operations of the permutation can be protected using DOMREP as introduced in the previous section. The interaction of the permutation and the duplex mode [Ber+12b] consists only of logical `XOR`s.

A detailed description of the hardware architecture of Gimli is given in [Gru+21], in contrast we will focus on DOMREP itself.

13.3. Results

We evaluate the effectiveness of the DOMREP countermeasure following the hardware architecture from [Gru+21]. The countermeasure is implemented with $n = 2$ shares and repetition code

length $k = 3$ on a CW305 Target Board featuring an Artix-7 (XC7A100TFTG256) running at a clock frequency of f_{clk}=1 MHz. To verify the resistance of the protected Gimli implementation against SCA and FIA, we provide results for TVLA and fault emulations in Section 13.3.1 and Section 13.3.2 respectively.

13.3.1. Side-Channel Analysis

The side-channel resistance of DOMREP is provided by Domain-Oriented Masking (DOM) as described in Section 12.1. In order to evaluate the side-channel security of our protected implementation, a PicoScope 6402D USB oscilloscope is used to perform trace measurements at a sampling frequency of f_s=156.25 MHz. We acquire power measurements through the SMA jack of the CW305 board, that provides the voltage drop of the FPGA's internal supply voltage over a 100 mΩ shunt amplified by a 20 dB low-noise amplifier. We did not use associated data and plaintext for all measurements. Thus, according to Fig. 9.1, only three permutations are processed: one during the initialization, one after associated data padding and a final one during the finalization of the tag. For trace alignment, a trigger event is output by the DUT after each permutation.

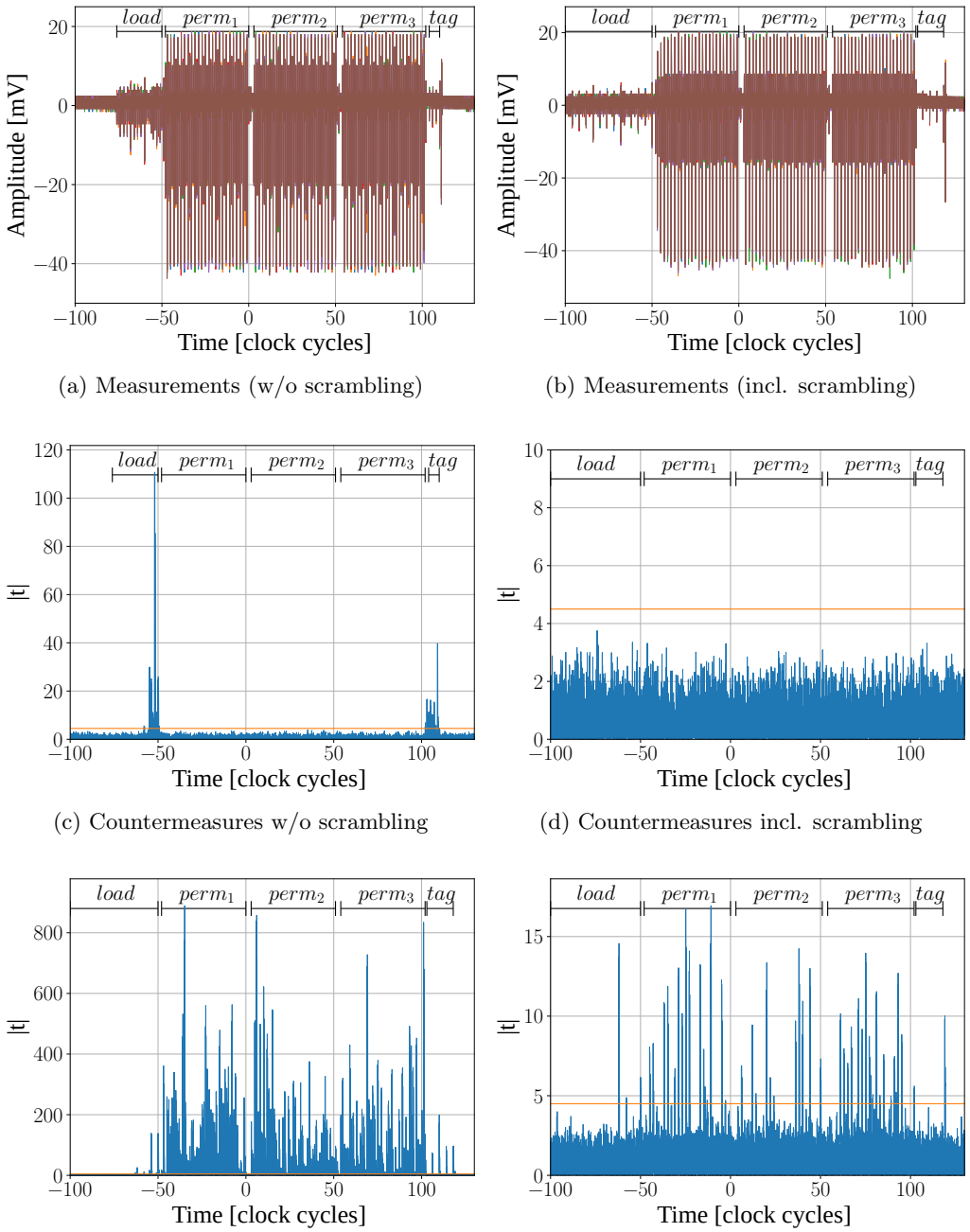
To ensure that the transmission of DOM-shared data to or from the DUT does not cause any leakage, we scrambled the communication with the DUT. We opted for this approach as transitional leakage, i.e., the HD between DOM-shares is caused when the shares are transmitted sequentially over the same bus, e.g., Universal Asynchronous Receiver Transmitter (UART). In our measurements, this type of leakage only occurs before and after the computation of Gimli. To prevent this transitional leakage, the data is scrambled before it is transmitted to the DUT, i.e., additional random masks are added to the shares. The random masks are generated by a Pseudorandom Number Generator (PRNG). We used the 32-bit version of `Xorshift` as proposed by Marsaglia et al. [Mar03]. As the data is only scrambled during the transmission, Gimli still operates on the descrambled data.

In Figs. 13.4a and 13.4b several raw measurements are depicted, with and without scrambling respectively. All measurements are aligned with respect to the first trigger that occurs at the end of the first permutation. The three Gimli permutations can be clearly distinguished. Before the first permutation, the key and nonce are loaded into the state, after the last permutation, the tag is generated. Both operations are visible through regular peaks. Note that the time for loading and tag finalization takes twice as long if scrambling is enabled. This is due to the extra data registers used to avoid glitches as described in [Gru+21].

In Figs. 13.4c to 13.4e the TVLA results (c.f. Section 5.4) for the protected Gimli implementation with different levels of countermeasures are shown. For each scenario 200,000 measurements are used, where for each measurement the key takes the same value and the nonce is randomly chosen to be either a random or fixed value. For all results $|t| > 4.5$ indicates side-channel leakage that is represented as an orange line. While a high t-value does not guarantee an attack to actually be possible, for a protected design t-values above the threshold should not occur.

First, Fig. 13.4e shows the TVLA results when countermeasures are disabled by setting all mask values to zero. The t-values exceed the threshold of 4.5 during all permutations and during loading and tag finalization. These results indicate that without further protection side-channel leakage occurs for the Gimli hardware implementation. Furthermore, the measurement setup is verified, i.e., it allows for acquiring measurements to detect leakage with TVLA.

Second, in Fig. 13.4c the TVLA results for the protected implementation, but without scrambling of the data during transmission, are depicted. Note that there are no t-values above the threshold during the permutation, while for the loading and finalization phases leakage can be observed. As outlined in [Gru+21], this leakage stems from transitional Hamming distance leak-



(a) Measurements (w/o scrambling)

(b) Measurements (incl. scrambling)

(c) Countermeasures w/o scrambling

(d) Countermeasures incl. scrambling

(e) Countermeasures disabled

(f) Countermeasures incl. scrambling (2nd order)

Figure 13.4.: Side-channel leakage assessment of the protected GIMLI hardware implementation: (a)-(b) example raw measurements, (d)-(f) TVLA results with fixed key and fixed-vs-random nonce for 200,000 measurements and different levels of protection.

age between two consecutive shares that are transferred over the same 32-bit bus to or from the *CryptoCore*. Most importantly, the results highlight that the protection mechanisms of DOMREP are indeed working as the countermeasure is aimed at protecting the permutations.

Finally, Fig. 13.4d confirms that the scrambling introduced in [Gru+21] entirely removes the transitional Hamming distance leakage during data transmission. In other words, using scrambling no univariate first-order leakage can be detected from data transmission until tag finalization. Additionally, the second-order TVLA in Fig. 13.4f indicates univariate second-order leakage during the permutations and data transmission. This is expected for a first-order secured hardware implementation with $n = 2$ shares and confirms the degree of protection.

13.3.2. Fault Injection Analysis

In order to verify the integrity of the DOMREP protected implementation of Gimli against FIA, the hardware implementation was attacked by means of fault emulation to simulate the strongest possible attacker which is able to influence single bits.

The SIFA of Gimli was already shown in Chapter 9. The proposed attack is able to attack the initialization phase of Gimli where the 256-bit key and the 128-bit nonce are loaded into the state as shown in Fig. 9.1. During the initialization phase two different rounds of the Gimli permutation can be chosen for an attack, i.e., round 22, 21 where round 24 denotes the first round.

In order to analyze the resistance of the FPGA design against SIFA, the error-correction capabilities must be evaluated. If we assume the implementation is able to correct all injected faults an attacker cannot distinguish between correct and faulty samples and thus cannot apply SIFA respectively FIA. We therefore assume that an attacker is capable to locally inject faults with bit-level precision, i.e., the strongest possible attacker.

In order to verify the resistance of DOMREP against FIA, and SIFA-1, several state bits were either tied to a fixed value (persistent) or inverted (transient) such that a fault injection is simulated, i.e., fault emulation. Applying these manipulations to the inputs of the ECC modules showed, that all faults were corrected successfully as far as they didn't exceed the repetition code's correction capabilities, i.e., correcting k faults for a codeword length of $2k + 1$. That means, a first-order secure design with a codeword length of 3 is able to correct 1 fault per codeword. This allows to correct up to 384 faults per state share, as long as no codeword is faulted twice. If multiple faults per codeword are injected, increasing the codeword length accordingly allowed to correct all faults. This approach was validated in simulation as well as on the hardware itself using fault emulation.

The resistance of DOMREP against SIFA-2 was verified with either a modification of the ECC module or the DOM protected `AND` gate in order to create a faulty codeword. Due to the fact that every bit of a codeword is (mutually) updated with a separate ECC module, this only leads to one faulty bit in the resulting codeword which is corrected in the following cycle. By manipulating either some bits on the ECC modules output, or the output of the DOM protected `AND` gate, this behavior can be verified. In case the faults were injected in some intermediate rounds of the permutation, the faults were corrected as expected using the mutual update step as shown in Fig. 13.2. Again, if multiple faults were injected, the codeword length had to be increased accordingly. Contrarily, when faulting the 0-th codeword bit in the last round, the cipher will output the faulty value. That is because the faults will not be corrected after the last round and the cipher always outputs the 0-th bit of a codeword. Nevertheless, this is no issue from a security point of view, as this would mean that an attacker modifies the computed output.

Therefore, SIFA is not applicable as long as the assumed attacker is not able to overcome

the capabilities of the repetition code. However, if the attacker is able to inject global faults by applying for instance clock glitching, then all bits of a codeword or all ECC instances updating one codeword might be faulted. In order to mitigate this, the error-correction must be implemented sequentially such that every bit of a codeword is updated in a consecutive clock cycle as shown in Fig. 13.3. Glitching the circuit once would then only lead to single bit errors which will be corrected in consecutive rounds. As updating a codeword of length k then requires k cycles, the resulting throughput is decreased by a factor of $1/k$.

13.3.3. Overhead

The introduced overhead of DOMREP is shown in Table 13.2. Where the overhead is shown as a scalar factor based on the comparison of an unprotected implementation with a first order secure implementation (SCA, SIFA-1, SIFA-2, and effective faults). Furthermore, the overhead is compared to the schemes of Saha et al. [Sah+20] and Ramezanpour et al. [RAD20]. As one can see, the countermeasure of Ramezanpour et al. introduces the lowest overhead due to the infective nature. In contrast, DOMREP scales similarly to the countermeasure of Saha et al. [Sah+20].

			First-Order	
Cipher	Metric	unprotected	SCA-secure	SCA, FIA-secure
Present [Sah+20]	GE	1.0 [Pos+11]	4.96	15.41
AES [RAD20]	Area	1.0	1.70	3.47
Gimli	LUT	1.0	2.51	8.05
	REG	1.0	4.78	11.96

Table 13.2.: Overhead of Combined Countermeasures

13.4. Summary

In this chapter we introduced DOMREP, a new robust solution for the simultaneous protection against SCA and FIA. Within DOMREP the individual countermeasures DOM and REP can be scaled independently in order to address the required security levels against side-channel and fault injection attacks according to application needs. In order to evaluate the side-channel resistance of our proposed countermeasure we protected a Gimli implementation to be first-order secure. TVLA results show an absence of any exploitable side-channel leakage using up to 200,000 measurements. As a fault attack model we addressed the most powerful attack model, i.e., SIFA-1 and SIFA-2 by means of fault emulation, which did not result in exploitable measurements. The overhead introduced by DOMREP requires a careful evaluation of the protection requirements. Fortunately, due to the orthogonality of the two schemes, the parameters can be chosen independently which allows for a very good adaptation.

14. TOFU Toggle Count Analysis of Cryptographic Implementations

Unprotected cryptographic implementations can cause potential side channels, due to data-dependent execution time or data-dependent power consumption. The most commonly used side channel is the device's power consumption. One approach to exploit this data-dependent power consumption is DPA as introduced by Kocher et al. [KJJ99]. Data-dependent power consumption occurs due to the physical properties of hardware, i.e., power consumption of CMOS circuits cf. Section 5.1. To verify the presence of leakage, one can measure the device's power consumption during the execution of a cryptographic algorithm and apply either DPA or TVLA. However, this approach has the disadvantage that some hardware is required to run the algorithm, as well as a device to record the power consumption, such as an oscilloscope. Furthermore, the measurements acquired by an oscilloscope are noisy by nature. An alternative to measuring a real device is to simulate it, which results in completely noiseless measurements that are also reproducible.

Related Work Veshchikov et al. proposed a simulator called SAVRASCA for the AVR architecture [VG17]. Mc Cann et al. proposed another simulator called ELMO [MOW17]. ELMO was developed as a profiled simulator and is specifically targeted at the ARM Cortex-M0 architecture. Another approach was taken by Le Corre et al. [CGD18] with their simulator MAPS which was tailored for the ARM Cortex-M3 architecture. MAPS was build based on the VHDL model of an ARM Cortex-M3 microcontroller where the pipeline was analyzed for instructions which exhibit data-dependent power consumption. In contrast, Sadhukhan et al. [Sad+19] focused on hardware implementations and proposed an SCA resistant design flow, they also proposed a toggle count based leakage model. Furthermore, Wamser et al. were able to show a toggle count based CPA of the AES [Wam19] using the `VCD2R` package available for the R programming language [Wam].

Contributions In this chapter we propose TOggle Foul-Up (TOFU), a versatile open-source tool to synthesize Value Change Dump (VCD) [IEE06] simulation traces into power traces, with an adjustable leakage model. Furthermore, we propose a security evaluation workflow based only on open-source tools. We verify the capabilities of TOFU at the example of the CPA of an AES hardware and software implementation. Although TOFU shares some similarities with `VCD2R` the focus of TOFU is on performance and the evaluation of protected implementations.

Organization The rest of this chapter is structured as follows: Section 14.1 introduces TOFU. Section 14.2 evaluates TOFU's performance. Section 14.3 introduces the proposed workflow, while Section 14.4 showcases the workflow at the example of an AES hardware implementation. Section 14.5 compares traces of an AES software implementation obtained by either simulation or measurement using the proposed workflow. Finally, Section 14.6 concludes this chapter.

Key	Description
vcdGlob	Glob to find VCD files.
signalsFileNameLiterals	Signals used for the leakage synthesis.
leakageModel	Leakage model used for the synthesis.
window	Use only samples from a window.
windowFrom	Specify window start.
windowTo	Specify window stop.
valueExtractFunction	Name of function which extracts values.
valueExtractIndex	Store the trace index as value.
writeTraces	Store traces in file or memory.
writeTracesBatchSize	How many traces to write at once.
traceFileName	Filename of the generated traces.
format	Format of the generated traces.

Table 14.1.: TOFU Settings Summary

14.1. TOFU

As TOFU is intended to be a helpful utility for research and teaching, the source code can be found here[1]. We will now introduce the most important settings of TOFU used during trace synthesis. TOFU is implemented in Python, but parsing is done in C++ alternatively parsing can be done in Python as well, which is helpful for testing, e.g., new leakage models. There are various settings which can be passed to TOFU, the most important ones are shown in Table 14.1. A regular expression describing the VCD files to parse is given by *vcdGlob*. Supported leakage models are either HW, or HD and can be chosen by *leakageModel*. It is possible to filter the signals used for the leakage generation by specifying them in *signalsFileNameLiterals*. The value of a signal can be extracted for any timestamp by specifying a *valueExtractFunction*. This can for instance be used to extract the value of, e.g., plaintexts, keys, or ciphertexts. Furthermore, values for multiple signals can be extracted as a single value, which is for instance useful if the plaintext consists of multiple signals like in masking. Alternatively, *valueExtractIndex* can be used as value extraction function, which is useful in TVLA [GGJR+11] for the differentiation between traces with fixed plaintext and random plaintext, based on the index of the current trace. If *window* is set, the generated leakage is restricted to all timestamps between *windowFrom* and *windowTo* which also speeds up the synthesis. Setting *writeTraces* to true indicates that the generated traces should be written to the file given by *traceFileName* with the batch size of *writeTracesBatchSize*. TOFU's default file format is *HDF5*, which is also the default format of Ledger's Advanced Side Channel Analysis Repository (LASCAR), other formats can be specified with *format*. New file formats can be easily integrated thanks to the modular architecture

14.2. Performance

In order to evaluate the performance of TOFU in terms of synthesis speed we evaluated the required time to parse VCD files of different sizes. Additionally, we used either the Python based parser, or the C++ version. Using C++ instead of Python is motivated by higher performance, which mainly arises due to the C++ code being compiled and optimized. Moreover, C++ offers more control than Python, for instance through pointers and manual memory management.

[1] https://gitlab.lrz.de/tueisec/tofu

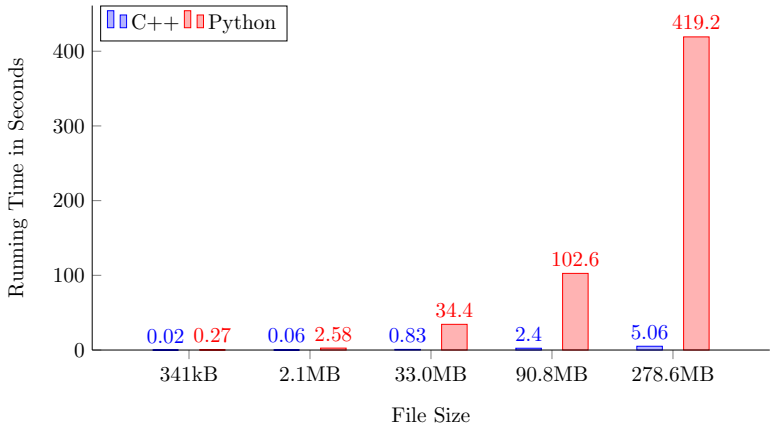


Figure 14.1.: Running Time Comparison

Besides using a different programming language, the C++ implementation employs additional optimizations. Firstly, VCD files are memory-mapped, thus allowing for fast read operations of the VCD file. This is because no system calls are necessary, among others. The time required by the C++ implementation to parse a VCD file only increases proportionally to the size of the VCD file, i.e., $\mathcal{O}(n)$. For instance, if parsing a 5 MB file takes 1 s, then parsing a 10 MB file should take about 2 s. This can only be achieved by constant lookup times of the values of symbols in the value change section of the VCD file. These lookups are necessary for calculating the leakage. For this reason, the C++ implementation uses a hash table which gives average constant-time insertion and search. Figure 14.1 compares the running time of the C++ implementation with the Python implementation for different VCD file sizes. For a better comparison, the time required for LASCAR to write the resulting traces is omitted. As one can see the required running time for the C++ implementation grows linearly with respect to the file size. TOFU was successfully used during the development of DOMREP cf. Chapter 13 which is indicated by the biggest file of Table 14.1.

14.3. Workflow

One of the main objectives of this chapter was to specify a workflow which is only based on open-source tools. The necessary steps of the workflow can be formulated as: *Simulation*, *Synthesis*, and *Analysis*.

Simulation For a hardware implementation the *Simulation* step (behavioral) is based on G Hardware Design Language (GHDL) an open-source VHDL simulator developed by Gingold et al. [Gin]. GHDL compiles Very High Speed Integrated Circuit Hardware Description Language (VHDL) files directly to machine code and allows native execution to allow high speed simulations. We have also verified the *Simulation* step of the workflow with *Vivado 2020.2*, which enables also simulations which take hardware specific delays into account, e.g., post-implementation simulation. For a software implementation the *Simulation* step is based on Unicorn an open-source Central Processing Unit (CPU) emulator based on QEMU.

Synthesis The *Synthesis* step is based on TOFU as introduced in this chapter. Prior to the *Synthesis* suitable settings must be chosen as specified in Section 14.1.

Analysis The *Analysis* step is based on LASCAR an open-source framework developed by Ledger [Led]. LASCAR is a versatile SCA framework which supports several SCA attacks, e.g., DPA, and CPA. Furthermore, LASCAR provides a convenient container format based on *HDF5* to store, and access acquired traces.

14.4. Exemplary Workflow

In the following section we will now outline the workflow introduced in Section 14.3 at the example of a CPA of an AES hardware implementation[2] written in VHDL using 10 000 traces. The AES implementation under attack applies all S-boxes simultaneously.

14.4.1. Simulation

The TOFU repository contains a testbench for the implementation of AES, which can be used as a basis for other VHDL projects. The integration of the AES's testbench and GHDL was done with parallelism in mind, to utilize several processor cores. Unfortunately, GHDL does not allow vectors as parameters [Gin]. Instead, it is possible to pass multiple integers as parameters and combine them into a vector. Passing parameters to the GHDL simulation avoids repetitive recompiling of the VHDL source to native code. The generation of the random plaintexts required by the CPA is done in a pseudo-random manner where a random number generator is seeded deterministically.

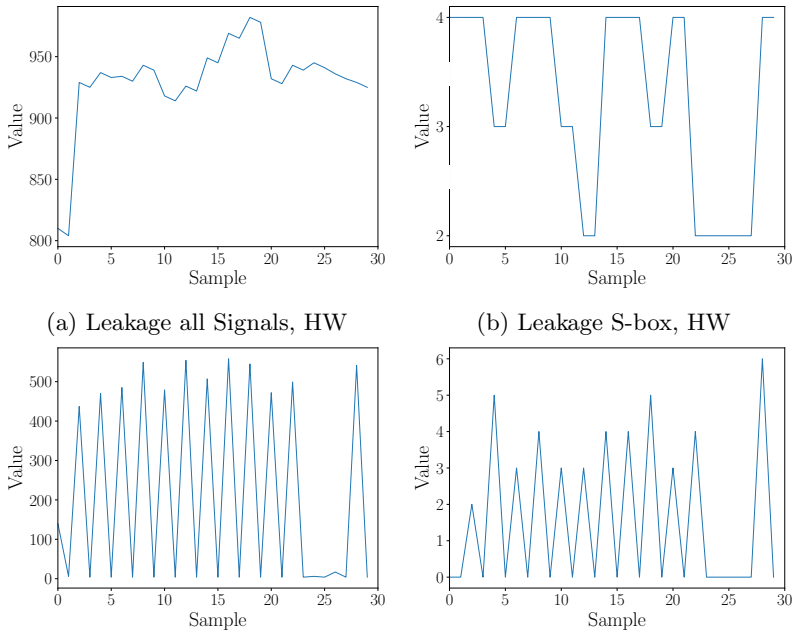
14.4.2. Synthesis

In order to demonstrate a subset of TOFU's features we have done several syntheses. For the leakage model we either used the HW, or HD. Also, for the filtering, i.e., *signalsFileNameLiterals* we either used all signals available in the VCD file, or only the signals of the first S-box's output only. The corresponding VCD has a size of 26 kB. The synthesized trace for an exemplary AES encryption (unfiltered) is shown in Fig. 14.2a under the assumption of a HW leakage model, while Fig. 14.2c shows the same encryption under the assumption of a HD leakage model. In contrast, if only the signals from the first S-box's output are used for the synthesis this is shown in Fig. 14.2b under the assumption of a HW leakage model, while Fig. 14.2d shows the same encryption under the assumption of a HD leakage model where the values take values between zero and eight as one may expect due to the S-box used in AES. Noteworthy, in Fig. 14.2c, and Fig. 14.2d every second sample has a value of zero, as the circuit is only sensitive to the clock's rising edge. The circuit's sensitivity to the clock's rising edges can also be used to fasten up the *Analysis* step if every second sample is discarded which results in a smaller power trace.

14.4.3. Analysis

In the last step of the proposed workflow the *Analysis* takes place, i.e., in the context of a CPA on AES the first round key is attacked (encryption). As usual, the CPA assumes the HW as underlying leakage model, the whole CPA is conducted by LASCAR. For the analysis we consider two different cases, i.e., traces generated from either all signals (*unfiltered*) or only the

[2] The AES VHDL implementation can be found inside TOFU's repository.



(a) Leakage all Signals, HW

(b) Leakage S-box, HW

(c) Leakage all Signals, HD

(d) Leakage S-box, HD

Figure 14.2.: AES Workflow – Synthesis

first S-box's output (*filtered*). The *unfiltered* correlation progression plot for all key hypotheses is shown in Fig. 14.3a, as one can see the absolute value of correlation of the correct key hypothesis converged to approximately 0.25 after 1000 traces.

14.4.4. Evaluation

According to Brier et al. the maximum value of 0.25 can also be calculated as shown in Eq. (14.1), where ρ_{WH} denotes the Pearson correlation coefficient $\rho_{WH} = \{x \in \mathbb{R} \mid -1 \leq x \leq 1\}$ [BCO04]. Especially for the case of AES $l = 8$ denotes a byte among a state of $m = 128$ bits

$$\rho_{WH_{l/m}} = \rho_{WH}\sqrt{\frac{l}{m}} \tag{14.1}$$

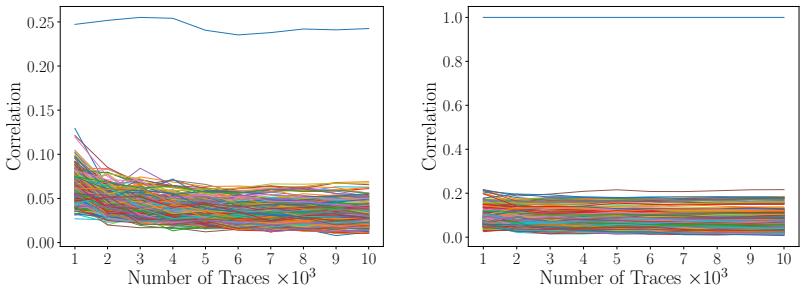
Consequently, for the *unfiltered* case the maximum absolute correlation is calculated as shown in Eq. (14.2), which confirms TOFU's correct behavior.

$$|\rho_{WH_{8/128}max}| = \sqrt{\frac{8}{128}} = 0.25 \tag{14.2}$$

Respectively for the *filtered* case Fig. 14.3b shows that the absolute correlation of the correct key hypothesis converges to 1.0 instantly cf. Eq. (14.3).

$$|\rho_{WH_{8/8}max}| = \sqrt{\frac{8}{8}} = 1 \tag{14.3}$$

None of the other 15 bytes can be recovered from such filtered traces correctly as the correlation of the remaining bytes does not converge towards a value in their respective correlation progression plots.



(a) Correlation, HW (b) Correlation filtered, HW

Figure 14.3.: AES Workflow – Analysis using 10 000 Traces

14.5. Leakage Simulation versus Leakage Measurement

We will now compare the power traces of an AES software implementation obtained either by simulation or by actual measurements. As the measured and simulated power traces differ by length and shape, we decided to use a comparison approach based on the Pearson correlation calculated for each set of traces during a CPA [BCO04].

14.5.1. Measurement

The measured traces are obtained with the help of a ChipWhisperer (Lite), which also contains a STM32-F303RCT7 as DUT [OC14]. The ChipWhisperer features hardware for power analysis, device programming, glitching, and serial communication. Furthermore, the ChipWhisperer uses a 10 bit ADC with a 105 MS/s sampling rate combined with an amplifier with up to 55 dB gain to measure small signals with ease. The used measurement setup is shown in Fig. 14.4.

14.5.2. Simulation

The simulated traces are obtained using a combination of two tools, Unicorn [Uni] and TOFU. In order to generate power traces from a software implementation, it is necessary to simulate the DUT's CPU and track the CPU's state accordingly. To achieve this, we used Unicorn [Uni]. Unicorn is designed to be a lightweight, multi-platform, multi-architecture CPU emulator framework based on the QEMU emulation engine. The state of the CPU as well as the memory is dumped in a suitable format for further processing, i.e., VCD. As a first step, we used Unicorn to simulate an AES implementation and dumped the state into a VCD file. As a second step, we used TOFU to synthesise power traces from the generated VCD simulation traces. The ability to observe register values and memory values makes it possible to visualize the contents over time. The content of the Programm Counter (PC) register is shown in Fig. 14.5, as one can see the key schedule takes place in the beginning followed by the key whitening and the subsequent ten rounds of AES. Noteworthy, the omission of *MixColumns* in the last round is visible as well. The used simulation environment[3] tracks the whole memory and the state of all registers which makes a reasonable choice as a proof of concept in order to tune the simulations to reflect actual measurements one may, e.g., only track memory read and write operations.

[3]The simulation environment to track the state of the DUT as well as the used software implementation of AES can be found inside TOFU's repository.

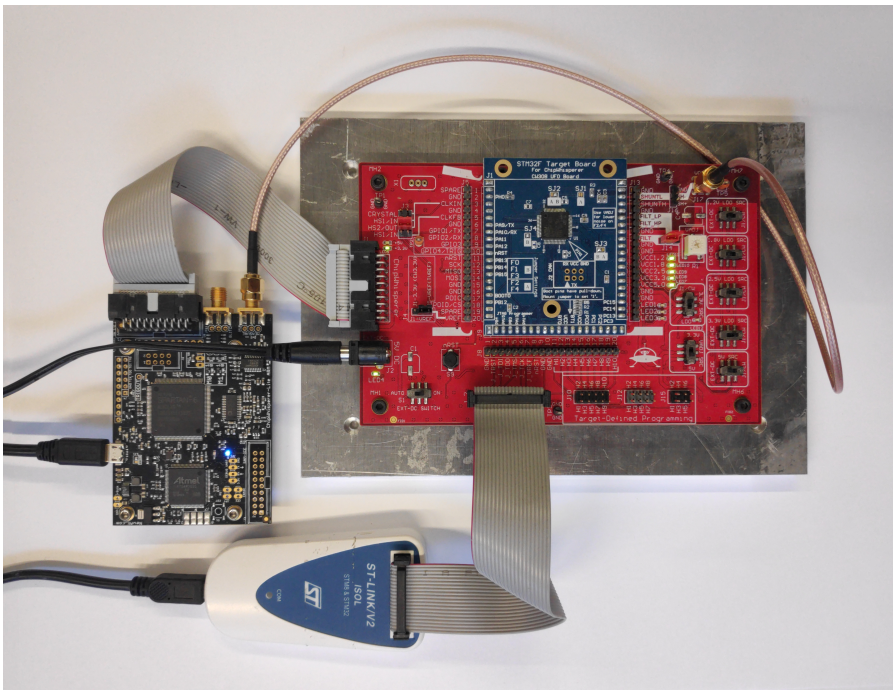


Figure 14.4.: ChipWhisperer Measurement Setup

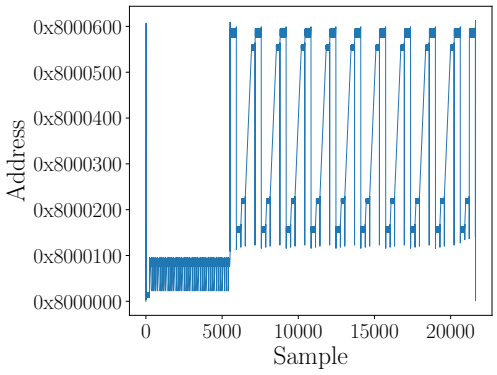
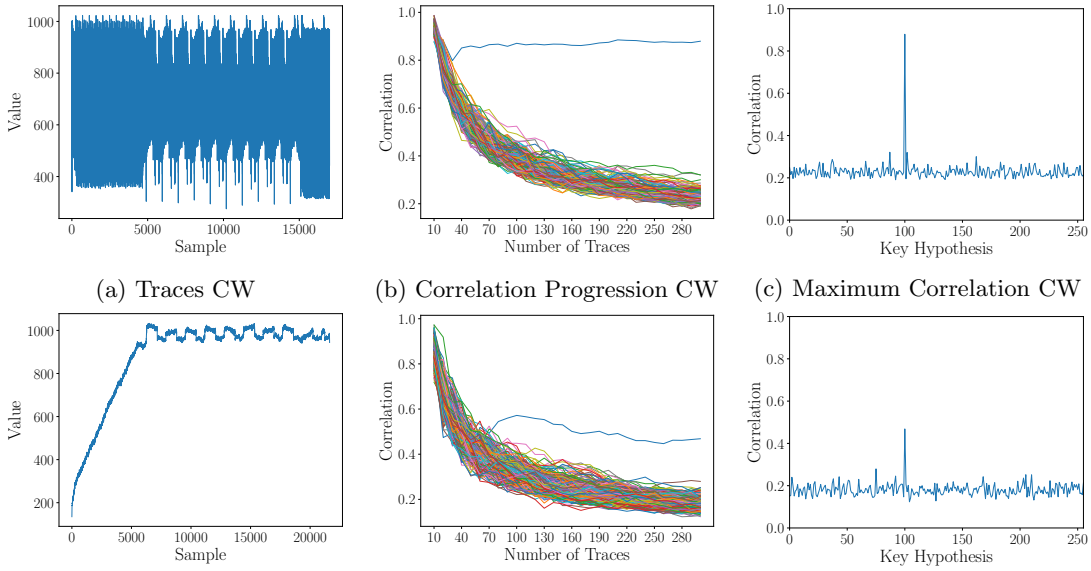


Figure 14.5.: Trace – Program Counter



(a) Traces CW (b) Correlation Progression CW (c) Maximum Correlation CW

(d) Traces UC (e) Correlation Progression UC (f) Maximum Correlation UC

Figure 14.6.: AES CPA, ChipWhisperer (CW), Unicorn (UC)

14.5.3. Analysis

The AES implementation under attack is written in *C*. For simplicity reasons we opted for a 8 bit implementation of AES as this makes optical inspection easier than an implementation based on T-tables [FIP01]. The generation of the random plaintexts required by the CPA is done in a pseudo-random fashion, where a random number generator is seeded deterministically. Therefore, the measurement approach and the simulation approach use exactly the same plaintexts during each run. In addition, the multiplications required by the *MixColumns* operation of AES are implemented as a lookup table to ensure a constant time implementation of AES, which is particularly helpful during analysis as it results in less temporal jitter. As usual, CPA assumes the HW as the underlying leakage model [BCO04]. The whole CPA is performed by LASCAR [Led]. The measured traces are shown in Fig. 14.6a, while the simulated traces are shown in Fig. 14.6d, as one can see both figures clearly show the ten rounds of AES. One may notice the slope at the beginning of Fig. 14.6d, which is caused by the key schedule which increases the HW of the memory. The corresponding correlation progression plots are shown in Fig. 14.6b for the measurement and in Fig. 14.6e for the simulation. Contrary to what one might expect, the right key hypothesis can be determined earlier in the measurements ≈ 40 than in the simulations ≈ 70, which is due to the fact that we tracked all registers and the entire memory, which introduces additional noise. The plots computed from all traces (first key byte) are shown in Fig. 14.6c for the measurement and in Fig. 14.6f for the simulation. The maximum correlation for the measurements is ≈ 0.9, respectively for the simulations ≈ 0.5.

14.5.4. Evaluation

The results obtained from the simulation prove that our approach to leakage simulation does indeed work as expected and exhibits exploitable leakage. However, the simulation-based results show less leakage than expected despite the completely noiseless environment during the simu-

lation. The main reason for this discrepancy between simulation and measurement is that our simulation is completely hardware independent and does not reflect the real switching behavior of an actual microcontroller. In contrast, when the leakage of a simulated hardware implementation is attacked, the results match the expected cf. Section 14.4

14.6. Summary

In this chapter we presented TOFU, a versatile tool for the synthesis of power traces from VCD files with different leakage models. In addition, we evaluated the running time of TOFU with respect to different VCD file sizes, and implementations of TOFU. Furthermore, we proposed a workflow which only relies on open-source tools to verify the security of an implementation. The open-source based workflow was successfully verified with the CPA of an AES hardware and software implementation.

15. Conclusion

Physical attacks on cryptographic implementations have become a serious threat over the last years. The reason for this recent trend can be attributed to the fact that modern cryptography is proven to be secure against cryptanalytic attack vectors, due to rigorous standardization processes. In contrast, the security assumptions of the standardization process only hold if an attacker has no further knowledge about the internals, i.e., in a black-box scenario, cf. Section 3.1. Consequently, if a cryptographic implementation is attacked we assume a gray-box where an attacker has access to additional knowledge, i.e., side-channel leakage which is acquired by either SCA or FIA.

The objective of this work was to investigate the application of state-of-the-art fault attacks, viz. DFA, PFA, AFA, and SIFA, on state-of-the-art cryptography, i.e., candidates of the CAESAR and LWC competition, to evaluate the threat posed by FIA. Usually, FIA is considered out of reach for an attacker with a low budget but in contrast as shown by Guillen et al. [GGS17] it is indeed possible to achieve rather precise fault injections even with a low budget. Furthermore, as demonstrated in Chapter 4, EMFI can be utilized to introduce precise and repeatable localized faults. Achieving suitable faulty behavior enables an attacker to mount the attacks introduced in this work cf. Part II. To defend against these attacks, it is necessary to implement cryptography in a protected manner with an appropriate countermeasure, as introduced in Chapter 13. Since FIA is only one of many threat-vectors for cryptographic implementations, it is also necessary to protect against other threat-vectors such as SCA at the same time.

Contributions This work's contributions can be categorized into three different types: Threats, Attacks, and Solutions. An overview of the different contributions is shown in Table 15.1. Furthermore, a detailed recap is given in the following.

Chapter	Type	Publication	Contribution
4	Threat	unpublished	EMFI evaluation of an ARM Cortex M0 microcontroller
7	Attack	[GS19]	Differential Fault Analysis (DFA) of KLEIN
8	Attack	[GPT19]	Persistent Fault Analysis (PFA) of COLM
8	Attack	[GPT19]	Persistent Fault Analysis (PFA) of Deoxys-II
8	Attack	[GPT19]	Persistent Fault Analysis (PFA) of OCB
9	Attack	[GPT20]	Statistical Ineffective Fault Analysis (SIFA) of Gimli
10	Attack	[GKS21]	Algebraic Fault Analysis (AFA) of SAE
13	Solution	[Gru+21]	DOMREP – Combined Countermeasure
14	Solution	[GS22]	TOFU – Toggle Count Analysis

Table 15.1.: This Work's Contributions

Chapter 4 showcased the capabilities of a commercial EMFI setup with the DFA of KLEIN based on Chapter 7. The EMFI setup was able to inject localized faults in an ARM Cortex M0 microcontroller, with high spatial precision and repeatability.

Chapter 7 introduced two DFAs of the lightweight block cipher KLEIN. Variant one targets the intermediate state of the cipher. Using at least five faulty ciphertexts, the attacker is able to determine the last round key. The second variant, which can only applied to KLEIN-64, injects a byte-fault in the key schedule and requires at least four faulty ciphertexts in order to determine the whole key. We verified the efficiency of both attack methods by means of simulation. Furthermore, we conducted the attacks practically on a microcontroller using EMFI.

Chapter 8 introduced the PFA of Deoxys-II, OCB, and COLM. We showed how to extend the original PFA to fit the needs of AE schemes and what makes them vulnerable to PFA. The targets for our attack on Deoxys-II was either the tag generation, or the message encryption. The targets for our attack on OCB was either the tag generation without associated data, or the message encryption of the last incomplete message block. The targets for our attack on Deoxys-II was either the tag generation, or the message encryption.

Chapter 9 introduced how the principles of SIFA can be applied to Gimli, an AE scheme participating the NIST-LWC competition. We identified two possible rounds during the initialization phase of Gimli to mount our attack. If we attacked the first location we are able to recover 3 bits of the key uniquely and the parity of 8 key-bits organized in 3 sums using 180 ineffective faults per biased single intermediate bit. If we attacked the second location we are able to recover 15 bits of the key uniquely and the parity of 22 key-bits organized in 7 sums using 340 ineffective faults per biased intermediate bit.

Chapter 10 introduced how the principles of AFA can be applied to the AE scheme Subterranean 2.0, a second round candidate of the NIST-LWC competition. In order to find the optimal parameters for a fault injection we investigated the fault model's influence on the solving time. The optimal fault parameters turned out as a single bit flip fault in conjunction with a known but randomly chosen fault location, where the fault is applied just one cycle before the tag generation. Conducting our proposed attack with optimal fault parameters requires only five fault injections to recover the secret key of Subterranean 2.0 in less than four seconds.

Chapter 13 introduced a novel generic solution for simultaneous protection against SCA and FIA of arbitrary order. We combined DOM and REP in an orthogonal way and call this approach DOMREP. The resistance against SCA and FIA can be scaled independently of each other, for the protection against higher-order SCA and the injection of multiple faults including SIFA. We developed the generic concept of orthogonal protection, and implemented the DOMREP concept on Gimli. Our implementation of Gimli was verified to be resistant against univariate first-order side-channel attacks by TVLA. The resistance against SIFA was verified by means of fault emulation of single as well as multiple bit faults.

Chapter 14 introduced a novel open-source tool called TOFU which synthesizes VCD simulation traces into power traces, with adjustable leakage models. Additionally, we proposed a workflow which is only based on open-source tools. The functionality of TOFU and the proposed workflow was verified by a CPA of an AES hardware and software implementation.

Bibliography

[AH20] Karim M. Abdellatif and Olivier Hériveaux. *SiliconToaster: A Cheap and Programmable EM Injector for Extracting Secrets*. Cryptology ePrint Archive, Paper 2020/1115. https://eprint.iacr.org/2020/1115. 2020. URL: https://eprint.iacr.org/2020/1115.

[AM11] Sk Subidh Ali and Debdeep Mukhopadhyay. "Differential Fault Analysis of AES-128 Key Schedule Using a Single Multi-byte Fault". In: *Smart Card Research and Advanced Applications*. Springer Berlin Heidelberg, 2011, pp. 50–64. DOI: 10.1007/978-3-642-27257-8_4.

[And+16] Elena Andreeva et al. "COLM v1". In: *Submission to the CAESAR Competition* (2016).

[AS10] Martin Albrecht and Mate Soos. *anf2cnf.py*. https://bitbucket.org/malb/research-snippets/src/master/anf2cnf.py. 2010.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. "Correlation Power Analysis with a Leakage Model". In: *Cryptographic Hardware and Embedded Systems - CHES 2004*. Ed. by Marc Joye and Jean-Jacques Quisquater. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 16–29. ISBN: 978-3-540-28632-5.

[BDL00] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Eliminating Errors in Cryptographic Computations". In: *Journal of Cryptology* 14.2 (2000), pp. 101–119. DOI: 10.1007/s001450010016.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)". In: *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*. 1997, pp. 37–51. DOI: 10.1007/3-540-69053-0_4. URL: http://dx.doi.org/10.1007/3-540-69053-0_4.

[BE+04] Hagai Bar-El et al. *The Sorcerer's Apprentice Guide to Fault Attacks*. Cryptology ePrint Archive, Report 2004/100. Version 20040507:081456. https://eprint.iacr.org/2004/100. May 2004. URL: https://eprint.iacr.org/2004/100/20040507:081456.

[Ber+12a] Guido Bertoni et al. "Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications". In: *Selected Areas in Cryptography*. Springer Berlin Heidelberg, 2012, pp. 320–337. DOI: 10.1007/978-3-642-28496-0_19.

[Ber+12b] Guido Bertoni et al. "Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications". In: *Selected Areas in Cryptography*. Ed. by Ali Miri and Serge Vaudenay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 320–337. ISBN: 978-3-642-28496-0.

[Ber+17] Daniel J Bernstein et al. "Gimli: a cross-platform permutation". In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2017, pp. 299–320.

[BGEC04] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. "Cryptanalysis of a White Box AES Implementation". In: *Selected Areas in Cryptography*. Springer Berlin Heidelberg, 2004, pp. 227–240. DOI: 10.1007/978-3-540-30564-4_16.

[Bog+] A. Bogdanov et al. "PRESENT: An Ultra-Lightweight Block Cipher". In: *Cryptographic Hardware and Embedded Systems - CHES 2007*. Springer Berlin Heidelberg, pp. 450–466. DOI: 10.1007/978-3-540-74735-2_31.

[Bre+19a] Jakub Breier et al. *A Countermeasure Against Statistical Ineffective Fault Analysis*. Cryptology ePrint Archive, Report 2019/515. Version 20190920:022400. https://eprint.iacr.org/2019/515. Sept. 2019. URL: https://eprint.iacr.org/2019/515/20190920:022400.

[Bre+19b] Jakub Breier et al. "A Countermeasure Against Statistical Ineffective Fault Analysis". In: (2019), pp. 1–5.

[Bru+20] Michaela Brunner et al. "Logic Locking Induced Fault Attacks". In: *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2020, pp. 114–119. DOI: 10.1109/ISVLSI49217.2020.00030.

[BS93] Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer New York, 1993. DOI: 10.1007/978-1-4613-9314-6.

[BS97] Eli Biham and Adi Shamir. "Differential fault analysis of secret key cryptosystems". In: *Advances in Cryptology — CRYPTO '97*. Ed. by Burton S. Kaliski. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 513–525. ISBN: 978-3-540-69528-8.

[Cae] *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*. http://competitions.cr.yp.to/caesar.html. Last accessed: 2022-09-07. 2012.

[CGD18] Yann Le Corre, Johann Großschädl, and Daniel Dinu. "Micro-architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors". In: *Constructive Side-Channel Analysis and Secure Design*. Springer International Publishing, 2018, pp. 82–98. DOI: 10.1007/978-3-319-89641-0_5.

[CH17] Ang Cui and Rick Housley. "BADFET: Defeating Modern Secure Boot Using Second-Order Pulsed Electromagnetic Fault Injection". In: *11th USENIX Workshop on Offensive Technologies (WOOT17)*. USENIX Association. 2017.

[Cho+03a] Stanley Chow et al. "A White-Box DES Implementation for DRM Applications". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 1–15. DOI: 10.1007/978-3-540-44993-5_1.

[Cho+03b] Stanley Chow et al. "White-Box Cryptography and an AES Implementation". In: *Selected Areas in Cryptography*. Springer Berlin Heidelberg, 2003, pp. 250–270. DOI: 10.1007/3-540-36492-7_17.

[Cho+18] Davin Choo et al. "BOSPHORUS: Bridging ANF and CNF Solvers". In: (Dec. 2018). arXiv: http://arxiv.org/abs/1812.04580v1 [cs.LO].

[Cho+19] Davin Choo et al. "Bosphorus: Bridging ANF and CNF Solvers". In: *Proceedings of Design, Automation, and Test in Europe(DATE)*. Mar. 2019.

[CJW10] Nicolas T Courtois, Keith Jackson, and David Ware. "Fault-algebraic attacks on inner rounds of DES". In: *E-Smart'10 Proceedings: The Future of Digital Security Technologies*. Strategies Telecom and Multimedia. 2010.

[Cla07] Christophe Clavier. "Secret external encodings do not prevent transient fault analysis". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2007, pp. 181–194.

[CP02] Nicolas T. Courtois and Josef Pieprzyk. "Cryptanalysis of Block Ciphers with Overdefined Systems of Equations". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2002, pp. 267–287. DOI: 10.1007/3-540-36178-2_17.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. "Template Attacks". In: *Cryptographic Hardware and Embedded Systems - CHES 2002*. Springer Berlin Heidelberg, 2003, pp. 13–28.

[CY03] Chien-Ning Chen and Sung-Ming Yen. "Differential Fault Analysis on AES Key Schedule and Some Countermeasures". In: *Information Security and Privacy*. Springer Berlin Heidelberg, 2003, pp. 118–129. DOI: 10.1007/3-540-45067-x_11.

[CYX15] Fan Cunyang, Wei Yuechuan, and Pan Xiaozhong. "A DIFFERENTIAL FAULT ANALYSIS METHOD FOR KLEIN CIPHER". In: *Computer Application and Software* 32 (June 2015). DOI: 10.3969/j.issn.1000-386x.2015.06.069.

[Dae+19] Joan Daemen et al. "Protecting against Statistical Ineffective Fault Attacks". In: *Cryptology ePrint Archive* Report 2019/536 (2019), pp. 1–27. URL: https://eprint.iacr.org/2019/536.

[De +19] Lauren De Meyer et al. "M&M: Masks and Macs against Physical Attacks". In: *TCHES* 2019.1 (2019), pp. 25–50. DOI: 10.13154/tches.v2019.i1.25-50.

[Deh+12] Amine Dehbaoui et al. "Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES". In: *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012*. 2012, pp. 7–15. DOI: 10.1109/FDTC.2012.15. URL: http://dx.doi.org/10.1109/FDTC.2012.15.

[DEM15] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. *Heuristic Tool for Linear Cryptanalysis with Applications to CAESAR Candidates*. Cryptology ePrint Archive, Report 2015/1200. https://eprint.iacr.org/2015/1200. 2015.

[DK10] Orr Dunkelman and Nathan Keller. *The Effects of the Omission of Last Round's MixColumns on AES*. Cryptology ePrint Archive, Report 2010/041. https://eprint.iacr.org/2010/041. 2010.

[DMR19] Joan Daemen, Pedro Maat Costa Massolino, and Yann Rotella. *The Subterranean 2.0 cipher suite*. https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates. 2019.

[DN16] Thomas De Cnudde and Svetla Nikova. "More Efficient Private Circuits II through Threshold Implementations". In: *Proceedings - 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016*. IEEE, 2016, pp. 114–124. ISBN: 9781509011087. DOI: 10.1109/FDTC.2016.15.

[Dob+16a] Christoph Dobraunig et al. "Ascon v1.2". In: *Submission to the CAESAR Competition* (2016).

[Dob+16b] Christoph Dobraunig et al. "Practical Fault Attacks on Authenticated Encryption Modes for AES." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 616.

[Dob+18a] Christoph Dobraunig et al. *Fault Attacks on Nonce-based Authenticated Encryption: Application to Keyak and Ketje*. Cryptology ePrint Archive, Report 2018/852. https://eprint.iacr.org/2018/852. 2018.

[Dob+18b] Christoph Dobraunig et al. "SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), pp. 547–572.

[DS16] François Durvaux and François-Xavier Standaert. "From Improved Leakage Detection to the Detection of Points of Interests in Leakage Traces". In: *Advances in Cryptology – EUROCRYPT 2016*. Springer Berlin Heidelberg, 2016, pp. 240–262. DOI: 10.1007/978-3-662-49890-3_10.

[Dwo07] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) for Confidentiality and Authentication*. SP 800-38D: National Institute of Standards and Technology, 2007.

[FIP01] NIST FIPS. "197: Advanced Encryption Standard (AES)". In: *Federal information processing standards publication* 197.441 (2001), p. 0311.

[Fuh+13] Thomas Fuhr et al. "Fault attacks on AES with faulty ciphertexts only". In: *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE. 2013, pp. 108–118.

[GGJR+11] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. "A testing methodology for side-channel resistance validation". In: *NIST non-invasive attack testing workshop*. Vol. 7. https://csrc.nist.gov/csrc/media/events/non-invasive-attack-testing-workshop/documents/08_goodwill.pdf. 2011, pp. 115–136.

[GGS17] Oscar M. Guillen, Michael Gruber, and Fabrizio De Santis. "Low-Cost Setup for Localized Semi-invasive Optical Fault Injection Attacks". In: *Constructive Side-Channel Analysis and Secure Design*. Springer International Publishing, 2017, pp. 207–222. DOI: 10.1007/978-3-319-64647-3_13.

[Gin] Tristan Gingold. *GHDL*. https://github.com/ghdl/ghdl. Accessed: 2021-07-11.

[GKS21] Michael Gruber, Patrick Karl, and Georg Sigl. "Algebraic Fault Analysis of Subterranean 2.0". In: *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*. 2021, pp. 45–55. DOI: 10.1109/FDTC53659.2021.00016.

[GMK16] Hannes Gross, Stefan Mangard, and Thomas Korak. *Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order*. Cryptology ePrint Archive, Report 2016/486. Version 20161115:152528. https://eprint.iacr.org/2016/486. Nov. 15, 2016. URL: https://eprint.iacr.org/2016/486/20161115:152528.

[GNL12] Zheng Gong, Svetla Nikova, and Yee Wei Law. "KLEIN: A New Family of Lightweight Block Ciphers". In: *RFID. Security and Privacy*. Springer Berlin Heidelberg, 2012, pp. 1–18. DOI: 10.1007/978-3-642-25286-0_1.

[Goo+11] Gilbert Goodwill et al. "A testing methodology for side-channel resistance validation". In: *NIST non-invasive attack testing workshop*. Vol. 7. 2011, pp. 115–136.

[GPT19] Michael Gruber, Matthias Probst, and Michael Tempelmeier. "Persistent Fault Analysis of OCB, DEOXYS and COLM". In: *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 2019, pp. 17–24. DOI: 10.1109/FDTC.2019.00011.

[GPT20] M. Gruber, M. Probst, and M. Tempelmeier. "Statistical Ineffective Fault Analysis of GIMLI". In: *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2020, pp. 252–261. DOI: 10.1109/HOST45689.2020.9300260.

[Gro+23] Mathieu Gross et al. "FPGANeedle". In: *Proceedings of the 28th Asia and South Pacific Design Automation Conference*. ACM, 2023. DOI: 10.1145/3566097.3568352.

[Gru+21] Michael Gruber et al. "DOMREP–An Orthogonal Countermeasure for Arbitrary Order Side-Channel and Fault Attack Protection". In: *IEEE Transactions on Information Forensics and Security* 16 (2021), pp. 4321–4335. DOI: 10.1109/TIFS.2021.3089875.

[GS19] Michael Gruber and Bodo Selmke. "Differential Fault Attacks on KLEIN". In: *The Urban Book Series*. Springer Singapore, 2019, pp. 80–95. DOI: 10.1007/978-3-030-16350-1_6.

[GS22] Michael Gruber and Georg Sigl. *TOFU - Toggle Count Analysis made simple*. Cryptology ePrint Archive, Report 2022/129. https://ia.cr/2022/129. 2022.

[GST12] Benedikt Gierlichs, Jörn-Marc Schmidt, and Michael Tunstall. "Infective Computation and Dummy Rounds: Fault Protection for Block Ciphers without Check-before-Output". In: *Progress in Cryptology – LATINCRYPT 2012*. Springer Berlin Heidelberg, 2012, pp. 305–321. DOI: 10.1007/978-3-642-33481-8_17.

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. "RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis". In: *Advances in Cryptology – CRYPTO 2014*. Springer Berlin Heidelberg, 2014, pp. 444–461. DOI: 10.1007/978-3-662-44371-2_25.

[HBG23] Tobias Holl, Katharina Bogad, and Michael Gruber. "Whiteboxgrind – Automated Analysis of Whitebox Cryptography". In: *Constructive Side-Channel Analysis and Secure Design*. Springer Nature Switzerland, 2023, pp. 221–240. DOI: 10.1007/978-3-031-29497-6_11.

[IEE06] IEEE. "IEEE Standard for Verilog Hardware Description Language". In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), pp. 1–590. DOI: 10.1109/IEEESTD.2006.99495.

[Ins97] Texas Instruments. "CMOS Power Consumption and Cpd Calculation". In: *SCAA035B June* (1997). https://www.ti.com/lit/an/scaa035b/scaa035b.pdf.

[Ish+06] Yuval Ishai et al. "Private Circuits II: Keeping Secrets in Tamperable Circuits". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 4004 LNCS. 2006, pp. 308–327. ISBN: 3540345469. DOI: 10.1007/11761679_19.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2729 (2003), pp. 463–481. ISSN: 03029743.

[Jea+16] Jérémy Jean et al. "Deoxys v1.41". In: *Submitted to the CAESAR Competition* (2016).

[JKP12] Philipp Jovanovic, Martin Kreuzer, and Ilia Polian. *An Algebraic Fault Attack on the LED Block Cipher*. Cryptology ePrint Archive, Report 2012/400. Version 20120723:115821. https://eprint.iacr.org/2012/400. July 2012. URL: https://eprint.iacr.org/2012/400/20120723:115821.

[JNP14] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. "Tweaks and Keys for Block Ciphers: The TWEAKEY Framework". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014, pp. 274–288. DOI: 10.1007/978-3-662-45608-8_15.

[KG21] Patrick Karl and Michael Gruber. "A Survey on the Application of Fault Analysis on Lightweight Cryptography". In: *2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. 2021, pp. 1–3. DOI: 10.1109/NTMS49979.2021.9432667.

[Kha+18] Mustafa Khairallah et al. *On Hardware Implementation of Tang-Maitra Boolean Functions*. Cryptology ePrint Archive, Report 2018/667. https://ia.cr/2018/667. 2018.

[Kim12] Chong Hee Kim. "Improved Differential Fault Analysis on AES Key Schedule". In: *IEEE Transactions on Information Forensics and Security* 7.1 (2012), pp. 41–50. DOI: 10.1109/tifs.2011.2161289.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Advances in Cryptology — CRYPTO' 99*. Springer Berlin Heidelberg, 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25.

[Koc96] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology — CRYPTO '96*. Springer Berlin Heidelberg, 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9.

[KR16] Ted Krovetz and Phillip Rogaway. "OCB v1.1". In: *Submission to the CAESAR Competition* (2016).

[KS05] François Koeune and François-Xavier Standaert. "A Tutorial on Physical Security and Side-Channel Attacks". In: *Foundations of Security Analysis and Design III*. Springer Berlin Heidelberg, 2005, pp. 78–108. DOI: 10.1007/11554578_3.

[LAN] LANGER. *ICI HH500-15 L-EFT Datasheet*. https://www.langer-emv.de/en/download/94/820/ici-hh500-15-l-eft-pulse-magnetic-field-source.pdf. Accessed: 2022-06-01.

[Led] Ledger. *LASCAR*. https://github.com/Ledger-Donjon/lascar. Accessed: 2021-07-11.

[Luo+17] Pei Luo et al. *Algebraic Fault Analysis of SHA-3*. Cryptology ePrint Archive, Report 2017/113. Version 20170214:183445. https://eprint.iacr.org/2017/113. Feb. 2017. URL: https://eprint.iacr.org/2017/113/20170214:183445.

[Man04] Stefan Mangard. "Hardware Countermeasures against DPA – A Statistical Analysis of Their Effectiveness". In: *Topics in Cryptology – CT-RSA 2004*. Springer Berlin Heidelberg, 2004, pp. 222–235.

[Mar03] George Marsaglia. "Xorshift RNGs". In: *Journal of Statistical Software* 8.14 (2003). DOI: 10.18637/jss.v008.i14.

[Mat94] Mitsuru Matsui. "Linear Cryptanalysis Method for DES Cipher ". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 765 LNCS (1994), pp. 386–397. ISSN: 16113349. DOI: 10.1007/3-540-48285-7_33.

[MBB11] Mohamed Saied Emam Mohamed, Stanislav Bulygin, and Johannes Buchmann. "Using SAT Solving to Improve Differential Fault Analysis of Trivium". In: *Communications in Computer and Information Science*. Springer Berlin Heidelberg, 2011, pp. 62–71. DOI: 10.1007/978-3-642-23141-4_7.

[McK+16] Kerry McKay et al. *Report on lightweight cryptography*. Tech. rep. National Institute of Standards and Technology, 2016.

[Mil+24] Michael Mildner et al. "Fault-Simulation-Based Flip-Flop Classification for Reverse Engineering". In: *2024 27th International Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*. 2024, pp. 53–56. DOI: 10.1109/DDECS60919.2024.10508905.

[MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. "Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 199–216. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/mccann.

[Muk09] Debdeep Mukhopadhyay. "An Improved Fault Based Attack of the Advanced Encryption Standard". In: *Progress in Cryptology – AFRICACRYPT 2009*. Springer Berlin Heidelberg, 2009, pp. 421–434. DOI: 10.1007/978-3-642-02384-2_26.

[MW78] Timothy C. May and Murray H. Woods. "A New Physical Mechanism for Soft Errors in Dynamic Memories". In: *16th International Reliability Physics Symposium*. IEEE, 1978. DOI: 10.1109/irps.1978.362815.

[Nev+03] Michael Neve et al. "Memories: A Survey of Their Secure Uses in Smart Cards". In: *2nd International IEEE Security in Storage Workshop (SISW 2003), Information Assurance, The Storage Security Perspective, 31 October 2003, Washington, DC, USA*. 2003, pp. 62–72. DOI: 10.1109/SISW.2003.10004. URL: http://dx.doi.org/10.1109/SISW.2003.10004.

[NIS17] NIST. *NIST-LWC: NIST Lightweight Cryptography (LWC) standardization process*. https://csrc.nist.gov/Projects/lightweight-cryptography. Last accessed: 2022-09-07. 2017.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. "Threshold Implementations Against Side-Channel Attacks and Glitches". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4307 LNCS (2006), pp. 529–545. ISSN: 16113349.

[OC14] Colin O'Flynn and Zhizhang (David) Chen. "ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research". In: *IACR Cryptology ePrint Archive* 2014 (2014), p. 204. URL: https://eprint.iacr.org/2014/204.

[O'F19] Colin O'Flynn. "MIN()imum Failure: EMFI Attacks against USB Stacks". In: *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA: USENIX Association, Aug. 2019. URL: https://www.usenix.org/conference/woot19/presentation/oflynn.

[OT17] Johannes Obermaier and Stefan Tatschner. "Shedding too much Light on a Microcontroller's Firmware Protection". In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, 2017. URL: https://www.usenix.org/conference/woot17/workshop-program/presentation/obermaier.

[PCM15] Sikhar Patranabis, Abhishek Chakraborty, and Debdeep Mukhopadhyay. "Fault Tolerant Infective Countermeasure for AES". In: *Security, Privacy, and Applied Cryptography Engineering*. Springer International Publishing, 2015, pp. 190–209. DOI: 10.1007/978-3-319-24126-5_12.

[Pea00] Karl Pearson. "X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50.302 (1900), pp. 157–175. DOI: 10.1080/14786440009463897.

[Pos+11] Axel Poschmann et al. "Side-Channel Resistant Crypto for Less than 2,300 GE". In: *J. Cryptology* 24 (2011), pp. 322–345. DOI: 10.1007/s00145-010-9086-6.

[PQ03] Gilles Piret and Jean-Jacques Quisquater. "A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 77–88. DOI: 10.1007/978-3-540-45238-6_7.

[Pro+24a] Matthias Probst et al. " Switch-Glitch : Location of Fault Injection Sweet Spots by Electro-Magnetic Emanation ". In: *2024 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*. Los Alamitos, CA, USA: IEEE Computer Society, Sept. 2024, pp. 22–27. DOI: 10.1109/FDTC64268.2024.00011. URL: https://doi.ieeecomputersociety.org/10.1109/FDTC64268.2024.00011.

[Pro+24b] Matthias Probst et al. "DOMREP II". In: *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2024, pp. 112–121. DOI: 10.1109/HOST55342.2024.10545417.

[Que14] Frank Quedenfeld. *Algebraic Fault Analysis of Katan*. Cryptology ePrint Archive, Paper 2014/954. https://eprint.iacr.org/2014/954. 2014. URL: https://eprint.iacr.org/2014/954.

[RAD19] Keyvan Ramezanpour, Paul Ampadu, and William Diehl. "A Statistical Fault Analysis Methodology for the Ascon Authenticated Cipher". In: *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2019. DOI: 10.1109/hst.2019.8741029.

[RAD20] K. Ramezanpour, P. Ampadu, and W. Diehl. "RS-Mask: Random Space Masking as an Integrated Countermeasure against Power and Fault Analysis". In: *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2020, pp. 176–187. DOI: 10.1109/HOST45689.2020.9300266.

[Rep+18] Oscar Reparaz et al. "CAPA: The Spirit of Beaver against Physical Attacks". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10991 LNCS (2018), pp. 121–151. ISSN: 16113349. DOI: 10.1007/978-3-319-96884-1_5.

[RGP22] Jonas Ruchti, Michael Gruber, and Michael Pehl. "When the Decoder Has to Look Twice: Glitching a PUF Error Correction". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.3 (2022), 26–70. DOI: 10.46586/tches.v2022.i3.26-70. URL: https://tches.iacr.org/index.php/TCHES/article/view/9694.

[Riv09] Matthieu Rivain. "Differential Fault Analysis on DES Middle Rounds". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 457–469. DOI: 10.1007/978-3-642-04138-9_32.

[SA03] Sergei P. Skorobogatov and Ross J. Anderson. "Optical Fault Induction Attacks". In: *Cryptographic Hardware and Embedded Systems - CHES 2002*. Springer Berlin Heidelberg, 2003, pp. 2–12. DOI: 10.1007/3-540-36400-5_2.

[Sad+19] Rajat Sadhukhan et al. "Count Your Toggles: a New Leakage Model for Pre-Silicon Power Analysis of Crypto Designs". In: *Journal of Electronic Testing* 35.5 (2019), pp. 605–619. DOI: 10.1007/s10836-019-05826-8.

[Sah+20] Sayandeep Saha et al. "A Framework to Counter Statistical Ineffective Fault Analysis of Block Ciphers Using Domain Transformation and Error Correction". In: *IEEE Transactions on Information Forensics and Security* 15 (2020), pp. 1905–1919. DOI: 10.1109/tifs.2019.2952262.

[SH07] Jörn-Marc Schmidt and Michael Hutter. "Optical and EM Fault-Attacks on CRT-based RSA: Concrete Results". In: *Austrian Workshop on Microelectronics – Austrochip 2007, Graz, Austria, October 11*. Ed. by Karl Christian Posch and Johannes Wolkerstorfer. ISBN 978-3-902465-87-0. Verlag der Technischen Universität Graz, 2007, pp. 61–67.

[SM15] Tobias Schneider and Amir Moradi. "Leakage Assessment Methodology". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2015, pp. 495–513. DOI: 10.1007/978-3-662-48324-4_25.

[SMG16] Tobias Schneider, Amir Moradi, and Tim Güneysu. "ParTI - Towards Combined Hardware Countermeasures against Side-Channel and Fault-Injection Attacks". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9815 (2016), pp. 302–332. ISSN: 16113349. DOI: 10.1007/978-3-662-53008-5_11.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. "Extending SAT Solvers to Cryptographic Problems". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 244–257. DOI: 10.1007/978-3-642-02777-2_24.

[TL22] Honghui Tang and Qiang Liu. "MPFA: An Efficient Multiple Faults-Based Persistent Fault Analysis Method for Low-Cost FIA". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.9 (2022), pp. 2821–2834. DOI: 10.1109/tcad.2021.3117512.

[TMA11] 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*. Springer Berlin Heidelberg, 2011, pp. 224–233. DOI: 10.1007/978-3-642-21040-2_15.

[Tou+20] J. Toulemont et al. *A Simple Protocol to Compare EMFI Platforms*. Cryptology ePrint Archive, Paper 2020/1277. https://eprint.iacr.org/2020/1277. 2020. URL: https://eprint.iacr.org/2020/1277.

[Uni] Unicorn. *Unicorn Engine*. https://github.com/unicorn-engine/unicorn. Accessed: 2023-03-01.

[Vaf+22] Navid Vafaei et al. "Statistical Effective Fault Attacks: The Other Side of the Coin". In: *IEEE Transactions on Information Forensics and Security* 17 (2022), pp. 1855–1867. DOI: 10.1109/tifs.2022.3172634.

[VG17] Nikita Veshchikov and Sylvain Guilley. "Use of Simulators for Side-Channel Analysis". In: *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2017. DOI: 10.1109/eurospw.2017.59.

[Wam] Markus Stefan Wamser. *VCD2R*. https://github.com/wamserma/VCD2R. Accessed: 2022-02-13.

[Wam19] Markus Stefan Wamser. "Serialisation of Inversion-Based S-Boxes". Dissertation. München: Technische Universität München, 2019.

[WRZ16] Y.-J Wang, Q.-Y Ren, and S.-Y Zhang. "Differential Fault Attack on lightweight Block Cipher Klein". In: *Tongxin Xuebao/Journal on Communications* 37 (Oct. 2016), pp. 111–115. DOI: 10.11959/j.issn.1000-436x.2016256.

[Wys+07] Brecht Wyseur et al. "Cryptanalysis of White-Box DES Implementations with Arbitrary External Encodings". In: *Selected Areas in Cryptography*. Springer Berlin Heidelberg, 2007, pp. 264–277. DOI: 10.1007/978-3-540-77360-3_17.

[Xu+21] Guorui Xu et al. "Pushing the Limit of PFA: Enhanced Persistent Fault Analysis on Block Ciphers". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.6 (2021), pp. 1102–1116. DOI: 10.1109/tcad.2020.3048280.

[Yos+] Hideki Yoshikawa et al. "Round Addition DFA on Lightweight Block Ciphers with On-The-Fly Key Schedule". In: -2006.9 (), p. 1. ISSN: eISSN:1307-6892. URL: http://waset.org/abstracts/.

[Zha+12] Xinjie Zhao et al. *Algebraic Differential Fault Attacks on LED using a Single Fault Injection*. Cryptology ePrint Archive, Report 2012/347. Version 20120622:195638. https://eprint.iacr.org/2012/347. June 2012. URL: https://eprint.iacr.org/2012/347/20120622:195638.

[Zha+13a] Fan Zhang et al. "Improved Algebraic Fault Analysis: A Case Study on Piccolo and Applications to Other Lightweight Block Ciphers". In: *Constructive Side-Channel Analysis and Secure Design*. Springer Berlin Heidelberg, 2013, pp. 62–79. DOI: 10.1007/978-3-642-40026-1_5.

[Zha+13b] Xinjie Zhao et al. "Improving and Evaluating Differential Fault Analysis on LED with Algebraic Techniques". In: *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2013. DOI: 10.1109/fdtc.2013.14.

[Zha+14] Xinjie Zhao et al. "Algebraic Fault Analysis on GOST for Key Recovery and Reverse Engineering". In: *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2014. DOI: 10.1109/fdtc.2014.13.

[Zha+16a] Fan Zhang et al. "A Framework for the Analysis and Evaluation of Algebraic Fault Attacks on Lightweight Block Ciphers". In: *IEEE Transactions on Information Forensics and Security* 11.5 (2016), pp. 1039–1054. DOI: 10.1109/tifs.2016.2516905.

[Zha+16b] Jinbao Zhang et al. "Against fault attacks based on random infection mechanism". In: *IEICE Electronics Express* 13.21 (2016), pp. 20160872–20160872.

[Zha+18] Fan Zhang et al. "Persistent Fault Analysis on Block Ciphers". en. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* Volume 2018 (2018), Issue 3–. DOI: 10.13154/tches.v2018.i3.150-172.