Automatic Methods for Protection of Cryptographic Hardware against Fault Attacks

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Maël Gay

aus Moulins, Frankreich

Hauptberichter: Prof. Dr. rer. nat. habil. Ilia Polian
Mitberichter: Prof. Dr. rer. nat. habil. Bernd Becker

Tag der mündlichen Prüfung: 03. Mai 2022.

Institut für Technische Informatik der Universität Stuttgart

2022

Contents

Acknowledgments

I would first like to thank Prof. Ilia Polian for his expertise and guidance throughout the years, as well as offering me the opportunity to work on interesting topics. I am also grateful for his help in settling in, both in Passau and in Stuttgart, as well as in Germany overall. Furthermore, I want to thank Prof. Bernd Becker from the University of Freiburg for agreeing to be my second adviser.

I would then like to thank my colleagues at the University of Stuttgart, and firstly Florian Neugebauer, whom has not only been a great colleague, but more importantly a great friend, from when he joined our group in Passau. I am grateful to have been able to work with all my colleagues at the Institut für Technische Informatik, and share good times with all of them. A kind thank you to Devanshi Upadhyaya and Sebastian Brandhofer, but also Nourhan Elhamawy, Roshwin Sengupta and Li-Wei Chen, as well as all other members of the institute.

In addition, I am grateful to Prof. Osnat Keren from the Bar-Ilan University, and Prof. Martin Kreuzer from the University of Passau for the collaboration we had together, and in the same way to Jan Burchard and Tobias Paxian from the University of Freiburg.

My thanks to Mirjam Breitling, Lothar Hellmeier and Helmut Häfner for their assistance on administrative and technical matters.

In a more personal way, I would like to thank two of my dear friends for their support, especially in recent difficult times. Thank you Elena Heldwein and Adrien Fuchs.

Finally, I would like to thank my parents for their eternal support during my thesis, but also throughout my entire life. I would not be where I am today without them.

Stuttgart, 03 May 2022

Maël Gay

Abstract

Since several years, the number of electronic devices in use has been strongly rising, especially in the field of embedded systems. From automotive applications or smartphones, to smaller area and power restricted embedded systems, such as Internet of Things (IoT) devices or smart cards, the wide availability of these systems induces a need for data protection. The implementation of hardware cryptographic primitives on Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA) aims to fulfil the security requirements, while providing faster and lower power encryption than software based solutions on microprocessors, especially in the case of constrained resources. However, cryptographic solutions can be attacked, even if the encryption scheme is proven secure. One possible way to do so is through physical attacks, such as Side-Channel Analysis (SCA), for example by analysing their power consumption, or fault injection attacks, which disturb the computation in a way that allows an attacker to recover the secret key. As such, it is of the utmost relevance to implement cryptographic algorithms in a way that minimises the risk of physical attacks, as well as implement some counter-measures to prevent them, for instance Error Correcting Codes (ECC). Moreover, the evaluation of aforementioned cryptographic hardware and counter-measures is not generally done automatically, but rather empirically. This results in a need for the automation of both counter-measures generation and physical hardware checking against attacks.

This thesis will focus on the automation of both aspects. Firstly, Error Detecting Code (EDC), as well as ECC, counter-measures are presented. Their goal is to stop faults from disturbing the encryption process. A discussion on the differences between natural (i.e induced by natural factors such as ageing or cosmic rays) and malicious faults is given in a subsequent chapter, as well as an analysis of the limitations of the evaluation of ECC. This is followed by the presentation of new architectures based on a new class of robust EDC, aimed at preventing multiple faults. They are scalable by construction, and as such it is possible to automatically choose an appropriate EDC implementation with regards to the constraint of the protected hardware. The architectures ensure the detection of faults

injected by a strong adversary (who has the ability to inject precise faults on a temporal and spatial level), as well as the correction of low-multiplicity faults. The structure of the implementation, an inner-outer code based construction, and more specifically an efficient decoding method are further detailed, as well as some additional tweaks. Finally, the implementation is validated against physical fault injection on a SAKURA-G FPGA platform, and the results further reinforce the need for such architectures.

The second part of the thesis will consider attack scenarios, and more precisely fault attacks. The automatic evaluation of hardware implementations of cryptographic primitives will be the main focus. In this regard, this thesis considers a particular type of fault attacks, hardware based Algebraic Fault Attacks (AFA). AFAs are at the border between mathematical cryptanalysis and physical fault injection attacks. They combine information from fault disturbed encryptions with some cipher description, in order to build an attack and recover the secret key. This work considers the hardware implementations of different ciphers as the source of algebraic information. In such regards, a framework for automated creation of AFAs has been developed in collaboration with the chair of computer architecture of the University of Freiburg. The framework takes the description of the cipher, in Hardware Description Language (HDL) or gate level, as well as a defined fault model as inputs, and through a series of steps, builds an attack in order to recovers the secret key. The detailed steps are presented in this thesis. The automatic generation of attack scenario for a considered cipher allows for an evaluation of any cipher implementation, including any potential changes or optimisation made against different attack scenarios. The framework itself was tested on a variety of different Substitution and Permutation Network (SPN), and some counter-measures. Physical realisation of fault attacks are also considered, from an implementation of the SAKURA-G FPGA platform, as well as software simulations of an idealised fault model. The constructed attacks were successful and the results are discussed, as well as the implication of multiple fault injections for solving. Finally, some counter-measures are considered, in order to validate or invalidate their effectiveness against AFAs.

Zusammenfassung

Seit Jahren ist die Zahl der elektronischen Geräte stark angestiegen, insbesondere im Bereich der Eingebetteten Systeme, von Automobilanwendungen und Smartphones, bis zu kleineren Systemen mit eingeschränkter Leistung, wie zum Beispiel Geräten im Internet of Things (IoT) oder Smartkarten. Die breite Verfügbarkeit dieser Systemen bedeutet, dass es eine Notwendigkeit für Datenschtutz gibt. Das Ziel der Implementierung kryptographischer Algorithmen auf Application Specific Integrated Circuit (ASIC) oder Field Programmable Gate Array (FPGA) ist die Erfüllung der Sicherheitsanforderungen, und zur gleichen Zeit eine schnellere und effizientere Verschlüsselung als Softwareimplementierungen bereitzustellen, insbesondere im Fall eingeschränkter Ressourcen. Trotzdem können kryptografische Implementierungen angegriffen werden, auch wenn die Verschlüsselung als sicher erwiesen ist. Eine Möglichkeit, dieses Ziel zu erreichen, sind physische Angriffe wie Side-Channel Analysis (SCA). Ein Angreifer kann beispielsweise den Stromverbrauch analysieren, oder einen Fehler injezieren, der die Berechnung auf eine Weise stört, die es dem Angreifer erlaubt, den geheimen Schlüssel zurückzuberechnen. Daher ist es von größter Bedeutung, den kryptografischen Algorithmus so zu implementieren, dass das Risiko physischer Angriffe minimiert wird, sowie einige Gegenmaßnahmen, wie zum Beispiel Error Correcting Codes (ECC), zu implementieren. Außerdem ist die Bewertung kryptographischer Hardware und Gegenmaßnahmen generell nicht automatisiert, sondern wird empirisch durchgeführt. Daher gibt es eine Notwendigkeit, die Generierung von Gegenmaßnahmen sowie die Überprüfung der Widerstandsfähigkeit der Hardware gegenüber Angriffen zu automatisieren.

Diese beiden Aspekte sind die Hauptgegenstände dieser Arbeit. Zuerst werden Error Detecting Code (EDC) und ECC vorgestellt, die die Gegenmaßnahmen zu Fehlerangriffen darstellen. Ihr Ziel ist es, die Störung der Verschlüsselung durch Fehlerinjektion zu verhindern. Im Anschluss folgt eine Diskussion der Unterschiede zwischen natürlichen (d.h. verursacht durch Alterung oder kosmischer Strahlung) und bösartigen Fehlern, sowie eine Analyse der Grenzen der Bewertung von ECC. Danach werden neue EDC Architekturen

basierend auf einer neuen Klasse robuster Codes, die Mehrfachfehler korigieren können, vorgestellt. Ihre Konstruktionen sind skalierbar, was es ermöglicht, automatisch eine die Hardwareeinschränkungen erfüllende EDC Implementierungen zu wählen. Die Architekturen stellen die Erkennung von Fehlern, die von einem starker Angreifer (der präzise zeitliche und räumliche Fehler einfügen kann) injeziert werden, sowie die Korrektur von Fehlern geringer Multiplizität, sicher. Die Struktur der Implementierung, basierend auf einem inner-outer Code Aufbau, insbesondere einer effizienten Dekodierung sowie einiger zusätzlicher Optimierungen, wird weiter detailliert. Schließlich wird die Implementierung gegenüber physischen Fehlerinjektionen auf einer SAKURA-G FPGA-Plattform validiert.
Der zweite Teil der Arbeit behandelt Angriffsszenarien, im Speziellen Fehlerangriffe. Der Schwerpunkt liegt auf der automatischen Bewertung von Hardwareimplementierungen kryptographischer Algorithmen. Zu diesem Zweck wird eine besondere Art von Fehlerangriffen, die hardwarebasierten Algebraic Fault Attacks (AFA), berücksichtigt. AFAs befinden sich an der Grenze zwischen mathematischer Kryptoanalyse und physischen Fehlerangriffen. Sie kombinieren Informationen aus fehlerhaften Verschlüsselungen mit einer Chiffrebeschreibung, um einen Angriff zu erstellen und den geheimen Schlüssel wiederherzustellen. Die Quellen algebraischer Informationen in diese Arbeit sind die Hardwareimplementierungen verschiedener Chiffren. Ein Framework für automatische Erstellung von AFAs wurde in Zusammenarbeit mit dem Lehrstuhl für Rechnerarchitektur der Universität Freiburg entwickelt. Das Framework erhält die Beschreibung der Chiffre in Hardware Description Language (HDL) oder als Beschreibung auf Gatterebene, sowie ein Fehlermodell als Input, und konstruiert daraus den Angriff. Die detaillierten Schritte dazu werden in dieser Arbeit vorgestellt. Die automatische Generierung eines Angriffsszenarios für ein Verschlüsselungverfahren ermöglicht eine Bewertung jeder individuellen Implementierung dieses Verfahrens, inklusive möglicher Änderungen oder Optimierungen, die gegen verschiedene Angriffsszenarien vorgenommen werden können. Das Framework wurde mit verschiedenen Substitution and Permutation Network (SPN) Chiffren, sowie mehreren Gegenmaßnahmen, geprüft. Physische Fehlerangriffe auf Implementierungen auf der SAKURA-G FPGA-Plattform, sowie Softwaresimulationen von idealen Fehlermodellen werden ebenfalls betrachtet. Die durchgeführten Angriffe waren erfolgreich und ihre Ergebnisse, sowie die Auswirkungen mehrerer Fehlerinjektionen, werden diskutiert. Schließlich werden noch einige Gegenmaßnahmen betrachtet, um ihre Wirksamkeit gegen AFAs zu untersuchen.

List of Figures

List of Tables

List of Abbreviations

AES	Advanced Encryption Standard
APN	Almost Perfect Non-linear (function)
ASIC	Application Specific Integrated Circuit
AFA	Algebraic Fault Attack
AMD	Algebraic Manipulation Detection (code)
ANF	Algebraic Normal Form
BCH	Bose–Chaudhuri–Hocquenghem (code)
CLB	Combinational Logic Blocks
CMS	CryptoMiniSat
CNF	Conjunctive Normal Form
CPC	Compact Protection Code
CPS	Cyberphysical System
CRT	Chinese Remainder Theorem
DCM	Digital Clock Manager
DFA	Differential Fault Analysis
DOM	Domain-Oriented Masking
DPA	Differential Power Analysis
DRAM	Dynamic Random Access Memory
ECC	Error Correcting Code
ECLT	Error Coefficient and Location Table
EDC	Error Detecting Code
EM	electromagnetic
FSA	Fault-Sensitivity Analysis

FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IFA	Ineffective Fault Analysis
IoT	Internet of Things
LED	Light Encryption Device
LUT	Lookup Table
MLBT	MapleLCMDistChronoBT
NFC	Near Field Communication
PI	Prime Input
PO	Prime Output
QS	Quadratic-Sum
RK	Rabii-Keren (code)
RNG	Random Number Generator
RSA	Rivest–Shamir–Adleman
SAT	Boolean Satisfiability
SCA	Side-Channel Analysis
SIFA	Statistical Ineffective Fault Attacks
SBox	Substitution Box
SPN	Substitution and Permutation Network
SRAM	Static Random Access Memory
SSAES	Small Scale AES
TMR	Triple Modular Redundancy

Chapter 1

Introduction

1.1 Motivation

In recent years, the importance of ensuring the security of sensitive data is undeniable, especially with the transition to the cyberphysical system (CPS) paradigm. A large number of digital circuits are now used for some applications where security, as well as safety, is crucial. For instance, smartphones are used for secure payments via a near field communication (NFC) chip or geolocation, while at the same time being connected to diverse social media and other applications. Another example which is becoming more and more important every year, is the car industry. Most cars nowadays have a wide array of safety features which are electronically controlled. From the anti-lock braking system to the electronic stability control, or even more critical drive-assist functionalities, they all aim at preventing deadly accidents. They are the potential targets for malicious attacks, even more so today, when car can be remotely accessed for updates or by social media features. It should therefore be clear that circuit designers for a large number of applications have to wisely consider how to guarantee data integrity and prevent any data leakages. However, guaranteeing privacy and integrity is a challenge in itself, especially for constrained systems, such as embedded devices or Internet of Things (IoT) applications.

In order to answer the data protection problem, cryptographic primitives are implemented. While they provide, at least up to a certain degree, data security, they still have to fit within the device constrains, for instance in terms of area or power [BKL+07, BBI+15]. In this regard, a balance has to be found between the security level and the system require-

ments. This trade-off can lead to vulnerabilities in the design, and those vulnerabilities can then be exploited by an attacker, which is especially true if the considered device can be physically accessed. There are a large variety of physical attacks which can be employed to recover the secret information being processed. One vector of attack is to use side-channel information from the device, such as power consumption or electro-magnetic radiations. The most common type of such attacks is passive side-channel analysis [MOP07], which includes timing attacks [Koc96], or differential power analysis (DPA) [KJJ99]. Another way to attack a device is hardware manipulations, such as fault injections attacks [BBKN12, RP17]. Of particular interest in the context of this thesis are AFAs [ZGZ+16]. Algebraic Fault Attacks (AFAs) are a class of physical attacks at the crossroad between mathematical analysis and conventional fault attacks. They utilise the algebraic description of the considered cipher, from which some equations are derived and used during solving, as well as some information from fault injection (i.e. the fault affected values and the fault model), in an attempt to recover the secret key. AFA frameworks can be divided in two categories. The frameworks which automatically analyse the fault propagation and the remaining size of the key space, such as the frameworks from [KRH17, SKMD17], and fully functional solvers which directly recover the secret key [ZZG+13, ZGZ+13, ZGZ+16]. For instance, the framework proposed in [ZGZ+16] considers some lightweight ciphers, mainly LBlock [WZ11] and PRESENT [BKL+07], and derives the algebraic equations directly from the functional description of the aforementioned ciphers. The framework then processes a given fault model and some fault affected values in combination of the derived equations, and forwards them to a Boolean Satisfiability (SAT) solver which recovers the secret key. However, none of the frameworks consider the hardware implementations themselves, with the exception of the framework presented in this thesis.

On the one hand, since multiple physical attacks exist, there are also counter-measures to prevent an attacker from recovering any sensitive information. Such counter-measures range from physical shielding at the chip level (discussed in [BECN+06]), in order to for instance prevent laser fault injections, to masking schemes [GMK16], aimed at prohibiting side-channel analysis, or Error Detecting Codes (EDCs) and Error Correcting Code (ECCs) [BBK+03, KKT04, KT04, WK11], to protect against fault attacks, and which are one of the focus of this thesis. In that regard, it is important to note that faults can be caused not

only by a malicious attacker, but also by natural failures (e.g., cosmic radiation or electromagnetic disturbances). The former can be much more difficult to prevent or even detect, since the attacker can be more precise than any natural causes, and carefully choose the location of the fault, as well as build a set of equations to retrieve the secret key. As such, a wide variety of defences have been suggested against both natural faults and fault injection attacks [BRC60, KKW07, BBKN12, SMG16]. In the case of malicious fault injection attacks, their goal can be to skip part of the execution, for instance jump over a password check [vWWM11], or more directly affect some intermediate values in order to change the output in a way that allows for the creation of specific equations and ultimately the key recovery, as is the case for Differential Fault Analysis (DFA) [BDL01]. In this thesis, the focus is however on faults that manifest themselves at the output, especially DFAs or AFAs (discussed in more details in Chapter 4). As such, fault attacks such as Fault-Sensitivity Analysis (FSA) [LOS12] or Statistical Ineffective Fault Attacks (SIFA) [DEK+18], while also relevant and potentially counter-acted by EDCs, are not considered in details in this work. As will be discussed in Chapter 3, EDCs are one of the counter-measures which can be deployed against fault attacks. Simple redundancy-based techniques are vulnerable to more advanced attacks (e.g. simultaneous fault injections [SHS16]). Consequently, special security oriented codes [WK11, KW14, TNK+14] have been developed. Moreover, conventional linear codes, like parity codes or Hamming codes, offer a limited protection against an attacker with the ability to inject specific faults. For instance, such an attacker could inject a fault which is the XOR difference between two codewords, and the results would therefore still be a codeword and hence go undetected. This shows the need for more powerful non-linear security-oriented codes. One class of such codes are called robust codes. They have the property that all non-zero faults are always detected with a probability greater than zero. This property is especially important to prevent a subset of fault to always go undetected, which would be catastrophic in the case of a malicious attacker.

During the design phase of a sensitive cryptographic device, counter-measure implementations, as well as reductions of the overall potential vulnerabilities of a circuit have to be considered. However, electronic design automation tools are not security aware, and while they can optimise the design in terms of area or power consumption in order to meet

the device constraints, they completely overlook potential physical vulnerabilities. The responsibility of evaluating hardware implementations against threats, such as fault attacks, therefore falls onto the circuit designers themselves, and is often done in an empirical manner. This task is therefore very tedious and requires a lot of time, as well as expertise. In order to improve the design phase of sensitive devices, automated tools need to be created for their evaluation in a security context. Such tools need to be able to mount a wide variety of attacks with particular sets of parameters in order to match diverse attack models, based on the capabilities of an attacker. In a second step, once some vulnerabilities have been highlighted, counter-measures also need to be generated automatically, once again dependent on the considered encryption scheme implementation. The choice between different counter-measure is also dependent on the device constraints, and also not leak any information itself. Therefore, having access to a scalable family of effective counter-measures is a requirement too. While such counter-measures exist theoretically, few automatic generation methods for hardware realisations exist to cover a large number of ciphers. In order to solve the lack of automation with respect to security, this thesis aims at both proposing an automated framework for the evaluation of cryptographic hardware, as well as multiple scalable security-oriented EDC architectures as counter-measures.

1.2 Research Contribution

The more specific focus of the work presented in this thesis is on fault attacks. Both the automation of the evaluation of cryptographic hardware implementations, via the automatic creation of AFAs, and the automatic generation of counter-measures, through the use of diverse security-oriented EDC architectures, is proposed with regards to malicious fault injections, and is summarised by Figure 1.1. In particular, new architectures aimed at protecting cryptographic circuits against natural and malicious faults will be presented in detail in Chapter 3. Each architecture can be applied to different ciphers by changing the parameters to fit the hardware and design constraints, and more precisely the size of the states being protected. The choice between one EDC implementations or another can therefore be done automatically, given proper information on the requirements. In a similar way, Chapter 4 relates to the work done on the automatic construction of AFAs on cryp-

tographic primitive implementations. The hardware-oriented AFA framework `AutoFault` is presented in this chapter. The framework was created in order to address the multiple differences which exist between different ciphers and their related fault attacks, and provide an automatic generation of solvable attacks for different ciphers. Such a framework is especially useful for the automatic evaluation of hardware implementations of encryption schemes at multiple stages during the design flow, as, if an attack is successfully mounted, the design can be changed in order to circumvent the newly discovered attack.

Contributions on Error Detecting Code Architectures

Originally, EDCs were used to detect and handle failures due to natural causes to occur, as they are efficient to detect disturbances that may happen while the circuit is operating. In a context of reliability and safety, they are therefore very effective, for example, to counteract the effect of ageing or failure due to radiations. Consequently, it is only logical to use such codes in a context of security, to protect sensitive data from an ill-intentioned third party. The use of EDC and ECC architectures for both detection, and when possible, correction of natural and malicious faults was therefore proposed [KGKP18], as well as the dedicated security-oriented non-linear code based architectures for this purpose (Section 3.1).

Evaluation of Security-Oriented Codes

The distinction between natural and malicious faults leads to a discussion on the limitation of the evaluation methods for EDC implementations [GKKP18] (Section 3.2). Mainly, evaluation via a random set of fault injection does not show well the security properties of more advanced EDC architectures. Simpler and less costly linear code implementations, such as Bose–Chaudhuri–Hocquenghem (BCH) codes [BRC60], appear to have essentially the same capabilities as security-oriented codes, when evaluated with randomly distributed fault injection. However, a strong attacker may inject a particular fault that goes undetected with conventional EDC implementation, while a security-oriented code would detect such an error with a non zero probability. Therefore, and while simpler implementations may seem more attractive to circuit designers, security-oriented architectures should be preferred in the context of cryptographic circuits, and as such, the

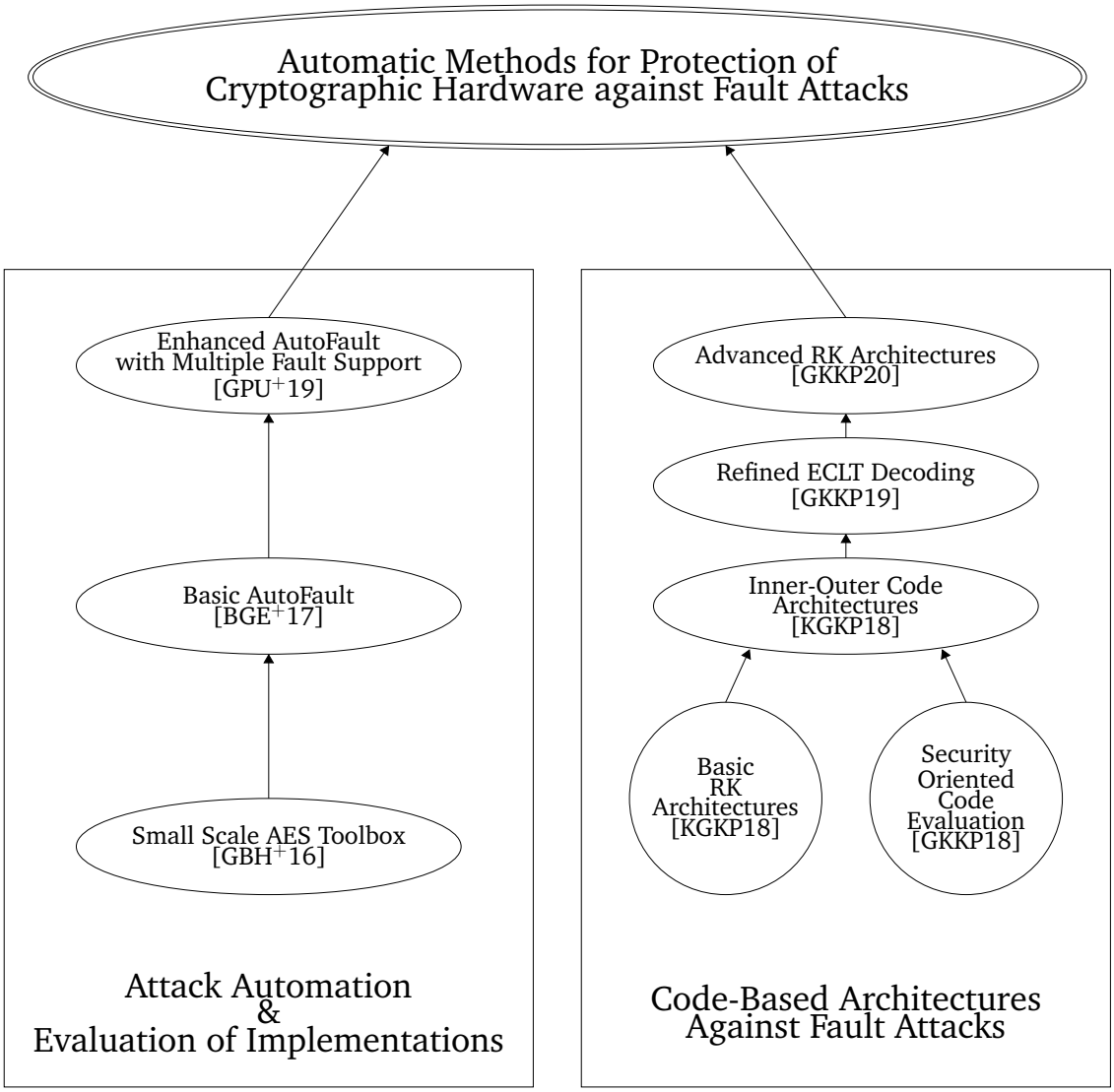

Figure 1.1: Overall Contributions and Work Goals

methodologies for the evaluation of EDC implementations should not only consider random fault injections, but mathematical analysis of the worst case scenario as well.

Initial Rabii-Keren Architectures

The previously mentioned proposed architectures utilise the Rabii-Keren (RK) codes first introduced in [RK17]. The RK codes are a new class of robust codes, combining a large distance, a low masking probability, and a high rate (ratio between data bits and check bits). Their robustness property is especially important for the architectures, as attackers have recently become better at injecting faults, and their capabilities may still improve in the years to come. Moreover, The RK codes had never been implemented in hardware prior to the work presented in this thesis. The base architecture (Section 3.3.1) has a focus on low cost ciphers, such as LED [GPPR11], with only states organized in 4-bit nibbles, but an analogue 8-bit version is also possible, for instance to protect the 8-bit substitution boxes (SBoxes) of the Advanced Encryption Standard (AES) [NIoSTN01]. For this purpose, RK codes are also ideal, as they are defined over symbols, not bits, and the size of the symbols can be chosen to match the size of the operations of the considered cipher (e.g. 8-bit symbols). However, it is costly and more difficult to protect larger states, hence another architecture aimed at byte-wise operations was needed and developed. Mainly, the former architecture uses a single predictor and decoder setup, for either 4-bit or 8-bit values, when, in comparison, the latter makes use of two predictors and decoders over 4-bit values only. While this approach may seem counter-intuitive to reduce the costs, and a slightly large area, the simpler decoding process makes up for the use of the dual decoder implementation, and the memory requirement is lower. This is the first milestone for the efficient implementation of the RK codes. Moreover, the architectures are not only capable of detecting errors, but also correcting low-multiplicity ones, which are usually the type of faults a skilful attacker tries to inject. The ability to correct such errors is an advantage compared to EDC only implementation, in the case of both natural faults, but also targeted attacks. The correction allows for the system to continue operating normally, while, if it was only detected, it needs to be handled differently, for example by stopping the execution and proceeding to a re-keying. Stopping any executions might however not be possible or cause more harm than good in critical applications, such as aeronautic

applications. In this regard, the proposed correction architectures are of particular interest for safety-critical applications, where one cannot simply wait for a reset. Despite the correction, the fault events should however still be recorded and monitored, for instance to classify between malicious and natural faults.

Inner-Outer Code Architecture

Similarly, even if the fault was corrected, a miscorrection can still occur, either normally, or due to a well handcrafted attack. To this end, the original base architectures are improved with a second layer of code, first using a Quadratic-Sum (QS) code [KKW07] (Section 3.3.2). The new inner-outer code architecture [KGKP18] allows for a further validation check on the correctness of the correction, or the non-detection of the fault. This is however only a detection step and no further correction is performed. Nonetheless, the advantages of the extra layer of code in terms of worst case scenario can't be overlooked, and which, as previously stated, are particularly important if an attacker manages to build an attack and inject a fault that would generate another valid code word. In such a case, and thanks to the proposed architecture, it would however be detected by the outer code. More specifically, such an inner-outer code based architecture combines the advantages of a robust inner code, for correction and high detection rate of faults (thanks to a code with a large distance), with an outer robust code for handling critical miscorrection or undetected faults.

Error Coefficient and Location Table Decoding

In order to implement the multiple architectures in hardware it is however necessary to take into consideration the costs of such architectures. For instance, one common way to decode BCH codes is the Berlekamp-Massey algorithm [Mas69]. This is however costly in hardware, and can therefore not be considered for all applications, especially in the case of constrained or low power ciphers. For this reason, a new decoding technique was developed [KGKP18], and then further refined [GKKP19]. The approach, called Error Coefficient and Location Table (ECLT), is oriented towards single errors (but could be adapted for larger errors at a higher cost) and has a lower cost than more conventional decoding methods (Section 3.3.3). The syndrome generated from the protected value is

compared with a few pre-computed values, and if a match is found, the location as well as the value of the error can be derived. The ECLT is pre-generated for single faults only (i.e. 4-bit nibbles or bytes corresponding to a single symbol of the code, in the context of this thesis), which is often the considered fault model for DFAs. Additionally, and even if larger faults were considered, the table would be smaller than usual tables. This is especially true when bytes are considered and for the dual decoder architectures. Moreover, the use of the ECLT avoids difficult computations, such as the hardware implementation of an inversion in a Galois field or a large number of multiplications.

Advanced Rabii-Keren Architectures

The QS codes previously used as outer code are robust and already allow for the detection of a large number of critical faults, however, this might not be sufficient for some applications and the QS codes are not scalable. To remedy this issue, the QS codes are replaced by a Compact Protection Code (CPC) [RNK19] based system level fault manager [GKKP20] (Section 3.3.4). CPCs are also robust, and have a low complexity and implementation cost. Moreover, they are scalable by construction, and by choosing the appropriate size, it is possible to drastically improve the detection rate of the CPCs, and thus avoid any well crafted malicious fault that would have went through the inner RK code undetected or miscorrected. Mainly, the CPCs have a similar detection rate to QS codes, for a size of 4 redundancy bits, but experimentally have up to 0.0001% misdetection rate for larger sizes. As such, the use of CPCs further strengthen the proposed architectures described in this thesis.

The previously mentioned architectures were evaluated in hardware on a SAKURA-G Field Programmable Gate Array (FPGA) board, and more precisely on the SPARTAN 6 LX75 FPGA, using a clock glitch based fault injector on multiple Substitution and Permutation Network (SPN) ciphers implementations. The experiments show that the architectures are well suited for the detection of fault attacks, as well as natural faults, and single fault injections are fully corrected (Section 3.4). Furthermore, the RK inner code is especially effective for a distance of at least 5, and the inner-outer code construction can reliably detect undetected faults or erroneous corrections, which reinforce the usefulness of such an architecture.

The Hardware-Oriented Algebraic Fault Attack Framework: AutoFault

The first fault injection attack was introduced in [BDL97] on the Rivest–Shamir–Adleman (RSA) cryptosystem [RSA78]. Since then, a large number of fault attacks have been proposed for multiple widely used ciphers. Such attacks are usually manually built by cryptographers, as each encryption scheme is sensible to different fault locations and positions, referring respectively to the location in the execution (i.e. in which round or operation for a given cipher) and the bits affected in the encryption state. For instance, if we consider the Small Scale AES (SSAES) [CMR05], which is a scalable cipher based on the widely used AES, same fault location and position affects each variant differently. However, prior to the work presented in this thesis, there were no attacks or hardware implementations of the different variants of the SSAES, and no generic tool to create AFAs on the SSAES.

Small Scale AES Toolbox

Therefore, the SSAES was implemented in hardware [GBH+16]. While the basic cryptographic blocks composing the SSAES are similar to the one of the full size AES, they are parametrisable, which allows for different size of states, number of rounds and the possible omission of the last mix column operation. This allows for a scalable cipher and the different implementations can be used in the appropriate context, for instance in low power devices or, in the context of this thesis, as a useful tool for research purposes (mainly as a benchmarking tool). One of the key differences, which reduces the complexity and constraints of the implementation, is the variable number of rows and columns (for a representation as a state matrix). This leads to a different fault propagation pattern, and, as such, conventional fault models, which are valid for the AES, are not valid for every SSAES variant. In this regard, the fault equation sets for every variant of the SSAES were proposed (Section 4.1). In order to attack the different variants of the SSAES, the circuit description was converted to Conjunctive Normal Form (CNF) clauses, using the Tseitin transformation [Tse68], mainly provided for use in conjunction with SAT solvers, and a first step towards automated hardware-oriented AFAs.

Basic AutoFault Framework

As a logical continuation, the first prototype version of the hardware-oriented AFA framework `AutoFault` was implemented [BGE+17]. `AutoFault` was developed in collaboration with the chair of computer architecture of the University of Freiburg, which has been working on Boolean Satisfiability (SAT) for many years. Their expertise in SAT solving for circuit-based problems brought in many ideas for the development, and efficient operation, of the framework. However, the contributions to the framework detailed in Chapter 4 of this thesis, unless stipulated otherwise (namely for SAT solving integration, and CNF conversion), are the work of the author. The framework itself was created in order to automatically construct AFAs for any proposed cipher implementations. The automation of the construction of AFAs is of particular interest for the analysis of hardware implementation of cryptographic primitives. For instance, a designer who implements a new cryptosystem can use `AutoFault` to quickly assess any potential vulnerabilities to fault injection attacks. Additionally, if a vulnerability has been found by the framework and a modification was implemented to counter it (for instance switching to a different variant of the SSAES, not vulnerable to the same fault model), the adequateness of the changes can be easily verified. In a similar way, any non-cryptographic optimisation of the design can also be introduced during the design phase, and as such the design needs to be re-tested against fault injection attacks, which can just as easily be performed by `AutoFault`.
Of course, another point of interest are counter-measures and the assessment of their usefulness against fault attacks. One way `AutoFault` can be used to verify that the implementation of a counter-measure did have an effect on a potential fault vulnerability is by restricting the fault location. Let's for instance imagine that, after a careful evaluation of a design, the designer decided to add some metal shielding to a specific part of the circuit to prevent any fault injection at this location. The fault model can be restricted accordingly in `AutoFault` and only the remaining non-shielded location can be considered. If the framework is then incapable of successfully recovering the secret key, contrary to a previous run without such limitations, then the shielding is indeed useful against fault injection attacks. Another type of counter-measure, directly related to this thesis' work, are EDC. As previously stated, EDCs can be used as a counter-measure to fault attacks, however, some faults may avoid detection [KW14]. It is possible to restrict `AutoFault`

to such faults, as well as some handcrafted ones. This is especially useful to assess the usefulness of an EDC architecture or even if the correct component was protected.

Mainly, the goal of `AutoFault` is to provide a quick and easy way to verify vulnerabilities without spending an extensive amount of time crafting fault equations derived from the encryption algorithm. More precisely, the framework does not require any algebraic description of the encryption scheme itself, but rather takes as input the hardware description of the implementation, either in Hardware Description Language (HDL) format or directly as a netlist, and a fault model. A time-frame expansion of the design is then performed by `AutoFault`. It alters the initial implementation of the cryptographic circuit in order to convert the cipher description to CNF clauses, as well as adding the fault relevant parts for the AFAs. For instance, for an SPN cipher, copies of the rounds are linked together, creating a combinational or fully unrolled implementation of the cipher, and XOR gates are introduced taking the fault as an input and feeding the output directly into new copies of the now fault affected rounds. The newly created circuit is then converted to CNF clauses via the Tseitin transformation (similarly to the case of the SSAES). The CNF clauses are then transferred to a SAT solver, which attempts to solve the given instance and recover the secret key.

Enhanced AutoFault Framework with Multiple Faults Support

While the first prototype version of `AutoFault` was only capable of processing a single fault injection during the attack construction phase, as well as the solving process, the framework was extended to support multiple faults [GPU+19], as well as new ciphers, and various optimisations were added (the overall framework is detailed through out Sections 4.2 to 4.9). The structure of the framework remains the same, but a new set of XOR gates and fault affected operations are introduced for each fault during the time-frame expansion step. Even though this increases the size of the processed data, since the number of CNF clauses will grow from the addition of more gates, it is important to note that only the fault affected operations are duplicated, not the full scheme. In addition, the solving process itself was also extended to be able to handle multiple faults, with the addition of new variables and the support for more general fault models.

Additionally, the framework was also extended to process data originating from physical on chip fault injections. Previously, all the results were simulated directly in `AutoFault`. This can be referred to as pre-silicon analysis, as it can be performed before the manufacturing of the cryptographic hardware, when only the HDL description of the implementation is available. With the addition of external inputs to the framework, `AutoFault` can assess the design against real life fault injection attacks. For example, the design can be implemented on an FPGA; then a laser can be used to disturb the encryption [BECN+06]; and then the output of the faulty encryption can be fed to the framework, which will try to mount the attack according to the given fault model. However, if the given fault model does not match the actual faults which were injected, it will most likely be impossible for the framework to successfully attack the considered cipher. This second application case for `AutoFault` is denoted as post-silicon analysis.

The initial prototype of `AutoFault` was only capable of solving small variants of the SSAES, as well as the LED cipher. However, it already showed the potential of such a framework for the automation of AFAs on hardware implementations, without any pre-cryptanalysis. The extended framework could successfully recover the secret key for PRESENT, as well as a full scale AES 128-bit implementation, thanks to the support of multiple faults and other optimisations, which also drastically improved the solving times. Finally, a physical fault injection run was performed on the previously mentioned SAKURA-G board, before transferring the data (restricted to single nibble successful fault injections) to `AutoFault`, which successfully recovered the secret key in most cases.

1.3 Structure of the Thesis

The rest of this thesis will be organised in three main Chapters. Chapter 2 will give some background information on both attacks and counter-measures. In Section 2.1, a description of power analysis attacks, as well as fault attacks, with a focus on DFA and the case of SPN cipher will be given. Then the principle of AFAs will be presented, as well as a discussion on the current state of the art for AFA frameworks. In the following Section 2.2, EDC and ECC counter-measures will be introduced, firstly with conventional codes, and then with security oriented codes, as well as the theory on the RK code, which is the base-

line code used for every architecture related to this work. Further counter-measures are discussed in the same section, as a bridge to counter-measure evaluation with `AutoFault`. In Chapter 3, the base idea behind the work on the ECC architectures from this thesis will be presented. A further discussion on natural faults and malicious faults in cryptographic circuits will be given in Section 3.1, as well as the limitations of the evaluation of ECC architectures in Section 3.2. Section 3.3 will follow with details on the different architectures themselves, such as the inner-outer code architecture or the different benefits of the ECLT. At the end of this chapter, in Section 3.4, experimental results for the multiple implementations will be presented. In Chapter 4 of this thesis, an in-depth presentation of the `AutoFault` framework will be provided. From the preliminary results on the SSAES in Section 4.1, which laid the ground floor for the automated generation of AFA, we will continue on the overall structure of the `AutoFault` framework in Section 4.2. The following Section 4.3 will detail each solving step related to the framework. Section 4.4 will be on the simulation of fault attacks in `AutoFault`, and will provide details on software based simulation compared to CNF simulation, and why choosing one over the other. Section 4.5 provides some information on how to employ `AutoFault` in the design flow. To showcase the effectiveness of the framework, some results on SPN ciphers, as well as the effect of multiple fault support will be presented in Section 4.6 and 4.7 respectively. Moreover, a discussion on counter-measure validation with `AutoFault` will be given in Section 4.8, and the last Section 4.9 will compare `AutoFault` to other state-of-the-art AFA frameworks. Finally, Chapter 5 will summarise the research contributions of this thesis, as well as discuss the possible future work based on the presented results.

Chapter 2

Preliminaries on Fault Attacks & Counter-Measures

Physical attacks are an ever growing threat to cryptographic primitive implementations. More precisely, fault injection attacks have been used since several decades to compromise otherwise thought secure encryption circuits. In order to better understand how to protect sensitive hardware against such threats, and more specifically in an automated fashion, the notions on side-channel and faults attacks, as well as respective counter-measures, will be introduced in this chapter.

2.1 Background on Side-Channel & Fault Attacks

In the following section, we will first give a brief introduction to side-channels attacks, before going more into details on fault attacks. In the context of this thesis, it is important to understand the scope of side-channel attacks, both from a general standpoint, and also regarding hardware counter-measures themselves. The main focus of the thesis being fault attacks, the rest of this section introduces the general concept of fault attacks, and then converges towards Algebraic Fault Attacks (AFAs). AFAs are of particular interest, since the framework later proposed in this thesis (Chapter 4) can be used as an automated tool for the generation of such attacks, and as such, the evaluation of cryptographic hardware implementations.

2.1.1 Side-Channel Analysis

Side-Channel Analysis (SCA) relates to a type of physical attacks, which makes use of extra information that can be derived from practical implementation of cryptographic primitives. The considered additional information does not relate directly to the theoretical cryptographic algorithm, but is rather based on the analysis, and measurement, of physical properties of the device on which it is implemented. Hardware characteristics such as timing related information [Koc96], or the power consumed during the encryption process [KJJ99], are among the most commonly used for SCA. An attacker can make use of side-channel measurements and infer information about the secret key by either by building and solving an equation system, or by statistical analysis. Moreover, SCA can be divided into two categories, depending on the capabilities of the attacker.

- If an attacker is only capable of observing and measuring physical leakage, then the attack can be referred to as Passive SCA.
- If, on the other hand, the attacker can, in addition, change the intermediate data, the attack can be classified as Active SCA.

More specifically, in the case of passive SCA, the device under attack processes the sensitive data, while, during the computation, the attacker observes the influence of the operations being processed on a side-channel (e.g. power [BCO04]). The attacker can then interpret the measured data in order to recover the secret information, but is at no point during runtime able to perturb the encryption process. This is the exact opposite for active SCA. In this case, at a specific point during the computation, the attacker is able to affect the operations being done on the device through a perturbation of a side-channel, such as making use of the photosensitivity of a cryptographic circuit. This may lead to either another observable side-channel leakage, or to a faulty output, which can then be analysed in order to retrieve the secret key. Active attacks, and more precisely fault attacks, are however discussed in more details in Section 2.1.2.
In order to be able to measure side-channel leakage, some preliminary steps may be required. For instance, it may be required to partially, or completely, decapsulate a chip in order to be able to record some side-channel data, or to influence the execution of a cryptographic algorithm. In this regard, both passive and active SCA can be further classified.

- Non-Invasive attacks do not require any modification of the targeted device in order to be successful.
- Invasive attacks are characterised by their requirement for depackaging or other damaging methods, so that the correct side-channel can be accessed.
- In some cases, only a partial decapsulation is necessary. In this case, the attack may be referred to as Semi-Invasive.

Furthermore, by their nature, invasive attacks (such as [SHS16]) are hard to repeat and require an expert knowledge in order to avoid destroying the chip altogether, while still removing enough protecting layer to be able to mount a successful attack. On the other hand, non-invasive attack, for instance EM-based attacks [AARR03], can easily be repeated and replicated, as only connecting the device under attack to the measurement device is necessary. However, the success rate of non-invasive attacks may be lower by comparison, or require much more measurements to be made. The use of semi-invasive methods allows to less significantly affect the chip, and are also less expensive than more destructive techniques. While not necessarily being as efficient as invasive SCA, semi-invasive attacks are a good middle ground and usually provide better results than non-invasive ones [QS01, GMO01]. With regards to this thesis's work, only non-invasive clock glitch-based active SCA attacks have been implemented, as a validation tool for EDC counter-measures and the AFA framework `AutoFault`.

In order to better understand the threat that SCA constitutes to secure hardware implementations, and thus the necessity of counter-measures, let's first discuss the general strategy which may be employed by an attacker in the case of a passive SCA. The device under attack is initially connected to a measurement tool. This can either be through the use of a physical connection, for instance an oscilloscope can be connected to some pins of the target, or remotely, such as an EM probe placed nearby the chip. Several measurements are then performed by the attacker during runtime, in order to record side-channel data for the considered hardware. The number of measurements is greatly dependent on the device itself, as well as the implementation choices, but also on the measurement resolution. Additionally, some counter-measure may be implemented, rendering the attack at least less efficient, and increasing significantly the number of measurements needed for a

successful attack. Once enough measurements have been performed, the collected data is confronted to a leakage model, and through key guesses, as well as statistical analysis, the correct key candidate can be derived. This method, or a derivative, is commonly used for most SCA, for example power analysis [MOP07].

An example of this process can be a Differential Power Analysis (DPA) on an unprotected AES implementation. Since the considered cryptographic circuit is unprotected, only a few thousands of traces are needed in order to recover the secret key without any pre-processing. As described previously in the general SCA method, the first step is to measure the power consumption of an hardware implementation of an AES. For this simple example, let's assume that the AES is implemented on an FPGA, connected to an oscilloscope and itself saving the recorded data on a PC, via the use of a script. While measuring the overall power consumption of a device can be easily achieved, for a DPA to be successful, not the entirety of the trace should be considered, but only a relevant portion. In the case of the AES, the first operation performed is the XOR with the master key, followed by the substitution layer of the first round. Considering this part of the encryption would allow for a successful DPA to be mounted, as key guesses can be made on the master key, and thanks to the non-linearity of the substitution function, the attack can be mounted successfully (given enough traces). Finding the exact spot in a power trace where the encryption is performed, and more precisely, the first SBox, is simple in the case of an unprotected implementation. Figure 2.1 depicts such a case. The AES encryption can be clearly identified by the repetition of ten similar patterns, which correspond to the 10 rounds of the AES (in red on the figure, the first round being longer due to the first master key XOR, and the last round shorter, thanks to the omission of the mix column operation). In addition, and since the rounds can be easily determined, restricting the analysis of the power trace to only the SBox operation can be done as well, as this is the first operation of the first round, and we assume no counter-measures (in green on Figure 2.1). It should be noted, that such clear identification is not always possible, dependent on the implementation itself, and potential counter-measures. Once the correct spot for the processing of the power trace is chosen, the analysis work can begin. First, a divide and conquer strategy is applied since a DPA requires some key guessing to be successful. In the case of the AES, making key guesses on the overall key wouldn't be helpful, as the

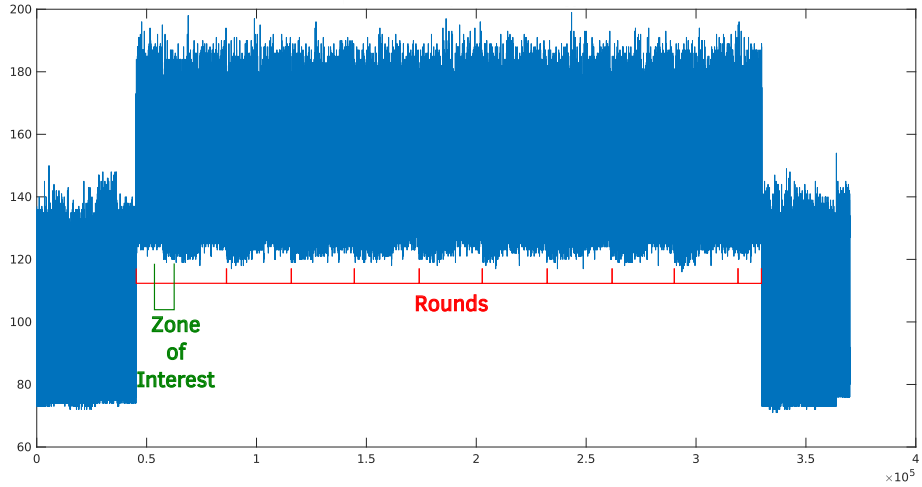

Figure 2.1: Power Consumption of an AES Encryption

key is 128 bits long, which would lead to too much processing. However, we consider the first substitution layer, which means that we can split our guesses over each single SBox, and in turns, over each key byte. The divide and conquer strategy is widely used in side-channel analysis, as it allows for a reduction of processing costs. In the considered case, a guess is made on a byte, also called key part, which allows for the computation of the output of the corresponding SBox, since the plaintext is known. Algorithm 2.1 shows the details of the DPA in this case. In short, the initial guess is made on each key part and the corresponding hypothetical value is computed. The recorded traces are then sorted accordingly to the value of a chosen distinguisher, in this case the least significant bit of the SBox output. The DPA trace, which is the difference of both averages, where the distinguisher is respectively 0 or 1, is then also computed, and finally, for each byte, the hypothesis which corresponds to the DPA trace with the highest peak value is recovered, which should be the correct key part, given enough data. The success of the DPA comes from the fact that, if the key guess is correct, power traces will be correctly sorted in the two different groups, according to the distinguisher. Since the distinguisher is a bit, and the power consumption of the device should be different depending if the bit was at 1 or 0, it results in a DPA trace with a high difference in power consumption for the correct key guess, while the consumption for other guesses will be randomly distributed (since the

hypotheses would be wrong, and thus the sorting too) and, as such, the DPA trace should be flat. This is of course in an ideal scenario. In reality, DPA traces may be much more noisy. As an example, Figure 2.2 depicts a DPA trace for a correct key guess, but computed with only 5000 traces. The low number of traces may for instance be due to a limited access to the device, and the resulting DPA trace, while still composed of peaks (highest one circled in red on the figure), is not composed of a single high peak. Nevertheless, The corresponding attack is successful, even with such a non-ideal trace.

Algorithm 2.1: Simplified DPA algorithm on an unprotected AES

```
1  for b = 0..15 do                                          // for all key parts
2      for i = 0..255 do                                  // for every possible byte value
3          hypothesis = SBox(plaintext[b] ⊕ i);
4          for t in Traces do                                          // for all traces
5              if hypothesis[0] = 0 then                // distinguisher: hypothesis LSB
6                  grp0 = grp0 + Traces[t];
7              else
8                  grp1 = grp1 + Traces[t];
9              end
10         end
11         dpaTraces[i] = average(grp0) − average(grp1);
12     end
13     Key[b] = getMaxDPA_index(dpaTraces);
14     // recovers the index of the DPA trace with the highest peak value
15 end
16 return Key
```

The success or failure of the attack depends on the quality and number of measurements. A good way to improve results is to record more power traces. It may however not always be possible to do so, due to hardware constraints, limited access to the device, or counter-measures. Moreover, it is still possible to improve the success rate of a DPA via different methods, but at the cost of processing. For example, applying some pre-processing techniques to the initial power traces, such as filtering or the creation of templates (so called template attacks [CRR03]) can improve the effectiveness of the analysis. Another way to get better results can be to choose a different distinguisher, or to consider several distinguishers, and take the most recurring value for each key part candidates (for instance, considering each bit of an SBox output, for the case of the AES). This however at the cost of computation, as this corresponds to mounting several DPAs.

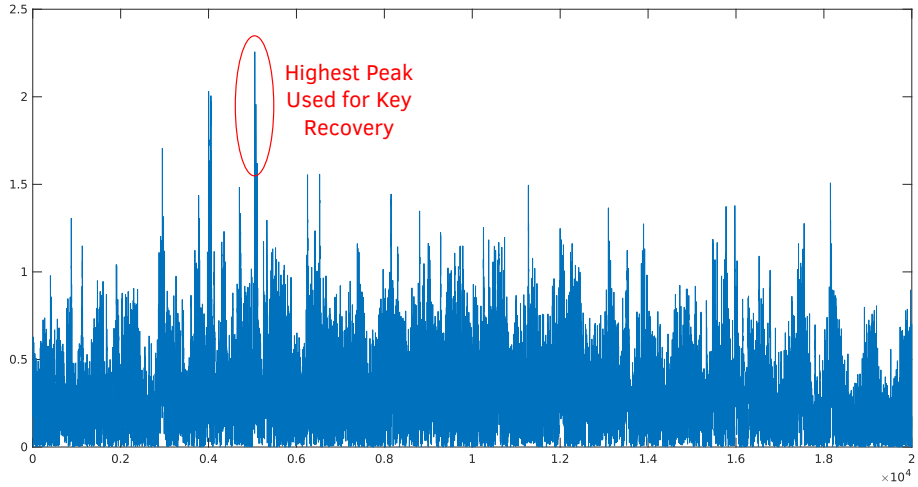

Figure 2.2: Noisy DPA Trace of an unprotected AES Encryption

Nevertheless, DPAs, and other passive SCAs, are powerful, and can recover the secret key with only a few thousand traces in the case of an unprotected implementation, for instance, only around 5000 traces were necessary in the previously discussed AES example. The fact that such a simple attack can be mounted, and generalised to other ciphers, shows the necessity of implementing counter-measures against passive SCAs. While passive SCAs are not the focus of this thesis, the implementation of a few counter-measures is discussed in Section 4.8 in the context of automatic circuit evaluation, and therefore passive SCAs are still relevant with regards to this work. Moreover, active SCAs, and more especially fault attacks, which are more directly relevant for this thesis, are considered in the following. They will also show why, in a more general way, some counter-measures need to be implemented in order to protect sensitive circuits against non-invasive, as well as invasive, active or passive SCAs.

2.1.2 Fault Injection Attacks

Fault injection attacks are another type of physical attacks, which makes use of a perturbation during the encryption process, the fault, to retrieve the secret key. The fault is deliberately induced at runtime to modify the behaviour of the device, for instance, by bypassing some counter-measures put in place, skip an operation or modify a value at a

given location in the encryption. In most cases, a post analysis is required to process the information derived from the fault injection. The fault themselves can be induced by multiple means, such as clock or voltage glitches, optical fault injection through the usage of a laser, or targeted electromagnetic disturbances. As such, the attack is an active attack, which may also be invasive, as, for instance, it may be necessary to decapsulate the chip in order to precisely and effectively shoot at a specific location with a laser. Even so, fault attacks are powerful and should not be overlooked when designing an encryption scheme.

The first fault injection attack was introduced in [BDL97]. The attack targets more specifically the RSA cryptosystem, however, it can be generalised to different other ciphers. The attack aims at the modular exponentiation computed in the RSA. Let's first briefly recall the RSA cryptosystem. $n = pq$ is the modulus chosen for the modular exponentiation, with p and q being two large prime numbers. The public key e is chosen such that $gcd(e, \phi(n)) = 1$, with $\phi(n) = (p-1)(q-1)$ and the private key d is defined by $d \cdot e \equiv 1 \pmod{\phi(n)}$. Then we have, for a private message m, $(m^e)^d \equiv m \pmod{n}$, and as such the ciphertext c can be computed by $c \equiv m^e \pmod{n}$, while the decryption is performed by computing $c^d \equiv m \pmod{n}$.

However, one of the most common way to implement the RSA cryptosystem, is to take advantage of the Chinese Remainder Theorem (CRT) in order to compute the exponentiation modulus p and q, instead of n, and thus perform more efficient operations. In fact, it is possible to find two integers a and b, such that:

$$c^d = a(c^d \pmod{p}) + b(c^d \pmod{q}) \pmod{n} \quad \text{with} \quad \begin{cases} a \equiv 1 \pmod{p} \\ a \equiv 0 \pmod{q} \\ b \equiv 0 \pmod{p} \\ b \equiv 1 \pmod{q} \end{cases} \tag{2.1}$$

While performing the exponentiation in such a way is less costly, since the numbers are smaller, it leads to a vulnerability if two decryptions are performed on the same message, but one is faulty. Let's assume that it is possible for a fault to occur during the computation of $c^d \pmod{p}$, but not for $c^d \pmod{q}$, then, it is possible to express the decrypted message m (and respectively, the faulty message m') as follows (where ϵ denotes the effect of the

fault):

$$\begin{cases} m = a(c^d \pmod p) + b(c^d \pmod q) \pmod n \\ m' = a(c^d \pmod p) + \epsilon) + b(c^d \pmod q) \pmod n \end{cases} \tag{2.2}$$

and as such:

$$m - m' = a(c^d \pmod p) + b(c^d \pmod q) - a(c^d \pmod p) + \epsilon) + b(c^d \pmod q) = a \cdot \epsilon \tag{2.3}$$

Finally, if ϵ is not divisible by p, which is likely if p and q are chosen randomly, then it is possible to retrieve q.

$$gcd(m - m', n) = gcd(a\epsilon, n) = q \tag{2.4}$$

Once q is retrieved, it is trivial to derive the secret key d. Moreover, in the case where the original ciphertext c is known, only a single faulty decryption is needed.

$$gcd(c - (m')^e, n) = q \quad \text{where} \quad m' \text{ is a faulty decrypted plaintext} \tag{2.5}$$

The attack laid the ground foundation for all the further fault attacks to come. It especially introduced the fact that a faulty intermediate computation, even if the value of the fault itself is unknown, can lead to a severe vulnerability, dependent on the design choices made for the implementation of a cryptographic scheme. Moreover, even though the attack was first targeted toward the RSA cryptosystem, it can be extended to a larger family of ciphers, as well as different types of RSA implementations, at the cost of more fault injections. Similarly, fault injection attacks are applicable to both dedicated cryptographic circuits, and processors running cryptographic software, as long as a fault can be induced.

It is also important to note that, at the time, the attack was purely theoretical, and was only considering potential errors occurring during the encryption, but not yet proposing a way to physically inject a fault. However, since then multiple techniques have been proposed and validated to practically inject faults during runtime. Such fault injection methods range from simple low cost glitch generation up to the usage of expensive equipments, such as focused ion beams.

2.1.3 Fault Injection Techniques & Fault Models

Glitches can be introduced by diverse methods, but the most common ones are power glitches and clock glitches. The former can be achieved by modifying the supplied voltage of the targeted device at a given point during the encryption process. This may cause some instructions to be skipped or some values to be incorrectly stored. The equipment required to cause such glitches is usually widely available and cost effective. However, the faults induced by power manipulations are usually imprecise. Even so, they are commonly used and fault injection with a reasonable precision can be achieved [O'F16]. A similar way of injecting faults is using an external clock, or modifying the internal clock, by introducing a clock glitch. Essentially, clock glitches are similar to power glitches in the sense that they can cause some instructions to be misread or some data to be wrongly accessed (for example, by accessing the data of round $n+1$ instead of round n for a specific operation), but also in terms of cost efficiency and lack of precision. An example of such clock glitches is the fault injector proposed in [MSI16]. On a SAKURA-G FPGA platform, two Digital Clock Managers (DCM) are used to create two asynchronous clocks, which are then XORed, creating a glitched clock, which can be used for fault injection.

Optical fault injection setups are also widely used to induce faults. A light source, usually a focused laser, is pointed at the decapsulated chip and the photon emitted by the light source disrupt the correct operation being done on the circuit. For example, targetting a Static Random Access Memory (SRAM) cell can cause bit-flips, which is the type of fault that an attacker may need. The main advantage of using this kind of fault injection method is the high precision of the injected fault. For instance, in [ADM+10] a single bit-flip fault is induce during an AES encryption, and thus successfully implement the attacked proposed in [Gir05], which would have been extremely difficult with power or clock glitches. This type of attack isn't limited to laser, and multiple optical sources can be used [ABC+17, PDL18]. Similarly, photons do not only affect SRAM cells, and many different platform can be targeted [SBHS16], as long as it is possible to target the relevant part of the circuitry. This however comes at a high cost in equipment, as well as expertise and time, since it may be a tedious process to find the correct location to target on the chip.

Other fault injection methods exist, such as electromagnetic-based (EM) injections [MDH+13, HHM+13], which rely on well timed EM pulses to disturb a specific operation. EM fault injectors also do not always require decapsulation of the chip, compared to optical ones, while still allowing for more precise fault injections than glitch-based attacks. Among other fault injection methods, one may also consider Row Hammer injections [KDK+14]. Row Hammer injections target the Dynamic Random Access Memory (DRAM), and by repeated accesses to a row, can flip a bit in an adjacent row. This attack was later extended to FPGA-based systems [KGT18]. Table 2.1, a simplified version of Table 1 available in [BBKN12], summarise most fault injection methods, as well as their advantages and drawbacks.

Table 2.1: Fault injection techniques summary

Technique	Spatial Accuracy	Temporal Accuracy	Skill Required	Cost	Implementation's Knowledge Required	Destructive
Underfeeding	High	None	Basic	Low	No	No
Power Glitch	Low	Moderate	Moderate	Low	Partial	No
Clock Glitch	Low	High	Moderate	Low	Yes	No
Light Radiation	Low	Low	Moderate	Low	No	Yes
Light Pulse	Moderate	Moderate	Moderate	Moderate	Yes	Possibly
Laser Beam	High	High	High	High	Yes	Possibly
Focused Ion Beam	Complete	Complete	Very High	Very High	Yes	Yes
EM Pulses	Low	Moderate	Moderate	Low	No	Possibly
Heating	Low	None	Low	Low	Yes	Possibly

It is usually not sufficient for an attacker to create a random disturbance. As pointed out in the first fault attack on RSA, choosing the correct fault model is important. The previously described fault model was specific to the CRT exponentiation, but it was also very lax, as any fault in one exponentiation would be sufficient for the key recovery. In a more general setting, multiple fault models exists, however two models in particular are the most commonly considered: single bit-flip faults and random byte (or more generally nibble) faults.

Single bit-flip faults are one of the strongest fault model. They refer to a powerful attacker, who is able to specifically target a chosen bit of a particular value during the encryption process. Such faults are extremely difficult to produce in practice, however they are not unrealistic [ADM+10]. Due to this difficulty to inject them, they are considered more in theoretical work. Moreover, for a cipher of key length n, if a single bit-flip fault can be

precisely injected during an operation dependent on the key, then n fault injections are sufficient to recover the secret key, by just flipping successively each bit of the key and observing a change in the output.

Random byte faults, or more generally random nibble faults, are the most commonly considered fault model [SH13, TBM14]. In this case, the attacker can only affect a portion of a value or an operation, and has no control over the value of the injected fault. The fault model is really lax and can match a wide variety of physical fault injections. In the case of block ciphers, the nibble size considered usually matches the size of some operations. For instance, int he case of the AES, byte faults are usually considered, as the internal state can be represented byte-wise and the the SBoxes operate on bytes as well. In some other cases, for example for stream ciphers, neighbouring bits can be affected by a fault targeting a specific location, resulting in a fault which can also be represented by this fault model.

In the context of this thesis, only random nibble faults are considered, and the size is dependent on the attacked cryptographic primitive. Similarly, only transient faults which can be represented as additive errors on the nibble are studied, unless stipulated otherwise.

Another important consideration for the fault model is the possibility to inject multiple faults simultaneously. For example, it is possible to consider two random byte faults injected simultaneously at two different locations during the encryption process. While this kind of fault injection was difficult to achieve, the recent advancement in terms of fault injection setup made such injections a practical reality [SHS16], and allowed for new attacks to be mounted. This is particularly relevant with regards to the SSAES attack presented in this thesis.

2.1.4 Types of Fault Attacks

The definition of a correct fault model, as well as the availability of divers physical fault injection setups, allowed to mount different types of fault attacks. The main ones are differential attacks, which are defined more in details in the following section, and are based directly on the approach from [BDL97]. Among other categories of fault attacks, two additional analysis methods are commonly used: statistical fault attacks and AFAs. The latter are the core of the work on fault attacks presented in this thesis, and as such

are presented in a later section (Section 2.1.5). The former are based on a statistical biased present during the encryption thanks to the fault injection. While statistical attacks are not directly related to the work done on attacks in this thesis, it is possible to use ECCs in order to protect the circuit against statistical attacks, as presented in [BKHL20]. As such, as well as for completeness, they are briefly described below.

First, Ineffective Fault Analysis (IFA) was introduced in [Cla07] as a way to derive the secret key of the Data Encryption Standard, without any knowledge on the value of its input or outputs. However, while the input-outputs values are not needed, it is required to be able to distinguish between the output of a fault free encryption and a faulty one. The attack relies on retrieving some information on the values of the key bits. A fault is injected during an XOR operation dependent on the secret key, and the output is forced to zero. Moreover, an extension of the attack was discussed, which later lead to more IFA on other ciphers [BG13, ADY15]. While IFAs are not making use of statistical differences directly, they were later extended to Statistical Ineffective Fault Attacks (SIFA) [DEK+18]. SIFAs similarly make use of the fact that a fault injection does not change the output of an encryption, however, the distribution of an intermediate value becomes non-uniform due to the fault effect. It is then possible to make key guesses on portions of the secret key, and revert the last operations of the encryption. If the key guess is correct, then the statistical bias can be observed, in contrast to the cases where the key guesses are wrong. One of the main advantage of SIFAs compared to IFAs is the required fault model. IFAs require a rather strong fault model, such as stuck-at-fault at a precise (and possibly hard to achieve) location, while SIFAs only require that the fault induced results in an unknown but non-uniform distribution of the intermediate value considered. A second advantage of IFAs is the possibility to circumvent some widely used counter-measures, such as majority voting or masking. Moreover, SIFAs were recently successfully applied to nonce-based ciphers from the CAESAR[1] competition, such as GIMLI [GPT20] or one of the winner of the competition, ASCON [RAD19].

Fault-Sensitivity Analysis (FSA) [LSG+10] is another widely use type of statistical fault attack. FSAs takes advantage of the correlation between the fault intensity, the strength

[1] Competition for Authenticated Encryption: Security, Applicability, and Robustness, organised in 2012 by the United States National Institute of Standards and Technology

of the disruption method used to inject the fault, and the sensitive data being processed. The attacker incrementally increases the fault intensity (for example by reducing the clock period), until a noticeable faulty output is produced. Once a faulty output is generated, the fault intensity for which the fault occurred, denoted critical fault intensity, is stored. The process is repeated multiple times for different randomly generated inputs, until a sufficient number of critical fault intensities are recorded. This results in a tuple of plaintext, ciphertext and critical fault intensity. The data can then be used to compute a predicted critical fault intensity, by making a key guess on a key part, and, for instance, reverting the last round operation. Finally, the correlation between the predicted, key guess dependent, intensities and the measured critical ones is computed. If the key guess is correct, then there should be a high correlation, and thus the key part can be recovered. The attack uses a divide and conquer method, as it is a more efficient way to make key guesses, similarly to a DPA, and is repeated until all the key part are recovered. It should be noted that, for better results, the incremental increase of the fault intensity should be done in small steps. A variant of FSA was later developed, called differential fault intensity analysis [GYTS14, GYS15], and has the advantage of not needing the fault intensity prediction stage of an FSA.

2.1.4.1 Differential Fault Analysis

In [BS97], the authors introduced the concept of Differential Fault Analysis (DFA). While the attack was practically implemented for the Data Encryption Standard (as well as some variants of the same family of ciphers), with at most 200 fault injections, the authors already showed that the attack could be extended to almost any symmetric encryption scheme available at the time. It showed how effective were, and still are, DFAs at attacking different type of ciphers.

In a more general way, DFAs rely on well chosen fault injections to propagate the fault throughout a larger portion of the intermediate states, in order to be able to derive some secret key-dependent equations, which are computationally feasible to solve. The equations are difference equations, meaning that a difference between a fault free encryption and a faulty one is computed. Most ciphers have a confusion layer, dependent on a non-linear function, as well as a diffusion layer. The difference equations are then derived

thanks to the non-linear function, as else, with a linear function, the secret key information would be lost during the computation of the difference, while the diffusion, also often referred to as permutation, layer allows multiple equations to be derived from a single fault injection.

The principle of the attack can be generalised as follows for most ciphers. Let's denote the secret key as k. Given a non-linear function f in use in the considered encryption scheme (for instance, an SBox), a well chosen fault ϵ and a difference Δ, if a fault is injected before the non-linear operation, one can derive a difference equation of the following form (Equation 2.6).

$$\Delta_{k,\epsilon} = f(x_k) \oplus f(x_k \oplus \epsilon) \quad \text{where} \quad x_k \text{ is an intermediate value dependent on } k \tag{2.6}$$

In this generalisation, Δ is an observable difference, for example the difference between fault free and faulty ciphertexts, while x_k is unknown. The fault value ϵ is either known or unknown depending on the considered fault model, and can be deterministic or not. Since Δ is known, it is possible to reduce the number of possible key dependent values significantly. Moreover, if the fault location was carefully chosen or multiple faults are injected, several of such equations can be derived and used in order to have a single candidate for the secret value.

While Equation 2.6 is a generalisation, the same principle applies to different categories of ciphers. DFAs are particularly efficient at recovering the secret key of an SPN-based encryption. Inherently, SPNs have well-defined permutation layers, often matrix-based, which allow for good propagation of the fault, and thus well-defined and bounded difference equations. This often allows for a fault injection in a preceding round to have the same affect as multiple faults in the next rounds, leading to a strong, but controlled, diffusion. However, the fault propagation itself, even though useful, also creates more complex equations to solve. For instance, one may have to revert multiple operations instead of a single one at the last round. This results in a trade-off between how early the fault is injected, leading to good propagation and hence a larger number of equations (or respectively, smaller number of injected faults), and the complexity of the equations to be solved. It is for example inadvisable to inject a fault at the beginning of an SPN cipher and

expect to be able to recover the secret key via a DFA. For instance, the earliest successful DFA on the AES is performed by injected a fault between the *MixColumns* operation of the sixth round, and the one of the seventh one [DFL11]. This attack is however not the most efficient DFA on the AES, which is a direct example of the described trade-off. In general, for most SPN schemes, the fault is injected three rounds before the end of the encryption. This is for instance the case for the AES [TMA11], LED [JKP12] or Midori [CZS16]. This once again comes from the fact that most SPN ciphers follow a similar structure, and if the fault goes through two permutation layers, it generally propagates through the entire state. However, not all SPN have a matrix-based permutation layer such as the previous ones. For instance, the permutation layer of the PRESENT cipher is constituted of a bit-wise shuffling of the internal state. In the case of a random nibble fault model, this implies that the fault will propagate unequally through different nibbles, depending on the value of the fault itself. Nevertheless, DFAs are still possible, and are usually performed at the 29^{th} round out of 32 [XjZ11, BEG13]. It is also important to note that a DFA may not directly recover the master key, but rather a round key. This is however of no concern, as the key schedule is known, and can thus be reverted, leading to a recovery of the master key.

DFAs aren't limited to SPN ciphers and can also be applied to any other type of ciphers, as long as it is possible to create relevant difference equations with a chosen fault model. For instance, stream ciphers can also be targeted by DFAs, such as in [HR08]. In the case of stream ciphers, the main difficulty comes from the fault injections themselves, which must be well timed (sometimes in successive clock cycles). More recently, nonce-based ciphers are of particular interest due to their resilience to DFAs and other differential attacks. However, despite this, several DFAs still exist on some nonce-based ciphers, such as one on NORX [JSP20] or one on ACORN [SSMC17], which is one of the winners of the CAESAR competition (however, the attack is for an earlier version of ACORN). Among other classes of encryption schemes of particular interest for DFAs, one may consider Feistel networks. Feistel networks are constituted of several intertwined branches acting as a permutation layer, as well as some inner operations in each branch. This may lead to a slightly trickier cryptanalysis in comparison to SPN ciphers, as the key, or more precisely a key part, only affects a portion of the internal state after each round. It turns out that a DFA may require

more than one fault injection, some times even at different location, to be able to recover the complete key, and it is as such difficult to generalise DFA, as is the case for SPN-based encryptions. Nevertheless, it is possible to mount some DFAs on Feistel networks, such as the DFA on SIMON [TBM14] or on Piccolo [Jeo12].
Finally, the main advantage of DFAs, compared to other type of fault attacks, is the relatively low number of fault injections required. Where statistical fault attacks may require thousands of injections [LSG+10], a DFA may be able to recover the secret key with only a single successful injection [TMA11]. However, the required faults may, in comparison, be more difficult to inject. Nonetheless, and thanks to the improvement of fault injection methods over the year, this trade-off isn't an obstacle and DFAs remain one of the most commonly used type of fault attacks.
In order to better understand the work presented in this thesis, especially with regards to the SSAES, the case study of the DFA on the AES is presented in the following section.

2.1.4.2 DFA on the Advanced Encryption Standard

The AES is one of the most widely used encryption schemes. As such, there have been multiple fault attacks targeted at the AES over the years. Several fault attacks, such as the ones in [TFY07b, TFY07a, KQ08], directly target the key schedule component of the AES. However, a key schedule might not always be implemented, or it may be harder to inject a fault during the key schedule operations, that is one of the reasons why there is a more significant focus on attacking the encryption operation itself in the literature. A number of such attacks require several distinct fault injections, such as the ones presented in [Gir05, BS03, DLV03], ranging from 40 to 250 fault injections needed. In [PQ03], only a single byte fault is used to recover the secret key. However, the attack requires that the same fault to be injected twice. A more refined single fault DFA was introduced in [Muk09]. It requires only a single fault injection at the beginning of the eighth round, which reduces the key space to 2^{32}. While the previous DFAs are all able to retrieve the secret key of an AES encryption, the different constraints, either in terms of number of faults injected, or the need for the exact injection performed twice, make this last attack much more realistic. However, the attack can be improved in order to further reduce the key space, and thus the retrieve the key in a timely manner.

A case of particular interest is the DFA on the AES presented in [TMA11]. It optimises the attack presented in [Muk09] and further reduces the key space to 2^8 candidates, which is trivial to brute force. This attack is also fundamental to understand how efficient DFAs can be built, as well as hardware-oriented AFAs presented later in this thesis.

First DFA Step

The AES is composed of 4 different operations: *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*. The *SubBytes* operation is composed of multiple 8-bit SBoxes and constitutes the non-linear operation of the encryption. The *ShiftRows* and *MixColumns* operations are linear and are used as a diffusion layer. The fourth operation, the *AddRoundKey* is only an XOR with a round key, derived from the master key. The details on the operations, which are processed in this specific order, can be found in [NIoSTN01]. However, one essential information to understand the DFA on the AES, is how to represent the internal states of the cipher.

The AES internal states, can be expressed in terms of state matrices. More precisely, each internal state can be divided into 16 bytes in the following fashion.

$$S = \begin{pmatrix} s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \\ s_4 & s_8 & s_{12} & s_{16} \end{pmatrix} \tag{2.7}$$

Where the s_i are one byte values and every operation is performed over the Galois field $GF(2^8)$, defined by $x^8 + x^4 + x^3 + x + 1$. This is especially important for the *MixColumns* operation, which is a pre-multiplication of the state matrix by the following matrix:

$$M = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \tag{2.8}$$

and which will come into play for the derivation of the fault equations.

The considered fault model is a random byte fault. That is to say that the fault is induced into a specific byte, however without any knowledge on the actual fault value. This fault model is more lax than other fault models aimed at flipping a single bit at a very specific location and position. In the context of this attack, knowing which bits of a specific byte are flipped is irrelevant for the solving process. It should only be ensured that the fault is a non-zero fault (i.e., it has an effect on the targeted byte). The considered fault position is in the first byte of the state matrix (s_1 in Equation 2.7). It should be noted that, while this is the position considered for the attack, any fault injection for any s_i in the state matrix is valid. It would only change the resulting equations.

Figure 2.3 shows how the chosen fault propagates from the beginning of round 8 until the encryption is done (the state matrices are represented as four by four squares for better visualisation). The *AddRoundKey* operations are omitted as the attack is a DFA, and as such they eliminate themselves if they are considered as the meeting point for the differential state.

The initial fault value is f and is unknown. The first *SubBytes* operation changes the value of the faulty byte, we denote as $\tilde{f}$ the new fault value after the first SBox. The eighth round *ShiftRows* operation doesn't shift the fault affected value in the state matrix, as the rows are shifted by $i - 1$ (i being the row index in this case), and the eighth round *MixColumns* propagates the fault throughout the first column, since it is a pre-multiplication of each column by the matrix M of Equation 2.8. Additionally, the fault is not propagated randomly after the *MixColumns* operation. Thanks to the coefficient of the matrix, it is possible to express the bytes in terms of their faulty values (i.e. the coefficient are the same as the ones from M). After the ninth round *SubBytes* layer, the faulty values are modified (hence becoming new unknown F_i), moreover, since there was previously an *AddRoundKey* operation, $F_2 \neq F_3$. The next *ShiftRows* operation allows each column to be affected by a single F_i, which is then propagated through each column fully by the last *MixColumns* operation (again, with specific coefficients). Hence achieving full propagation after the last, ninth round, *MixColumns* operation, which is the chosen point for the differential state. The last operations until the end of the encryption do not further propagate the fault and only change the values of each byte. It should be noted that, due

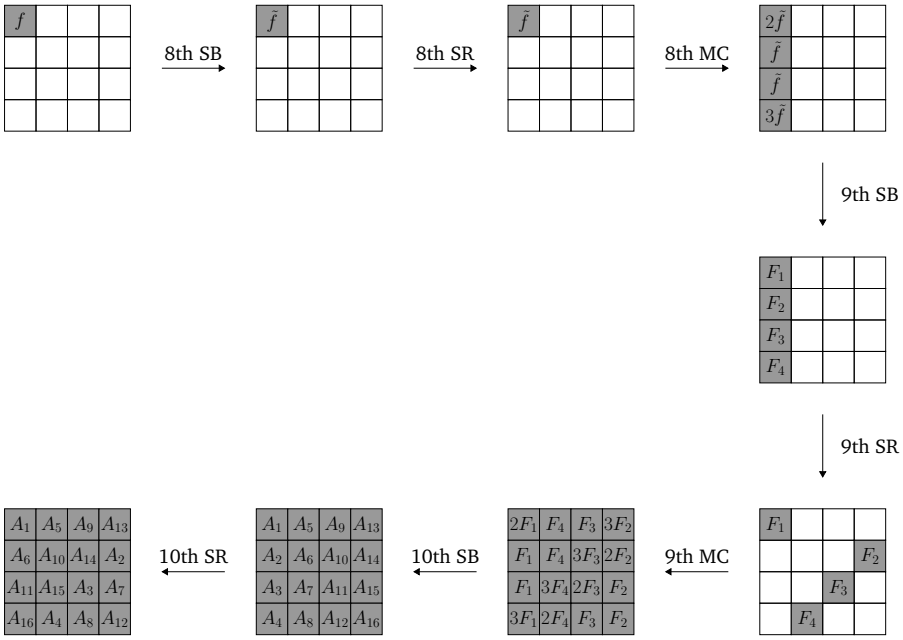

Figure 2.3: Differential fault attack on the AES

to the fault propagation pattern, if the fault is injected after the eighth round *SubBytes*, or after the *ShiftRows* operation, it would still be successful.

As such, it is possible to derive an expression for each byte of the differential state, by reverting the encryption until the last *MixColumns* operation. Let's denote by x and x' the fault free and faulty ciphertexts respectively, as well as x_i and x'_i their respective nibbles. The last round key is k and its parts are denoted k_i. The *SubBytes* operation is represented by the function $S()$ and its inverse by $S^{-1}()$. Finally, δ_i stands for the XOR difference at the chosen differential point. First the *AddRoundKey* operation with the last round key is inverted. It is only an XOR between the last round key bytes k_i and the ciphertext bytes x_i (or x'_irespectively), which is then shifted in an inverted pattern to the AES *ShiftRows* operation, before being processed by an inverse SBox. Both of the expressions for each bytes of the state matrix are then finally XORed, giving the following expression for a byte of the differential state.

$$a.\delta_i = S^{-1}(x_j \oplus k_j) \oplus S^{-1}(x'_j \oplus k_j) \qquad \text{with } a \text{ the } \delta_i \text{ coefficient (either 1, 2 or 3)} \quad (2.9)$$

An equation set is derived for each δ_i (i.e. each column in Figure 2.3), and a total of four equation sets, independent from each other, is available (such as the one in Equation 2.10). All sets can be found in Appendix A (the SSAES case $(r, c) = (4, 4)$ has the same equations).

$$\begin{aligned} 2\delta_1 &= S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) \\ \delta_1 &= S^{-1}(x_{14} \oplus k_{14}) \oplus S^{-1}(x'_{14} \oplus k_{14}) \\ \delta_1 &= S^{-1}(x_{11} \oplus k_{11}) \oplus S^{-1}(x'_{11} \oplus k_{11}) \\ 3\delta_1 &= S^{-1}(x_8 \oplus k_8) \oplus S^{-1}(x'_8 \oplus k_8) \end{aligned} \tag{2.10}$$

For each equation, x_i and x'_i are known, as they are nibbles of the fault free and faulty output respectively, but the respective δ and k_i are unknown. In order to solve the equation, a key hypothesis must be made, as well as a guess for the value of δ. Algorithm 2.2 details how the key candidates are derived from the equation sets.

It should be noted that $\delta_i \neq 0$, as otherwise it would mean that $x_i = x'_i$, meaning that the fault injection would have been unsuccessful. The algorithm outputs a set of key candidates. Thanks to the discard mechanism described in Algorithm 2.2, each equation set generates, on average, 256 32-bit long partial key candidates. The combination of all the partial key candidates therefore constitutes the remaining key space, which is of size $(2^8)^4 = 2^{32}$. While they can be brute forced by simply encrypting a known plaintext and checking if the output ciphertext matches the known correct plaintext, a second attack step can be performed to further reduce the key space.

Second DFA Step

In order to further reduce the key space, more equations are needed. From Figure 2.3, it is possible to distinguish another possible meeting point for the DFA. The differential state after the second to last *MixColumns* operation has a similar structure has the later one. However, it depends on the ninth round key, and, as such, it can not directly be expressed with the last round key values, for which the previous key candidates have been derived. Nevertheless, it is possible to express the ninth round key in terms of the tenth round key

Algorithm 2.2: DFA algorithm for the AES (first step)

```
1  for i = 1..4 do                                              // for each equation set
2      for δ_i = 1..255 do                                      // for all δ_i possible values
3          for k_{j1} = 0..255 do      // for all k_{j1} (first equation key part) values
4              if EQ_1(k_{j1}) = true then                      // first equation has a solution
5                  for k_{j2} = 0..255 do
6                      if EQ_2(k_{j2}) = true then
7                          for k_{j3} = 0..255 do
8                              if EQ_3(k_{j3}) = true then
9                                  for k_{j4} = 0..255 do
10                                     if EQ_4(k_{j4}) = true then
11                                         Store the partial key candidate
                                             kc_i = {k_{j1}, k_{j2}, k_{j3}, k_{j4}}
12                                     else
13                                         Discard δ_i and proceed to the next value of δ_i
14                                     end
15                                 end
16                             else
17                                 Discard δ_i and proceed to the next value of δ_i
18                             end
19                         end
20                     else
21                         Discard δ_i and proceed to the next value of δ_i
22                     end
23                 end
24             else
25                 Discard δ_i and proceed to the next value of δ_i
26             end
27         end
28     end
29 end
30 Compute all the key candidates kc from all the combination of all the kc_i
31 return Set of all kc
```

by inverting the key schedule (see Equation 2.11, with κ_{10} the last round key constant in the key schedule).

$$\begin{pmatrix} k_1^9 & k_5^9 & k_9^9 & k_{13}^9 \\ k_2^9 & k_6^9 & k_{10}^9 & k_{14}^9 \\ k_3^9 & k_7^9 & k_{11}^9 & k_{15}^9 \\ k_4^9 & k_8^9 & k_{12}^9 & k_{16}^9 \end{pmatrix} = \begin{pmatrix} k_1^{10} \oplus S(k_{14}^{10} \oplus k_{10}^{10}) \oplus \kappa_{10} & k_5^{10} \oplus k_1^{10} & k_9^{10} \oplus k_5^{10} & k_{13}^{10} \oplus k_9^{10} \\ k_2^{10} \oplus S(k_{15}^{10} \oplus k_{11}^{10}) & k_6^{10} \oplus k_2^{10} & k_{10}^{10} \oplus k_6^{10} & k_{14}^{10} \oplus k_{10}^{10} \\ k_3^{10} \oplus S(k_{16}^{10} \oplus k_{12}^{10}) & k_7^{10} \oplus k_3^{10} & k_{11}^{10} \oplus k_7^{10} & k_{15}^{10} \oplus k_{11}^{10} \\ k_4^{10} \oplus S(k_{13}^{10} \oplus k_9^{10}) & k_8^{10} \oplus k_4^{10} & k_{12}^{10} \oplus k_8^{10} & k_{16}^{10} \oplus k_{12}^{10} \end{pmatrix} \quad (2.11)$$

Thanks to this relationship between the last two round keys, it is possible to invert the encryption one round further and obtain a new set of equations. It should be noted that inverting the ninth round requires inverting the last *MixColumns* operations, introducing new coefficients.

$$\begin{aligned} 2\tilde{f} = \; & S^{-1}(14(S^{-1}(x_1 \oplus k_1) \oplus (k_1 \oplus S(k_{14} \oplus k_{10}) \oplus \kappa_{10})) \oplus \\ & 11(S^{-1}(x_{14} \oplus k_{14}) \oplus k_2 \oplus S(k_{15} \oplus k_{11})) \oplus \\ & 13(S^{-1}(x_{11} \oplus k_{11}) \oplus k_3 \oplus S(k_{16} \oplus k_{12})) \oplus \\ & 9(S^{-1}(x_8 \oplus k_8) \oplus k_4 \oplus S(k_{13} \oplus k_9))) \oplus \\ & S^{-1}(14(S^{-1}(x'_1 \oplus k_1) \oplus (k_1 \oplus S(k_{14} \oplus k_{10}) \oplus \kappa_{10})) \oplus \\ & 11(S^{-1}(x'_{14} \oplus k_{14}) \oplus k_2 \oplus S(k_{15} \oplus k_{11})) \oplus \\ & 13(S^{-1}(x'_{11} \oplus k_{11}) \oplus k_3 \oplus S(k_{16} \oplus k_{12})) \oplus \\ & 9(S^{-1}(x'_8 \oplus k_8) \oplus k_4 \oplus S(k_{13} \oplus k_9))) \end{aligned} \quad (2.12)$$

$$\begin{aligned} \tilde{f} = \; & S^{-1}(9(S^{-1}(x_{13} \oplus k_{13}) \oplus k_4 \oplus k_9) \oplus 14(S^{-1}(x_{10} \oplus k_{10}) \oplus k_{10} \oplus k_{14}) \oplus \\ & 11(S^{-1}(x_7 \oplus k_7) \oplus k_{15} \oplus k_{11}) \oplus 13(S^{-1}(x_4 \oplus k_4) \oplus k_{16} \oplus k_{12})) \oplus \\ & S^{-1}(9(S^{-1}(x'_{13} \oplus k_{13}) \oplus k_4 \oplus k_9) \oplus 14(S^{-1}(x'_{10} \oplus k_{10}) \oplus k_{10} \oplus k_{14}) \oplus \\ & 11(S^{-1}(x'_7 \oplus k_7) \oplus k_{15} \oplus k_{11}) \oplus 13(S^{-1}(x'_4 \oplus k_4) \oplus k_{16} \oplus k_{12})) \end{aligned} \quad (2.13)$$

$$\tilde{f} = \begin{array}{l} S^{-1}(13(S^{-1}(x_9 \oplus k_9) \oplus k_9 \oplus k_5) \oplus 9(S^{-1}(x_6 \oplus k_6) \oplus k_{10} \oplus k_6) \oplus \\ 14(S^{-1}(x_3 \oplus k_3) \oplus k_{11} \oplus k_7) \oplus 11(S^{-1}(x_{16} \oplus k_{16}) \oplus k_{12} \oplus k_8)) \oplus \\ S^{-1}(13(S^{-1}(x'_9 \oplus k_9) \oplus k_9 \oplus k_5) \oplus 9(S^{-1}(x'_6 \oplus k_6) \oplus k_{10} \oplus k_6) \oplus \\ 14(S^{-1}(x'_3 \oplus k_3) \oplus k_{11} \oplus k_7) \oplus 11(S^{-1}(x'_{16} \oplus k_{16}) \oplus k_{12} \oplus k_8)) \end{array} \tag{2.14}$$

$$3\tilde{f} = \begin{array}{l} S^{-1}(11(S^{-1}(x_5 \oplus k_5) \oplus k_5 \oplus k_1) \oplus 13(S^{-1}(x_2 \oplus k_2) \oplus k_6 \oplus k_2) \oplus \\ 9(S^{-1}(x_{15} \oplus k_{15}) \oplus k_7 \oplus k_3) \oplus 14(S^{-1}(x_{12} \oplus k_{12}) \oplus k_8 \oplus k_4)) \oplus \\ S^{-1}(11(S^{-1}(x'_5 \oplus k_5) \oplus k_5 \oplus k_1) \oplus 13(S^{-1}(x'_2 \oplus k_2) \oplus k_6 \oplus k_2) \oplus \\ 9(S^{-1}(x'_{15} \oplus k_{15}) \oplus k_7 \oplus k_3) \oplus 14(S^{-1}(x'_{12} \oplus k_{12}) \oplus k_8 \oplus k_4)) \end{array} \tag{2.15}$$

The key candidates derived in the first attack step can then be used as input for the equations from Equations 2.12 to 2.15. Similarly to the first step, for a given $\tilde{f}$, each equation has a solution with probability $\frac{1}{2^8}$. Meaning that a given key candidate is solution of the equation set with probability $\frac{2^8}{(2^8)^4} = \frac{1}{2^{24}}$, since it is necessary to go through all possible values for $\tilde{f}$, and thus the remaining key space has a size of $\frac{2^{32}}{2^{24}} = 2^8$. Brute forcing such a small number of keys is trivial, proving the efficiency of this optimised two-step DFA.

2.1.5 Algebraic Fault Attacks

Induced fault can be used in conjunction with algebraic descriptions of the encryption scheme under attack, in order to successfully recover sensitive data, which might not be possible with only fault information. Fault attacks which combine both aspects are called Algebraic Fault Attacks (AFAs). As the name suggests, AFAs are at the crossroad between algebraic attacks, which are attacks based solely on the algebraic properties of an encryption algorithm, and break the cipher at a theoretical level, and more conventional fault attacks, which only consider fault affected values in order to retrieve the secret key. They have the advantage of having a high automatisation potential compared to either of the other two types of attacks they are related to, which make them a prime choice as an automated tool for the evaluation of cryptographic hardware.

In the context of this thesis, it is therefore important to introduce the core principle behind AFAs, as well as how are such attacks practically mounted. Moreover, different type of AFA frameworks exist and they are not all equal in their abilities. In this regards, Section 2.1.5.1 is dedicated to the theory behind this type of attack, while Section 2.1.5.2 offers an overview of the state-of-the-art AFA frameworks available at the time of writing. The `AutoFault` framework presented in this thesis, as well as the concept of hardware-based AFAs, are mostly excluded here, as this is one of the research contributions of this work (Chapter 4), and a direct comparison to other frameworks is given in Section 4.9.

2.1.5.1 Principle of Algebraic Fault Attacks

The first AFA was introduced in 2010 by Courtois et al. [CJW10]. The initial goal of AFAs was to reduce the number of needed fault injections for a successful attack, as well as simplifying the solving process. The original authors pointed out that both the hardware fault injections themselves, and the cryptanalysis work necessary for an attack, are becoming harder with each new cipher. To this end, reducing the number of required fault injections would allow for less costly setups and less efforts spent on repeating what is often a difficult task. At the same time, improving the solving process would increase the scope of attacks which may be performed. Those are exactly the respective advantages of algebraic cryptanalysis and fault injection attacks (or SCA in a more general way). On the one hand, algebraic cryptanalysis, by its nature, is computationally intensive, as the encryption scheme has to be broken on the theoretical level, but does not require any specific external data. On the other hand, fault attacks, such as DFAs, require much less processing power to be successful, but are data dependent, and may need a large number of fault injections to be able to recover the secret key. It would therefore seem logical to combine both type of attacks in order to benefit from both of their advantages. This is the main idea behind AFAs: using the cipher description, as well as some fault injection data, in order to retrieve the sensitive information contained within the encryption scheme.

In order to mount a successful AFA, there are therefore three major steps.

1. Choose or find a suitable fault model

2. Express the cipher and the chosen fault in a proper algebraic formalism, according to the selected solver

3. Run the solver and eliminate the incorrect key candidates if necessary

It should be clear that, for any considered cipher, not every possible faults can result in a successful attack. Similarly to DFAs, or any fault injection attack, the fault location and position are an important factor for AFAs as well, and even more so if the complexity of the cipher is high for the chosen solver. The first step of finding a fault model which reduces significantly enough the key space is therefore of the utmost importance. To achieve this first step, there are different possibilities. If the encryption scheme is well known by the attacker, for instance, for evaluation purposes by the cipher designer, it can be known which location is the most susceptible to be vulnerable. In a similar way, if the encryption algorithm is part of a larger family, such as SPN schemes, the attacker may have some insights on possibly successful locations, for example, three rounds before the output in many SPN cases. However, this is not always the case, and manually finding a correct fault model for newly developed encryption schemes is not an easy task. To this end, one may evaluate the size of the key space for a given fault model. This is also a functionality of several AFA frameworks. In order to evaluate the key space, given a fault model, the cipher is expressed in the correct format for the chosen solver (similarly to step 2 of an actual attack, only with fewer equations), and either the number of possible solutions are counted, or the fault propagation patterns studied, in order to see if the considered fault model could lead to a sufficient key space reduction. By trying different fault locations and positions, the attacker can chose the most relevant fault model. This is however usually only an estimation of the remaining key space, and does not necessarily guarantee a successful attack if the fault model which has the smaller estimated remaining key space is chosen.

Once a fault model is chosen, the second step of an AFA is to derive a set of equations which describe the ciphertext, as well as the fault effects. These equations have to be in the correct format for the chosen solver. For instance, if the solving technique of the AFA is SAT solving, the formulas which describe the cipher have to be CNF clauses. SAT solvers are most widely used in AFA literature, however other type of solvers can be used instead,

such as algebraic solvers like SAGE [SJ05]. In this case, formulas would be in Algebraic Normal Forms (ANF). Generally, encryption algorithm are given in a behavioural manner. For example, the substitution layer is often given directly as an SBox look-up table. The expression of each operation in the required format is thus not always straightforward. Several techniques can be employed to express every encryption operations, but the most commonly used representation method is the polynomial form called ANF. 4-bit SBoxes can for example be expressed with a set of four AND equations [KM10]. Despite the fact that ANF descriptions are commonly used, and even if it would seem logical to use an algebraic solver, the memory requirements of such solvers, as well as their often slower runtime compared to in most cases SAT solvers, do not make them a good candidate for solving. Instead, converting ANF formulas to CNF clauses, in order to be able to use a SAT solver is more advisable, and the most common practise in the literature. The conversion methods, as well as the original ANF derivation, should be carefully performed, as introducing too many variables, or having too many clauses can lead to an increase in solving time. It is also possible to derive directly CNF clauses from the cipher description, and this is especially true in the context of this work, where the CNF clauses are derived directly from the hardware implementation (more details in Chapter 4). Moreover, the fault itself, and its effect on the computation also need to be mapped to the correct format. This is usually a much simpler process, especially in the case where the considered fault is an additive fault, modelled as an XOR. The fault model should also take into account if the exact fault position is known (i.e. which bits of the considered intermediate state). The knowledge of the fault position has to be mapped to the formulas as well. This can either be done by introducing new variables, which are either 1 or 0 if the fault manifests itself at the respective position, or by adding a complete fault vector over the full intermediate state and treating it as an input.

The previous two generic sets of equations, for the cipher description, and the fault model, can then be used in conjunction with some plaintext-ciphertext pairs to create the final sets for equations representing the AFA. It should be noted that a complete description of the cipher may not be required. For example, only representing the last rounds of an SPN may be sufficient to mount a successful attack. It may also be more efficient to only consider a partial representation of the encryption scheme, as less equations over less variables

are produced that way, reducing the overall solving complexity. Furthermore, in order to constrain the key space to only a single key candidate, another set of equations may be added. If a set of equations based on a known correct plaintext-ciphertext pair is added, only the correct key will satisfy all the equation sets.

Finally, and the last step of an AFA, the equation sets, or CNF clauses, can be fed to the solver of choice, which should return at least a single key candidate. In the case where multiple key candidates are returned, they need to be processed in order to find the correct key. This processing can easily be done by encrypting a known plaintext with the key candidate and verifying if the obtained ciphertext matches the corresponding known correct ciphertext. If both ciphertexts match, then the correct key was found, and no further processing is needed (if the key candidates are verified iteratively). It may be more efficient to proceed that way, instead of constraining the key space by adding more equations, such as is the case in step 2, if a set of equations for a correct encryption is added. The more equations are added, the longer is the processing time, while a single encryption for the verification of the key candidate can be done easily and swiftly. For SAT solver specifically, this may also change the way CNF clauses are processed, depending on the SAT solver heuristic.

Let's consider the original example of the AFA on the DES. The first step is to specify a fault model. At the time, some of the best fault attacks required bit-flip fault injections at round 16 of the DES. Due to the way the faults propagate in the cipher, this resulted in thousands of faults being needed for a successful key recovery. The fault model of the first AFA presented in [CJW10] however considered faults injected at round 13 and 14. Such faults would propagate in a way which would affect all the SBoxes of the DES, and, as such only 1 or 2 fault injections would be required, as opposed to thousands, which is much more realistic from an actual attack point of view. The attack with a single fault at round 13 is especially interesting, since the considered fault is a single bit-flip, as opposed to two double bit-flips, for the attack at round 14, but both attacks work in the exact same manner. With this fault model selected, the authors expressed the DES SBoxes as a system of cubic equations, which were then converted to CNF. The sets of CNF clauses with the correct plaintext-ciphertext pairs considered (either a single pair, or two pairs respectively, for the previously mentioned fault models), were created, achieving the second step of

the AFA. Finally, the clauses were fed to a SAT solver, with the addition of several key bits, in order to make the processing time shorter. In the case of the attack at round 13, if 24 key bits are known prior to the attack, the attack in [CJW10] was found to require 0,01 hours. The complete attack would therefore take approximately 2^{19} hours. While this first attack AFA may not seem impressive by the solving time alone, it laid the bases for much more powerful attacks and automated frameworks, such as the hardware-oriented one presented in this thesis, which follows the same three major steps.

2.1.5.2 State of the Art on Algebraic Fault Attack Frameworks

Over the last decade, a few AFA frameworks have been proposed, following the the first attack from Courtois et al. [CJW10]. Those frameworks follow the same principle, even though they differ in the way they handle the problem of AFAs. In this section we will review some of the current approaches concerning the automatic construction of AFAs, as an overview of the state-of-the-art techniques similar to this thesis's related work. First, it is important to distinguish two types of frameworks: key space evaluation frameworks, which only realise the first step of AFAs, and fully automated AFA frameworks, aimed at completely solving a given AFA instance. In this thesis, frameworks which are capable to automatically recover the secret key are denoted as AFA solvers. Concerning the former case, two frameworks have been proposed: the XFC framework [KRH17] and also the one described in [SKMD17]. Both frameworks can be used to find suitable scenarios for the realisation of fault attacks, and focus on the evaluation of the remaining key space for a given fault model. They were specifically designed to this extent, in order to make fault injection attacks easier. They are not automatic, as they require to manually construct the attack after finding an appropriate fault model. As for fully fledged AFA solvers, in addition to the one presented in Chapter 4, three frameworks have been proposed in the last years [ZZG+13, ZGZ+13, ZGZ+16], all based on the same principle, while the `AutoFault` framework is hardware-oriented.

Frameworks for Key Space Evaluation

As discussed in Section 2.1.5.1, the first step in AFA, and more generally fault injection attacks, is to identify a suitable location and position for a fault injection. It is therefore

extremely beneficial to be able to automatically derive fault injection parameters, which are likely to result in a successful attack. The XFC framework proposed in [KRH17] offers a colouring-based approach to fault characterisation for block ciphers.

XFC takes as input the specification of the considered block cipher, as well as the fault model which should be evaluated. The encryption scheme description consists of the different operations composing the block cipher under attack, namely linear and non-linear functions. By making the distinction between linear and non-linear functions, according to their respective inputs, and forwarding this information to XFC, the framework is capable of generating a colour-based cipher description, which describes the fault propagation properties of the considered fault model. Different colours are used to trace the fault effects at different position of the intermediate states, related to how useful the fault information should be. In more detail, when a fault model is considered, and a fault is injected accordingly, XFC assigns a new colour to the affected parts of the block cipher. For each subsequent operations, and depending on their inputs, as well as their linearity, the framework assigns once again a new colour for their outputs, if there is some fault propagation. While this stage of the evaluation is a good visual clue for the effectiveness of a given fault model, it is not sufficient for a proper evaluation. To this end, the next step takes the coloured output of the first stage and estimates the remaining size of the key space. In order to give such an estimation, and evaluate the complexity of the related attack, the framework approaches the problem in a backward fashion, referring to the previously generated colours of the states and nibbles. Each colour refers to a specific variable and XFC tries to derive related equations which may lead to the recovery of some portions of the secret key. The process itself needs some additional inputs from the user, and is as such only partially automated, but it allows for a good estimation of the attack complexity. XFC was successfully applied to different block ciphers, including AES. In this specific case, if the fault is injected in one byte of the beginning of the 8^{th} round, the size of the remaining key space found by XFC is the same as the one from the conventional single fault DFA from [TMA11]. This result strengthens the fact that automated tools can find suitable attack scenarios, which otherwise take a significant amount of time to manually craft, and showcases the correctness of the framework. More precisely, the overall flow of XFC is exactly what is performed in the case of DFA, only with an automated first

step, which gives an estimation of the attack's, and also the fault model's, potential, but still requires to implement a solver for any reasonable output of XFC.

While colour-based approaches are functional and can lead to good key space estimations in some cases, they are limited. One example of this would be the fact that XFC would not consider a fault injection at the beginning of round 7, in the case of the AES, to be a suitable fault model. This was shown by the authors of a different key space evaluation framework in [SKMD17] which overcomes this issue. Their tool supports additional fault models compared to XFC, but also removes the need for extra inputs from the user in order to produce a good evaluation. As such, the framework fully automates the evaluation of the remaining key space for a given fault model. This is achieved by using some data mining-like approaches according to the fault propagation, instead of colouring the intermediate states. The framework takes as input a functionnal description of the cipher and a fault model, similarly to XFC. However, more fault models are supported compared to XFC, which makes it more versatile. In order to output a an estimation for the size of the key space, the framework goes through three stages. First, a distinguisher for the attack needs to be identified. Once this is the case, a divide and conquer strategy on the various operations of the cipher is performed in order to identify the key parts involved in faulty computations. Finally, this information is used to provide an evaluation of the remaining key space, and thus the feasibility of the attack. The first step of identifying the distinguisher is performed by computing states entropies related to the chosen fault model, and comparing them to the maximum state entropy, in the fault-free case. Using the measure of entropy, instead of a colouring approach, allows for a better evaluation of the distinguishers. The entropy gives an indication on the fault propagation paths, as well as the automatically considered distinguisher. In other word, a suitable differential state is a distinguisher if its entropy is inferior to the maximum entropy of the same state. This identification stage is especially important as it removes the need for additional user inputs to find a differential state, given a fault model. This first stage returns several potential candidates for a distinguisher, which are then processed by the divide and conquer step for identification of the related key bits involved. To do so, the cipher description is broken down into a graph, and each operation in itself into another sub-graph. The overall graph is denoted as cipher dependency graph and is used to evaluate the compatibility

of the considered distinguisher with the fault model. The graph is searched to identify which key parts are present on the faulty path and in the computation of the distinguisher. Thanks to this step, the portions of the key which are not involved in the computation of the differential distinguisher can also be identified, and thus the size of the key space which needs to be guessed. Lastly, this information is forwarded to the last stage, which computes the final reduction of the key space, also considering the information which can be derived from the best differential state. Contrary to XFC, the evaluation not only provides an evaluation of the size of the remaining key space, but also an estimated number of faults which may be required to achieve a successful key recovery. The framework was applied to the AES, as well as PRESENT, and, as opposed to XFC, it was able to find a suitable attack scenario at the beginning of round 7, with an estimation of the key space's size of roughly $2^{32} - 2^{26}$. Despite the fact that the authors did not provide any information on the required number of faults for this attack, it should be noted that this attack was not found by XFC, which shows more potential for entropy-based methods, rather than colour-based ones. However, and similarly to XFC, even if a suitable fault model and distinguisher are found, the attack itself still needs to be manually implemented.
Both previous frameworks constitute a first step towards the automated construction of fault attacks, but they do not proceed to implement an actual attack and only provide a complexity estimation for the attack at the found location and position.

Complete AFA Solvers

The complete automation of fault attacks requires not only an evaluation of the key space for a given fault model, but also methods to automatically solve selected fault injection attack's instances. To this end, fully fledged AFA solvers require a description of the cipher as input, as well as the fault model, usually in a CNF format. This allows for automated solving of given attacks, since, after some processing, the inputs can be forwarded to a solver, often a SAT solver, and therefore no manual implementation of an attack is needed. With the exception of the `AutoFault` framework (presented in Chapter 4), three AFA frameworks are available at the time of writing [ZZG+13, ZGZ+13, ZGZ+16]. They all must be provided with a suitable fault model, as this is the first major step of any AFA. In the case of these frameworks, the first input is a functional description of the cipher

(unlike `AutoFault`), which can be achieved through know methods to derive algebraic expressions of encryption operations, and then convert them to CNF, since all of the considered frameworks use SAT solver to recover the secret key. The second input is the fault model. The framework presented in [ZZG+13] assumes a specific random nibble fault model. The other two tools do not consider specific fault models, and only require them to be given by the user.

The first complete AFA framework was presented in [ZZG+13]. The authors propose a tool aimed at attacking the Piccolo cipher [SIH+11], which can however be extended to other lightweight ciphers. Both the functional description of Piccolo and the fault model are expressed as ANF formulas. The framework itself is restricted to random nibble faults in the 23^{rd} round of Picollo, which is the chosen fault model for the attack. This is however a restriction of this specific framework, to only support random nibble faults. Additionally, the authors proposed a method to represent the fault injection in ANF format. A new variable is introduced for each fault injection and is expressed as an XOR in the equations, while another set of variables represents the presence, or the omission, of a fault at a chosen position. Once all the equations for both the cipher description and the fault model are derived, they are converted to CNF and fed to a SAT solver (in this case CryptoMiniSaT [SNC09]). The SAT solver then proceeds to fine a satisfiable assignment for the secret key given the known plaintexts and ciphertexts, as well as the fault position, thus recovering the correct key. While the authors evaluated their framework on both the encryption and the decryption of the Piccolo cipher, the framework was only capable to recover the secret key for the decryption process. The key was however recovered with a single fault injection in around five hours, and two fault injections would decrease the solving time to two hours. The authors also provide additional benchmarks for a few other ciphers, but it should be noted that the extension of the framework to new ciphers requires some amount of work, as the equation creation step is not fully, but only partially automated, and cipher plus fault model dependent. The partial automation is however a crucial step towards fully automated AFA solvers, and the newly introduced method for ANF fault modelling can be generalised for any cipher.

In [ZGZ+13], a new approach to AFA was suggested. The denoted algebraic differential fault analysis (ADFA) builds upon the previous framework by introducing DFA equations

to the inputs of the solver. The framework focuses on attacking the LED cipher, but can be extended to any other cipher and does not consider a restrictive fault model. Similarly to the previous tool [ZZG+13], ANF equations are created for both the fault model and the cipher description; extra equations for reversed operations, such as an expression for the inverse SBox, are created as well. This step is especially important for the introduction of DFA-based equations in the ANF formulas, as differential fault equations require reversing some operations (as described in Section 2.1.4.1). During the solving process, those DFA equations are processed in a way which is similar to going backwards for a portion of the encryption and meeting at a given differential location. This process is similar to what is done in DFAs, or in the XFC framework [KRH17], for the evaluation of the key space. Once again, all the equations, including the DFA-based equations, are converted to CNF and fed to a SAT solver (CryptoMiniSat). The attack on LED, with a solving time constraint, shows a 97.2% success rate, considering a fault injection at the beginning of round 30 in a single 4-bit nibble. The experimental results therefore shows a good performance improvement compared to standard DFA attacks, but also showcase the possibility to add additional cryptanalytic input to conventional AFA to improve the performances in this case as well. Moreover, the key space reduction was improved, by almost two orders of magnitude, compared to DFAs, supporting the fact that ADFA automation is an efficient method for automated attacks of ciphers.

More recently, a framework for the evaluation of AFAs on lightweight ciphers was introduced [ZGZ+16]. This specific framework expands on the framework proposed in [ZZG+13], making it more versatile and not cipher restricted. To showcase this versatility, the authors successfully applied their framework to various lightweight ciphers and fault models. Similar to any other AFA, the first step of creating equations for the cipher description and the fault model is the same, and they are fed to CryptoMiniSat. However, the fault model is not constrained to any specific one, contrary to the first framework [ZZG+13]. An interesting feature of this specific framework is the presence of two solving modes. Similarly to the two previous frameworks, Mode A uses the CNF clauses derived from the cipher description and the fault model to retrieve the secret key. However, Mode B doesn't take any plaintext-ciphertext pairs as input, and instead uses a modified version of the SAT solver to evaluate the size of the remaining key space, given a fault model, like

in [KRH17, SKMD17]. As discussed previously, this can be an important feature to automate fault attacks, and more specifically AFAs, as first an evaluation step can be performed to choose an appropriate fault model, further increasing the degree of automation. Moreover, this specific framework is much more versatile than any other previously mentioned frameworks, due to the large number of possible inputs. Not only the framework supports unrestricted fault models, but it also can be applied to various ciphers, and also support additional inputs (similarly to [ZGZ+13]). The authors considered multiple fault injection scenarios, such as an injection during the key schedule, or in the round counter itself, in addition to the more conventional fault at the beginning of a round. Similarly, bit-based and nibble-based fault models were analysed, as well as several different SPN ciphers (LBlock, DES, PRESENT and Twofish). In each case, the proposed framework managed to solve the different instances, with faults at different locations and positions, all in an automated fashion. This supports the versatility of AFA frameworks, especially in the case of unconstrained inputs.

In the context of this thesis, the focus will be on the hardware-oriented AFA framework `AutoFault` for automated evaluation of hardware implementations of cryptographic primitives. Chapter 4 will describe the framework in more details. However, the frameworks presented in this section already showcase the usefulness of AFA framework for the creation of automated fault attacks. From key space evaluation to complete solvers, the data on the different attacks, or at least fault models, can already be used to evaluate ciphers to a certain degree, even without considering their hardware implementations. Moreover, the versatility of AFA frameworks, such as [ZGZ+16], further strengthens the finding that the automation of AFA can be used to remove the need for manually crafted fault equations for a large number of ciphers, which is a time consuming and difficult step for cryptanalysts.

2.2 Background on Error Correcting Codes & Other Counter-Measures

Physical attacks introduced in Section 2.1 can be mitigated by using diverse counter-measures. One of such counter-measures, which is directed more towards fault attacks,

is the use of an Error Detecting Code (EDC), or an Error Correcting Code (ECC), module. While code-based counter-measure are widely used against a variety of different faults, not every code is equal and, from a security perspective, more specialised EDC architectures should be used. In this section, a brief background on conventional code-based counter-measures will be given, before a discussion on security-oriented codes, and more precisely the Rabii-Keren (RK) code, as it is the base of the architectures presented in Chapter 3 of this thesis. Additionally, a few subsequent counter-measures against fault attacks, as well as other side-channel attacks, will be discussed, in order to better understand their characteristics, and the relevance of the implementations analysed in Section 4.8.

2.2.1 Error Detecting & Correcting Codes

In a general context, EDCs and ECCs are used to ensure that the correct data is transferred over a communication channel. They were developed to avoid data loss or corruption over noisy channels. For example, if a binary message is sent over a noisy channel, some bits which were initially 0s in the original message may be flipped to 1s. To ensure that the correct message is transmitted, some redundant elements are introduced. The added redundancy, while costing some extra data bits to be transmitted, can be used to either detect that an error occurred, or, in the case of ECCs, to even correct the error (or errors) which may have been introduced during transmission. In the case of detection only, the redundancy bits can be checked in order to verify that the correct message has been received, by encoding the data one more time and verifying that the redundancy bits match the one received (only possible for codes where redundancy bits are easily identifiable), or by using a decoding algorithm. If the considered code has some correction capabilities, a decoding algorithm can be used too, in order to compute the potential error which may have occurred, and correct it. However, ECCs can not correct any arbitrary error. They are limited by their error correction capability, and can only correct up to a certain number of errors, highly dependent on the size of the introduced redundancy.

In the context of this thesis, EDCs and ECCs are considered for security applications, and more especially protection against fault injection attacks. A fault can be considered in the same way as a disruption on a communication channel, since, at a given instant during the runtime of an encryption, a value is altered, similarly to how a transmitted message

over of noisy channel can be modified. In this regard, and especially for this section as well as Chapter 3, the notion of faults or errors have the same meaning. Both words are used to describe the disruption that occurs during runtime, being malicious or not.

Some key notions concerning EDCs and ECCs first need to be introduced. Let's consider a code $\mathcal{C}$, which may relate to either an EDCs or an ECCs. $\mathcal{C}$ is defined over an alphabet $\mathcal{A}$. The alphabet consists of all the symbols used to express a word. For instance, binary codes are expressed in binary, as their name suggests, and therefore their alphabet is $\mathcal{A} = \{0, 1\}$. Generally, $\mathcal{A}$ is a finite field $\mathbb{F}_q$ of size q, often a power of a prime number. A code of this form is denoted as q-ary code (or binary in the case where $q = 2$). Therefore, a q-ary code $\mathcal{C} : \mathcal{A}^k \rightarrow \mathcal{A}^n$, which is equivalent to $\mathcal{C} : \mathbb{F}_{q^k} \rightarrow \mathbb{F}_{q^n}$ in the context of this thesis, is a subset of size $|\mathcal{C}|$ of the vector space $\mathbb{F}_{q^n}$ of dimension n over $\mathbb{F}_q = GF(q)$. From this definition, it can be noted that k is the size of the initial data in terms of symbols of $\mathcal{A}$, while n is the corresponding size of the encoded data. Consequently, the number of redundancy symbols r is defined as $r = n - k$. Any word $c \in \mathcal{C}$ is called a codeword. Additionally, the distance d of a code is defined as the minimum Hamming distance, in terms of symbols of $\mathcal{A}$, between two distinct codewords of the considered code $\mathcal{C}$.

$$d = \min_{\substack{c_1, c_2 \in \mathcal{C} \\ c_1 \neq c_2}} \{\Delta(c_1, c_2)\} \quad \text{where } \Delta \text{ is the Hamming distance function} \tag{2.16}$$

A code $\mathcal{C}$ can be identified as an $[n, k, d]_q$ in order to quickly identify its parameters.

The distance d of a code $\mathcal{C}$ defines the detection capabilities of the code. Since, by definition, the minimum distance between two codewords is d, any word c' such that $\Delta(c, c') < d$, with c a valid codeword, is not part of $\mathcal{C}$, and $c' \notin \mathcal{C}$ is thus a non-codeword. This means that any $[n, k, d]_q$ code has a detection capability of at least $d - 1$ symbols over $\mathbb{F}_q$. Similarly, for ECCs, the error correction capability t can be defined according to d. If an invalid codeword c' is received instead of c, then the closest valid codeword in terms of Hamming distance is a good candidate for c, however this candidate may not be unique. In order for the candidate to be unique, and therefore the correct c, it should be clear from the definition of d that $\Delta(c, c') < \lfloor \frac{d-1}{2} \rfloor$ should be satisfied. Consequently, $t = \lfloor \frac{d-1}{2} \rfloor$ for any code of distance d.

Different types of codes exist, however, in this thesis the focus is on linear codes and security-oriented codes, which are derived from them. A linear code $\mathcal{C}$ over a vector space $\mathbb{F}_{q^n}$, is a vector sub-space of $\mathbb{F}_{q^n}$. That is to say that every linear combination of codewords of $\mathcal{C}$ is a valid codeword. In the case of a linear code $[n, k, d]_q$, the sub-space is of dimension k. A basis of the sub-space is a set of linearly independent codewords, of which the linear combination can generate any other codeword of $\mathcal{C}$ (for example, the set of all c_i in Equation 2.17). A basis of a linear code is however not necessarily unique.

$$\mathcal{C} = \sum_{i=0}^{k-1} a_i c_i : a_i \in \mathbb{F}_q \; ; \; c_i \in \mathcal{C} \tag{2.17}$$

For a given basis, it is possible to create a generator matrix of the linear code. Since any codeword is a linear combination of the k vectors of the basis, then let's define G as follows:

$$G = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{k-1} \end{pmatrix} \quad \text{with } \{c_i = (c_{i,0}, .., c_{i,n-1})\} \text{ a basis of } \mathcal{C} \text{ over } \mathbb{F}_{q^n} \tag{2.18}$$

Any vector $v \in \mathbb{F}_{q^k}$ can therefore be encoded by performing a multiplication by the generator matrix G.

$$vG = (v_0, .., v_{k-1}) \begin{pmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,n-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,n-1} \\ \vdots & \vdots & & \vdots \\ c_{k-1,0} & c_{k-1,1} & \cdots & c_{k-1,n-1} \end{pmatrix} = c \quad \text{with } c \in \mathbb{F}_{q^n} \text{ a valid codeword} \tag{2.19}$$

Another useful matrix which can define a linear code $\mathcal{C}$ is the parity-check matrix H. The parity-check matrix is the generator matrix of the dual-code of $\mathcal{C}$, denoted as $\mathcal{C}^\perp$. $\mathcal{C}^\perp$ is a vector sub-space of $\mathbb{F}_{q^n}$, for which all vectors are orthogonal to any vector of $\mathcal{C}$. In other words:

Definition 2.1. Let $\mathcal{C}$ be a linear code over $\mathbb{F}_{q^n}$. Then $\mathcal{C}^\perp$, the dual-code of $\mathcal{C}$, is defined as follows.

$$\mathcal{C}^\perp = \{c^\perp \in \mathbb{F}_{q^n} \mid c^\perp \cdot c^\top = 0 \quad \forall c \in \mathcal{C}\}$$

The resulting dual-code is of dimension $n-k$ and, if H is a parity-check matrix, then it satisfies $cH^\top = 0_{\mathbb{F}_{q^{n-k}}} \quad \forall c \in \mathcal{C}$. In particular, the parity-check matrix can be used to verify if an error occurred thanks to the previous expression. Let's consider an input vector x, and its corresponding encoded codeword $c \in \mathcal{C}$. If an error ϵ occurred, then c is modified into c' and can be modelled as $c' = c + \epsilon$. Unless c' is also a codeword (e.g $c' \in \mathcal{C}$), then $c'H^\top \neq 0_{\mathbb{F}_{q^{n-k}}}$ and therefore ϵ can be detected. More precisely, and for a linear code, we get the following equation:

$$c'H^\top = (c+\epsilon)H^\top = cH^\top + \epsilon H^\top = 0_{\mathbb{F}_{q^{n-k}}} + \epsilon H^\top = \epsilon H^\top = s \tag{2.20}$$

In this case, $s \in \mathbb{F}_{q^{n-k}}$ is called the syndrome of ϵ. The syndrome can be used by a decoding algorithm to try to correct the error ϵ. However, if ϵ is unknown, which is the case in most scenarios, $s = \epsilon H^\top$ doesn't have a unique solution (n unknowns, for $n-k$ equations), and therefore recovering the correct codeword c is not a trivial task. For this purpose, many different techniques and algorithms exist, but if the error affected less symbols than the error correction capability of the code, then ϵ, and thus c, are uniquely computable.

A linear code can be defined by both matrices, and thanks to a generator matrix, the codewords of a linear code $\mathcal{C}$ can be easily computed. Yet, the resulting codeword may be of an unpractical form for some applications. It is possible to express any codeword with the information portion and the redundancy portions distinct from each other. This form is called systematic form. $\mathcal{C}$ is a systematic code if every codeword is of the form $c = (x, w(x))$ where $x \in \mathbb{F}_{q^k}$ is the information portion and $w \in \mathbb{F}_{q^{n-k}}$ is the redundancy portion. In this case, the generator matrix G of the linear code in its systematic form is constituted of the identity matrix $I_k \in \mathbb{F}_{q^{k\times k}}$ and another matrix $A \in \mathbb{F}_{q^{k\times(n-k)}}$.

$$G = (I_k|A) \tag{2.21}$$

If this is the case, the corresponding parity-check matrix can be easily derived.

$$H = (-A^{\top}|I_{n-k}) \tag{2.22}$$

Therefore, expressing a linear code in its systematic form is preferable and such a form always exists, since a basis of $\mathcal{C}$, and the corresponding generator matrix, can always be reduced in that way through Gaussian elimination. Example 2.1 shows an example of conversion to systematic form.

Example 2.1. *Consider the Hamming code* $[7, 4, 3]_2$. *The generator matrix of the code is:*

$$G = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{matrix} \leftarrow l_1 \\ \leftarrow l_2 \\ \leftarrow l_3 \\ \leftarrow l_4 \end{matrix}$$

Let's apply Gaussian elimination to G *to express the generator matrix of the code in its systematic form.*

$$G_{sytematic} = \left(\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right) \begin{matrix} \leftarrow l_3 + l_4 \\ \leftarrow l_2 + l_4 \\ \leftarrow l_1 + l_2 + l_3 \\ \leftarrow l_2 + l_3 + l_4 \end{matrix}$$

There are different criteria for the efficiency of a code, such as the detection rate, which will be discussed more in details in Sections 2.2.1.1 and 2.2.1.2, but one important criteria is the code rate. Implementation-wise, the number of added redundancy symbols can be an issue, especially in hardware, where resources are constrained. Therefore, the fewer redundancy symbols are present, the better it is from an implementation point of view. The ratio between the number of information and redundancy symbols is the code rate $R = \frac{k}{n}$. A higher rate hence indicate a more efficient code, since less redundancy symbols need to be computed.

Finally, and in order to evaluate EDCs and ECCs architectures, let's define the error multiplicity as the number of erroneous symbols in the data. If the error ϵ is defined as an

additive error, the error multiplicity is the number of non-zero elements of ϵ. The error multiplicity is especially important for the evaluation of the code, in terms of detection and correction capabilities, as will be discussed in Section 3.2.

2.2.1.1 Conventional Codes

Many different EDCs and ECCs are used for different applications, however, in the context of this work, the considered application is security, and more precisely the use of such codes as counter-measure against different attacks. In addition to dedicated security-oriented codes, such as the Rabii-Keren codes (RK) [RK17] or the Algebraic Manipulation Detection codes [CDF+08], some conventional EDCs are also used for security applications. Conventional codes can be problematic for certain attacks, but have the advantage to be simpler to implement and less costly. Thus, it is of particular interest to first understand such codes and discuss their drawbacks in comparison to security-oriented codes. Moreover, conventional codes are used as basic building blocks for security-oriented codes, for example the Bose–Chaudhuri–Hocquenghem codes (BCH) [BRC60], and are as such also of particular interest.

One of the first obvious way to protect a sensitive implementation would be to use some plain redundancy for the critical part of a cryptographic circuit. In 1956, Von Neumann introduced an ECCs based on n-modular redundancy [VN56]. The principle is simple, in order to correct an error which may occur during runtime of a component, this same component is duplicated $n-1$ times. The output of each of the modules is then given as input to a majority voting system, and the most recurring value is chosen as the correct output. In this case, n stands for the total number of redundant modules. Figure 2.4 shows a 3-modular redundant decoder, otherwise known as Triple Modular Redundancy (TMR). TMR ECC are the most commonly used modular redundancy systems. Form Figure 2.4, it should be clear that if an error occurs in a single module, but the other two are error-free, then the output will be the correct one. A TMR implementation can therefore handle a single erroneous module. More generally, an n-modular redundancy ECC can reliably correct up to $\lfloor \frac{n-1}{2} \rfloor$ errors, in terms of components (similarly to the error correction capability of linear codes). Despite this, TMR codes are widely used. This is due to the fact that, while more redundancy implies more correction capabilities, it also implies a higher

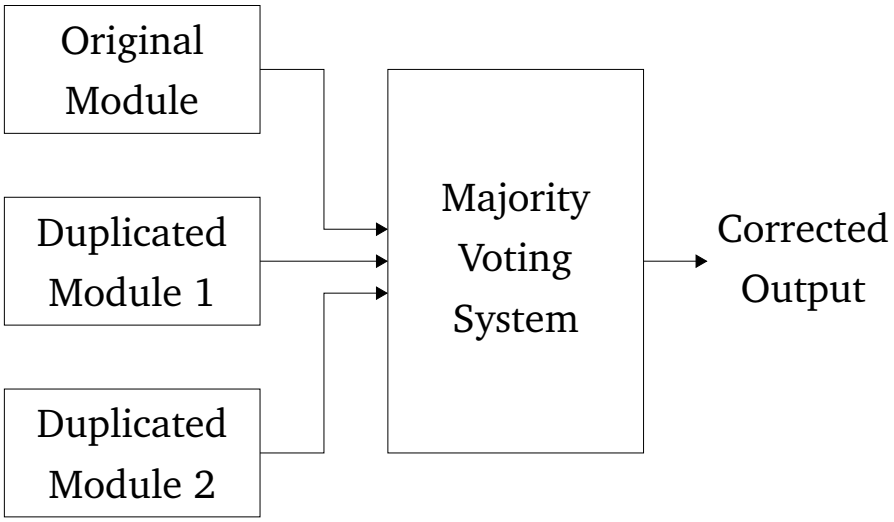

Figure 2.4: Triple Modular Redundancy Decoder [VN56]

implementation cost, which is often undesired. Moreover, in a security context, TMR have a major flaw. If an attacker is able to inject simultaneously twice the same fault, then the output of the TMR module would be the faulty output, and the attack would be successful. Since such attacks have been proven to be mountable in the recent years [SHS16], TMR should consequently not be used as a counter-measure against fault attacks.

Another simple way to, this time, detect errors, is to introduce a parity check bit at the end of the data which needs to be protected. This is denoted as a parity code. Parity codes compute the Hamming weight of the data modulus 2 and append the resulting bit at to the sensitive data. For example, let's consider that an attacker is targeting a 4-bit SBox of an SSAES [CMR05], which is protected by a parity code. If the input is 2 (0010 in binary), then the output is 5 (0101), and thus the parity bit, if an even parity is considered, is 0 (two ones in the original data). The output of the SBox should therefore be 01010 (the parity bit is introduced as the least significant bit). If an attacker is able to induce a single bit flip, for instance on the most significant bit, then the output becomes **1**1010, and it is evident that a fault was injected since the value of the parity check bit should be 1 according to the observed data (three ones in the information portion). However, parity codes suffer from the same drawback as TMR codes, since simultaneous faults can hide the occurrence of an error. In the case of parity codes, this goes even further, since faults which affect an even number of bits are undetectable. If we consider the same example

as previously, but with an error which flips the first two bits, then the output would be **10**010, which is a valid codeword. In terms of physical fault injections, faults which affect more than a single bit, and especially neighbouring bits, are the most common. Therefore, parity codes should also be avoided as potential counter-measure against fault attacks.

A related family of EDCs are the Hamming codes [Ham50]. However, contrary to the parity codes, the Hamming codes are capable of detecting up to two bit errors, and, more importantly, correcting single bit errors. They are also high rate codes, making them much more versatile than parity codes. Hamming codes are linear codes $[n, k, 3]_2$. Hamming codes can be extended to different size of inputs k, but the distance is always 3 (hence the detection and correction capabilities). Originally, the Hamming codes were defined according to the parity check bits position and the related bits they checked, but firstly, it is needed to determined the number of parity bits necessary for a given input size. Hamming codes require $r = \lfloor \log_2(k) \rfloor + 1$ redundancy bits to encode an input data of size k. Once the number of required redundancy bits has been chosen, the redundancy bits are placed at the positions $r_i = \log_2(2^i) + 1 \; \forall i \in \mathbb{N}^* \mid i \leq r$, while the information portion of the encoded word constitutes the remaining bits. Finally, each r_i is the sum modulus 2 of the bits such that the i^{th} bits of the binary expression of the position is 1 (of course, not counting r_i itself). For example, if $x = (x_1, .., x_n)$ is the encoded codeword, $r_1 = x_3 \oplus x_5 \oplus x_7 \oplus ...$, $r_2 = x_3 \oplus x_6 \oplus x_7 \oplus ...$, and so on. Due to the encoding, x can also be expressed as $x = (x_1, .., x_n) = (r_1, r_2, x_3, r_3, x_5, .., x_n)$.

In order to better understand the Hamming code, and later their vulnerability in a security context, let's consider the example of the Hamming code $[7, 4, 3]_2$, which is a commonly described and used instance of Hamming (and also taken as an example in the original paper [Ham50]). As per definition, this Hamming code requires $r = \lfloor \log_2(4) \rfloor + 1 = 3$ redundancy bits (as a linear code, it should be clear that $r = n - k = 7 - 4 = 3$). Let $m = (m_1, m_2, m_3, m_4)$ be the input vector, and $x = (x_1, x_2, x_3, x_4, x_5, x_6, x_7) = (r_1, r_2, m_1, r_3, m_2, m_3, m_4)$ the encoded codeword. From the definition of the code:

$$\begin{aligned} r_1 &= x_3 \oplus x_5 \oplus x_7 = m_1 \oplus m_2 \oplus m_4 \\ r_2 &= x_3 \oplus x_6 \oplus x_7 = m_1 \oplus m_3 \oplus m_4 \\ r_3 &= x_5 \oplus x_6 \oplus x_7 = m_2 \oplus m_3 \oplus m_4 \end{aligned} \tag{2.23}$$

Following this, if the binary input is $m = 0101$, then the encoded codeword is $x = 010\ 0101$. Let's now assume that x_4 is flipped by an error, then the codeword becomes $x' = 010\ \underline{\mathbf{1}}101$. In order to correct a single bit error, it is sufficient to only recompute the redundancy bits r_i''. The position of the error is then given as a binary value by $r_i' \oplus r_i''$ (most significant bits ordered from largest to smallest i). In the previous example, $r_1'' = x_3' \oplus x_5' \oplus x_7' = 0 \oplus 1 \oplus 1 = 0$, $r_2'' = x_3' \oplus x_6' \oplus x_7' = 0 \oplus 0 \oplus 1 = 1$, $r_3'' = x_5' \oplus x_6' \oplus x_7' = 1 \oplus 0 \oplus 1 = 0$ and the position of the single error is therefore at position:

$$\begin{aligned} p &= (r_3' \oplus r_3'', r_2' \oplus r_2'', r_1' \oplus r_1'') \\ &= (1 \oplus 0, 1 \oplus 1, 0 \oplus 0) \\ &= (1, 0, 0) \\ p &= 4 \end{aligned}$$

and the correct codeword is thus indeed x.

Hamming codes are linear codes and can therefore be defined by a generator matrix and a parity check matrix, and also in systematic form. Consequently, let's consider the previous example of the Hamming code $[7, 4, 3]_2$, reordered in its systematic form. As a reminder, the systematic matrix is as follows.

$$G = (I_4|A) = \left(\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right) \tag{2.24}$$

If this is the case, the corresponding parity-check matrix can be easily derived.

$$H = (-A^\top|I_3) = \left(\begin{array}{cccc|ccc} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right) \tag{2.25}$$

Still following the previous example, if $m = 0101$, then $x = mG = 0101010$ (which is the same value as previously with only the redundant portion being shifted at the end). Similarly, let's consider $x' = \underline{\mathbf{1}}101010$, then it is possible to compute the syndrome $z = x'H^\top = 110$ (and it should be noted that a codeword is declared error-free if $z = 0_{\mathbb{F}_{2^r}}$). z is equal to the first column of H, which means, thanks to the definition of the matrices and their respective bases, that the error occurred in the first bit. Flipping back the corresponding bit indeed leads to the correction of x' into x, and goes to show the convenience of using matrix representation and computation of a linear code.

Hamming codes are still widely used nowadays, for instance for ECC RAM, but also in a security context, as a counter-measure to mitigate fault injection-based attacks [SMG16, BKHL20, POTSC+20]. However, Hamming codes are linear codes, and as such, by definition, any sum of codewords is itself a codeword. This is a major vulnerability in terms of security. If an attacker knows that a Hamming code is used as counter-measure against fault attacks, then, if he is skilful enough, he can engineer a fault which is also a codeword. This error would therefore always go undetected. Let's take an example, using the previous matrices (Equations 2.24 and 2.25) and input 4-bit vector $m = 0101$. In this case, it is still true that $x = mG = 0101010$, but let's assume that the attacker, instead of injecting the faulty vector 1000000 (similarly to the decoding example), injects the fault $\epsilon = 0001111 = (0001)G$ (and thus is a valid codeword), as he knows about the ECC counter-measure. Then, since the Hamming code $[7, 4, 3]_2$ is linear, then Equation 2.26 applies.

$$z = x'H^\top = (x \oplus \epsilon)H^\top = mGH^\top \oplus \epsilon H^\top = \epsilon H^\top = (0001)GH^\top = 0_{\mathbb{F}_{2^3}} \tag{2.26}$$

As such, the fault injection would go undetected and this is true for any input $m \in GF(2^4)$. This scenario is also realistic as often a physical fault injection can flip the neighbouring bits of the targeted bit. Therefore, in this case, if the attacker wanted to flip the fourth information bit, then it may have also flipped the next three redundancy bits. Moreover, such an attack scenario can be applied to 4-bit SBoxes protected by the Hamming code $[7, 4, 3]_2$ (i.e in the case of an SSAES or the LED cipher).

This can be generalised to any linear code, since, in their case, a sum of two codewords is also a codeword. Consequently, linear codes should not be used in a security context, or at least with the knowledge that a skilful attacker may be able to completely circumvent the corresponding EDC or ECC counter-measure which has been implemented, as their usually lower implementation cost, compared to security-oriented codes, may be a good trade-off in the case of extremely hardware constrained devices.

The final family of conventional ECC of interest in this thesis is the Bose-Chaudhuri-Hocquenghem (BCH) codes [BRC60]. BCH codes are $[n, k, d]_{q^m}$ linear codes, with q a prime number and m an integer (as defined in Section 2.2.1). However, in the context of this thesis, the focus is on the case where $q = 2$, and hence only BCH codes over $GF(2^m)$ are of interest, but it should nonetheless be noted, for completeness, that BCH codes, in a broader scope, can be generalised to other finite fields, or even for q non prime. BCH codes are of particular interest as building blocks for the Rabii-Keren (RK) codes, which the ECC architectures presented in Chapter 3 are based on.

Without going into details, as the construction of linear codes is not the focus of this thesis, and the example of the Hamming codes (which is a particular case of BCH codes over $GF(2)$) was already discussed, for a given $n = 2^m - 1$ and a chosen distance d, there always exists a BCH code over $GF(2^m)$ [BRC60]. The code can easily be derived from the primitive polynomial of the considered Galois field, and the corresponding generator matrix G, and the respective parity check matrix H, can be computed. Moreover, since BCH codes are linear codes, those matrices can also be expressed in their systematic form, and is the form which we will consider through out the remainder of the thesis. Once both matrices have been generated for the considered case, depending on the length of the data to be protected and the alphabet, the encoded data can be computed by multiplying the input data by G. As for the decoding process, and the error correction, the syndrome can also be easily computed by multiplying the encoded codeword by the $H^\top$ and then different methods can be used to correct the codeword, as long as the Hamming weight of the error, in terms of elements of the alphabet, is lower than $t = \lfloor \frac{d-1}{2} \rfloor$. However, those techniques are usually costly, and in Section 3.3.3 a new method for efficient error correction will be presented.

One of the main advantage of BCH codes over some other codes, such as the Hamming codes, is their higher error correction capability. BCH codes can be constructed such that errors of higher multiplicities can be corrected, even though this of course comes at the cost of a higher number of redundancy symbols. Since they are linear codes, they achieve this in a rather simple way, as the code matrices G and H can be easily computed. However, the linearity of BCH codes lead to the same limitations as previously discussed. That is to say that the sum of two codewords is another codeword. As such, they are also not directly usable in a security context, if a strong attacker is considered (i.e. able to inject faults which are codewords). Even so, an interesting property of BCH codes compared to Hamming codes, which are sometimes used for cryptographic applications, is the fact that they are codes over $GF(2^m)$. This can be particularly useful for the protection of block ciphers, as their internal states can usually be expressed as blocks of size 2^m (for instance, to protect a non-linear layer constituted of SBoxes).

Of course, there exist many different other conventional codes with good detection and correction rates, or in general good performances. However, while those codes are particularly useful in the context of communication over noisy channels, they often have drawbacks in terms of security. More precisely, and to summarise, if an attacker knows which conventional code is being used to protect a sensitive cryptosystem, he may be able to inject faults such that they are undetected by the chosen EDC or ECC counter-measure. As a consequence, security-oriented codes are necessary.

2.2.1.2 Security-Oriented Codes

Security-oriented codes are EDC or ECC which are aimed to being used to protect cryptographic implementations. They are designed to avoid the common problems of conventional codes, and circumvent different type of attacks, such as fault attacks, even in the presence of a strongly capable attacker. In order to do so, they satisfy different properties. One of such properties is the robustness of a code [KKT04]. First, let's define the error masking probability. For an EDC $\mathcal{C}$, the error masking probability of an error ϵ is the probability that the error will be masked by a codeword $c \in \mathcal{C}$. It is denoted as $Q(\epsilon)$ and

defined as follows.

$$Q(\epsilon) = \sum_{c \in C} Pr(c)\delta_C(c \oplus \epsilon) \tag{2.27}$$

$Pr(c)$ is the probability of the codeword $c \in \mathcal{C}$, and δ_C is the characteristic function of the considered code $\mathcal{C}$.

$$\delta_{\mathcal{C}}(x) = \begin{cases} 0 \text{ if } x \notin \mathcal{C} \\ 1 \text{ if } x \in \mathcal{C} \end{cases}$$

In other word, $Q(\epsilon)$ is the probability that the error ϵ will be undetected by the EDC. If an error ϵ is never detected by the EDC, then $Q(\epsilon) = 1$. All the errors which are never detected form the kernel K_d of the code $\mathcal{C}$. Finally, the error masking probability of the code itself, $\overline{Q}$ is the maximal error masking probability for all possible errors $\epsilon \notin K_d$.

$$\overline{Q} = \max_{\epsilon \notin K_d} Q(\epsilon) \tag{2.28}$$

Definition 2.2. A code $\mathcal{C}$ is called robust if any non-zero error can be detected with a probability greater than zero. That is to say that, for any error $\epsilon \neq 0$, $Q(\epsilon) < 1$. Consequently, $K_d = \{0\}$ if $\mathcal{C}$ is a robust code.

The robustness property of a code $\mathcal{C}$ therefore refers to the resilience of the code to the worst case scenario of undetected errors. Robust codes do not have any error which is always undetectable, as such, it is much harder for an attacker to find an error vector which would lead to a successful fault injection attack. This is especially true comparatively to linear codes. In fact, it should be clear from the definition of robust codes that linear codes are not robust. Since any codeword can be expressed as a linear combination of the other codewords, and more particularly, the sum of two codewords is a codeword, then the kernel of a linear code is $K_d = \mathcal{C}$. A code can also be partially robust.

Definition 2.3. A code $\mathcal{C}$ is partially robust if the size of its kernel is smaller than the code itself: $1 < |K_d| < |\mathcal{C}|$. Essentially: $\exists \epsilon \mid Q(\epsilon) = 1$.

Partially robust codes, even if not as resilient to cleverly crafted fault injection attacks (since there exist at least a fault which is always undetected), are still better than non-robust codes in a security context, and may be less costly to implement.

While robust codes seem to be the perfect candidates as a counter-measure against malicious attacks, very few robust codes exist. Only four robust code families are known at the time of writing: the Quadratic Systematic codes, also denoted as Quadratic-Sum (QS) codes [KKW07], the Punctured Cubic codes [ALK12, NK14], the Compact Protection Code (CPC) [RNK19] and the Rabii-Keren (RK) codes [RK17]. Any other robust code uses one of the previously mentioned codes as a base. The former three robust codes do not however have any correction capabilities, and are thus restricted to only detection of faults. The detection of injected fault by robust codes is of course already a step forward in term of security application, but error correction capabilities can further improve the protection offered by code-based counter-measures against such attacks. Some partially robust ECC are available, such as the Vasil'ev code [Vas65], or the Phelps code [Phe83], but only the newly introduced RK codes are robust and also offer error correction capabilities. The RK code were as such chosen for the ECC architectures presented in this thesis (Chapter 3), and are detailed in their own Section 2.2.1.3.

The previously described codes are all deterministic codes, but security-oriented codes can also rely on some randomness to avoid worst case scenarios and leaking some sensitive information [CDF+08, WK11, NBD+15, DPW18]. Such codes are not the focus of this work, but should be discussed nonetheless, as a comparison to the RK codes, and for completeness of security-oriented code coverage.

Algebraic Manipulation Detection (AMD) codes were introduce in [CDF+08] and later their application for cryptographic devices was discussed in [WK11]. They are non deterministic EDCs/ECCs since they introduce a random element in the encoding process, and thus require a true Random Number Generator (RNG) to be implemented alongside the encoder and decoder. The advantage of the AMD codes over conventional codes, for security sensitive applications, is their resilience to a stronger attacker model. If an attacker knows all the implementation details, and including the (deterministic) ECC used, and can not only precisely control the injected fault, but also the input of the encryption scheme, he can therefore also choose the fault-free output of the device under attack. Clearly, this is a weakness for conventional codes, as stated in Section 2.2.1.1, but the authors of [WK11] also discuss how this is a vulnerability for robust codes. They point out that if an attacker is able to control both the input and the injected fault, and has

a complete knowledge of the cryptographic circuit, as well as the ECC counter-measure, then he has the capability of finding a fault which will be masked. This does not mean that the fault constructed by the attacker will always be undetected, but it will be for the chosen input. Thus, the authors argue that, even for a robust code, it is possible for a skilful attacker to mount a successful fault injection attack even in the presence of a robust code-based counter-measure. While this is true in theory, in practice, it is not always easy to find such a fault, even with control over the input, and injecting the constructed fault with a physical fault injector may also be difficult. This is however not completely impossible, and as such AMD codes have an advantage over robust codes in this even stronger attacker model. Nonetheless, it should be noted that this only takes into account the code itself, and not the architecture which may be deployed around the ECC. For instance, the inner-outer code architecture presented in this thesis (Section 3.3.2) would also tackle this vulnerability.

AMD codes are therefore relevant security-oriented codes, as they provide a stronger security than conventional codes, but also robust codes. They achieve this stronger security claims by introducing some randomness into the information portion of the code. This way, the computed redundancy is not only dependent on the output of the protected cryptographic module, but also on a random data generated by a true RNG. The definition of an AMD code is based on the definition of security kernel of a code.

Definition 2.4. Let $x \in GF(2^m)$ be the randomness generated by a true RNG. Then a (k, m, r) EDC $\mathcal{C}$, where k is the size of the information portion $y \in GF(2^k)$, m the size of the random portion and r the size of the corresponding redundant portion, computed by the function $f(y, x)\colon GF(2^{k+m}) \rightarrow GF(2^r)$. The security kernel K_S of $\mathcal{C}$ is the set of errors $\epsilon = (\epsilon_y, \epsilon_x, \epsilon_f) \in GF(2^{k+m+r})$, for which y exists such that $f(y \oplus \epsilon_y, x \oplus \epsilon_x) \oplus f(y, x) = \epsilon_f$ for all x.

$$K_S = \{\epsilon \mid \exists y, f(y \oplus \epsilon_y, x \oplus \epsilon_x) \oplus f(y, x) = \epsilon_f, \forall x\} \tag{2.29}$$

From this definition, a secure EDC in relation to the previously defined attacker model should therefore have no non-zero errors in its security kernel. This is the definition of an AMD code.

Definition 2.5. A (k, m, r) EDC $\mathcal{C}$ is an AMD code if and only if $K_S = 0_{GF(2^{k+m+r})}$.

It is possible to define an analogue error masking probability for AMD codes and, in [WK11], an example of AMD code based on the generalized Reed-Muller codes and the extended Reed-Solomon codes, which are both linear codes used as primary building block for AMD codes, was given, and applied to protect a Galois field multiplier (which is used for elliptic curve cryptosystems). The authors claim a low area overhead for the implementation of such an AMD-based counter-measure. The proposed architecture has a 112% to 155%, which is indeed low, however, this does not account for the implementation of a true RNG. The implementation cost related to the introduced randomness of AMD codes was ignored based on the assumption that true RNG are available in most cryptographic hardware. This is highly dependent on the considered hardware and encryption scheme. It is true that some encryption algorithm require some random inputs as well, or that some other counter-measures also need a random number, but not all cipher are built this way. Additionally, true RNG are hard to implement, especially for highly constrained devices [Sch08]. Therefore, the area overhead of an AMD code architecture should also consider the implementation of a true RNG, and not assume that it is already present in most cryptographic devices. Nevertheless, AMD codes are still strongly capable security-oriented EDCs, and offer a higher level of security compared to robust codes in the case of a stronger attacker model.

One more final property of a particular class of security-oriented code, which is of interest in comparison to robust codes, is the non-malleability of an EDC or ECC. The notion of non-malleability was first introduced in [DPW18], at the same time as an example construction of non-malleable code. Without going into details, an ECC is non-malleable if the output of the decoder is either the correctly corrected codeword or a completely independent value (i.e. a random value). This property is of particular interest as a secondary counter-measure property against passive SCA. Passive side-channel, as discussed in Section 2.1.1, infer some sensitive information, such as the secret key, via some side-channels, with or without access to the data itself. If an EDC detects an error, or if an ECC is incapable of correcting an error, a decision has to be made on how to proceed. It can be chosen to output all zeros, stop the encryption, or many other kind of system-level actions. However, such actions may lead to some information leakage (see discussion of FSA in Section

2.1.4). Similarly, if the attacker has access to the data, and the output of the decoder is statistically correlated to the key, even if the ECC architecture outputs such a value, it may be possible to statistically recover the secret key. Therefore, if the output of the decoder is completely independent from the processed data (i.e. random), then this kind of attacks can be prevented. Hence the relevance of non-malleable codes. However, other architectures may be resilient against passive SCA without being non-malleable. Moreover, due to the required random nature of the output, non-malleable code, similarly to AMD codes, require the implementation of a true RNG (the proposed architectures of non-malleable codes are actually based on AMD codes), and even rely on a random oracle to satisfy the non-malleability properties (as of the time of writing). Due to these factors, non-malleable codes are theoretically a strong counter-measure against both passive and active SCA, but they remain difficult to implement in practice.

For either of the previous codes, the true RNG can be attacked. If the attack is successful, the cryptographic circuit becomes vulnerable to aforementioned attacks. Therefore, the true RNG needs to be protected as well, increasing further the implementation costs.

In comparison to both AMD codes and non-malleable codes, deterministic robust codes do not require the implementation of any true RNG or random oracle. They are thus applicable even to constrained devices, while providing stronger security properties than conventional codes. More precisely, their robustness property makes renders mounting successful attacks much more difficult, since no fault is always undetected. The only requirement for robust codes, but also any other type of EDC or ECC used as counter-measure against fault attacks, is that parts of the detection or correction module itself is tamper-proof. This means that, for example, no fault can be injected on the detection flag, in the case of EDC, or no fault can change directly the corrected code word for an ECC. This model is however standard, as else any code-based counter-measure can be easily circumvented (for instance by simply flipping the detection flag bit). Consequently, they were chosen for the architectures presented in Chapter 3 and as main ECCs for this work.

2.2.1.3 Rabii-Keren Code

The Rabii-Keren (RK) [RK17] codes are a new class of q-ary security-oriented robust codes. RK codes are constituted of two main parts: a q-ary linear systematic code, and a specifi-

cally chosen function over the corresponding field $\mathbb{F}_q$, which provides the robustness property. RK codes take advantage of those two components in order to preserve the generally good properties of the considered linear code, such as the detection and correction capabilities, or even the simplicity of implementation, while providing a higher security level for cryptographic applications compared to the stand-alone linear code. The RK codes are the ECCs chosen for all the architectures presented in this work (Chapter 3), due to their properties and scalability, thus being a good candidate for the automated generation of counter-measure architecture for cryptographic circuits. Consequently, they are presented in detail in this section to provide the necessary background for this work architectures.
In order to prove the robustness properties of the RK codes, a metric for the non-linearity of a function $f\colon \mathbb{F}_q \to \mathbb{F}_q$ is necessary. In the remainder of this section, $q = 2^m$, unless stipulated otherwise. This is a standard choice for cryptographic applications. The non-linearity of a function $f\colon \mathbb{F}_q \to \mathbb{F}_q$ can be measured by the value of $\Delta(f)$, defined as follows.

$$\Delta(f) = \max_{a \in \mathbb{F}_q^*, b \in \mathbb{F}_q} |\{x \in \mathbb{F}_q \mid f(x+a) \oplus f(x) = b\}| \tag{2.30}$$

The lower the value of $\Delta(f)$, the higher is the non-linearity of f. Moreover, f is perfectly non-linear if $\Delta(f) = \frac{1}{q}$. However, in the case of this work, where the considered field is binary, there are no perfectly non-linear function over $\mathbb{F}_q$. Since the robustness property of the RK code is tightly tied to the non-linearity of the chosen function, and binary field are the primary application of encryption schemes, the function chosen for the construction of the ECC needs to be as non-linear as possible. Therefore, let's define Almost Perfect Non-linear (APN) functions.

Property 2.1. *If $q = 2^m$ and $f\colon \mathbb{F}_q \to \mathbb{F}_q$ then, in the general case, we have $\Delta(f) \geq 2$, and if $\Delta(f) = 2$ then f is an APN function.*

Now that a metric for the non-linearity of a function has been introduced, let us construct an RK code. The first building block of an RK code is an $[n, k, d]_q$ linear code $\mathcal{C}$. $\mathcal{C}$ must be in systematic form. That is to say that its generator matrix is of the form $G = (I_k|A)$. The second part of the code is a bijective function f, which will make the code robust.

Definition 2.6. If $f\colon \mathbb{F}_q \to \mathbb{F}_q$ is a bijective function, and $G = (I_k|A)$ is the generator matrix of an $[n, k, d]_q$ systematic linear code $\mathcal{C}$. $A = \{a_{ij}\}^{k,r}$ with $a_{ij} \in \mathbb{F}_q$. The corresponding

Rabii-Keren code is denoted $\tilde{\mathcal{C}}$, and a codeword can be denoted as $\tilde{c} = (x, w)$, such that $x = (x_1, .., x_k) \in \mathbb{F}_q^k$ is the information portion of the code and $w = (x_1, .., x_r) \in \mathbb{F}_q^r$ is the corresponding redundant portion. Then $\tilde{\mathcal{C}}$ is defined as follows:

$$\tilde{\mathcal{C}} = \{\tilde{c} = (x, w), w_j = \sum_{i=1}^{k} a_{ij} f(x_i)\} \quad \text{where } f \text{ is an APN function} \tag{2.31}$$

Moreover, since f is bijective, $\mathcal{C}$ and $\tilde{\mathcal{C}}$ are equivalent codes, and thus the RK code also has a distance of d, same as the linear code $\mathcal{C}$. It should be noted that, if f wasn't bijective, this does not hold. Additionally, the constructed RK code is systematic as well. Therefore, if a non-zero error $\epsilon = (\epsilon_x, \epsilon_w)$ would occur such that a codeword $c = (x, w)$ would be mapped to $c' = (x', w') = (x \oplus \epsilon_x, w \oplus \epsilon_w)$, then if $\epsilon_x = 0_{\mathbb{F}_q^k}$, $\epsilon = (0, \epsilon_w)$ is always detected. Otherwise, if $\epsilon_x \neq 0_{\mathbb{F}_q^k}$, it should be clear from the definition of the RK code that an error is masked if:

$$\sum_{i=1}^{k} a_{ij}(f(x_i \oplus \epsilon_{x_i}) \oplus f(x_i)) = \epsilon_{w_j} \qquad \forall j \tag{2.32}$$

While only a bijective function is required for the construction of an RK code, good robustness properties are only given by APN functions over $GF(2^m)$ (or more generally highly non-linear functions). Consequently, only such functions should be used in a security context. If a function $f : \mathbb{F}_q \to \mathbb{F}_q$ is an APN function, then there are two cases: either $q = 2^m$ with m odd or m is even. If m is odd, it can be shown that, thanks to Equation 2.32, the error masking probability $Q(\epsilon)$ of the RK code is smaller than $\frac{2}{q}$, while, if m is even, $Q(\epsilon) \leq \frac{4}{q}$ for any APN function and all possible errors ϵ (see [RK17]). Consequently, the RK code is indeed a robust code according the Definition 2.2, and there is no error that is always undetected. In the case where m is odd, one of the best function to be used, thanks to its relatively low implementation cost, is $f(x) = x^3$. If m is even, it is possible to use the function $f(x) = x^{-1}$ over $GF(2^m)$. The latter function was chosen for the architecture presented in Chapter 3, due to its simplicity of implementation and low implementation cost.

Finally, RK-based architectures presented in this work may also use either a QS code [KKW07] or a CPC [RNK19] as a second layer of code (See Section 3.3). QS codes are

optimum robust codes constructed directly from the definition of robust codes. The code is defined as follows. Let $x = (x_1, ..., x_{2n}) \in GF(2^{(2n)k})$ with k and n two integers be the input data, as well as c be the corresponding codeword after a QS code encoding.

$$c = (x, w) \text{ with } w = x_1x_2 + ... + x_{2n-1}x_{2n} \in GF(2^k) \tag{2.33}$$

CPCs are high-rate low-complexity robust codes constructed from a set of i $[n_i, k_i]$ systematic robust codes $\mathcal{C}_i = \{(x, w_i)\}$, where w_i is the respective redundancy introduced by each code. Let's define $r = \max_i r_i$, where $r_i = n_i - k_i$, and $k = \sum_i k_i$. The corresponding CPC code $\mathcal{C}$ is then define as follows.

$$\mathcal{C} = \{(x, w) : w = \sum_i w_i\} \tag{2.34}$$

$\mathcal{C}$ is an $[n = k + r, k]$ robust code. Note that each code $\mathcal{C}_i$ can be padded with zeros to ensure correct data sizes.

2.2.2 Other Counter-Measures

The aim of this section is to present a few additional counter-measures against both passive side-channel and fault attacks. The considered counter-measures are relevant in the context of the automated AFA framework `AutoFault` presented in Chapter 4, as possible use cases for the framework. A brief discussion on physical counter-measures at the circuit level is given first, followed by an description of masking and the specific case of Domain-Oriented Masking (DOM) [GMK16]. Finally, nonce-based cryptosystem are presented in this section, as they offer a protection against fault injection attacks, even though they are not typically considered as a counter-measure in the strict sense of the term.

2.2.2.1 Physical Counter-Measures

Counter-measures against SCA, both passive and active, can be implemented at the circuit level. For instance, in order to prevent a malicious attacker from recovering the secret key of an encryption scheme through the implementation of a DPA or CPA attack, it is possible to inject some noise into the power supplied to the hardware implementation of the cipher.

This noise renders the analysis of power traces much harder, and thus the success rate of passive SCA attacks which measure the power consumption of the device lower. This counter-measure is implemented outside of the hardware implementation of the cryptographic primitive, at the circuit level, and is denoted as Noise-Injector counter-measure [DMN+17, DMN+18]. More relevant circuit level physical counter-measures in the context of this thesis are shielding [LM06, Tar10] and sensor-based protections [RE04]. Both protection methods can be respectively categorised as attack prevention and attack detection physical counter-measures. In the latter case, sensors are placed on the circuit, at locations which are deemed as critical from a security point of view (for example, at a location where a fault injection is known to lead to a successful attack). If the sensor detects any tampering, then the system has to decide what course of action to take. One may stop the encryption, proceed to some re-keying or even have some kind of self-destruct mechanism. Many different kind of sensors can be used to achieve this, such as temperature or voltage sensors, or in the case of protection against laser-based fault injections, photo-sensors. As for shielding methods, they can be divided into two different types: passive and active shielding. Passive shielding consists into placing a metal shield over the sensitive parts of a circuit, as to prevent any EM or optical tampering. However, due to heat dissipation constraints, it is not possible to cover a complete circuit with a shield, which is a limitation to this approach. Furthermore, an attacker may be able to remove the metal protection layer, or circumvent it by other means, such as indirect laser beam at a different angle. Active shields, on the other hand, are usually constituted of a mesh structure, which raises a flag if it detects any tampering by being interrupted. In both cases, as well as in the case of sensor-based counter-measures, the effect can be modelled as an impossibility to inject a fault at a given location, which, in turn, can be considered in the fault model of an attack or, in the context of this work, in the `AutoFault` framework.

2.2.2.2 Masking

At an algorithmic level, one technique which may be used to circumvent SCA is masking. The idea of masking was discussed and introduced by various papers, such as [CJRR99] and [AG01], roughly at a similar time. The basic principle is to prevent the correlation of sensitive values to any side-channel, such as their relation to the power consumption of

the device, by introducing a random element and splitting the sensitive values, as well as computations, into separate instances. In more details, let $K = K_i$ be the secret key of an encryption scheme and its respective key parts. If no counter-measure against SCA is implemented, then it may, for instance, be possible to mount a DPA (similarly to Section 2.1.1). If this is the case, the key parts k_i can be correlated to the power consumption of the device and recovered. However, if the values of each k_i is randomised at each encryption, then only the randomised value of the key part can be recovered, which is of no use for the attacker. Of course, this is a simplified model, but this is the main idea behind masking.

Secret-sharing approaches are one simple way to implement masking. Let r be a random number, then it is possible to compute $K \oplus r = x$, in this case r and x are the denoted as the shares of K, and the value of the key can be easily reconstructed by computing $r \oplus x = K$. During the encryption, only r or x should be computed at a given clock cycle. Otherwise, it may be possible to correlate the value of K directly, despite the sharing. Moreover, this can be generalised to more shares, then each operation should only be computed with a single share and no multiple shares should be processed in the same clock cycle, to avoid any leakage. This means that every operation (cryptographic functions), which is performed during the encryption, needs to process masked values such that correct, unmasked, values can be recomputed at the end of the encryption. However, it should be clear that every linear function used in a cipher can easily be masked and unmasked by the scheme mentioned above, as, if f is linear, $f(x \oplus r) = f(x) \oplus f(r)$. The most difficult part of the implementation of a masking scheme is the handling of non-linear functions, such as SBoxes, as they cannot be mapped so simply. A naive method would be to implement an SBox for each possible random value, and choose the SBox based on the generated random value. This is costly or even impossible to achieve for highly constrained devices, therefore, better methods exist and should be employed. Such methods are also often masking scheme dependent, as the way the mask are applied, and consequently how the values are unmasked, depends on the specific construction [MPL+11, DCBR+15, GMK16].

Many different masking schemes exist, however, let's consider the Domain-Oriented Masking (DOM) scheme [GMK16], since it was presented as one of the most efficient masking scheme in a recent evaluation of multiple masking techniques [MMSS19], and is also a

case study for the automated validation of counter-measures through the use of `AutoFault` (Section 4.8). DOM implementations, as the name suggest, are based on so called share domains. If a variable x is for instance divided into two shares, they can be denoted as A_x and B_x such that $x = A_x \oplus B_x$ (in this case, A_x is for instance the generated random number, and B_x can be trivially derived). Additionally, another random number is used to create two shares for another variable y, it can also be expressed as $y = A_y \oplus B_y$. In this example, A and B are the shared domains, for both variables x and y, and this can be generalised to any number of shares. More specifically, a DOM implementation with $d+1$ shares achieves d^{th}-order of security (i.e. secure against attack which can correlate d intermediate values to a single side-channel measurement [CJRR99, BGN$^+$14]). Without going too much into details, as this is not the primary focus of this work, DOM scheme introduce a re-sharing step if a computation is done between two different domains (cross-domain operation). While this is trivial for linear operations, the main difficulty for DOM implementations is to mask non-linear operations (similarly to any masking scheme). To this end, the authors proposed two main components, a DOM-indep, and a DOM-dep multiplier. The DOM-indep multipliers require independently shared inputs, but can be implemented at a lower cost, and does not need as much new random variables, as opposed to the DOM-dep multipliers. Figure 2.5 shows the example of a DOM-indep multiplier. As previously mentioned, DOM implementations make use of a re-sharing step to avoid leaking sensitive data when a cross-domain operation is performed. In this case, the multiplier computes $q = x \cdot y = (A_x \oplus B_x) \cdot (A_y \oplus B_y) = A_x A_y \oplus A_x B_y \oplus B_x A_y \oplus B_x B_y = A_q \oplus B_q$. It is clear that $A_x A_y$ and $B_x B_y$ are inner domain multiplication, and thus non-critical, while $A_x B_y$ and $B_x A_y$ are cross-domain operations which may leak some sensitive information. Therefore, a new share Z_0 is introduced and XORed with the previous values, in order to prevent any leakage. Since the same new share is used for the computation of A_q and B_q, and in both cases it is only an XOR, it is clear that $q = A_q \oplus B_q$, even after the re-sharing operation. The DOM-dep multiplier is based on the same principle and the authors also provide an example of protected AES SBox with a DOM implementations.

DOM implementations have many advantages compared to other masking schemes, while having a comparatively low overhead. They also offer a scalable protection against d^{th}-order attacks, as they can be generalised to any number of shares. This masking scheme is

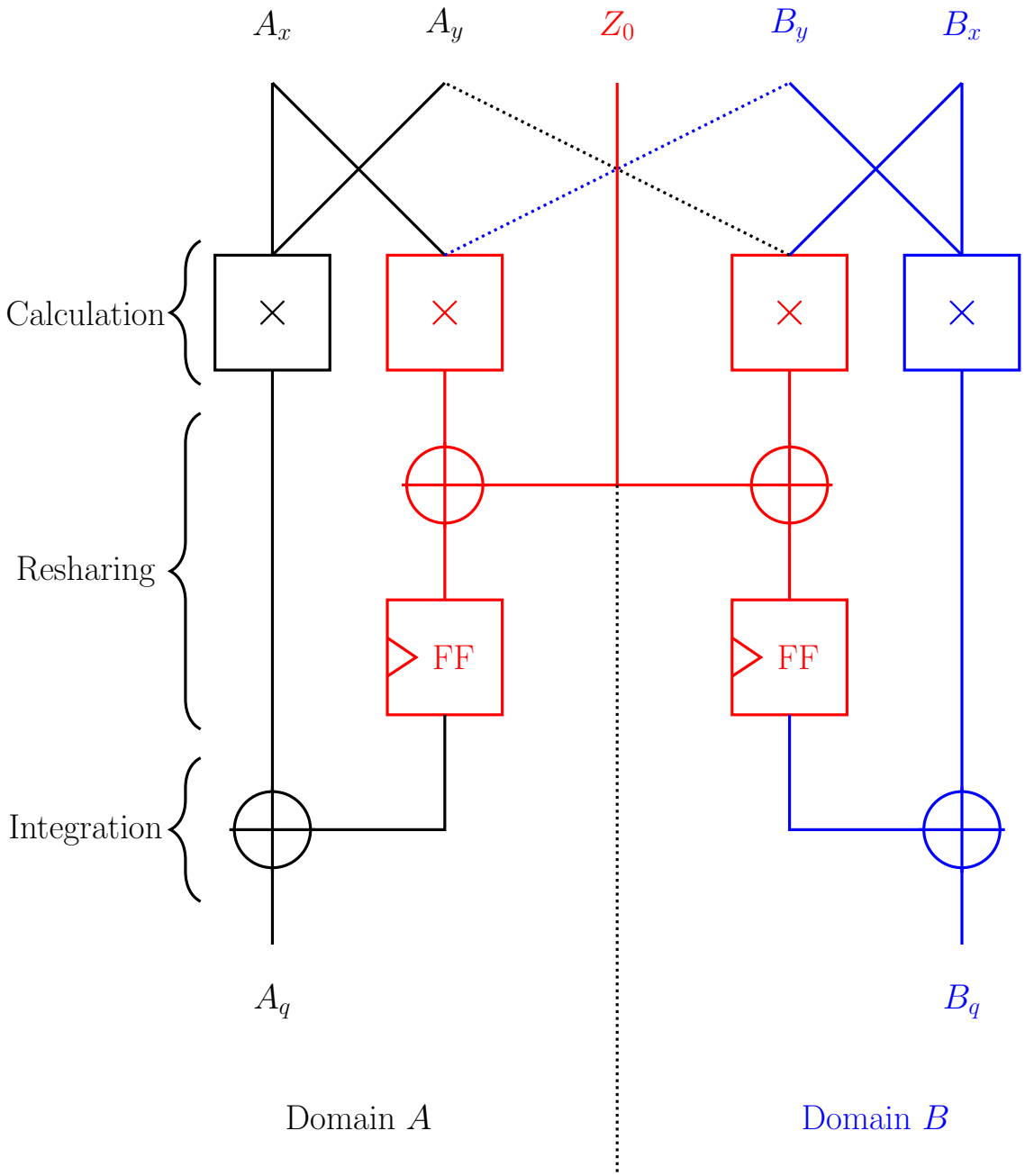

Figure 2.5: First Order DOM-indep $GF(2^n)$ Multiplier

therefore of particular interest in the context of this thesis. Furthermore, masking schemes are only aimed at protecting a hardware implementation of a cryptographic primitive against passive SCA, but not active ones, such as fault injections attacks. They may however lead to further weaknesses against such attacks, and the specific case the interaction between a DOMAES (DOM protected AES implementation) and the framework proposed in this thesis will be discussed in Section 4.8.

2.2.2.3 Nonce-based Ciphers

A nonce is a "number used once". For cryptographic applications, this means that a number is generated and used during the encryption process. This number can be either randomly or sequentially generated. However, in the former case, a failure of the RNG may lead to the same nonce being re-generated, while a sequential generation (for example using a counter [Rog04]) ensures that the numbers are not repeated between each encryption. A nonce can relate to stream ciphers or nonce-based symmetric block ciphers. In the context of this work, stream ciphers were not studied so far, and the reference to nonce-based ciphers relates to symmetric block ciphers which use a new nonce for each encryption. The goal is to provide a stronger security level, at the expense of an RNG. Nonce-based encryption schemes have a few additional properties compared to more conventional symmetric block ciphers, but one characteristic of particular interest is the impossibility for an attacker to generate pairs of fault-free and faulty ciphertexts dependent on the same inputs (since the nonce changes at each encryption). This circumvents differential attacks, and, for instance, the DFA on the AES presented in Section 2.1.4.2 would be impossible to mount if a nonce was introduced during the encryption process. However, this advantageous property against differential attacks is only valid if the nonce is used properly, that is to say not reused or misused. If the nonce would be reused a differential attack would be possible again, since several pairs of fault-free and faulty ciphertexts could be generated. Despite this fact, state-of-the-art attack models often consider that the nonce is misused, or can be forced to be reused. Some models even consider the nonce as known. As such, these attack scenarios are perfectly valid, or can at least be a starting point for more complex attack with an unknown nonce model.

Recently, the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) took place in order to proposed new secure authenticated ciphers (encryption schemes which ensure the authenticity of the data). After a few years of competition and several attacks successfully mounted on most of the competitors, two nonce-based encryption schemes were chosen as the winner of the CAESAR competition, for the lightweight applications category: ACORN [Wu16] and Ascon [DEMS16]. While both ciphers have been carefully designed and have withstood different attacks, in this work the focus will be on Ascon and its interactions with the AFA framework `AutoFault`. Even if purely differential attacks are circumvented in the case of Ascon, since it is a nonce-based cipher, it is not clear how an AFA would stand against such a cipher, considering known and unknown nonce fault models, as well as nonce reuse. This will be discussed in Section 4.8.

As a brief description of Ascon (for Section reference in Section 4.8), it is constituted of four different phases. The first phase is the initialisation, where an initialisation vector IV, the nonce N and the secret key K are processed to create the initial intermediate state. The second main process is the introduction of the associated data A (additional data mainly used to avoid attacks which would reuse the encrypted message), divided in parts A_i, to the internal state, if such data is present. Then comes the processing of the plaintext P (by the processing of its respective parts P_i), and creation of the corresponding ciphertext C (and the related parts C_i). In this phase, however, the key is not used directly, but the ciphertext is still dependent on it at this step. The intermediate state is computed by XORing the intermediate state of the previous phases with the plaintext. Thus, there is a dependence to the key in this stage too. The last phase, the finalisation, is similar to

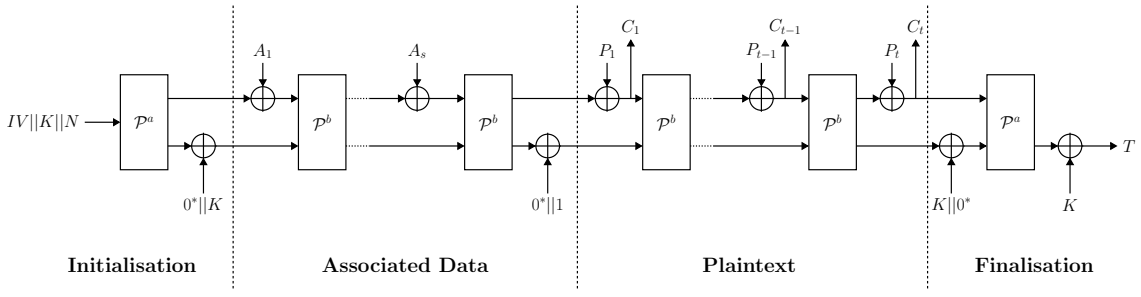

Figure 2.6: Ascon Cipher (Encryption)

SPN ciphers, in the sense that SBoxes and permutations are used to compute the output, constituted of the modified cyphertext and a tag T. Figure 2.6 shows the overall flow of the Ascon cipher, for an encryption. The last finalisation phase is of particular interest for AFAs, as the key is processed and a non-linear operation is performed (the substitution layer, constituted of 5-bit SBoxes, is included the defined permutation operation), which is once again similar to other, more conventional, SPN ciphers, even though the values are also nonce-dependent.

Chapter 3

Security Oriented Code based Architectures for Fault Attack Mitigation

In order to protect against the plethora of different fault injection attacks (Section 2.1), cryptographic circuit designers need to implement efficient counter-measures. Code-based counter-measures are one way to protect a circuit against fault attacks. They comprise of Error Detecting Codes (EDCs) and Error Correcting Codes (ECCs). While both conventional code-based counter-measures and security-oriented ones were discussed in Section 2.2.1, this chapter focuses on the investigation of the Rabii-Keren (RK) code at a hardware level, to mitigate the impact and success rate of fault injection attacks. More precisely, discussions on the use case of this kind of ECCs (Section 3.1), as well as the limitations of the conventional evaluation methods for EDCs in a security context (Section 3.2) are given, before a detailed presentation of the implementation of the RK codes. The implementation itself is based on the RK codes presented in Section 2.2.1.3, but at a practical level and with error correction capabilities, which was not the case in the original paper [RK17]. The proposed implementations can be easily scaled to any cipher, thanks to the construction methods. As such, the architectures presented in this chapter can be automatically chosen given a cipher and a desired detection rate. Throughout Section 3.3 are presented the multiple main aspects of the architectures, from the basic implementation, to several improvements over the base idea. Finally, Section 3.4 presents experimental results for the architectures against physical fault injections for multiple SPN ciphers.

3.1 Natural Fault & Malicious Fault Scenarios

Code-based counter-measures are implemented according to assumptions on the nature of the faults they are aimed at handling, but also on the requirements of the design and the hardware. In this thesis, we consider that faults can be categorised as either natural faults, or malicious faults, and, depending on the category, as well as the desired protection, detection or correction may be needed. Recent breakthroughs in coding theory, such as RK codes [RK17] and CPC [RNK19], laid the theoretical background for physical implementations of scalable code-based counter-measures for the architectures proposed in this thesis, which are dependent on the aforementioned assumptions and prerequisites.

Natural faults are caused by factors such as ageing or cosmic rays radiations, and can be assumed to only affect a small number of bits. In other words, the Hamming weight of the induced fault is low, and as such, an ECC with low correction capability is sufficient to deal with naturally induced faults. This is especially true in the case of q-ary ECC implementations, as a single error is in fact a symbol of q bits. For instance, if a natural fault affects an SBox of a cipher, it will affect at most the number of bits of this specific SBox (for instance, 8 bits in the case of the AES). If an ECC is implemented over an alphabet of the same size ($q = 2^8$ in the case of the AES), then only a single error correction capability is required. Moreover, in this case, the correction of the erroneous values is enough to counteract natural faults entirely, since the data under attack and only reliability is required.

By contrast, maliciously injected faults can have a wide-variety of effects and affect either a larger array of bits, or be extremely precise and limited to as little as a single bit-flip. This usually comes down to the attacker fault injection set-up. A low cost fault injector, such as a clock-glitch based injector, will not be able to target a very specific bit in the circuit, but rather a specific location, and in hope of usable faults, while, in comparison, costlier equipment, will allow the attacker to stick with a constraint fault model. In terms of EDC or ECC counter-measure, this means that the chosen code should be able to detect arbitrary errors with high probability. Correcting the injected fault can also be achieved by some ECC architectures, however, not for arbitrary faults. In addition, EDC architectures can leak sensitive information on their own, if the designer is not careful [RBIK12],

but this specific case isn't considered in the remainder of this chapter. Another important aspect in the case of malicious fault, is the possibility for a dual fault injection [SHS16]. An attacker with access to a dual-beam laser set-up can inject a fault in both the EDC/ECC component, as well as in the encryption, which may cripple the error detection, or correction, capability.

In this regard, EDCs can be divided into two categories, reliability oriented codes and security oriented codes. The former are aimed at correcting natural faults, while the latter are used to guarantee detection against maliciously injected faults. Reliability oriented are evaluated in terms of error correction capabilities. For an ECC of minimum distance d, the error correction capability is defined as $t = \left\lfloor \frac{d-1}{2} \right\rfloor$ (see Section 2.2.1). On the other hand, since security oriented codes are aimed toward the detection of arbitrary faults, and not their correction, they are evaluated in term of their error masking probability $\overline{Q}$, which is the probability that an error is not detected by the EDC architecture (Equations 2.27 and 2.28). For a security oriented code, the lower the value of $\overline{Q}$, the better, as no error should go undetected with a high probability. As such, robust codes (Definition 2.2) are a good choice for security application, as their error masking probability is inferior to 1 by definition.

While the detection of maliciously injected faults is relevant, a lot of security oriented code implementations do not attempt, or have the capability, to correct the faulty data, even though it can, for instance, circumvent some statistical attacks, such as SIFA [DEK+18]. In this regard, the architectures proposed in this chapter provide both detection of any arbitrary fault, as well as correction of low multiplicity ones. This allows for an application in the context of security, as well as reliability, and is a further counter-measure to different kind of attacks compared to only detection based EDC implementations, while also being robust. As of the knowledge of the author, they are the first security oriented EDC architectures to provide simultaneously detection and correction capabilities of faults, being natural or malicious.

3.2 Limitations of Error Detecting Code Evaluation in Security Context

How to evaluate EDCs is generally important before finalising a design, and this is especially true in a security context. A commonly used method to measure the effectiveness of an EDC implementation is to empirically check the detection rate of the implemented code. A large number of randomly generated faults are injected into the protected hardware, or simulated in software. The encoding of the data, or a detection flag, is then checked to verify if the EDC did detect the fault or not. For such an evaluation, it is important that the randomly generated faults are approximately uniformly distributed. However, and while it may be a valid consideration for reliability oriented codes, assuming that maliciously injected faults are uniformly distributed is not realistic. Depending on the the fault injection setup, an attacker may be unable to accurately inject faults, which may result in a wide variety of injected faults. This is usually the case for low cost fault injectors. If the attacker is able to inject very precise and specific faults, the modelling of faults as random for the evaluation of EDC architectures in the context of security is invalid. For a non-robust code, the fault may be chosen to be a known fault with masking probability $Q(e) = 1$, and as such always undetected.

A low cost, clock glitch-based, fault injector on the SAKURA-G platform (basic version of the injector detailed in Section 3.4) for instance shows that the fault is distributed close to uniformly in the case of an injection on the SSAES, while it is not the case for the full scale AES (Figure 3.1, bytes 3, 4 and 11 have a much higher probability to be faulty). Similarly, in the case of the fault multiplicity for the same full scale AES implementation, Figure 3.2 reveals that not only single byte or low multiplicity faults are injected, but also a significant number of sixteen byte faults. Moreover, a more refined fault injector was implemented on the LX75 FPGA of the same platform, which exhibited once again a different distribution pattern for the same implementation of the cipher. This differences between different injectors and considered ciphers, while expected from a security point of view, are often not considered in a coding theory context. If, however, the implemented EDC counter-measure has a weaker detection rate for smaller or larger multiplicity faults, then this might be overlooked if only random faults are considered.

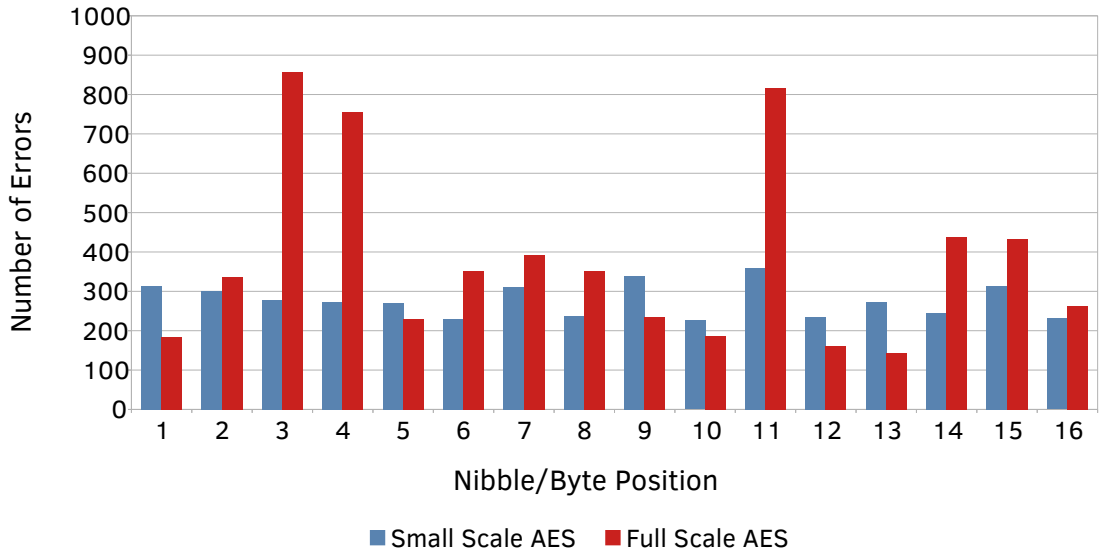

Figure 3.1: Error Distribution for Clock-based Fault Injector on LX9 FPGA

Additionally, only looking at the average detection rate for EDC architectures might also hide the benefits of using a stronger code. For instance, a circuit designer may consider using a BCH code over an RK architecture to protect an AES implementation, since the BCH codes avoid the use of additional functions and are thus slightly less costly to implement. In order to validate his choice, the designer implements both codes and uses a low-cost fault injector to inject a large number of faults, while recording the detection flag of the said EDC. The results of such a comparison can be seen in Table 3.1, which would probably direct the designer to select the BCH code architecture.

However, the RK code is robust, while the BCH code is not, which results in some faults always being undetected by the BCH code. A mathematical analysis would show that 256^{16} faults are never detected by the BCH code, while they are detected with a non-zero probability by the RK code. This is especially important in a security context, where the goal is to prevent any fault injection. If a fault always goes undetected, then an attacker could make use of this, and given a sufficiently powerful fault injector, may be able to specifically target such faults, circumventing the EDC counter-measure. Therefore, if no mathematical analysis of the chosen code counter-measure is done beforehand, and only the detection rate based on randomly chosen faults is performed, critical properties of EDC

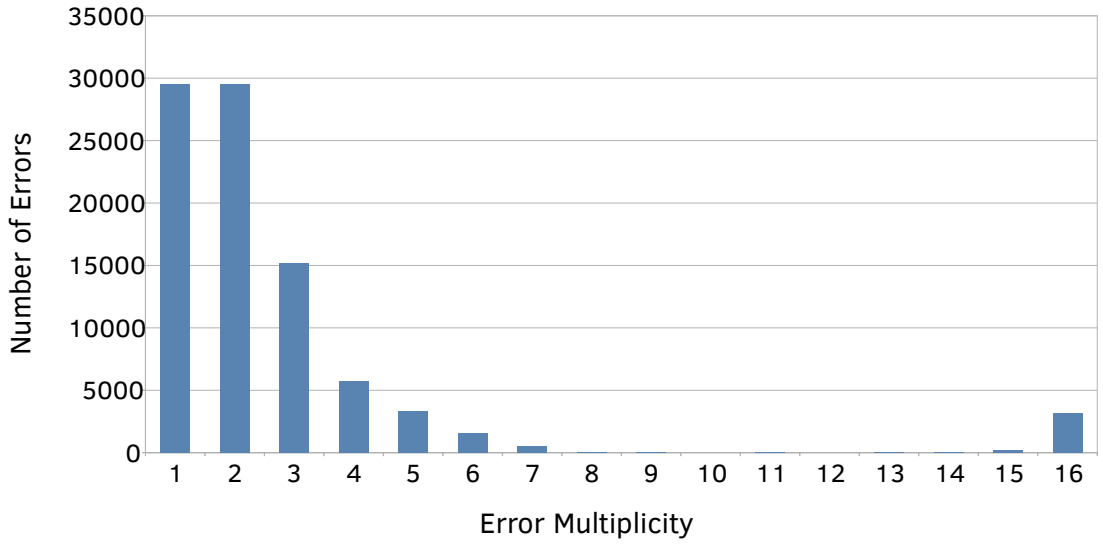

Figure 3.2: Full Scale AES Error Multiplicity for Clock-based Fault Injector on LX9 FPGA

implementations may be missed and just as critical vulnerabilities may be left in the final design.

Table 3.1: Detection Rate of BCH and RK Codes for Two Fault Injection Campaigns

Experiment	# Injections	Detected faults		Detection rate	
		BCH	RK	BCH	RK
1	1000000	999824	999821	99.98%	99.98%
2	10000000	9997515	9997558	99.98%	99.98%

To summarise on the evaluation of security oriented EDCs, the capability of the attacker should be considered during the design phase, in order to choose the appropriate EDC, since a wide variety of faults can be injected, and they are dependent on the injection setup, as well as the implementation of the cipher itself. Additionally, both a mathematical analysis of the worst case scenarios, as well as an evaluation of the detection rate are required to assess the effectiveness of the counter-measure, and avoid undetectable faults. In this thesis, the proposed architectures based on the RK code, for which the mathematical analysis is available in [RK17, RNK19], are evaluated in the next sections.

3.3 Rabii-Keren Code Hardware Architectures

Prior to this work, the RK codes were only proposed theoretically and no encoding or decoding algorithm were given. Similarly, no error correction capability had been discussed and the implementation of this robust code had not been evaluated for a practical use case, in a security context. The architectures in this Chapter are the first proposed implementations of the RK codes, as a scalable counter-measure against fault injection attacks. An architecture can be automatically generated for a considered cipher.
In this section, the basic architectures for the RK code implementations are presented first. An improvement over the basic architectures is proposed next with the introduction of a second layer of code to handle undetected errors or possible miscorrections. However, even with this improvement, implementing the RK codes in hardware may be difficult and costly. In order to reduce the implementation costs, without decreasing the security level of the counter-measure, an efficient decoding method, the Error Coefficient and Location Table (ECLT), is introduced in Section 3.3.3. Finally, with the goal to provide an even better detection rate for the architectures, while remaining scalable to any cipher or constraints, a modification of the second layer code by a Compact Protection Code (CPC) is proposed in Section 3.3.4.

3.3.1 Basic Structure of the Rabii-Keren Architectures

The base architecture proposed in this work are based on the RK code constructions described in Section 2.2.1.3. The first choice to be made for an RK code, is to choose a linear code to be the main building block upon which the implementation can be built. The linear code also needs to be in its systematic form, such that a generator matrix G can be expressed in the form $G = (I_k|A)$. Even though it is possible to choose any $[n, k, d]_q$ systematic linear code as the primary building block of an RK code (since all linear codes can be expressed in such a way, see Section 2.2.1), for the architectures presented here, the choice was made to use a shortened BCH code (a sub-family of the BCH code presented in Section 2.2.1.1). The shortened BCH codes were chosen as they are easily scalable for any size of data over binary Galois fields, making them good candidates for security sensitive applications. Moreover, they have an excellent detection rate and are capable

of correcting errors of multiple sizes, dependent on the code parameters. By setting the code's parameters, it is possible to satisfy stricter requirements in terms of detection rate and correction capabilities, even if this has a hardware cost in terms of area and power. For example, if a designer of a cryptographic circuit requires a correction capability $t = 1$, then since $t = \lfloor \frac{d-1}{2} \rfloor$, he should at least choose an $[n, k, 3]_q$ (distance $d = 3$) shortened BCH code, and if he requires $t = 2$, then at least an $[n, k, 5]_q$ one. Since the construction of the BCH codes is well defined and known, a designer can select the corresponding generator matrix, according to the security requirements, and the size of the data, such that it fits the cipher being protected. Consequently, thanks to the use of the shortened BCH codes, the construction of the corresponding RK code architecture can be automated for any cryptographic primitive.

3.3.1.1 Architecture Overview

Since the chosen linear code for the architecture is a shortened BCH code, it is possible to construct the generator matrix of the code, in its systematic form, by different methods. Algorithm 3.1 shows how to construct A, the non-identity part of the generator matrix $G = (I_k|A)$, from the parity check matrix of the code. This method is well known and is general for any $[n, k, d]_q$ (shortened) BCH code [BRC60]. Once A is generated, and thus G, it is necessary to proceed to the second step of the RK code, the choice of an APN function. The APN function (Property 2.1), as discussed in 2.2.1.3, is the second main component of an RK code, which makes the code robust. Therefore, it is of the utmost importance to choose a function which is as non-linear as possible, while being simple to compute and implement.

In this work, the fields considered for the multiple ciphers are either $GF(2^4)$ or $GF(2^8)$, based on the size of the SBoxes in use. The original paper on the RK codes [RK17] states that, in the case of a Galois field $GF(2^m)$, if m is even, then one possible function to be used is the inversion function $f(x) = x^{-1}$. This function is not only simple to realise, as it can be implemented as a Lookup Table (LUT) for small fields, but also low cost for a large number of ciphers, since the inversion of the considered Galois field might already be implemented for the considered encryption scheme. For example, in the case of the AES, the SBoxes can be implemented by computing the inverse of the input data, pre-

Algorithm 3.1: Construction of A (shortened BCH code)

1 Choose the size of the alphabet, q.
2 Choose the dimension of the code, k.
3 Choose the distance of the code, d.
4 Determine the root field using the rule: $q^m - 1 > k + (d-1)m$ with an m chosen minimal.
5 Choose b the power of the first root in the sequence of $d-1$ consecutive roots of the generator polynomial. Usually $b = 1$ is chosen for a simple code, however if $b = 0$, a smaller number of redundancy symbols are required.
6 Once d, b, q, m are determined, find r the degree of the generator polynomial.
7 Shorten the code by defining $\tilde{n} = k + r$ given the r.
8 Represent the shortened check matrix

$$H_{orig,d} = \left(\begin{array}{cccc|ccc} 1 & \alpha^{b} & \cdots & \alpha^{b(k-1)} & \alpha^{bk} & \cdots & \alpha^{b(\tilde{n}-1)} \\ 1 & \alpha^{(b+1)} & \cdots & \alpha^{(b+1)(k-1)} & \alpha^{(b+1)k} & \cdots & \alpha^{(b+1)(\tilde{n}-1)} \\ \vdots & & & \vdots & \vdots & & \vdots \\ 1 & \alpha^{(b+d-2)} & \cdots & \alpha^{(b+d-2)(k-1)} & \alpha^{(b+d-2)k} & \cdots & \alpha^{(b+d-2)(\tilde{n}-1)} \end{array}\right) = (H_l | H_r)$$

9 Compute the check matrix $H_d = (A_d \mid I)$, where $A_d = H_r^{-1} H_l$. In this case, H_r is an invertible matrix since shortened BCH codes are cyclic.

multiplying it by a given matrix and adding a constant. This means that, in such a case, the inversion module is already implemented, making the inversion function over $GF(2^m)$ a perfect choice as the APN function for the RK code. It should however be noted that, in the case where m is odd, the function $f(x) = x^3$ can be used, thus not breaking the generality and the possibility to automatically generate an RK code counter-measure in the case where a different field is considered (for instance, 5-bit SBoxes, as in the Ascon cipher [DEMS16]). Similarly, if the considered cipher is not directly divided into states of smaller sizes via SBoxes or any other functions, the internal state can simply be divided arbitrarily in a size that divides the size of the internal state itself. For example, let us consider an encryption scheme which would have a 320 bits internal state, without any clear sub-divisions. It is of course possible to work over the field $GF(2^4)$ or $GF(2^8)$, similarly to what is presented in this thesis, but it is also possible to generate an RK code over $GF(2^5)$. In this case, the information portion of the data would be constituted of 64 symbols of 5 bits each. The choice should however be carefully considered in order to match possible real world attacks.

Once both the APN function and the generator matrix G are computed, the encoding of a word can simply be done by first applying the APN function to the data and then multiplying by the sub-matrix A of G. This results in the computation of the redundant portion of the data.

The base architecture [KGKP18] is described in Figure 3.3. The original component (e.g. the encryption module) in this figure is the component which needs to be protected. This is a design choice. A particular component may be critical and after a careful inspection of the implementation in the early stages of the design phase, it may have been found that a fault injection is particularly successful on this component. Another example would just be a known critical location for fault injection at the algorithmic level for the cipher. For instance, the SBoxes are a known injection location in the case of the AES (see 2.1.4.2), and thus it would be beneficial to protect them with an ECC counter-measure. However, if we still consider the AES, an attack may also be successful if a fault is injected after the substitution layer (but before the *MixColumns* operation), and consequently, it may be necessary to protect more than just the SBoxes. If the choice is made to protect a larger portion of the circuit, the corresponding predictor component would be larger as well. This is because the predictor replicates the computation of the original component, but is implemented in a different way to avoid easy dual fault injections resulting in the same fault in both components. For example, in the smaller case of protecting an AES substitution layer, the original component may use Canright's compact SBoxes [Can05] and the predictor LUT-based SBoxes. If a complete AES round is protected, the whole round module of the predictor needs to be implemented differently than the one of the original component. This would lead to an increase in area and power consumption of the implementation compared to only protecting the substitution layer. In general, it may only be possible to protect a smaller portion of the circuit for constrained devices. Nevertheless, the predictor directly computes the redundancy associated with the duplicated computation, which is then taken as input to the decoder (non-linear checker and decoder on Figure 3.3), at the same time as the output of the original component. Both can be faulty, simultaneously or not (denoted by ϵ_x and ϵ_w). The decoder first computes the redundancy associated with the original component output $w' = (x')^{-1} \cdot A$. Once both w and w' have

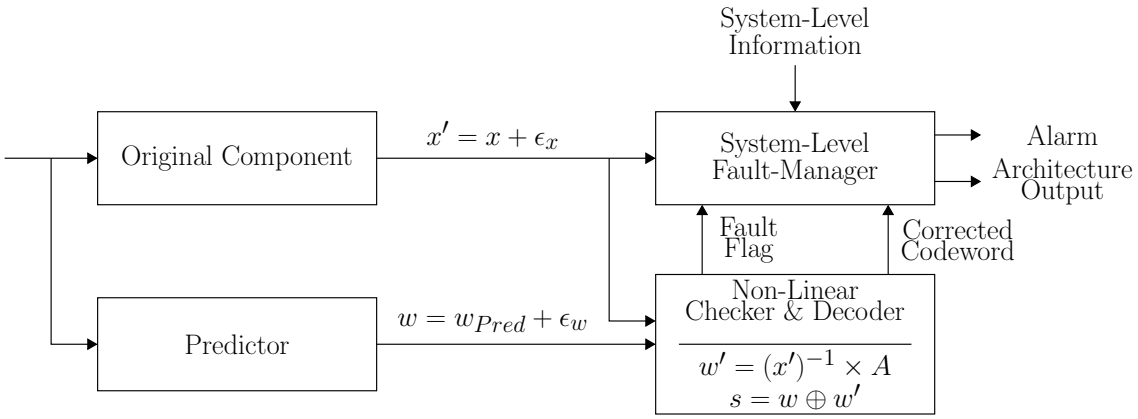

Figure 3.3: General RK-based Architecture for Detection & Correction

been computed, the syndrome s associated with the input data can be derived.

$$s = w \oplus w' \tag{3.1}$$

If $s = 0$ then no fault occurred, or at least no fault was detected, as a fault may have been masked. However, the architecture implements a robust code (Definition 2.2), and therefore this probability is always inferior to 1 (and even significantly lower than 1 in the case of the RK code). If the syndrome is non-zero, a fault occurred, meaning that the codeword needs to be processed further. Additionally, a fault flag is raised and forwarded to the system-level fault manager. At this stage, the proposed architecture already achieves error detection, but it is possible to take advantage of the shortened BCH code correction capabilities to, if possible, correct the invalid codeword. In order to do so, the syndrome can be fed to a decoding algorithm implemented in the decoder. Many decoding techniques can be used, and are available in the case of the BCH codes. Any method can be used here, however most of them are costly to implement in hardware, and to circumvent this issue a decoding method aimed at security applications is presented in Section 3.3.3. With consideration to the chosen decoding method, and consequently the correction capabilities of the architecture, the injected fault (or a natural fault) may be correctable. If that is the case, the corrected codeword is forwarded to the system-level fault manager which, depending on the system-level parameters, can then output the corrected codeword to the next component. In the case where the fault is uncorrectable, the

fault manager itself can decide of the correct course of actions. For example, an all zero codeword can be chosen as output of the architecture, but this may lead to some information leakage. If this is a concern, it would also be possible to implement an RNG and output a random value instead, this would however increase the implementation costs. Moreover, the system-level manager can also decide of the course of action to follow if a fault was detected, no matter if it is correctable or not. If a fault is detected repeatedly, it may be wiser to stop the encryption module completely or proceed to some re-keying of the cryptographic hardware. This is however not always possible as some system cannot simply be stopped, for instance some airplane sub-systems, and in this case, it may be better to let the encryption continue, while still recording the fault events in a log. This must however be decided at a system-level and is not part of the architecture itself.
It should also be noted that the decoder is assumed to be tamper-proof, and therefore the fault flag, as well as the corrected output are assumed to be fault free. Otherwise, an attacker could directly attack the output of the decoder, or at least flip the detection flag. This is a common consideration for any EDC or ECC counter-measure, but it needs to be considered during the design of the cryptographic circuit.

3.3.1.2 Single Decoder Architecture

As proven in the original RK code paper [RK17], in addition to the robustness property, the code capabilities of the RK code are inherited from the original linear code used. In this case, the detection rate of the base architecture presented in Figure 3.3, and any of the architectures proposed in this thesis, is therefore the same as the one of the shortened BCH code chosen (from the code parameters). This also goes for the correction capability, and this architecture takes advantage of this fact. For a designer, this means that, if a higher detection rate is needed, a shortened BCH code with a higher distance can be chosen. Similarly, a larger distance also increases the correction capabilities of the RK code, and reduces the miscorrection rate. Alternatively, a decoding method which focuses on smaller fault multiplicities (such as the one in Section 3.3.3) can be implemented to save on implementation costs. The architectures presented in this work can be automatically chosen to meet any of those criteria, thanks to the possibility of choosing the distance of the shortened BCH code, and thus the RK code, but also thanks to the variety of decoding

methods which can be deployed. Moreover, the architecture itself can be implemented without any added latency, given a chosen decoding algorithm which can also be computed at runtime.

This general architecture can protect diverse encryption schemes. For example, block ciphers often have an internal state which can be divided according to the SBoxes which are used. The substitution layer of block ciphers is often constituted of 4-bit SBoxes, such as for some variants of the SSAES or the LED cipher [GPPR11], or take a byte as input (i.e. the AES). While other SBoxes do exist, or other sub-divisions are possible, the example architectures proposed in this work are aimed at those two most common sizes. It is, however, possible to generalise the architectures to any sizes (based on the SBoxes, other operations, or any other consideration).

In this regard, Figure 3.4 details a generic architecture for the protection of S-bit SBoxes, which was more specifically implemented for 4-bit and 8-bit ones, as well as in the case of 64-bit (and respectively 128-bit) ciphers (LED, PRESENT, SSAES and full scale AES). The input data of the ECC architecture is therefore of the size of the internal state of the ciphers, that is to say also either 64 or 128 bits. The data is then split into 4-bit nibbles or bytes before entering the original substitution layer. The division is dependent on the cipher and the respective size of the SBoxes, but, in contrast, the complete input data is forwarded to the predictor. In this case, the output of the original component is thus constituted of 16 S-bit blocks, which is taken as the information portion for the non-linear checker and decoder. Similarly, the redundancy for the predictor is based on a 16 S-bit symbol words, and fed to the checker. For this architecture, all the computations are therefore performed in $GF(2^S)$, meaning that both the detection and correction capabilities affect S-bit symbols (i.e. if $t = 1$, then a single S-bit symbol can be corrected, corresponding to a single SBox in this case). Finally, the checker computes the syndrome, as well as the potential corrected codeword, and outputs the relevant flags.

This architecture can be generalised for any layer of S-bit operations, but in this case the translation of the substitution layer to a codeword is straightforward, making this architecture an ideal and simple ECC counter-measure for such encryption schemes. The distance d can be chosen to be any value, at the cost of more computations. In this work, the focus being on 4-bit and 8-bit SBoxes, let us introduce the following notation for each

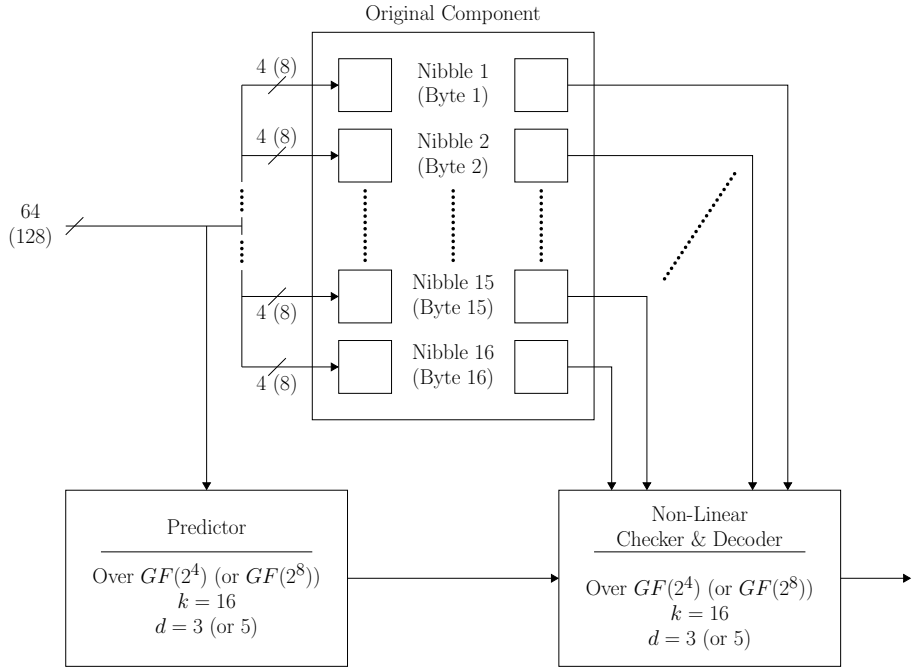

Figure 3.4: Architecture for S-bit SBoxes (here $S = 4$ or $S = 8$)

architecture. $S_{i_1}d_{i_2}m_{i_3}$ refers to an architecture for i_1-bit SBoxes, a distance i_2 for the code, over a Galois field $GF(2^{i_3})$. Furthermore, the distance considered for the RK code in the remainder of this chapter is either $d = 3$ or $d = 5$, as the detection rate is already extremely good for $d = 5$ (see experimental results in Section 3.4). Figure 3.4 illustrates the architectures $S_4d_3m_4$, $S_4d_5m_4$ and $S_8d_3m_8$.

Table 3.2: Single Decoder Architectures

Architecture	Galois field	Symbols			Bits		Distance	Masking Probability
	$GF(2^q)$	n_q	k_q	r_q	n	r	d	$\overline{Q}$
$S_4d_3m_4$	$GF(16)$	19	16	3	76	12	3	$\leq 1/8$
$S_4d_5m_4$	$GF(16)$	23	16	7	92	28	5	$\leq 1/8$
$S_8d_3m_8$	$GF(256)$	18	16	2	144	16	3	$\leq 1/128$

A summary of each architecture from Figure 3.4 is given in Table 3.2. More precisely, the different sizes, in terms of symbols and bits, as well as the error masking probabilities for a given distance are detailed. One important fact to note from this table is that the increase in distance d does not change, the error masking probability of the code (which is inferior to 1, since the code is robust). However, a larger distance increases the implementation

costs (more redundant symbols and operations). Consequently, if the hardware constraints can be met, a larger distance should be chosen to improve the detection rate of the code, as well as the correction capability.

3.3.1.3 Dual Decoder Architecture

An architecture like $S_8 d_3 m_8$ can be employed to protect a 128-bit cipher, such as the AES. However, operations over $GF(2^8)$ can be expensive to implement. A solution to this issue is the implementation of an RK-based ECC architecture over $GF(2^4)$ instead of $GF(2^8)$. This way, the computations remain in a smaller field, and are thus easier, and it is still possible to correct nibbles-faults with a clever implementation. Figure 3.5 shows the correct way to implement an RK-based ECC over $GF(2^4)$, which can still correct faults potentially injected in a complete byte of the internal state. Each output byte of the original component is split into two 4-bit nibbles. Each even and odd 4-bit nibble, where an even nibble corresponds to the four least significant bits of a byte, and an odd nibble to its four most significant bits, is then fed respectively to a distinct non-linear checker. A similar split is performed inside the predictor (or predictors, dependent on the implementation choices) and then forwarded to the corresponding checker as well. In turn, this architecture has two different decoders, instead of a single one, but can still detect faults which affect a same byte. This would not be the case if, for instance, the output of the original SBox layer was split into two 64-bit large chunks, each fed to a dedicated decoder (in that case, only 4-bit faults over each byte could be detected and corrected). Moreover, this architecture, either $S_8 d_3 m_4$ or $S_8 d_5 m_4$ in the case of the experimental results of this work, has a slightly better detection rate than a single decoder architecture, independently of the distance d (see Section 3.4).

Table 3.3: Dual Decoder Architectures

Architecture	Galois field	Symbols			Bits		Distance	Masking Probability
	$GF(2^q)$	n_q	k_q	r_q	n	r	d	$\overline{Q}$
$S_8 d_3 m_4$	$GF(16)$	38	32	6	152	24	3	$\leq 1/8$
$S_8 d_5 m_4$	$GF(16)$	46	32	14	184	56	5	$\leq 1/8$

Table 3.3 summarises the different sizes and parameters of the dual decoder architectures in the case of a 128-bit internal state, for both $d = 3$ and $d = 5$. Table 3.4 on the other hand

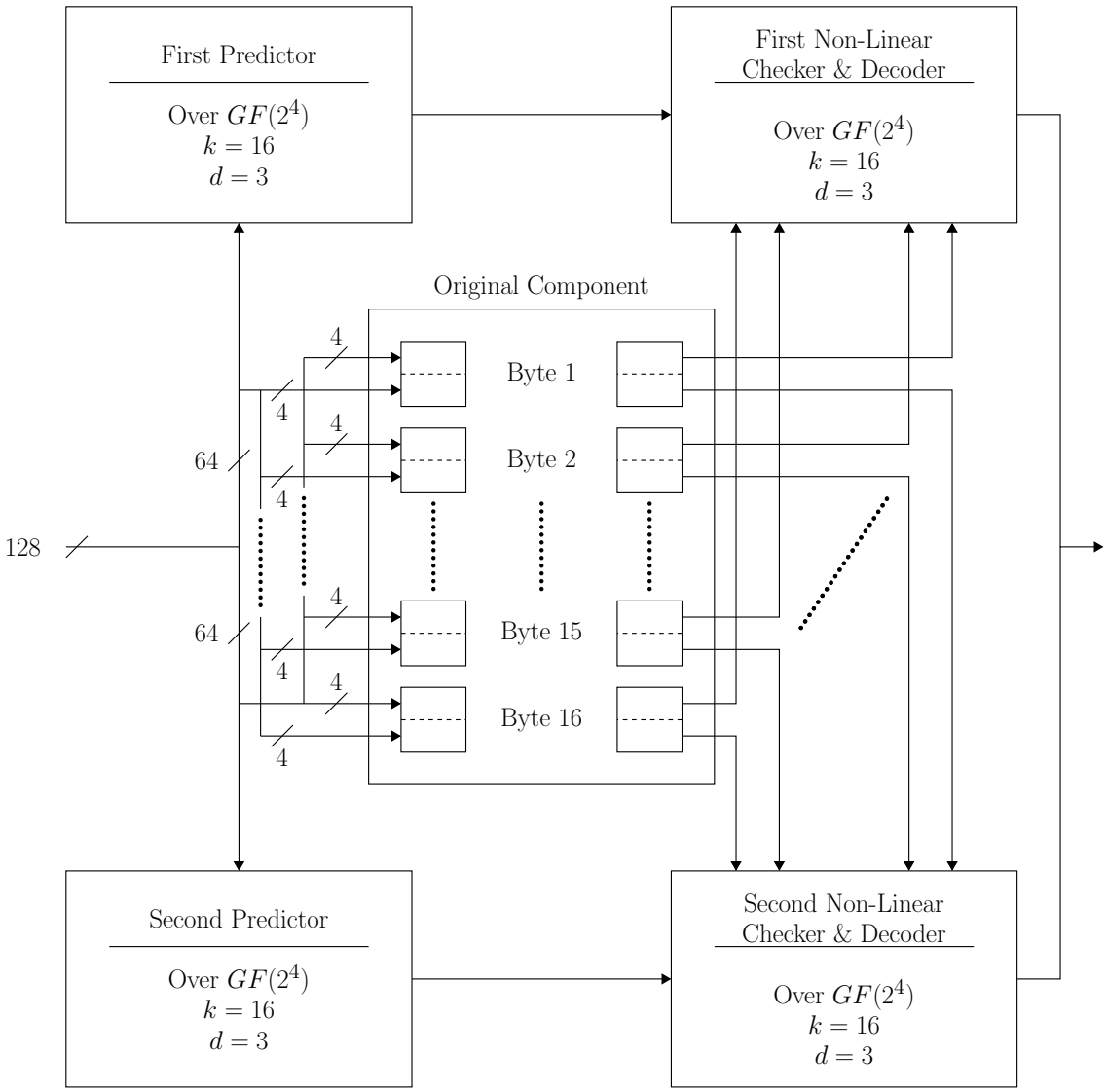

Figure 3.5: Dual Decoder Architecture for 8-bit Sboxes

shows the complexity of each architecture in terms of number of operations over $GF(2^4)$. While the table clearly showcases that a higher distance implies a higher number of operations (which is expected), and thus an increase in implementation cost, the comparison between $S_8d_3m_8$ and $S_8d_3m_4$ is of particular interest, as both architecture are aimed at correcting single byte errors, thanks to the choice of distance $d = 3$, and are potential candidates for the same application for a designer. The dual decoder setup $S_8d_3m_4$ has double the number of divisions, but also requires far less multiplications and additions to be performed (in bold in the table), compared to $S_8d_3m_8$. Therefore, it is advisable to implement an RK-based architecture with dual decoders over $GF(2^4)$ to protect ciphers with operations performed on byte oriented internal states (unless $GF(2^8)$ operations modules are already implemented and available during the computation of the syndrome). A further discussion on implementation costs, especially in terms of Combinational Logic Blocks (CLBs) of an FPGA, is available in Section 3.4.

Table 3.4: Complexity of each Architecture in terms of Operations

Architecture	Galois field	Bits			Number of Operations in $GF(2^4)$		
	$GF(2^q)$	n	r	d	#DIV	#MUL	#ADD
$S_4d_3m_4$	$GF(16)$	76	12	3	16	48	45
$S_4d_5m_4$	$GF(16)$	92	28	5	16	112	105
$S_8d_3m_8$	$GF(256)$	144	16	3	16	**240**	**188**
$S_8d_3m_4$	$GF(16)$	152	24	3	32	**96**	**90**
$S_8d_5m_4$	$GF(16)$	184	56	5	32	224	217

The general RK-based architecture from Figure 3.3, as well as the two example architectures, either single decoder-based or dual decoders, showcase how to build a robust ECC architecture for application over diverse ciphers. Each architecture can be generalised to any cipher and benefits from the good properties of the shortened BCH code, as well as the robustness property of the corresponding RK, which is of the utmost importance for cryptographic applications. The construction of these architectures can therefore easily be automated to fit the need of any hardware implementation of an encryption scheme, offering a simple way to implement ECC counter-measure with a higher security level, thanks to the robustness property of the code.

3.3.2 Inner/Outer Code Architectures

The basic RK-based architecture, while perfectly functional, is subject to miscorrection, similarly to other ECC. This is of course dependent on the implemented correction method, but it can be a problem in the few cases where a fault is identified as correctable, even though it is not (i.e. it exceeds the correction capability of the code). In such a case, a faulty value is passed to the next component, which can result in system-level issues, but also potential security threats, as the output of the ECC architecture is then faulty in a way which may still be leverage-able by a malicious attacker. Handling faults which may result in a miscorrection can be done at a system-level, since the fault was detected faulty in the first place. In this case, the system-level fault manager could, for instance, proceed to some re-keying of the encryption scheme. This is however still a limitation of the basic architecture. Moreover, a highly capable attacker who fully knows the hardware ECC counter-measure employed, in this case, the RK-based architecture chosen, can potentially create such a miscorrected or undetected fault for a given input. This case, for which the author of [CDF+08] first raised a concern (and described in Section 2.2.1.2), assumes a total control over the input and the capacity to inject very specific fault (i.e. mathematically constructed undetected faults). The physical realisation of an attack of this kind is therefore challenging. Nonetheless, a solution to this issue is needed.

A method to avoid miscorrections as much as possible, under strong attacker models, where the attacker is able to control both the input and the fault at the same time, consists in implementing a second layer of code, which checks if the correction was indeed successful or potentially incorrect. This improvement is also possible to implement in the case of the basic RK-based architecture [KGKP18], but first it is important to clearly identify the different cases which arise during the decoding process of an RK-based ECC architecture. Fault events for ECCs can be divided into four classes, denoted C1 to C4 in the remainder of this thesis.

- **Class C1: Undetected Faults by the RK code:** Faults for which the computed syndrome was zero, and thus were undetected by the architecture. Fault events of class C1 occur when an erroneous codeword is mapped into a valid codeword by the fault.

- **Class C2: Single Faults:** In this work, the focus was on the correction of single faults (i.e. faults affecting a single nibble). Therefore, this class only concerns single faults in the context of this thesis (decoding method described in 3.3.3). However, this classification can be extended to any fault which is smaller than the correction capability of the code, and thus always successfully corrected. Fault events of class C2 correspond to the case where a fault transforms a codeword into an invalid codeword, which is still at distance inferior or equal to $t = \left\lfloor \frac{d-1}{2} \right\rfloor$, the correction capability of the code ($t = 1$ in this work).

- **Class C3: Recognized as Suspicious:** Faults which affect more than t symbols, and for which the correction algorithm was not able to provide a corrected output.

- **Class C4: Miscorrection:** If an erroneous codeword, of multiplicity superior or equal to t, is mapped into an invalid codeword at a distance inferior or equal to t of a valid codeword, then the correction algorithm will miscorrect the codeword into an incorrect, but valid, codeword (i.e. the codeword is different from the original codeword). Any fault of this type belongs to class C4.

This fault classification can also be applied to dual decoder architectures, such as the ones from Table 3.3. In this case, the classification of a fault event for the overall architecture is dependent on the impact of the fault on each decoder, and the corresponding classification for each decoder. Table 3.5 shows how the fault event can be classified for the overall architecture.

Table 3.5: Dual Decoder Fault Classification

		First decoder fault classification			
		C1	C2	C3	C4
Second decoder fault classification	C1	C1	C4	C3	C4
	C2	C4	C2	C3	C4
	C3	C3	C3	C3	C3
	C4	C4	C4	C3	C4

Faults which belong in class C1 and C4 are critical, as they are not properly dealt with by the RK-based architectures alone, and correspond to the case discussed at the beginning of this section, where an attacker may be able to leverage those faults. One possible way to decrease the probability that a fault belongs in either class C1 or C4 is to implement an RK code with a higher distance d, while limiting the correction capability t, such that $t \leq \left\lfloor \frac{d-1}{2} \right\rfloor$. However, while this solution reduces the number of critical faults for a specific input, it also implies a higher implementation costs, and worsen the code rate. Therefore, this is not an ideal solution for constrained devices, and another method to deal with faults of class C1 and C4 is needed for such applications. A detection mechanism for faults belonging in those two classes would result in a better handling high-risks faults. To this end, it is only logical to implement a second layer of code, able to detect potential miscorrections, as well as undetected errors. Two-layers construction are known, for example the Single Error Correction Double Error Detection (SECDED) Hamming code [Ham50], which uses a single additional parity bit to improve on the basic Hamming code. In the remainder of this section, an inner-outer code RK-based architecture construction is proposed to handle critical faults.

Figure 3.6 shows an RK-based implementation with an added Quadratic-Sum (QS) code as outer code. The QS code takes the original component data as input and computes a single additional redundant symbol, in the considered field and per decoder, which is appended to the input data of the RK code. The RK code then proceeds as in the basic architecture case. The output of the RK code is then processed by the QS decoder, which recomputes the redundancy of the QS code and compares it to the symbol computed by the QS predictor. If both symbols are equal, then the RK correction is assumed to be successful, else a miscorrection occurred.

The QS code is also a robust code, and consequently, no non-zero fault go undetected with probability 1, and the probability that a fault goes undetected is dependent on the input data. In this specific use case, this means that no undetected fault for the RK code layer, or no miscorrected code word go through the second, outer, layer of code with high probability, and independently of the input. While theoretically, it would be possible for an attacker with complete knowledge of the ECC architecture to compute a fault which goes through both layers of code, either undetected or miscorrected, which would still be

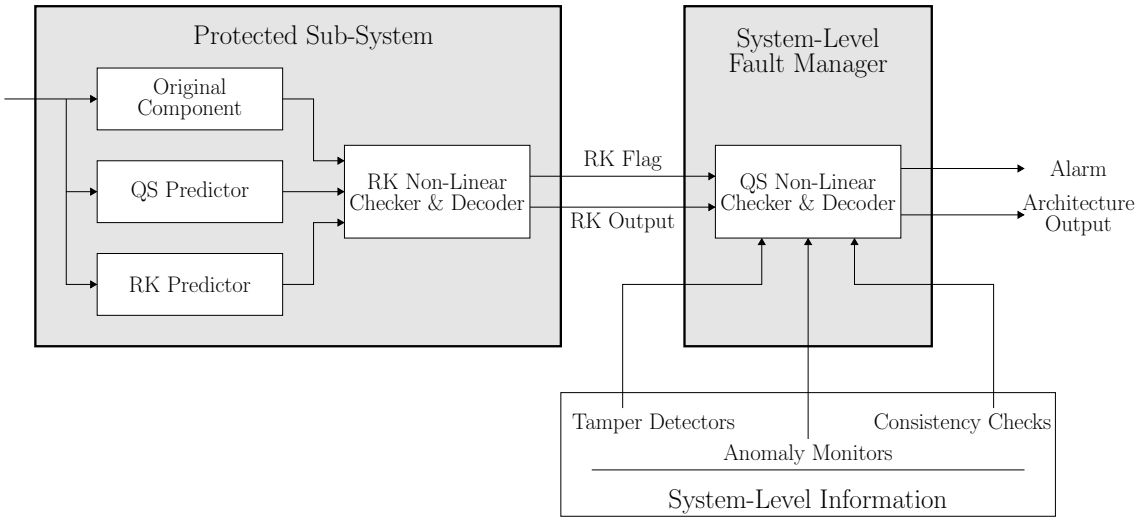

Figure 3.6: Inner-Outer Code Architecture

a threat for the sensitive data processed by the architecture, the number of such faults is low. Moreover, an attacker is usually unaware of precisely which (key-dependent) internal state is affected by the fault, and faults which affect the architecture in this way may be hard to inject. In the case where several fault injections are necessary, this becomes a daunting task for an attacker. In practice, such an architecture circumvents even more cases where the attacker knows the full code and can choose the input to create a fault that is undetected, or miscorrected, by the RK code, rendering even fault injection attacks with a strong attacker model ineffective.

The inner-outer code architecture achieves a protection mechanism against both fault events of classes C1 and C4. Figure 3.7 described an extension of the classification of fault events to include the outer code system-level fault manager. Two new classes of faults are introduced, S1 and S2, corresponding respectively to a successful detection of fault events of the two former classes, and unsuccessful detection. In more details, classes S1 and S2 are as follows.

- **Class S1: Faults Detected by the Outer code:** undetected or miscorrected faults by the RK inner code layer, which are successfully identified as erroneous by the outer code.

- **Class S2: Faults Undetected by the Outer code:** Faults undetected or miscorrected by the RK inner code, and further undetected by the outer code. This is the worst case scenario, where both codes did not manage to handle the fault properly.

At a system-level, it is clear that fault events of classes C2 and C3 do not require any further handling, and that fault events of class S2 are the only potential threats to the system, and may be used by an attacker.

The choice of the QS code as outer code for this architecture was therefore made with regard to its robustness property, but also its low implementation cost. Let us denote the redundancy symbol of the QS code by r_{QS}. For an input data $x = (x_1, .., x_k) \in GF(2^m)^k$, and k even (which is the case for the architectures presented in this work), r_{QS} is computed as defined in Equation 2.33.

$$r_{QS} = \sum_{i=1}^{\frac{k}{2}} x_{2i-1} x_{2i} \tag{3.2}$$

The hardware cost for such a computation is low, especially if multipliers over $GF(2^m)$ are already available (which is the case for many block ciphers). There is however an other additional cost to the implementation of an inner-outer code architecture. After the encoding of the QS code, the input data for the RK code becomes $(x_1, .., x_k, r_{QS})$. The additional symbol means that a slightly larger RK encoder and decoder have to be implemented as well. This increase in data size is however low, since it is only a single symbol, especially for larger encryption schemes (for instance, an architecture for the AES). It would be possible to skip the RK encoding for this specific symbol, and only compare the predictor output in the QS decoder, thus avoiding the extra symbol for the RK code. The drawback of doing so, would be that any potential tampering with the QS predictor would then go undetected, and consequently nullify the improvement of the inner-outer code architecture in the presence of a strongly capable attacker. The hardware costs of the inner-outer code architecture are discussed in details in Section 3.4.

Since the computation of the additional redundancy symbol introduced by the QS code can be done over any Galois field, an inner-outer code architecture can still be implemented for any cipher. It should be noted that, in the case where k, the number of symbols of the input data, is odd, other cheap to implement robust codes can be used, such as

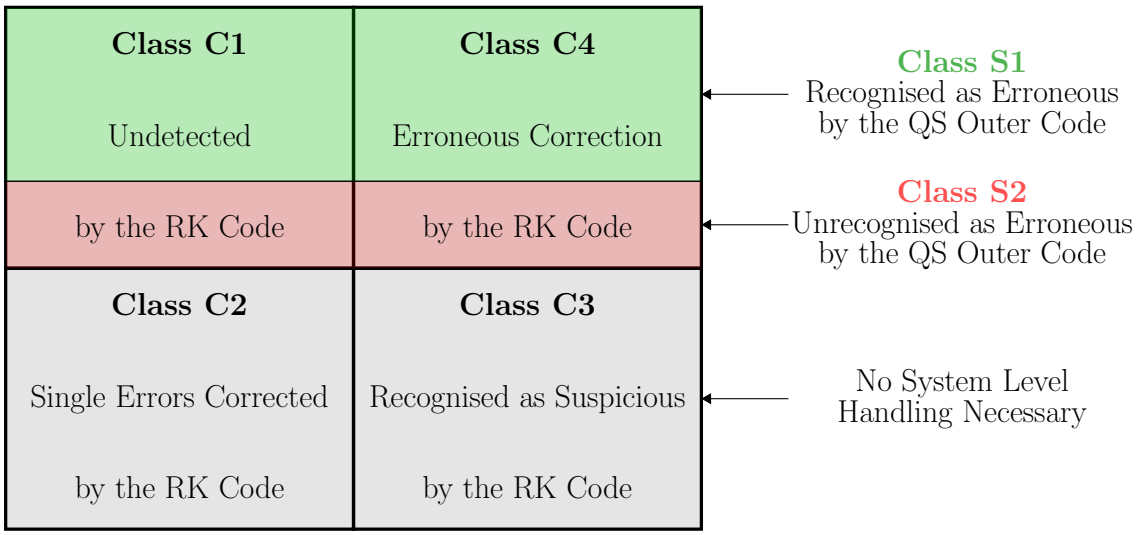

Figure 3.7: Classification of Fault Events at Code and System-Level

Compact Protection Codes (CPC) presented in Section 3.3.4, which can be constructed to suit any sizes. Therefore, and in the continuation of the basic RK-based architecture, the inner-outer code architecture can be implemented in an automated fashion for any cryptographic application, given the correct parameters. Even in the case of extremely constrained devices, where every gate counts, this improvement can simply be omitted. Of course, this is at the expense of security, but it can be a necessary trade-off for this use case.

Finally, compared to AMD codes [CDF+08], the inner-outer code architecture is theoretically vulnerable to a highly capable attacker able to precisely inject a fault and control the inputs. However, in a practical scenario with physical fault injections, the inner-outer code architecture protects the cryptographic circuit against even strong attacker models. Additionally, no true RNG are required, avoiding any risk of RNG failure and making this architecture a compelling choice for protection against fault injection attacks.

3.3.3 Error Coefficient and Location Table (ECLT) Decoding

The error correction process for the RK-based architecture is the same as for the linear code it is composed of, with the addition of an APN function at the end (here an inversion in $GF(2^m)$). This means that, for the shortened BCH code, different correction algorithms

can be applied. Decoding techniques such as the Berlekamp-Massey algorithm [Ber84] can be implemented, but they are very costly in hardware and better suited for software decoding. Another decoding method is based on the syndrome and is a better choice for hardware implementation of RK-based architectures.

Figure 3.8 shows the classical way to perform decoding based on the syndrome. w and ϵ_w refer respectively to the redundancy given by the predictor and a potential fault injection effect on this redundancy. Similarly, x and ϵ_x are respectively the output of the protected component and a potential related fault, as defined in Figure 3.3. The matrix A_d is computed as described in Algorithm 3.1, and is used instead of the complete H_d to reduce the complexity of the implementation. This simplification of the implementation is possible, since the other half of H_d is the identity matrix and, for this stage of this decoding method, only the redundant portions are of interest to compute the syndrome. For simplicity sake, the outer code presented in Section 3.3.2 is omitted here. The difference in the case of an inner-outer code architecture would be that the input of the inversion function, here $x + \epsilon_x$, would be expanded to $(x, r) + \epsilon_{(x,r)}$, with r the redundancy added by the outer code. The rest of the RK decoding process would then be the same.

The decoding process goes through different stages in order to correct the fault which may have occurred. First, to provide the robustness property of the RK code, the output of the original component is inverted (more generally, the APN function chosen for the RK code is applied to the input), before being multiplied by A_d in order compute the redundancy w' associated with the input $x + \epsilon_x$ (shortened BCH encoding). w and w' are then XORed into the syndrome s. Equation 3.3 gives the expression of the syndrome, as a vector.

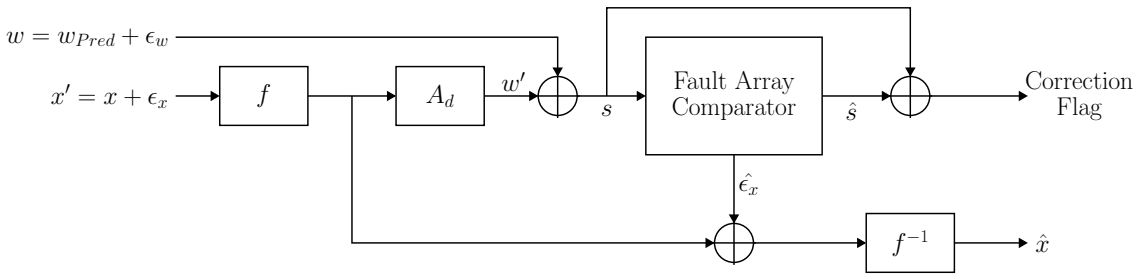

Figure 3.8: Classical Syndrome Decoding Process for an RK-based Architecture

$$
\begin{aligned}
s &= w \oplus w' \\
&= (A_d \cdot f(x)^\top \oplus \epsilon_w) \oplus (A_d \cdot f(x+\epsilon_x)^\top) \\
s &= A_d \cdot (f(x)^\top \oplus f(x+\epsilon_x)^\top) \oplus \epsilon_w
\end{aligned}
\tag{3.3}
$$

If $s = 0$, then no fault were detected (i.e. $\epsilon_x = 0$ and $\epsilon_w = 0$), and the data can be forwarded to the next component. If this is not the case, then a fault occurred and the decoder attempts to correct it. Within the error correction capability t of the RK code, each possible fault of multiplicity inferior or equal to t can be associated to a unique syndrome. Therefore, in order to correct the invalid codeword, s is compared with an array of faults $\hat{\epsilon} = (\hat{\epsilon_x}, \hat{\epsilon_w})$, and if a match is found, then the codeword is corrected as follows.

$$
\hat{x} = f^{-1}(f(x \oplus \epsilon_x) \oplus \hat{\epsilon_x}) \tag{3.4}
$$

Note that, in this case, the potentially faulty redundancy $w \oplus \epsilon_w$ can also be corrected by XORing it with $\hat{\epsilon_w}$. Moreover, in the context of this work, since $f(x) = x^{-1}$, then $f^{-1}(x) = f(x) = x^{-1}$. If the fault multiplicity was inferior or equal to the correction capability of the RK code, then $\hat{x}$ is successfully corrected ($\hat{x} = x$). Otherwise, a miscorrection occurred.

The classical syndrome decoding method allows for correction of faults of multiplicity up to t, the correction capability of the code, but they require the computation of the fault array for matching with the syndrome. On the one hand, this computation can be done at runtime, saving memory space at the price of area and potentially latency, depending on the implementation. On the other hand, the array can be pre-computed, but then needs to be stored on the device. Since the array can be very large, this solution would have a high memory cost. Both methods are therefore costly, and a more efficient decoding technique for hardware constrained scenarios is needed.

The main drawback of syndrome decoding in term of hardware implementation is the size of the table which needs to be computed for syndrome matching. In order to reduce the cost of such a table, a smaller Error Coefficient and Location Table (ECLT) can be used, with only small modification to the decoding algorithm [GKKP19]. Algorithm 3.2 describes a new decoding algorithm which only requires the smaller ECLT, instead of a

complete table for containing every possible fault and its matching syndrome. This new decoding method is aimed at single faults (i.e. faults of multiplicity 1). In a security context, fault injection attacks generally target a single symbol of an intermediate state. This is because, as explained in Section 2.1, precise fault injections lead to less overlapping in the fault propagation, and thus the possibility to generate better equations for solving. Consequently, the correction of single faults is of the utmost importance, and focusing on such low multiplicity faults reduces the size of the decoding table, as less entries are needed (even without the implementation of an ECLT). In the remainder of this chapter, ECLT-based decoding is thus only implemented for single fault corrections. Faults of larger multiplicity can however still be detected, since the syndrome is intact, meaning that the ECLT decoding does not impair the detection capability of the architectures.

Algorithm 3.2: Error Coefficient and Location Table (ECLT) Single Fault Decoding

1 Compute $s = A_d \cdot (f(x)^\top \oplus f(x + \epsilon_x)^\top) \oplus \epsilon_w$
2 Normalize the first $(m + 1)$ symbols of the syndrome $\hat{s} = (\frac{s_1}{s_j}, \frac{s_2}{s_j}, ..., \frac{s_{m+1}}{s_j})$.
3 Find normalized syndrome in ECLT and determine f_i and i.
4 If found, correct the fault at position i: $\hat{x_i} = f(f(x_i \oplus \epsilon_{x_i}) \oplus s_j f_i)$.
5 Update the syndrome $\tilde{s} = s \oplus s_j f_i h_i$.
6 If the syndrome is equal to zero, there was a single fault. Else, more faults occurred.

ECLT-based RK Architecture - Case Distance $d = 3$

In order to better understand the improvement brought by the ECLT, the case where $d = 3$ is considered first. The initial step in the proposed ECLT-based decoding (Algorithm 3.2), which is the same as the previous method, is to compute the syndrome s. Once s is computed, the first non-zero symbol of s, denoted s_j, is extracted and the syndrome is normalised into $\hat{s} = (\frac{s_1}{s_j}, \frac{s_2}{s_j}, ..., \frac{s_{m+1}}{s_j})$ (here, m is from line 4 of Algorithm 3.1). In the case where $d = 3$, the initial steps therefore consist of normalising the complete syndrome, since the syndrome is constituted of $m + 1 = 3$ symbols. The decoding algorithm then reads through the ECLT to find the corresponding normalised syndrome, and if there is a match, fetches the corresponding fault position i and coefficient for fault correction f_i. The coefficient is multiplied by s_j and XORed with the symbol of the faulty data at position i in order to correct the fault. Finally, the syndrome is updated to verify that the fault has

been corrected. If $s \neq 0_{GF(2^m)}$, more faults occurred and the correction was invalid. The ECLT-based decoding algorithm is therefore only slightly different from the more standard decoding method of Figure 3.8, as it only introduces very few more operations (namely an inversion and a few multiplications, see Figure 3.9 for comparison) and a new table.

$$H_{orig,3} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 11 & 5 & 9 & 3 & 1 & 5 & 0 & 13 & 7 & 3 & 3 & 8 & 11 & 6 & 11 & 4 & 6 & 15 \\ 0 & 7 & 11 & 2 & 2 & 1 & 3 & 4 & 3 & 9 & 4 & 10 & 7 & 2 & 12 & 4 & 7 & 0 & 1 \end{pmatrix}$$

Matrix 3.1: Original non-systematic BCH check matrix $H_{orig,3}$

Let us now detail why such a decoding method is valid, especially why $\epsilon_i = s_j f_i$, and how to construct the ECLT. Recall the parity check matrix in its systematic form, $H_3 = (A_3 \mid I)$ (used instead of $H_{orig,3}$ in Matrix 3.1). By definition of the BCH codes, we have $s = h_i \epsilon_i$, where i is the position of the fault ϵ_i, and h_i the corresponding column of H_3. It should be noted that this reasoning can be extended to larger distances and faults of higher multiplicities, even though the focus in this work is only on single faults. Let us now consider the first non-zero coefficient of h_i, g_i. Since $g_i \neq 0_{GF(2^m)}$, it is possible to normalise h_i into $\hat{h_i} = \frac{h_i}{g_i}$, and thus we also have the following.

$$h_i = \hat{h_i} g_i \tag{3.5}$$

The syndrome s can now be expressed in terms of $\hat{h_i}$ and g_i.

$$s = h_i \epsilon_i \Leftrightarrow s = \hat{h_i} g_i \epsilon_i \tag{3.6}$$

The normalised syndrome $\hat{s} = (\frac{s_1}{s_j}, \frac{s_2}{s_j}, \frac{s_3}{s_j}) = \frac{s}{s_j}$ gives us $s = \hat{s} s_j$. Moreover, by definition $\hat{s} = \hat{h_i}$, therefore we get the following equation.

$$\begin{aligned} s = \hat{h_i} g_i \epsilon_i \Leftrightarrow \hat{s} s_j &= \hat{h_i} g_i \epsilon_i \\ \Leftrightarrow s_j &= g_i \epsilon_i \\ s = \hat{h_i} g_i \epsilon_i \Leftrightarrow \epsilon_i &= \frac{s_j}{g_i} \end{aligned} \tag{3.7}$$

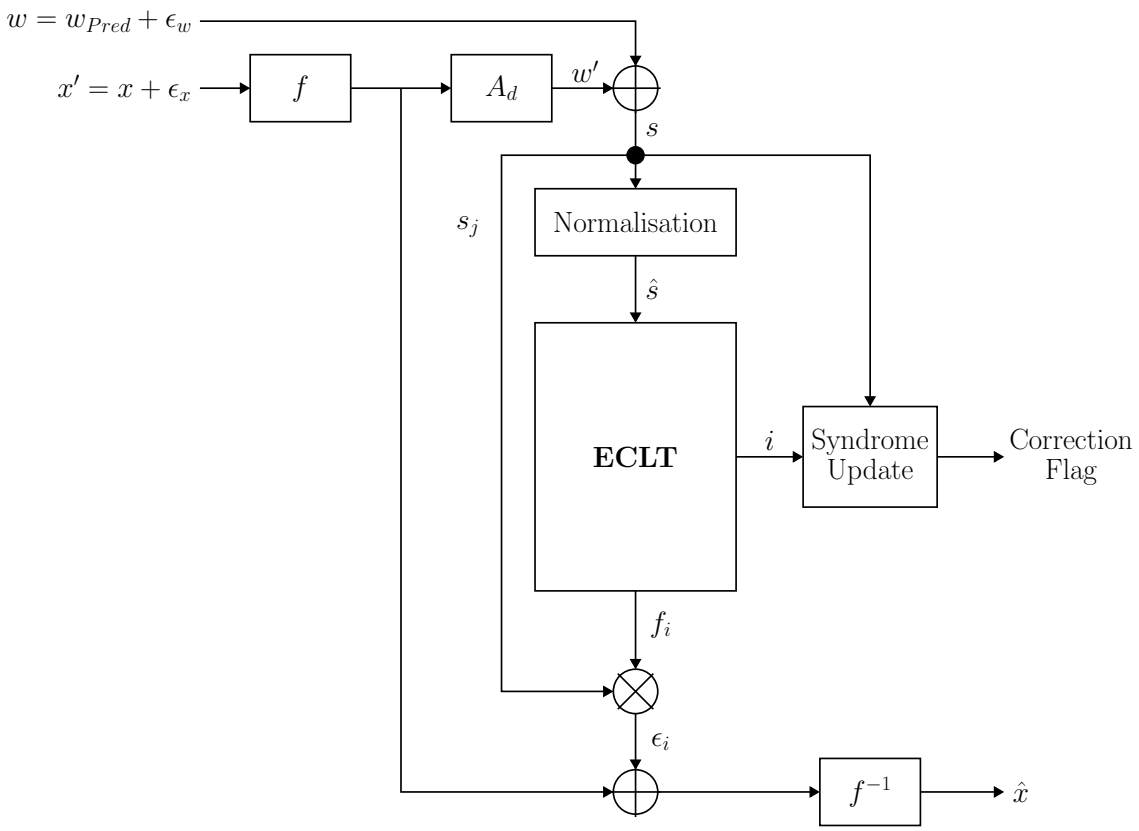

Figure 3.9: Low-Complexity ECLT-based decoder

The inverse of g_i in $GF(2^m)$ can be denoted as f_i.

$$\epsilon_i = s_j \cdot g_i^{-1} = s_j \cdot f_i \tag{3.8}$$

The fault can therefore indeed be expressed in terms of the first non-zero value of the syndrome s, its position i, and a coefficient f_i. Consequently, an ECLT can be generated by computing f_i, and storing the position i, for every possible $\hat{s} = \hat{h_i}$. Moreover, during the decoding process, extracting s_j and matching $\hat{s}$ is simple, and only consists of a limited overhead in terms of operations, while using a much smaller table compared to more classical syndrome decoding techniques. The ECLT is in fact a table of size $n \cdot ((m+1)+2)$ (m from Algorithm 3.1). Each row contains a normalised syndrome $\hat{s}$ ($m+1$ symbols), the position i and the coefficient f_i. For instance, a table for the standard decoding method of Figure 3.8, in the case of an SSAES $SR^*(10, 4, 4, 4)$ (see Section 4.1), and a $[19, 16, 3]_{16}$ RK code, would be constituted of $n \cdot (q-1)$ entries. Each entry would be the concatenation of a syndrome, a position and the value of the fault to be corrected. In terms of bits, this would amount in $19 \cdot 15 \cdot (12+5+4) = 19 \cdot 15 \cdot 21 = 5985$ bits. In comparison, an ECLT-based decoder (Figure 3.9) would only have a $n \cdot (9+5+4) = 19 \cdot 18 = 342$ bit table, or more than 17 times smaller. Note that the number of bits for the normalised syndrome $\hat{s}$ is only 9, since, thanks to the normalisation, the first symbol is always at most 1.

In the case of larger ciphers, the reduction is even more significant. For example, for a full scale AES, and thus with 8-bit SBoxes, the conventional table would have $n \cdot (q-1) = 19 \cdot 255 = 4845$ entries of 37 bits, against only 19 entries of 30 bits for the ECLT (more than 300 times smaller), further reinforcing the benefits of such an approach for decoding.

Example 3.1. *In this example, an ECLT-based decoder for a* $[19, 16, 3]_{16}$ *RK architecture is considered. For simplicity sake, a single fault is injected at the output of the component to be protected, for instance the substitution layer, and the output of the predictor is considered untouched. On the one hand, the faulty output is as follows.*

$$x' = x \oplus \epsilon_x = (9, B, 9, 3, B, \underline{\mathbf{3}}, 2, 2, C, 7, 1, D, 1, 9, 3, 5)$$

Table 3.6: ECLT for a $[19, 16, 3]_{16}$ RK Code Architecture

$\hat{s}$			i	f_i
1	A	C	0	7
1	9	1	1	9
1	D	6	2	A
1	C	4	3	9
1	F	5	4	B
1	E	5	5	A
1	E	1	6	E
1	D	0	7	C
0	1	D	8	C
1	A	3	9	8
1	6	D	10	A
1	5	8	11	C
1	1	D	12	D
1	5	3	13	7
1	6	5	14	2
1	E	7	15	8
1	0	0	16	1
0	1	0	17	1
0	0	1	18	1

On the other hand, the predictor computes the correct value.

$$x = (9, B, 9, 3, B, \underline{\mathbf{E}}, 2, 2, C, 7, 1, D, 1, 9, 3, 5)$$

The corresponding redundancy is $w = (3, 9, 5)$. *The fault itself only affects the sixth symbol of the data (highlighted in bold).*

The following for the decoding of x' *steps refer to the steps of Algorithm 3.2.*

1. *First, the syndrome* s *is calculated. To do so,* x' *is first inverted and then multiplied by* A_3*, in order to get the corresponding redundancy* w'*. In this case,* $w' = (0, 8, A)$*, and thus* s *can be computed as follows.*

$$s = w \oplus w' = (3, 9, 5) \oplus (0, 8, A) = (3, 1, F)$$

2. *Since for this example $d = 3$, then $(m + 1) = r = 3$ and thus the complete syndrome is normalised. The first non-zero value of s is $s_j = 3$.*

$$\hat{s} = \frac{s}{s_j} = (\frac{3}{3}, \frac{1}{3}\frac{F}{3}) = (1, E, 5)$$

3. *Table 3.6 is the ECLT for a $[19, 16, 3]_{16}$ RK code. A match for $\hat{s}$ is found in the table, and the corresponding position and coefficient are $i = 5$ (which correspond to the sixth symbol of the original vector, and indeed the fault position) and $f_i = A$.*

4. *The effect of the fault after the inversion, denoted ϵ (note that $\epsilon \neq \epsilon_x$), can now be computed. $\epsilon_i = s_j \cdot f_i = 3 \cdot A = D$ and thus:*

$$\epsilon = (0, 0, 0, 0, 0, D, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$$

With this value of ϵ, the data can be corrected after a final inversion (in the following equation ϵ is truncated, since the corrected symbol is part of the information portion).

$$\begin{aligned}
\hat{x} &= f(f(x \oplus \epsilon_x) \oplus \epsilon) \\
&= f((2, 5, 2, E, 5, \underline{\mathbf{E}}, 9, 9, A, 6, 1, 4, 1, 2, E, B) \\
&\quad \oplus (0, 0, 0, 0, 0, D, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)) \\
&= f((2, 5, 2, E, 5, \underline{\mathbf{3}}, 9, 9, A, 6, 1, 4, 1, 2, E, B)) \\
&= (9, B, 9, 3, B, \underline{\mathbf{E}}, 2, 2, C, 7, 1, D, 1, 9, 3, 5) \\
\hat{x} &= x
\end{aligned}$$

5. *Finally, the syndrome is updated.*

$$\tilde{s} = s \oplus s_j f_i h_i = (3, 1, F) \oplus D \cdot (12, 4, 9) = (0, 0, 0)$$

6. *Since the syndrome is zero, a single fault occurred and was successfully corrected.*

ECLT-based RK Architecture - Case Distance $d > 3$

In the case where $d = 3$, it is clear how to proceed for the correction of single faults, since the correction capability of the code is by definition $t = 1$ and the complete syndrome is used for decoding. However, while a small distance $d = 3$ is sufficient for the correction of single faults, choosing an RK code with a distance $d > 3$ has some advantages in combination to the ECLT decoding method. Namely, if the distance is larger than 3, then decoding with an ECLT allows to reduce even further the size of the table, especially for single faults, and the number of miscorrections is reduced (i.e. miscorrections are avoided up to $d - 2$ symbols, thanks to the syndrome update function).

In order to evaluate the benefits of an ECLT-based decoding, the correctness of decoding first needs to be generalised for any distance $d > 3$. In this regard, the following theorem shows how the ECLT for single fault correction can be generalised to any RK architecture.

Theorem 3.1. *The first $(m + 1)$ symbols of the syndrome s, uniquely define the location of a single fault and its value.*

Proof. Consider two faults ϵ_1 and ϵ_2 of multiplicity 1, such that $\epsilon_1 \neq \epsilon_2$. Let us denote by s_1 and s_2 the respective syndromes associated to ϵ_1 and ϵ_2.

$$s_1 = H_d \cdot \epsilon_1^\top \qquad \text{and} \qquad s_2 = H_d \cdot \epsilon_2^\top$$

Since the code is a linear code of distance $d > 3$, $s_1 \neq s_2$.

Let us define $s_{1_{[0:m]}}$ and $s_{2_{[0:m]}}$ the partial syndromes constituted of the first $(m+1)$ symbols of their respective complete syndromes, and assume that $s_{1_{[0:m]}} = s_{2_{[0:m]}}$ (i.e. the partial syndromes do not uniquely define the fault). Then, it is possible to write the following equation.

$$H_d \cdot (\epsilon_1 \oplus \epsilon_2)^\top = (0_{m+1}, \Delta_s)^\top \Leftrightarrow (A_d | I_d) \cdot (\epsilon_1 \oplus \epsilon_2)^\top = (0_{m+1}, \Delta_s)^\top \tag{3.9}$$

$\Delta_s \in \mathbb{F}_q^{r-(m+1)}$ (with r the number of redundant bits of the code) relates to the difference of the remaining symbols of the syndromes. Moreover, by definition $A_d = H_r^{-1} H_l$ and

$H_{orig,d} = (H_l|H_r)$ (Algorithm 3.1). Therefore:

$$\begin{gathered} H_r \cdot (A_d|I_d) \cdot (\epsilon_1 \oplus \epsilon_2)^\top = H_r \cdot (0_{m+1}, \Delta_s)^\top \\ \Leftrightarrow \\ H_{orig,d} \cdot (\epsilon_1 \oplus \epsilon_2)^\top = H_r \cdot (0_{m+1}, \Delta_s)^\top \\ \Leftrightarrow \\ H_{orig,d} \cdot (\epsilon_1 \oplus \epsilon_2)^\top \oplus H_r \cdot (0_{m+1}, \Delta_s)^\top = 0_{\mathbb{F}_q^r} \end{gathered} \tag{3.10}$$

Therefore, at least some columns of $H_{orig,d}$ are linearly dependent, which is impossible since $H_{orig,d}$ is the parity check matrix of a linear code of distance d. Consequently, for any faults ϵ_1 and ϵ_2 of multiplicity 1, $s_{1_{[0:m]}} \neq s_{2_{[0:m]}}$, and thus the first $(m+1)$ symbols of the syndrome of the RK code uniquely define a single fault. □

Moreover, this property can also be seen through the expression of the $H_{orig,d}$ and A_d matrices. By definition of the BCH code, the top left $(m+1) \cdot n$ sub-matrix of any parity check matrix $H_{orig,d}$ is in fact $H_{orig,3}$ (see Matrix 3.2 for an example of a $[23, 16, 5]_{16}$ RK code), and even though $A_d \neq A_3$, the first $(m+1)$ rows of any A_d have the property that any two or less columns are linearly independent (as can be seen in Matrix 3.3, also proving Theorem 3.1 for the case of a $[23, 16, 5]_{16}$ RK code).

$$H_{orig,5} = \left(\begin{array}{cccccccccccccccccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 11 & 5 & 9 & 3 & 1 & 5 & 0 & 13 & 7 & 3 & 3 & 8 & 11 & 6 & 11 & 4 & 6 & 15 & 13 & 3 & 10 & 6 \\ 0 & 7 & 11 & 2 & 2 & 1 & 3 & 4 & 3 & 9 & 4 & 10 & 7 & 2 & 12 & 4 & 7 & 0 & 1 & 15 & 12 & 12 & 6 \\ \hline 1 & 5 & 3 & 5 & 13 & 3 & 8 & 6 & 4 & 15 & 3 & 6 & 0 & 1 & 10 & 15 & 15 & 7 & 8 & 9 & 8 & 5 & 9 \\ 0 & 11 & 2 & 3 & 3 & 4 & 7 & 12 & 7 & 1 & 12 & 6 & 11 & 3 & 9 & 12 & 11 & 0 & 4 & 14 & 9 & 9 & 15 \\ \hline 1 & 9 & 5 & 7 & 8 & 11 & 15 & 10 & 0 & 10 & 15 & 11 & 8 & 7 & 5 & 9 & 1 & 1 & 9 & 5 & 7 & 8 & 11 \\ 0 & 2 & 3 & 9 & 7 & 4 & 1 & 12 & 11 & 11 & 12 & 1 & 4 & 7 & 9 & 3 & 2 & 0 & 2 & 3 & 9 & 7 & 4 \end{array}\right)$$

Matrix 3.2: Original non-systematic BCH check matrix $H_{orig,5}$ (in red $H_{orig,3}$)

In the general case of an RK architecture, it is therefore sufficient for single fault correction to only consider the first $(m+1)$ symbols of the computed syndromes. Nonetheless, the detection of faults of larger multiplicity is still handled by the complete syndrome of size r. This implies that the better detection rate (see Section 3.4) can be maintained for distances $d > 3$, while the ECLT decoding can be applied for single fault corrections,

$$A_5^\top = \begin{pmatrix} 1 & 12 & 0 & 15 & 0 & 14 & 13 & 6 & 15 & 12 & 3 & 9 & 15 & 10 & 0 & 15 \\ 11 & 12 & 12 & 3 & 15 & 8 & 8 & 2 & 5 & 2 & 2 & 15 & 10 & 13 & 10 & 3 \\ 5 & 2 & 12 & 10 & 3 & 12 & 4 & 5 & 4 & 12 & 13 & 9 & 9 & 14 & 13 & 12 \\ 10 & 4 & 2 & 0 & 10 & 5 & 7 & 13 & 9 & 5 & 1 & 8 & 5 & 1 & 14 & 1 \\ 7 & 8 & 4 & 9 & 0 & 6 & 0 & 6 & 6 & 11 & 12 & 11 & 3 & 6 & 1 & 5 \\ 15 & 15 & 8 & 14 & 9 & 5 & 1 & 4 & 12 & 14 & 9 & 2 & 1 & 15 & 6 & 11 \\ 12 & 0 & 15 & 0 & 14 & 13 & 6 & 15 & 12 & 3 & 9 & 15 & 10 & 0 & 15 & 14 \end{pmatrix}$$

Matrix 3.3: Matrix A_5 derived from $H_{orig,5}$

which are usually the faults which are injected by a malicious attacker, thus reducing the hardware implementation costs. Additionally, the ECLT can be extended to faults of larger multiplicities at the cost of the table's size. In this case, the first $(m+1)$ symbols are not sufficient to uniquely define a fault of higher multiplicity, but, for instance, the first 7 symbols would be sufficient to correct any fault of multiplicity inferior or equal to 2, for an RK architecture of distance $d \geq 5$ and $m = 2$. In this work, this case has however not been implemented, since the faults the architectures were aimed at were single faults.

In conventional syndrome decoding, for an $[n, k, d]_q$ RK code, a table of $(q-1)n$ entries of

$$\underbrace{r \log_2 q}_{\text{syndrome}} + \underbrace{\log_2(n)}_{\text{error-location}} + \underbrace{\log_2 q}_{\text{error-value}}$$

bits would be needed. In the general case for an ECLT-based decoder, and for the same code, only a table of $k + (m+1)$ entries of

$$\underbrace{(m) \log_2 q + 1}_{\text{syndrome}} + \underbrace{\log_2(n)}_{\text{error-location}} + \underbrace{\log_2 q}_{\text{error-value}}$$

bits is sufficient to correct single faults. For instance, in the case of a $[23, 16, 5]_{16}$ RK architecture, the conventional method would require $15 \cdot 23 \cdot (7 \cdot 4 + 5 + 4) = 12765$ bits to be stored, against only $19 \cdot (9+5+4) = 342$ bits for the ECLT. Consequently, an ECLT-based decoder reduces significantly the hardware cost of an RK architecture, for any chosen distance for the considered code, allowing the implementation of this counter-measure on more constrained devices.

Table 3.7: ECLT for a $[23, 16, 5]_{16}$ RK Code Architecture

$\hat{s}$			i	f_i
1	B	5	0	1
1	1	7	1	A
0	1	1	2	A
1	B	F	3	8
0	1	B	4	8
1	B	7	5	3
1	6	3	6	4
1	E	8	7	7
1	E	6	8	8
1	7	1	9	A
1	F	A	10	E
1	D	1	11	2
1	F	4	12	8
1	3	4	13	C
0	1	3	14	C
1	B	A	15	8
1	0	0	16	1
0	1	0	17	1
0	0	1	18	1

Example 3.2. *Similarly to Example 3.1, let us consider the following inputs, for a* $[23, 16, 5]_{16}$ *ECLT-based RK architecture and correct a single fault (shortened example).*

$$x = (5, 5, 0, \underline{\mathbf{2}}, D, C, 1, 2, A, C, B, D, E, D, C, 4)$$

$$w = (A, F, 2, B, 4, 6, 8)$$

$$x' = (5, 5, 0, \underline{\mathbf{6}}, D, C, 1, 2, A, C, B, D, E, D, C, 4)$$

1. **Computation of the syndrome s:** $w' = (F, E, 4, B, 3, D, 8)$ *and thus:*

$$s = w \oplus w' = (A, F, 2, B, 4, 6, 8) \oplus (F, E, 4, B, 3, D, 8) = (5, 1, 6, 0, 7, B, 0)$$

2. **Normalisation (first $(m + 1)$ symbols of s):** $s_j = 5$ *(first non-zero symbol of* s*).*

$$\hat{s} = \frac{s}{s_j} = (\frac{5}{5}, \frac{1}{5}\frac{6}{5}) = (1, B, F)$$

3. **ECLT matching:** *From the table, the fault's position is* $i = 3$ *and the coefficient is* $f_i = 8$.

4. **Correction of the fault:** *Computation of the fault effect:*

$$\epsilon_i = s_j \cdot f_i = 5 \cdot 8 = E$$

$$\epsilon = (0, 0, 0, E, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$$

The data can then be corrected:

$$\begin{aligned}
\hat{x} &= f(f(x') \oplus \epsilon) \\
&= f((B, B, 0, \underline{\mathbf{7}}, 4, A, 1, 9, C, A, 5, 4, 3, 4, A, D, 8, 3, D, 5, E, 4, F) \\
&\quad \oplus (0, 0, 0, E, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)) \\
&= f((B, B, 0, \underline{\mathbf{9}}, 4, A, 1, 9, C, A, 5, 4, 3, 4, A, D, 8, 3, D, 5, E, 4, F)) \\
&= (5, 5, 0, \underline{\mathbf{2}}, D, C, 1, 2, A, C, B, D, E, D, C, 4) \\
\hat{x} &= x
\end{aligned}$$

The ECLT-based decoding method can be employed in conjunction with the inner-outer code architecture from Section 3.3.2. Moreover, an ECLT-based architecture is scalable, since it can be applied for any distance d. That is to say, with regards to the device constraints and the security level required for a given implementation, a designer can choose to use an RK code with a higher distance, which would provide a better detection rate, but also lower the number of potential miscorrections, at only a slightly higher hardware cost (the size of the ECLT remaining the same). Or, if the device is highly constrained, an ECLT-based architecture with a distance $d = 3$ can be implemented. The generation of one or the other can be automated, and only the ECLT itself needs to be precomputed prior to the implementation. Overall, this new decoding method, aimed in this work at the correction of single faults, is therefore optimised for constrained hardware implementations.

3.3.4 CPC based Outer Code

The detection rate of the previous RK architectures is greater if a larger distance is chosen for the implementation (see 3.4). However, choosing a base code with a larger distance also implies a higher implementation cost, as more operations are performed and slightly larger values need to be stored. The inner-outer code architectures from Section 3.3.2 can detect masked faults, as well as miscorrections, but the original second layer of code, even though efficient in this regard, was not scalable, since it used a fixed QS code. For constrained devices which would still require a higher detection rate, there is therefore a need for a scalable inner-outer code architecture. In this section, the original QS outer code is replaced by a Compact Protection Code (CPC), which are also robust codes, in order to offer a scalable inner-outer code architecture [GKKP20], with better detection rate than the original QS-based one. The overall inner-outer code architecture remains the same as Figure 3.6.

While CPC [RNK19] can be implemented in many different ways (see Equation 2.34), in this work, for simplicity of computation, as well as a lower hardware cost, Punctured Cubic (PC) codes [ALK12] are used as ground codes for the CPC. Let us denote by r_{CPC_i} the redundancy introduced by the outer CPC, where the index i refers to the number of symbols composing the redundancy in $GF(2^m)$. In order to compute r_{CPC_i}, the non punctured value of the code needs to first be calculated. For an input data $x = (x_1, .., x_k) \in GF(2^m)^k$, the value of the non-punctured code, denoted v_{npc}, is computed as follows.

$$v_{npc} = \sum_{i=1}^{\frac{k}{4}} (x_{4i-3}, x_{4i-2}, x_{4i-1}, x_{4i})^3 \tag{3.11}$$

The cubic function is computed in $GF((2^m)^4)$ (i.e. over 4 symbols), and hence the result is also a value $v_{npc} \in GF((2^m)^4)$, which can also be expressed in $GF(2^m)^4$ as $v_{npc} = (v_1, v_2, v_3, v_4) \mid v_i \in GF(2^m)\ \forall i \in \mathbb{N}$. r_{CPC_i} is then derived by puncturing v_{npc} into i symbols of $GF(2^m)$. That is to say, $r_{CPC_i} = (v_{1..i}) \in GF(2^m)^i$ (for instance $r_{CPC_2} = (v_1, v_2)$). For all the ciphers considered in this work, k is always divisible by 4, and therefore such a CPC can be computed. In the case were k would not be divisible 4, the construction from Equation 3.11 can be generalised to any number of symbols (i.e.

any i). Moreover, it is also possible to puncture the code in a number of bits which is not a multiple of m. This case is however not considered in this chapter, as a redundancy in terms of symbols was implemented for all the architectures from Section 3.4, both for consistency and easiness of computing the RK code. It should also be noted that, by definition, a CPC exists for any word length, and as such they can always be implemented as outer code, no matter the size of the input data.

Consequently, a CPC-based inner-outer code RK architecture can have up to i redundancy symbols, as opposed to only a single symbol for the QS-based inner-outer code. This results in a higher detection rate for faults which are masked for the RK code, or miscorrected, if more symbols are considered. In the case where only a single CPC symbol is introduced (i.e. r_{CPC_1}), the detection rate is similar to the QS-based architecture (Table 3.9). Experiments on the overall capability of a CPC-based inner-outer code architecture is available in Section 3.4.

The scalability of the CPC-based architecture allows more flexibility for a designer in terms of implementation. Dependent on the encryption scheme which is considered, and more especially the size of the internal states or operations to be protected, a designer may make the choice to chose to implement an RK code architecture with distance $d = 3$ in order to to be within the hardware constraints of the device. In this case, the detection rate would be lower than with a code of higher distance, however, by using a CPC-based outer code with more than one symbol, instead of a QS one, the system-level manager can detect a higher number of critical faults of class C1 and C4 (Figure 3.7), and thus only a very low number of faults of class S2 remain, approaching the detection rate of a code of higher distance. Nevertheless, introducing more CPC redundancy bits has a hardware cost too, and thus, it is advisable to implement an RK architecture with a higher distance to begin with, rather than introducing a large CPC as outer code. The outer CPC should be used as a more granular approach, when the security level offered by an RK architecture of lower distance is insufficient, but a higher distance is too costly to implement, In this case, using a CPC-based architecture can allow to improve the security level further than a QS outer code would, while remaining less costly to implement than a higher distance RK code. However, a QS outer remains cheaper to implement and should thus be used in the case where a single additional symbol is wanted.

3.4 Experimental Results of the RK Architectures

As stated in Section 3.2, security oriented codes need to be evaluated experimentally, but also at a theoretical level, in order to assess the worst case scenario that can occur. In the case of the different architectures presented in Section 3.3, since they are based on a specific RK code implementation, the mathematical discussion on the efficacy of the code against a strong attacker model (i.e. an attacker able to inject very precise and specific faults) has already been provided in the original RK code paper [RK17]. Moreover, in Section 3.3.2, the theoretical handling of miscorrections and masked faults (which could both be used by a skilled attacker) for the RK code, by an outer code, has been discussed as well. In this section, detection and correction rates of the implemented architectures against a physical fault injector are detailed, as well as hardware costs. In addition, a comparison to linear codes is provided.

The evaluation of the architecture was performed against a low cost clock-based fault injector. The fault injection was performed on a SAKURA-G Field Programmable Gate Array (FPGA) board. The board consist of two Xilinx Spartan-6 FPGAs. The smaller LX9 FPGA handled the communication between the host computer and the board, while the larger LX75 FPGA performed the encryption and the fault injection. The fault injection was realised by using a Digital CLock Manager (DCM) module present on the Spartan-6 FPGA family. The DCM was connected to the on-board clock, and used to generate a slightly faster clock, by adjusting the coefficients of the DCM. Finally, a clock multiplexer (BUFGMUX) was used to switch between the nominal clock and the newly generated clock at the desired fault location. The fault injection method being clock manipulation, and since the resolution at which the clock can be modified is dependent on the DCM coefficient used, it results in an imprecise fault injection setup. In more details, during the encryption process, when the switch between both clocks occurs (it only lasts for a single cycle) on the one hand, the next clock edge can arise rapidly. This results in a large number of circuit paths failing, and thus a high multiplicity fault. On the other hand, the difference between both the nominal and the generated clock edge can be minimal, and the operation can end only slightly before when it would have usually ended. In this case, only one long (critical) path will fail, resulting in a low multiplicity fault (e.g. a single

fault). Even if the injector is not extremely precise, the cost of implementing such a fault injector is low compared to other fault injection methods (see Section 2.1.2 and Table 2.1). Moreover, faults ranging from random single nibble to high multiplicity ones can be injected by the injector, hence encompassing both the faults which can be injected by a highly capable attacker (i.e. single faults) and any other possible faults, which is a perfect test case for the RK architectures.

The fault injection setup itself is described in Figure 3.10. The host computer sends a plaintext data to the FPGA, using an asynchronous First In First Out interface programmed on the LX9 FPGA. The FPGA then sends the complete data to the LX75 FPGA which proceed to the faulty encryption, and the data is sent back to the host computer for processing. The data comprise of the plaintext, the ciphertext (faulty in this case), as well as auxiliary data such as the fault free round input of the fault affected round and the faulty round output. An attacker would not have access to the latter, and this data is only generated in order to gather statistics on the architectures. The hardware costs (without the communication or statistic overhead) presented later in this section are based on an implementation on the same FPGA platform, and their proper functionality was also validated. In any case, the data is then processed by the software RK architecture, and both a fault flag, corresponding to the detection of the fault, as well as a corrected output are given as output of the testing setup. It should be noted that, during the fault injection phase, fault injection which were not successful are discarded by the host computer, and a new fault injection is performed. This way, only successfully faulted values are considered by the RK architecture, eliminating any unwanted processing.

Figure 3.10 also shows that, for the considered setup, a fault is injected during a complete round. This is due to both the imprecise nature of the injector, and also the implementation of the ciphers under test. In order to gather experimental results for the architectures, four SPN encryption schemes were considered. Three low cost 64-bit ciphers: LED [GPPR11], PRESENT [BKL$^+$07] and the SSAES [CMR05], as well as the 128-bit full scale AES [NIoSTN01]. Each of the ciphers is constituted of multiple rounds, and one round is processed in a single clock cycle. Therefore, a clock-based fault injector can inject a fault at any stage during the round. This means that, even if only a single critical path failed, resulting in a low multiplicity fault, but the fault occurs before a diffusion component,

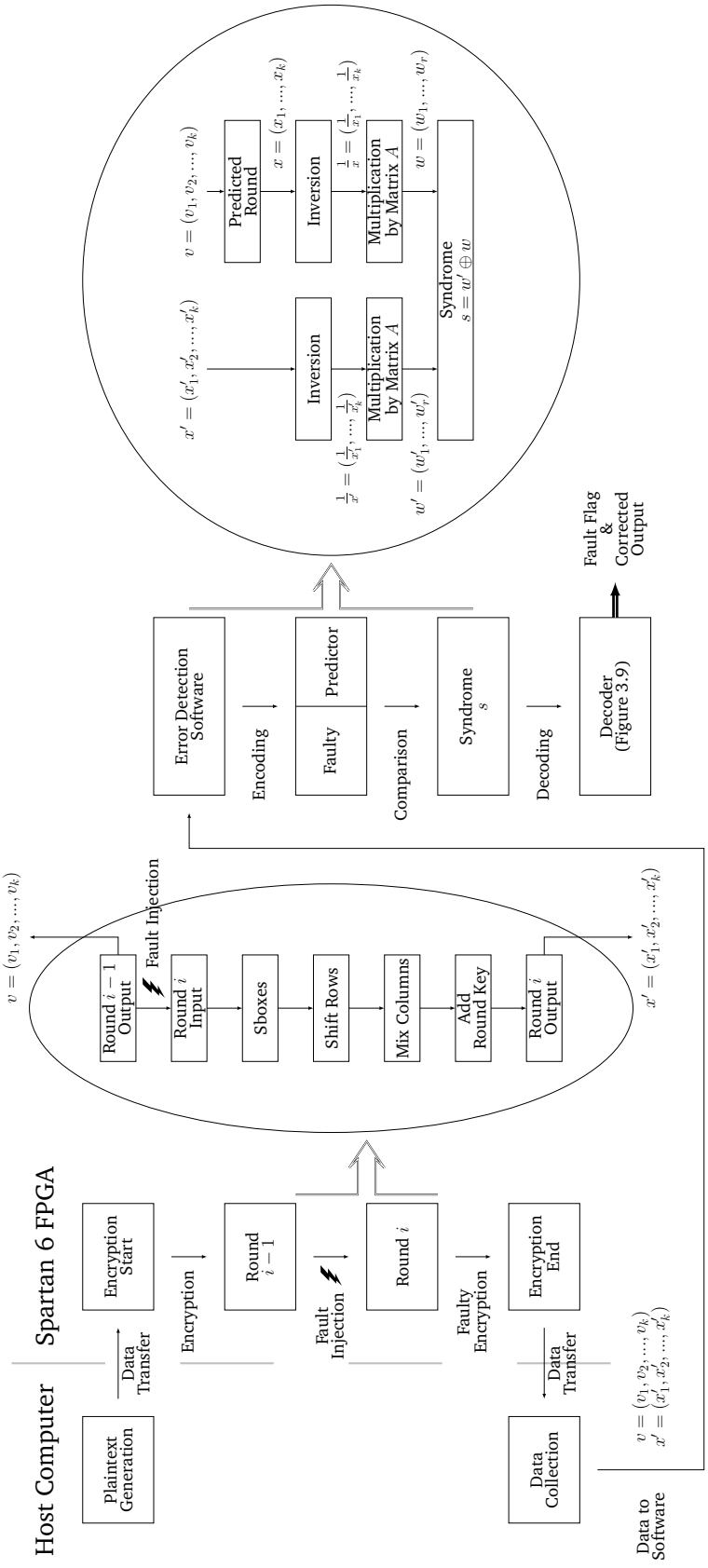

Figure 3.10: Fault Injection Setup for RK Architecture Evaluation

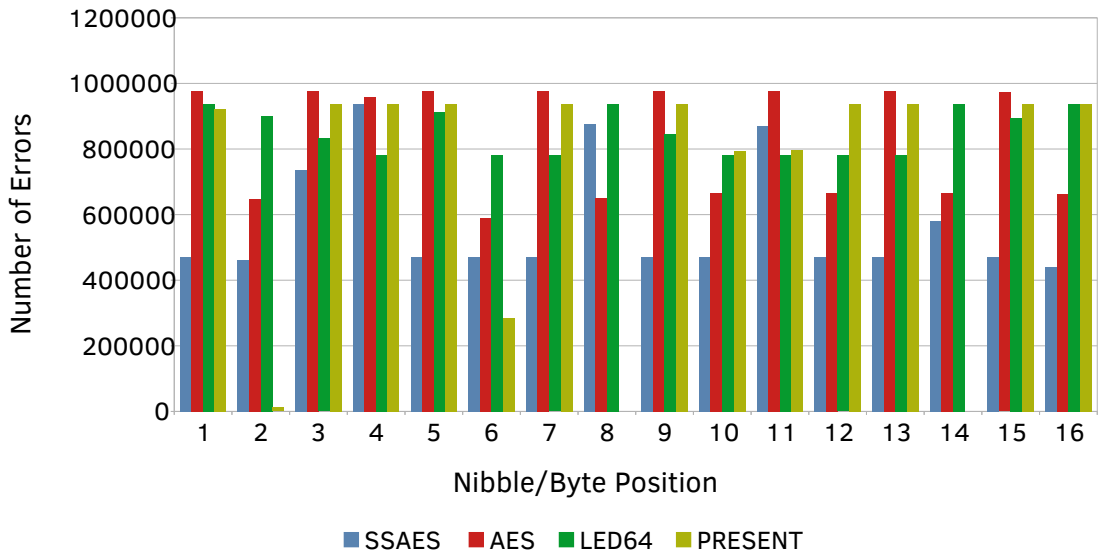

Figure 3.11: Fault Position for Each ECC Protected Cipher

then the multiplicity may increase at the output of the round. Consequently, the choice of protecting a complete round of the encryption schemes was made. This is similar to the choice a designer would make to protect its implementation, as he would need to consider the cipher, and its vulnerabilities to fault injection attacks, as well as the potential capabilities of an attacker.

For each cipher, one million successful fault injections were performed. Figure 3.11 shows the probability of a specific nibble (or respectively byte for the AES) to be faulty. From the figure, it is evident that some nibbles are more vulnerable than others. For instance, the eighth and fourteenth nibble of PRESENT are almost never faulty, while most nibbles of the AES have a high probability to be faulty. This is to be expected, as each implementation is different, and thus different critical paths are failing, leading to a wild variety of fault effects. Similarly, as expected by the nature of the fault injection method, the multiplicity of the injected fault is high, and most of the time exceeds the correction capabilities of the architectures (Figure 3.12). This is also dependent on the hardware implementation, and as such notable differences can be observed. For instance, for PRESENT, no fault of low, or high, multiplicities were injected. Overall, both figures showcase that, in the case of a low cost fault injector, faults can be considered arbitrary, and all faults need to be detected.

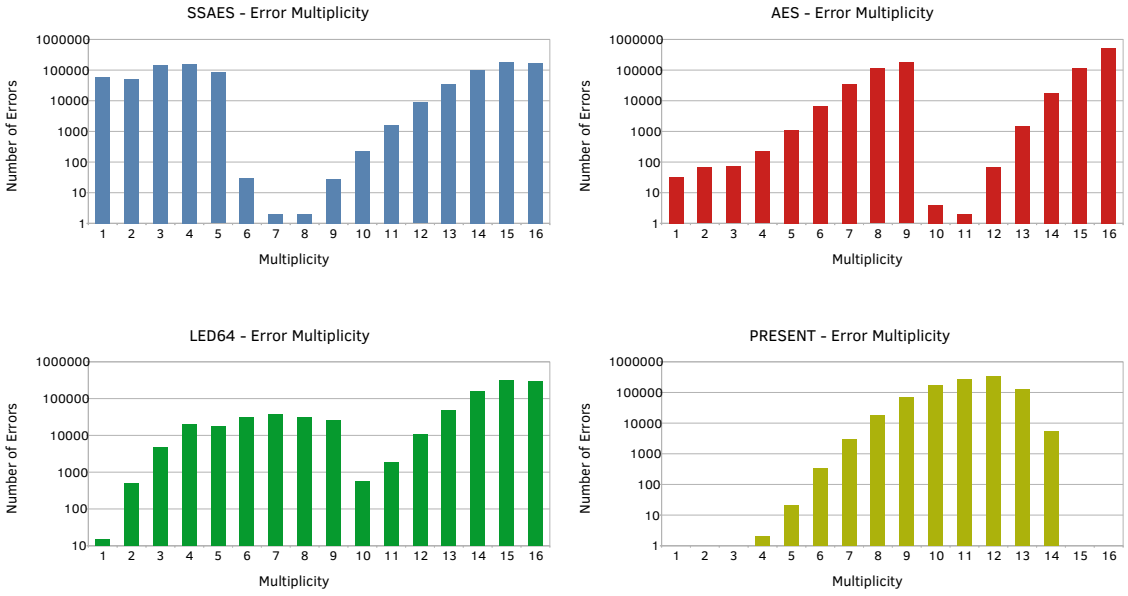

Figure 3.12: Fault Multiplicities for Each ECC Protected Cipher

In addition, Figure 3.13 confirms that faults can be modelled as additive errors, which is the fault model considered in this work and throughout this chapter. In other words, the fault model assumes that the probability of a bit to flip from 0 to 1 is the same as the probability to flip from 1 to 0, which is the case for the presented experiments.

Table 3.8: Comparison Between a Robust RK Architecture and a Linear BCH Code

Circuit	Distance d	Bits		Undetected faults	
		k	r	BCH	RK
SSAES	3	68	12	176	179
	5	68	28	0	0
AES	3	136	16	234	239
	5	136	56	0	0
LED	3	68	12	239	234
	5	68	28	0	0
PRESENT	3	68	12	231	229
	5	68	28	0	0

The injection of one million faults was first used to verify practically (since no RK code hardware implementation were designed prior to this work) that the detection rate of the RK architecture would not worsen by the introduction of the APN function. To this

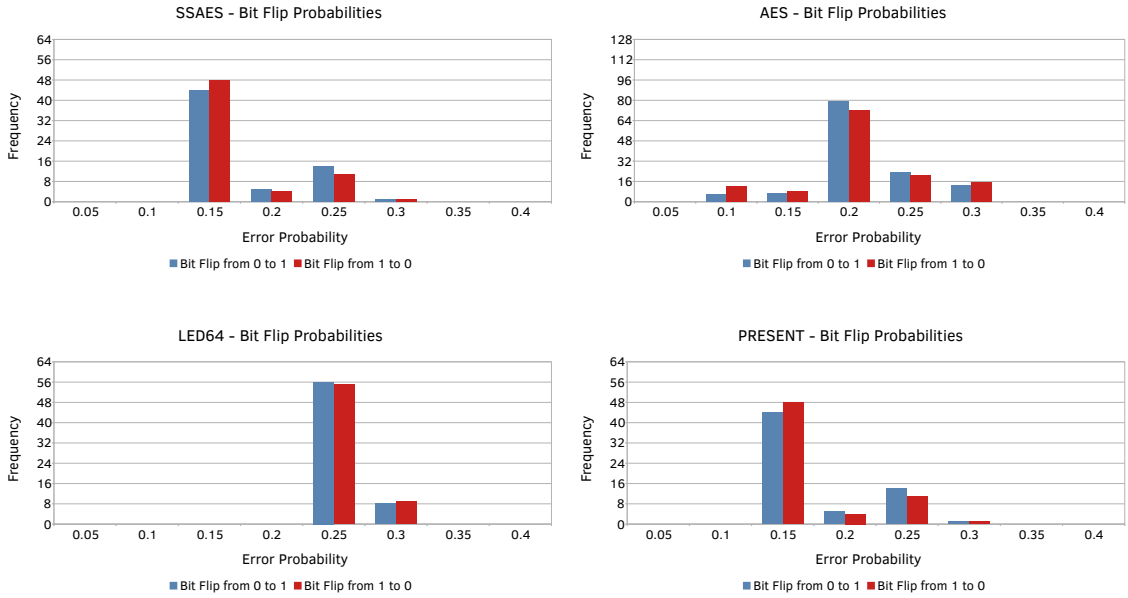

Figure 3.13: Bit Flip Characterisation for Each ECC Protected Cipher

end, both a linear, non-robust, BCH code implementation, and the robust RK architecture presented in Section 3.3.1 were compared. The results are presented in Table 3.8, and it can be seen that both codes have only a very low number of undetected faults for a distance $d = 3$, while no faults remained undetected for larger distances. It is consequently clear that from a detection rate perspective, both codes are equivalent, but the RK code has the advantage to be robust, and thus are better suited for security applications, since no fault always goes undetected.

Table 3.9 shows the results for a complete RK-based architecture. That is to say, an inner-outer code architecture with an ECLT decoder. The chosen code for the outer layer is either a CPC with a single redundancy symbol, or a QS code. In the case of the AES, both a single decoder over $GF(2^8)$ (i.e. $q = 8$ for this case only), and dual decoder architectures over $GF(2^4)$ were implemented. The number of symbols (and respectively bits) correspond to the input of the RK inner code. Since a single symbol outer CPC is used (or respectively a QS code, which also adds a single symbol), this means that the input of the RK code is constituted of one additional symbol (or two in the case of the dual decoder architectures). For the non-linear checker, which correspond to the RK inner layer, the number of faults in each class are presented in terms of percentages over one million fault injections. The

Table 3.9: Experimental Results on Fault Detection & correction

Architecture			**Symbols**		**Bits**		**Fault Events**							
							Non-linear Checker				**CPC System-level Fault Manager**		**QS Code System-level Fault Manager**	
Circuit	d	#Dec	k_q	r_q	k	r	**Class C1** Undetected (by RK code)	**Class C2** Single Faults (corrected by RK code)	**Class C3** Recognized as Suspicious (by RK code)	**Class C4** Erroneous Corrections (by RK code)	**Class S1** Recognized as Erroneous (by CPC Outer Code)	**Class S2** Unrecognised as Erroneous (by CPC Outer Code)	**Class S1** Recognized as Erroneous (by QS Outer Code)	**Class S2** Unrecognised as Erroneous (by QS Outer Code)
SSAES	3	1	17	3	68	12	0.0179	5.9337	87.4101	6.6383	62463 (93.8%)	4099 (6.2%)	62499 (93.9%)	4063 (6.1%)
	5	1	17	7	68	28	0	5.9337	94.0662	0.0001	1 (100%)	0 (0%)	1 (100%)	0 (0%)
AES	3	1	17	2	136	16	0.0239	0.0033	86.7272	13.2456	124347 (93.7%)	8348 (6.3%)	124286 (93.7%)	8409 (6.3%)
	3	2	34	6	136	24	0.0067	0.0033	88.1006	11.8894	117540 (98.8%)	1421 (1.2%)	117475 (98.8%)	1486 (1.2%)
	5	2	34	14	136	56	0	0.0033	99.9956	0.0011	11 (100%)	0 (0%)	11 (100%)	0 (0%)
LED	3	1	17	3	68	12	0.0234	0.0015	92.6682	7.3069	68923 (94.0%)	4380 (6.0%)	68734 (93.8%)	4569 (6.2%)
	5	1	17	7	68	28	0	0.0015	99.9985	0	0	0	0	0
PRESENT	3	1	17	3	68	12	0.0229	0	92.7018	7.2753	68427 (93.8%)	4555 (6.2%)	68376 (93.7%)	4606 (6.3%)
	5	1	17	7	68	28	0	0	100	0	0	0	0	0

data for the system-level manager (i.e. the outer code) is given in terms of raw number, and also of percentages over the critical cases of the inner layer, the sum of faults of classes C1 and C4 (see Figure 3.7).

First, if only faults of class C1 are considered (corresponding column of Table 3.9), it can be seen that only a very low number of faults are undetected by the RK code, less than 0.1%. This equates to saying that the detection rate of the RK code alone is over 99.9%. If the architectures were implemented for detection only, for example for extremely constrained devices, where implementing a decoder is impossible area-wise, this is an excellent detection rate. However, in such a case, the correction capabilities of the code are omitted, and thus the architecture is not used at its full potential. It is nevertheless a possible consideration for a designer.

Column C2 of Table 3.9 corresponds to all the single errors which occurred and were successfully corrected by RK code. Such faults are always successfully corrected, as they fall within the correction capability of the ECLT-based architecture (i.e. single faults). Column C3 refers to the faults which were uncorrectable for the RK code, and were also successfully recognised as uncorrectable. That is to say, faults for which the corresponding normalised syndrome did not have any match in the ECLT. Finally for the non-linear checker, Column C4 shows the number of miscorrections which were carried by the inner code. Overall, all single faults were successfully corrected, eliminating the threat of precise single nibble (or byte) fault attacks (for example, the ones presented in Section 4.1), while only a small portion of faults were miscorrected, especially in the case of architecture based on codes of distance $d = 5$. In this latter case, not only no faults were undectected by the RK code, but almost no miscorrection occurred (at most 11 in the case of the AES). This supports the fact that employing a larger distance, even with an ECLT limited to single faults, is beneficial for the architecture, and should thus be preferred, if the hardware constraints can be met.

Columns S1 and S2 of both system-level fault managers, in Table 3.9, refer to the detection rate of the outer code. It can be seen that most faults of class C1 and C4 are detected by either outer code constituted of a single symbol (only 6% remain undetected for $d = 3$). While this can be improved by increasing the number of symbols of the CPC (see Table 3.10), the results for a single additional symbol are already good, and overall, for the

considered architectures, at most 0.8% of faults remain critical, in the case of architecture $S_8d_3m_8$ (single decoder) for the AES. Moreover, the table shows that both the single symbol CPC and the QS code have the same performances. Therefore, in the case where no extra security is required for the outer code (as discussed in Section 3.3.4), the QS code should be preferred, as it is less costly to implement.

With regard to architectures $S_8d_3m_8$ and $S_8d_3m_4$ implemented for the AES, Table 3.9 shows that the dual decoder architecture is superior to the single decoder over a larger field. For this reason, only the dual decoder architecture was extended to distance $d = 5$. Only two redundancy symbols are necessary for distance $d = 3$ over $GF(2^8)$, therefore the single decoder has slightly less redundancy bits, but this is solely due to the BCH codes used, and the number of redundancy bits for $d = 5$ would be the same.

Table 3.10: Probability of Class S2 Faults with Different outer CPCs

Circuit	d	#Dec	k_q	r_q	k	r	$r_{CPC_1} = 4$	$r_{CPC_2} = 8$	$r_{CPC_3} = 12$	$r_{CPC_4} = 16$
SSAES	3	1	17	3	68	12	0.4099%	0.0246%	0.0017%	0.0001%
	5	1	17	7	68	28	0%	0%	0%	0%
AES	3	1	17	2	136	16	0.8348%	0.0539%	0.0035%	0.0002%
	3	2	34	6	136	24	0.1421%	0.0085%	0.0004%	0.0001%
	5	2	34	14	136	56	0%	0%	0%	0%
LED	3	1	17	3	68	12	0.4380%	0.0288%	0.0024%	0.0001%
	5	1	17	7	68	28	0%	0%	0%	0%
PRESENT	3	1	17	3	68	12	0.4555%	0.0290%	0.0021%	0.0001%
	5	1	17	7	68	28	0%	0%	0%	0%

Table 3.10 show the evolution of the misdetection rate for an increasing size of outer CPCs. Four different CPCs were implemented, each adding an additional redundancy symbol for miscorrection verification, compared to the previous one. The redundancy in terms of bits is denoted by r_{CPC_i} for each CPC, where i corresponds to the number of equivalent symbols. Moreover, Figure 3.14 shows that the misdetection rate decreases exponentially with the number of redundancy bits used. Overall, a single symbol leads to at most 0.8% of faults which remain undetected, while adding more symbols results to at most two masked faults per million (both for the AES and architecture $S_8d_3m_8$). However, the table also shows that a distance $d = 5$ results in no misdetection. This reinforces the fact that a larger distance should be prioritised, when the hardware constraints allow it. Nevertheless, the use of CPC can further bring down the misdetection rate in the cases where this is not possible, and in a scalable way.

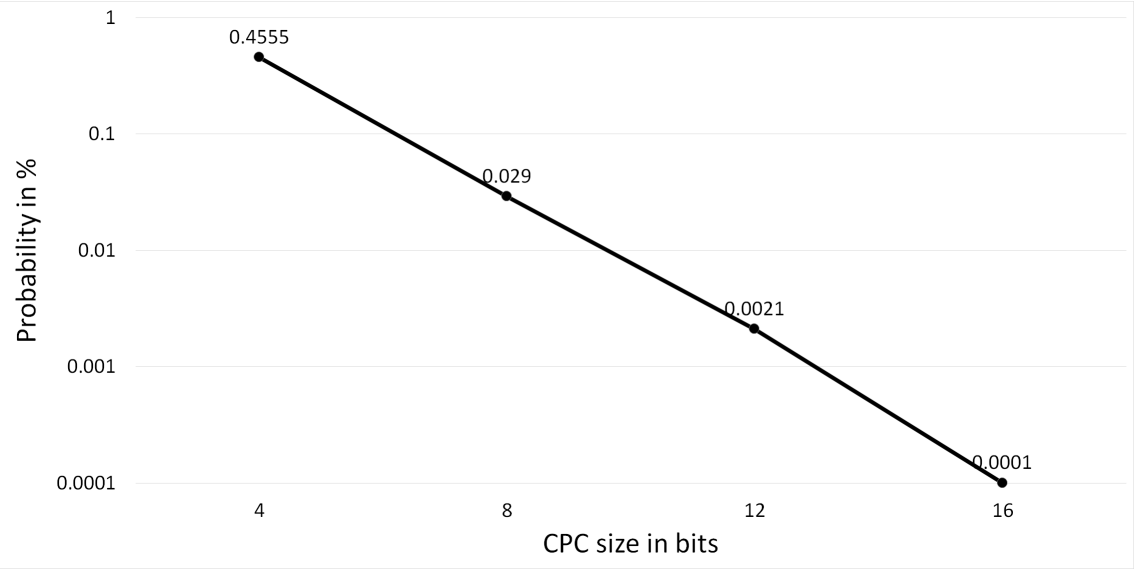

Figure 3.14: Evolution of the Probability of Class S2 Faults for Different CPCs on PRESENT

Table 3.11: Size Comparison in terms of Configurable Logic Blocks (CLBs)

Cipher	Unprotected Round	RK ($d = 3$)	BCH ($d = 3$)	RK ($d = 5$)	BCH ($d = 5$)	TMR
SSAES	37	202	169	328	267	108
AES (1 decoder)	173	388	392	–	–	421
AES (2 decoders)	173	465	419	572	462	421
LED	39	221	213	257	248	133
PRESENT	23	165	148	240	243	57

The different architectures have excellent detection and correction properties, and the use of ECLT-based decoders allow for an efficient hardware implementation. This is shown in Table 3.11, which compares the previous RK-based architectures with diverse alternatives in terms of Configurable Logic Blocks (CLBs). The number of CLBs needed for each implementation is derived from the synthesised design implemented on the Spartan-6 LX75 FPGA used in the setup described in Figure 3.10. First, it is clear from the table that the hardware cost of an ECC counter-measure is high, especially for lightweight block ciphers, such as PRESENT. A single round of PRESENT protected by an RK architecture with distance $d = 3$ is more than seven times larger than the same unprotected round. However, this is also the case if a non-robust BCH code architectures is implemented (in both cases, the decoder was implemented using an ECLT, for fairness of comparison). The table actually shows that the robustness property comes at a very low extra cost in area. The highest

increase in cost between a non-robust BCH ECC and an RK architecture is of 24%, for the AES with two decoder and a distance $d = 5$. In the case of the single decoder AES with distance $d = 3$, the RK architecture is even slightly smaller than the BCH one. Even though this is due to synthesis optimisation from Xilinx ISE Design Suite, the overall increase in cost is negligible compared to the advantages, in term of security (Section 2.2.1.2), of using a robust code.

Table 3.11 also compares the architectures to equivalently implemented TMR architectures. However, TMR are not only non-robust, but also extremely vulnerable to simultaneous fault injections. Nevertheless, it can be seen that, for larger circuits (e.g. an AES implementation), TMR architectures are of a comparable size to the proposed RK architectures with a lower distance. This reinforces the fact that the architectures presented in this chapter are not only efficient at detecting and correcting faults, but also implementable in practice, thanks to the ECLT. Moreover, the architecture are based on q-ary codes, which also allow for more practically implementable ECC counter-measure, in comparison to commonly used binary codes. For instance, in the case of the two decoder architecture with distance $d = 5$, used for the AES, 56 redundancy bits are required. In comparison, a binary BCH-code decoder would require 112 bits in order to achieve the same detection and correction capability. Even though operations over $\mathbb{F}_q$ are costlier than binary operations, the sheer number of additional bits also has a cost. In addition, the ECLT-based decoding method cannot be implemented for binary codes, and as such a more complex decoding algorithm, such as the Berlekamp-Massey algorithm, has to be used. This is not only costlier, but it cannot be implemented in a single cycle, thus increasing the latency of the design, compared to the proposed RK architectures.

To conclude this chapter, the experimental results encourage the use of the RK architectures developed in this work. All architectures are efficient at both detecting faults, and correcting critical single fault injections, while being practically implementable thanks to the use of the ECLT. Moreover, the architectures can be scaled to any cipher, and any required level of security. The generation of the architectures can easily be automated, since they are based on well known BCH codes, and only the ECLT and the generator matrix need to be pre-computed prior to the implementation. Therefore, they are a strong choice for ECC-based counter-measure against fault injection attacks, especially compared

to other security oriented codes, such as the AMD codes [CDF+08], which require the additional implementation of a random number generator.

Chapter 4

AutoFault: Hardware-Oriented Algebraic Fault Attack Framework

The focus of this Chapter is on the automation of fault attacks on cryptographic hardware, and more especially AFAs. For this purpose, the `AutoFault` framework was created. However, in order to automate attacks on different ciphers, it is important to first understand the subtle differences which exist between different cryptographic primitives and their implementations, and how they impact the generation of fault attacks. In this regard, the first consideration will be on the Small Scale AES (SSAES). The fault attacks on the SSAES presented in Section 4.1 can be seen as multiple special cases which do not occur for the baseline AES, while remaining closely related. Such fine disparities for a single cipher family showcase the need for an automated method for fault attack generation, as well as giving many points of reflection, which needs to be considered in `AutoFault`. While this is the first consideration step for the framework, the following sections focus on multiple aspects of the `AutoFault` framework itself. Details are firstly given on the overall structure of the framework, followed by explanations on the solving steps and the CNF-based simulation of attacks. In Section 4.5, use cases for the usage of `AutoFault` are discussed. Experimental results and a comparison to other state-of-the-art AFA framework are presented in the last Sections, to further validate and reinforce the need for the hardware-oriented AFA framework `AutoFault`.

4.1 Preliminary: Fault Attack on Small Scale AES

The SSAES [CMR05] was designed as a research tool in order to analyse AES equation system. To this end it shares the same components as the AES, but in a scaled down and fully parametrised manner. That is to say, the same operation are performed in the same order: *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*.
The SSAES is defined as either $SR(n, r, c, e)$ or $SR^*(n, r, c, e)$. The latter variant is the same as the former in terms of state size and possible number of rounds, but, similarly to the AES, the *MixColumns* operation is omitted in the last round. The different parameters, (n, r, c, e), are defined as follows:

- n denotes the number of rounds, which can range from 1 to 10
- r and c respectively refer to the number of rows and columns composing the state matrix
- e is the word size of the SSAES, either 4 or 8. It corresponds to the Galois field in which the computation is done, respectively $GF(2^4)$ and $GF(2^8)$, as well as the size of the SBoxes used, either 4 or 8-bit SBoxes

For example, the full scale AES is equivalent to the SSAES $SR^*(10, 4, 4, 8)$, the only slight difference being the definition of the key schedule.
The scalable nature of the SSAES makes it a perfect first candidate for new hardware-oriented solving methods. However, while the algorithm for the SSAES was introduced prior to this work, no hardware implementation had been provided. Therefore, a VHDL implementation was introduced [GBH+16] as a first step towards hardware-oriented AFAs. Since the SSAES operations are dependent on the parameters r, c and e, a specific implementation for each variant was designed. Each implementation differs on the following operations:

- The SBoxes are different depending on the word size e, and are implemented as simple look-up tables.
- The key schedule is computed with regards to the number of rows and columns of the state matrix. Mainly, the XOR operations are performed column-wise. Additionally, an SBox is used and the key schedule is therefore dependent on e as well.

- The *MixColumns* operation consists of a pre-multiplication of a column by a fixed matrix (e.g. Equation 4.1 in the case where $c = 2$), and as such relates to the number of rows r, as well as the word size e for the Galois field multiplications. As such, multiple variants share the same *MixColumns* module (same (r, e), but different c). The matrix is only composed of ones, twos and threes, no dedicated Galois field multiplier is implemented, but rather the operations are directly done in the *MixColumns* module.

$$M = \begin{pmatrix} 3 & 2 \\ 2 & 3 \end{pmatrix} \tag{4.1}$$

- The rounds are themselves dependent on every parameters, and therefore every variants disposes of its own round module. The *ShiftRows* operations is directly done here, as it is only a nibble-wise shift. Similarly, the *AddRoundKey* operations is only an XOR and is performed directly in the round module as well.

Each of the previous operations is implemented as a distinct VHDL module, as can be seen in Figure 4.1. While they are all related to the encryption process, the overall implementation also includes a decryption module, and therefore the corresponding inverse modules are provided as well. Finally, the presence of a *MixColumns* operation in the last round ($SR(n, r, c, e)$ or $SR^*(n, r, c, e)$) is simply taken as a further input parameter in each of the previous implementations and therefore a single SSAES implementation is given for both variants.

It is important to note that the proposed implementation does not consider any countermeasures to any kind of physical attacks (being side-channel or fault injections). The

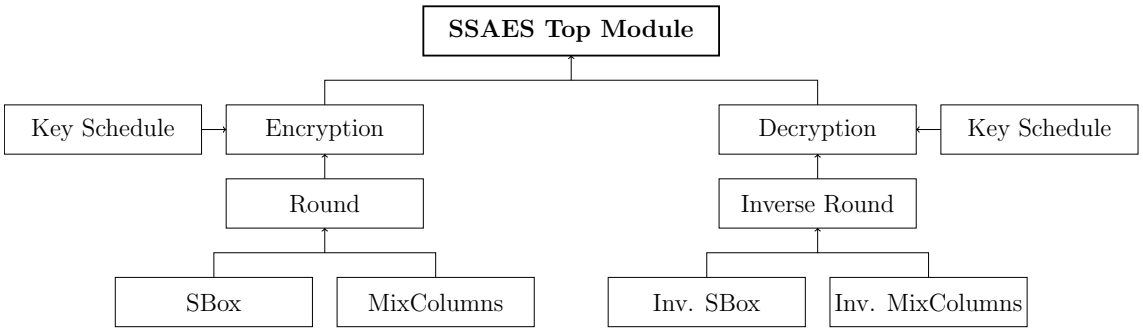

Figure 4.1: Modules of the SSAES VHDL Implementation

implementation is directed towards an academic usage, either in a research context or for education purposes, and as such is not protected. For a similar purpose, a gate-level implementation based on the 45nm NanGate cell library [Sil] is provided as an easy way to implement the SSAES on any hardware.

Before considering AFAs, more conventional fault injections attack on the SSAES are to be considered, especially to gain some insights on the solving process. The fault attack introduced in [TMA11] (and detailed in Section 2.1.4.2) can be generalised to the whole family of SSAES. However, the fault propagation, and thus the fault equations, is dependent on the parameters of each SSAES. For instance, a smaller number of rows r than columns c reduces the propagation rate.

Fault Injection Attack on SSAES

The proposed attack is a DFA with the following variables (similarly to Section 2.1.4.2). In the subsequent equations, x stands for the output of the fault free encryption, the correct ciphertext, and x' for the faulty one. The respective nibbles (either 4-bit nibbles or bytes, depending on the value of e) are denoted as x_i and x'_i. Similarly, the key parts of the last round key k are denoted k_i. $S()$ represent the SBox of the considered SSAES variant, and $S^{-1}()$ the inverse of that SBox. δ_i will be the XOR difference of two words at the chosen meeting point for the DFA.

The parametric nature of the SSAES implies a variable number of rounds, as well as a variety of different state matrices. The attack can be achieved for any SSAES variant with $n \geq 3$ rounds, as enough fault propagation is needed to recover the secret key. The state matrix size depends on the values of (r, c, e). However, the word size only affects the solving part, and not the equation derivation. As such, all equations are valid for any value of e. The number of rows and columns does affect the fault propagation patterns, and therefore in the next sections, all possibilities for the tuple (r, c) are considered. Additionally, a visualisation of the fault propagation is given for the cases $(r, c) = (2, 4)$ and $(r, c) = (4, 2)$ only, as they are the most relevant examples of a different fault propagation pattern, and thus the solution needs to still be able to recover the secret key. Similarly, the presence or the omission of the last *MixColumns* operation has an effect on the fault equations. The focus of this work is on the $SR^*(n, r, c, e)$ variant of the SSAES (without a *MixColumn*

operation in the last round), but a discussion on the $SR(n, r, c, e)$ variant, as well as the fault equations, are provided.

Case without last *MixColumns*

Let's first consider the SSAES variants $SR^*(n, r, c, e)$, which omits the last *MixColumns* operations, similarly to the AES. The chosen meeting point for the DFA is right after the last *MixColumns* operation (second to last round). All the equations can be found in Appendix A, but few are given here as well to better showcase the differences.

1. Case $(r, c) = (1, 1)$:
   In this case, the state matrix is only composed of a single element, rather than multiple rows or columns. Therefore, there is of course no fault propagation, and the processed data is only a single word. As such, the last round can simply be inverted in order to get the following equation, from a single fault injection in round 9 (either before or after the SBox):

$$\delta_1 = S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) \tag{4.2}$$

2. Case $(r, c) = (1, 2)$ and $(r, c) = (1, 4)$:
   Similarly to the previous case, there is only a single row in the state matrix, and even though there are multiple elements, no *ShiftRows* or *MixColumns* operations are performed. Despite this, and in order to recover the complete key, a single fault injection is not sufficient, since there is no fault propagation, and therefore a fault injection would be needed for each nibble. This would result in two (or respectively four) similar equations as in equation 4.2, with different indices. Due to the small nature of those variants, it would however be possible to brute force the remainder of the secret key, rather than performing more fault injections.

3. Case $(r, c) = (2, 1)$:
   The state matrix is a single column composed of two elements. Therefore, if the fault is injected at the beginning of round 9, in the first element, there will be enough fault propagation to cover the full state matrix, without any overlap.

4. Case $(r, c) = (2, 2)$:

   $r = c$, meaning that the state matrix is square. Consequently, thanks to the *ShiftRows* and *MixColumns* operations, a single fault injection at the beginning of round 8, in the first element, will propagate through both rows and columns of the state matrix. This is a reduced case of the AES fault attack.

5. Case $(r, c) = (2, 4)$:

   In this case, $r < c$. If a fault is injected in the first element of the state matrix, at the beginning of round 8, then it will only propagate through half of the state matrix by the end of the encryption (see Figure 4.2, grey fault only). While it would be possible to brute force the second half of the key, a second fault injection in the fifth element of the state matrix (either simultaneously or two distinct fault injections) would result in a full propagation, and is as such a possibility to recover the complete secret key (Figure 4.2, orange fault). A set of equations for the first fault injection is as follows:

$$\begin{aligned} 3\delta_1 &= S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) \quad & 2\delta_2 &= S^{-1}(x_7 \oplus k_7) \oplus S^{-1}(x'_7 \oplus k_7) \\ 2\delta_1 &= S^{-1}(x_8 \oplus k_8) \oplus S^{-1}(x'_8 \oplus k_8) \quad & 3\delta_2 &= S^{-1}(x_6 \oplus k_6) \oplus S^{-1}(x'_6 \oplus k_6) \end{aligned} \tag{4.3}$$

6. Case $(r, c) = (4, 1)$:

   Similarly to the case $(r, c) = (2, 1)$, a single fault injection at the beginning of round 9 is sufficient to recover the full key.

7. Case $(r, c) = (4, 2)$:

   Since $r > c$, if a fault is injected at the beginning of round 8, it will propagate through the complete state matrix. However, the fault-affected values will overlap each other due to the *ShiftRows* operation, which will lead to more complex equations (Figure 4.3). To avoid over-propagation, a fault can be injected one round later, at the beginning of round 9. In this case, the fault will only propagate through half the state matrix (similarly to the case $(r, c) = (2, 4)$), and a second fault should be injected in the fifth element of the state matrix to achieve full propagation (Figure

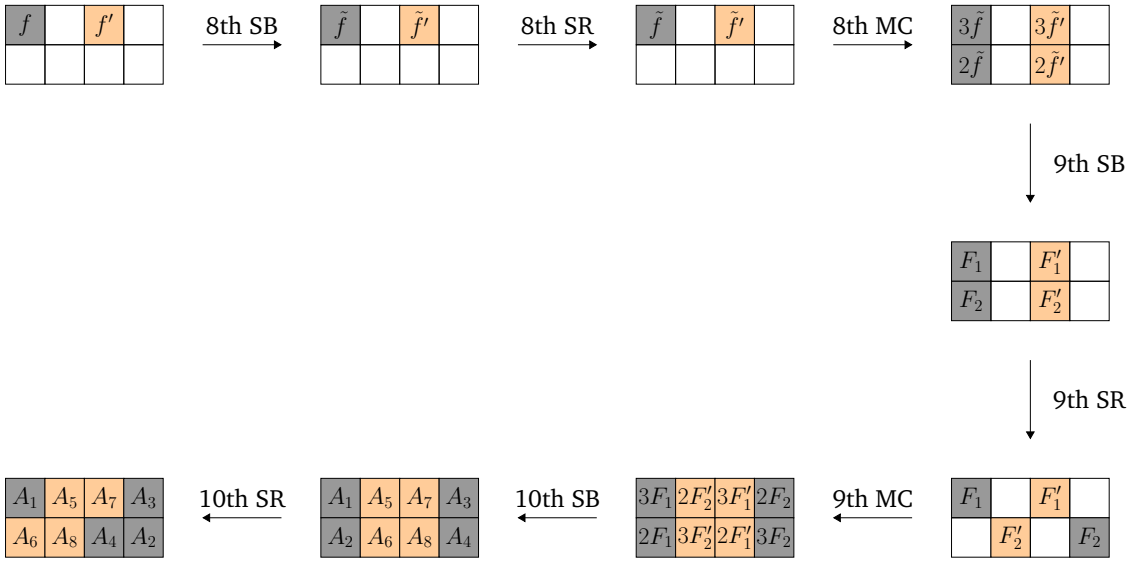

Figure 4.2: Dual fault injection at round 8 in the $SR^*(10, 2, 4, e)$

4.4). Below is a set of equations corresponding to the first fault injection.

$$\begin{aligned} 2\delta_1 &= S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) \\ \delta_1 &= S^{-1}(x_6 \oplus k_6) \oplus S^{-1}(x'_6 \oplus k_6) \\ \delta_1 &= S^{-1}(x_3 \oplus k_3) \oplus S^{-1}(x'_3 \oplus k_3) \\ 3\delta_1 &= S^{-1}(x_8 \oplus k_8) \oplus S^{-1}(x'_8 \oplus k_8) \end{aligned} \tag{4.4}$$

8. Case $(r, c) = (4, 4)$:

   This case is simply the same case as the DFA on the AES

Thanks to the new locations for the faults, or the additional faults. The solving process for each $SR^*(n, r, c, e)$ variant is the same as the AES (Section 2.1.4.2). The same discarding process for the δ_i values can be applied, hence reducing significantly the complexity of the attack. The main difference, in the case of the SSAES, comes from the number of equations. While there may be less equations available, leading to a smaller reduction of the key space, the key space itself is also smaller for every variant which differs from $SR^*(10, 4, 4, 8)$, thus every attack is still mountable.

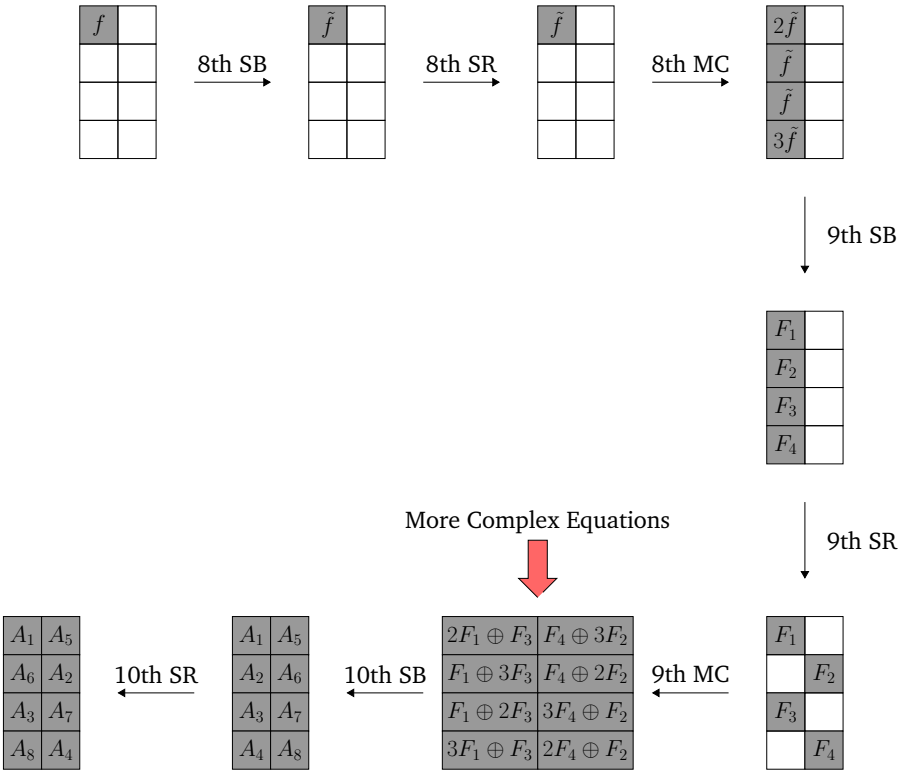

Figure 4.3: Single fault injection at round 8 in the $SR^*(10, 4, 2, e)$

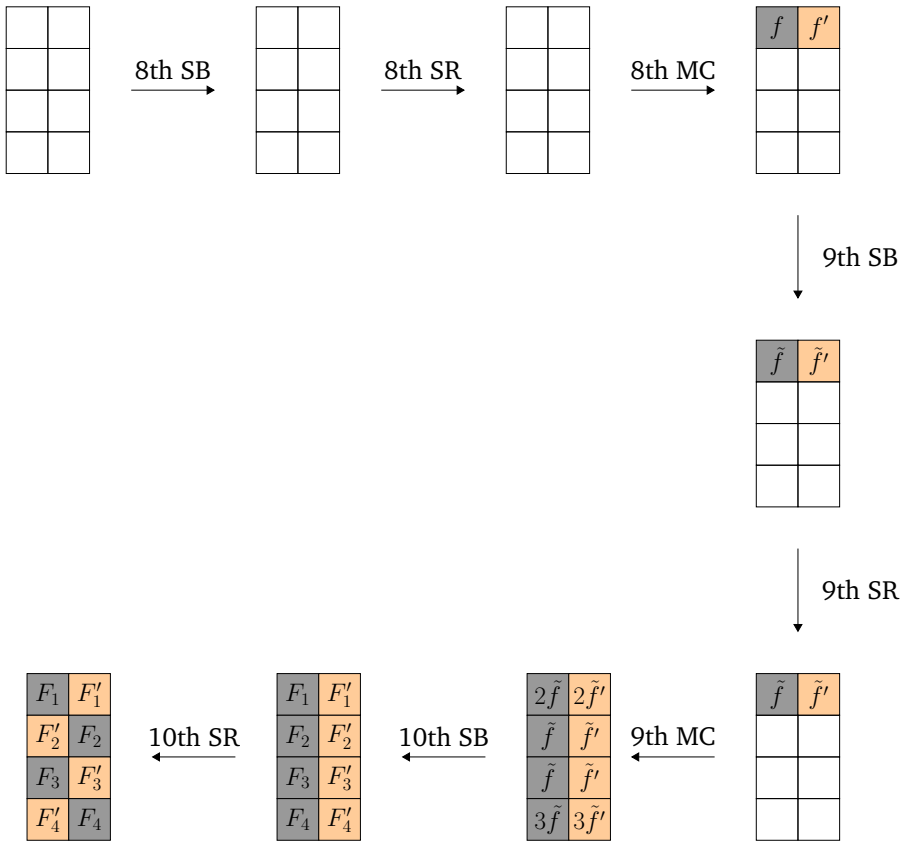

Figure 4.4: Dual fault injection at round 9 in the $SR^*(10, 4, 2, e)$

Similarly to the case of the AES, it is also possible to proceed with a second reduction step for the attack. However, the key schedule of the SSAES is slightly different from the AES, even in the full scale case. The equations for the reverse key schedule can be found in Appendix A as well, and can be applied similarly to Section 2.1.4.2.

Case with last *MixColumns*

For the SSAES variants $SR(n, r, c, e)$, the *MixColumns* operation is not omitted in the last round. The presence of the last round *MixColumns* operation propagates the fault more than needed, if the fault injection locations are the same as in the previous section. One intuition would be to use this, and inject the fault a round later in order to achieve full propagation throughout the state matrix. However, the meeting point is then shifted to right after the last *MixColumns* in the last round. No information can be gained from this meeting point since it depends directly on x_i, x'_i and k_i. More precisely, a possible equation would be:

$$2\delta_1 = (x_1 \oplus k_1) \oplus (x'_1 \oplus k_1) = x_1 \oplus x'_1 \tag{4.5}$$

It is clear that the δ_i are independent from the value of k_i, and therefore this would not lead to any key recovery. Consequently, and despite this fact, the faults should still be injected at the previously discussed locations, even if this results in more complex equations, and the meeting point is situated after the second to last *MixColumns* operation. The derived equations are dependent on a full column, and thus several key parts k_i, instead of only a single one for the $SR^*(n, r, c, e)$ variants. Equation 4.6 showcases a single equation from one of the four equation sets of an $SR(10, 4, 4, 8)$ variant (see Appendix A for the other equations).

$$2\delta_1 = \oplus \begin{array}{l} S^{-1}(14(x_1 \oplus k_1) \oplus 11(x_2 \oplus k_2) \oplus 13(x_3 \oplus k_3) \oplus 9(x_4 \oplus k_4)) \\ S^{-1}(14(x'_1 \oplus k_1) \oplus 11(x'_2 \oplus k_2) \oplus 13(x'_3 \oplus k_3) \oplus 9(x'_4 \oplus k_4)) \end{array} \tag{4.6}$$

While the equations are more complex to solve, several equations, from different equation sets, also depends on the same key part tuple (see Appendix A). This can be used to

further reduce the number of key candidates. In this case, only the candidates from the first set need to be tested with the second set, and so on until the last set (if more sets are available).

Further key space reduction

In a similar fashion to the second attack step for the AES, it is possible to further reduce the key space by expressing the key of the second to last round in terms of the last round key, via a reverse key schedule. This step is only required if the remaining key space is too large, and, as such, the equations for the cases where $r = 1$ are omitted from Appendix A. Furthermore, and since this step is the same as in the AES attack, only with a different reverse key schedule, it won't be discussed further here.

A first case towards automatic hardware AFA solving

For a conventional DFA, the previous equations need to be derived for every SSAES case, and in a more general way, for every different encryption scheme. The generation of the equations requires a cryptanalyst to carefully work on them for an extended amount of time. While this is true for DFAs, it is also true for AFAs, as conventional AFAs make use of handcrafted fault equations. However, AFAs also use the cipher description as an input. As such, and to avoid deriving the description related equations for every SSAES case, the hardware implementation can be used to directly generate the related equations in the form of Conjunctive Normal Form (CNF) clauses. To do so, the Tseitin transform is applied to a synthesised gate-level unrolled implementation of every SSAES, resulting in a set of clauses for every SSAES variant. However, even though such clauses can theoretically be solved by a Boolean Satisfiability (SAT) solver, it is not sufficient for a key recovery. Given a plaintext and a ciphertext, it is usually computationally impossible to derive the secret key solely from the equations derived from the hardware implementation, and that is to be expected for any cryptographic element. Therefore a fault component of the attack is needed. To this end, is possible to introduce a new XOR gate in the circuit, which takes the fault as an input, as well as the corresponding intermediate value to the chosen fault location, and duplicate the fault-affected parts of the circuit, in order to get more CNF clauses for a specific SSAES variant. The newly derived clauses are related to the

fault, and as such constitute the fault equation part of the AFA. The whole process can be automated for any SSAES variant, as well as any encryption scheme in general, and is the foundation of the `AutoFault` framework.

In the case of the SSAES, and as seen previously, each variant has a different fault propagation pattern. While this is especially important for manually crafted fault equations, it also translates to equations derived automatically from the hardware implementation. It is similarly possible to distinguish between three cases: a perfect (or close to perfect) fault propagation, an under-propagation and an over-propagation.

The former case corresponds to any of the optimisation cases presented in the previous section for the DFA on the SSAES. In terms of CNF clauses, it translates into having enough conflicting clauses derived from the duplication of the fault affected rounds. Conflict clauses are clauses which imply an opposite assignment for the same variable in order to be satisfied. For example, if the following set of clauses $(a \vee b \vee c) \wedge (a \vee b \vee c')$ is considered, and assuming that the solver assigned $a = 0$ and $b = 0$ previously, then c needs to be equal to 1 to satisfy the first clause, but 0 to satisfy the second one, leading to a conflict. In this case, the solver would backtrack and try a different assignment for a or b. For AFAs, if enough clauses are conflicting with each other, it is possible for the SAT solver to rapidly eliminate key candidates and recover the correct secret key. However, choosing a fault location or position that leads to over-propagation doesn't speed-up the process either. Similarly to the case of DFAs, the solving process becomes harder if more CNF clauses which don't directly conflict with each others are introduced. This can for example be the case if a single fault is injected at round 8 for $SR^*(10, 4, 2, e)$. For this same case, injecting a single fault one round later would lead to an under-propagation of the fault. In this case, it should be clear that not enough CNF clauses are conflicting, and thus only one half of the key space can be reduced and such under-propagation is not desired in the case of hardware AFAs either.

The process of finding a good fault propagation, or in other words, a good fault model, can be done automatically thanks to the automatic generation of CNF formulas. For instance, in the case of $SR^*(10, 4, 2, e)$, a first step would be to try to inject the fault at the conventional location (round 8), but also in the neighbouring rounds (7 and 9).If two random nibble faults are injected at round 9, one can observe low solving times. However, all

dual faults injected at round 9 are not equal. If the faults overlap, there will be very little gain compared to a single fault at round 9. By observing the specific faults leading to the lowest runtimes, one can choose an appropriate fault model (which would be similar to the previous DFA case). The results are detailed in Section 4.6, as part of the `AutoFault` framework, and generalised for different ciphers.

4.2 AutoFault Structure

The idea behind the `AutoFault` framework [BGE+17] is not only to be able to mount AFAs automatically on diverse variants of the SSAES, but also for any cryptographic primitive. One of the goals is to have a framework which can be used during each design phase of a cipher hardware implementation, in order to assess potential vulnerabilities to fault attacks. In this regard, Figure 4.5 shows the overall structure of `AutoFault`.

`AutoFault` has two main inputs, the circuit model and the fault description. The former consist of the cipher description in a Hardware Description Language (HDL) format. The cryptographic circuit is time-frame expanded and adapted in order to mount the attack (see Section 4.3.1), before being converted to CNF and processed by the main software component of the framework. While only behavioural HDL descriptions of ciphers, in both Verilog and VHDL, have been used in this work, it is also possible to give a gate-level netlist (for instance obtained as output of a synthesis tool) as input to `AutoFault`. The second primary input of `AutoFault` , the fault description, encompass both the fault model and the fault parameters. The fault model refers to the type of injected fault simulated by the framework, such as bit flips or random byte/nibble faults. The fault parameters con-

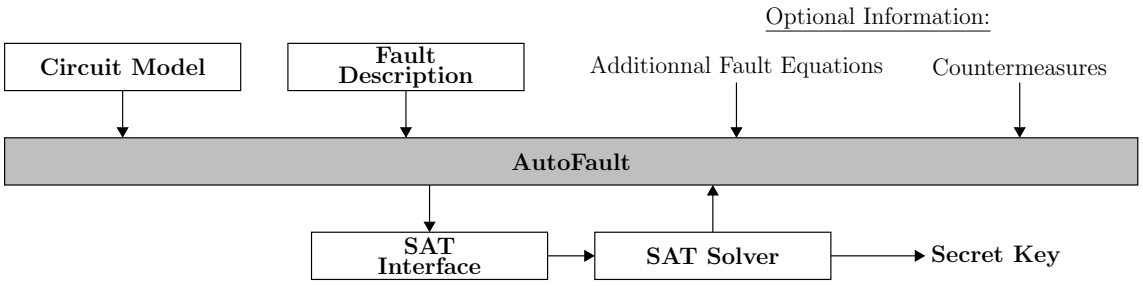

Figure 4.5: Overall `AutoFault` Structure

tain to the fault location and position, respectively the operation before which the fault is injected, and the affected bits of the internal state, as well as the number of injected faults. The fault-related information is mapped to CNF as well before solving. In addition to the two previous inputs, the framework can be used in conjunction with some optional information. One example would be the implementation of counter-measures to mitigate vulnerabilities. In this case, the user can give the hardware description of the implemented counter-measure, which can then be converted to CNF, similarly to the cipher, or restrictions on the fault description can be given as well, for example to simulate shielding (as discussed in Section 2.2.2.1), and the impossibility to inject faults at specific locations. Furthermore, manually crafted fault equations, if they are available, can also be given in order to improve the attack effectiveness.

Once all inputs are defined, the framework processes the data and outputs CNF clauses to a SAT interface. The SAT interface was developed by the chair of computer architecture of the University of Freiburg, and is used as an interface to several SAT solvers used by `AutoFault`. If the SAT solver finds a satisfiable assignment for the given data, it means that a key candidate was found. The key candidate is then fed back to the framework itself, and a fault free encryption is performed. If the ciphertext matches a known correct ciphertext, then the correct key was found, and the frameworks outputs it. On the contrary, if both ciphertexts differ, the key is incorrect. In this case, the framework forwards corresponding CNF clauses to the SAT solver (more details in Section 4.3.3), and the solving process continues until the correct key was found. It should be noted that, in this work, only SAT solvers were used, but other solvers, such as algebraic solvers, can be used to perform the AFA. Other solvers may however require adaptations and conversions of the data (e.g. from CNF to ANF for algebraic solvers).

4.3 Detailed Solving Steps

In this section, details on the different stages `AutoFault` goes through are given in processing order, from the time-frame expansion of the design, to a brief discussion on SAT solving.

4.3.1 Time-Frame Expansion

The first step in mounting an AFA through `AutoFault` is to process the given circuit model. Throughout the remainder of this thesis, this stage will be denoted as the time-frame expansion. It is composed of two main parts, the time-frame expansion itself, and the attack construction. Both respectively modify the circuit under attack, so that it can be expressed in the correct format for the SAT solver used in the later stages, as well as modelling the fault attack component into the circuit itself.

A combinational circuit with n inputs and m outputs can be expressed as a Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$, and any Boolean function can be rewritten in CNF. Consequently, any combinational circuit can be expressed in terms of CNF clauses, which can then be fed to a SAT solver. However, the direct conversion of the circuit to CNF clauses is inefficient. A solution to this problem is to use the Tseitin transform [Tse68]. Every gate constituting the circuit is expressed in CNF, and a new variable is introduced for every gate output. This allows for shorter CNF clauses, at the expense of more variables. Moreover, for combinational circuits, each input which is not an output of a previous gate is a Primary Input (PI), and respectively, every output not connected to another gate is a Primary Output (PO).

After the Tseitin transform is applied, the set of CNF clauses can be fed to a SAT solver and used to solve justification problems. A justification problem consists of finding an assignment for the circuit's inputs and outputs, which is consistent with given logic values at specific locations. More precisely, in the context of this work, specifying some PI and/or PO variables to known values allows to solve for the remaining unspecified ones. Specifically for AFAs, the faulty ciphertexts and the fault free ciphertext POs can be set to known values, as well as the plaintext PI in some instances, while the key variables remain unknown. The SAT solver will then look for a satisfiable assignment for the key variables, in accordance to the given PIs and POs,and thus recover the secret, unknown, key. CNF variables can be easily set to a specific value as each variable represent a single bit. For instance, if one wants to set x_0, the first bit of the fault free ciphertext, to 0, the clause $\{\neg x_0\}$ can be added to the set of CNF clauses.

However, while any circuit can be expressed in this way, only the CNF conversion of combinational circuit would result in a set of CNF clauses which represent the full functionality

Figure 4.6: Time-Frame Expansion & Attack Construction in `AutoFault`

of the circuit. Sequential circuits are composed of combinational modules and clocked memory. Consequently, if a combinational module is reused during runtime, a direct application of the Tseitin transform to the circuit would result in only a single set of CNF clauses for this module, and thus not represent the functionality of the circuit. For example, the SSAES $SR^*(n, r, c, e)$ implementation from Figure 4.1 consist of a single round module, which is clocked and used n times. If only a single set of CNF clauses for this module is available, and if all the PIs and POs corresponding to a full encryption are given, the instance will be UNSAT, since the equivalent of only a single round is performed. As a remedy to this problem, the circuit can be unrolled. An unrolled circuit is constituted of as many repetitions of the combinational modules as needed, glued together. In the previous example of the SSAES (and corresponding to Figure 4.6), this means having n copies of the round module (respectively ten on the figure), and a conversion of this new circuit to CNF clauses would correctly represent an encryption. This unrolling is called time-frame expansion.

Thanks to the time-frame expansion, any circuit, combinational or sequential, can be represented in CNF format. Even so, if the CNF clauses corresponding to the circuit are fed directly to a SAT solver, recovering the secret key will not be possible. Encryption schemes are designed to be computationally difficult to solve, else there would be no point in using them for secure communications. Therefore, it is practically impossible to recover the secret key from the circuit description alone, and as required for any AFA, the addition of some fault information is required for a successful attack. This fault information can be modelled directly into the circuit. Several fault models exist, such as stuck-at faults, bit flip or random nibbles. In this work, only bit flips and random nibbles are considered, which can be represented as an XOR. Consequently, the fault can be simulated by adding an XOR gate at the specific location in the circuit where a fault would be physically injected, and the new fault input is considered as a PI during the Tseitin transform. `AutoFault` can be extended easily to different fault models, for example by using either an AND gate or an OR gate instead of the XOR gate for stuck-at 0, and respectively 1, faults. Nevertheless, in the case of `AutoFault`, and as depicted in Figure 4.6, the XOR gate introduced is of the same width as the intermediate value it is XORed to. During solving, in order to simulate a fault of unknown value, but known position, all the bits (and thus corresponding CNF

variables) of the fault PI (e.g. F_1 and F_2 from 4.6) are set to 0, except the ones corresponding to the actual fault position. For instance, if the fault model is a random 4-bit nibble fault, and the fault is injected in the first 4 bits, then the first 4 variables corresponding to those bits remain unset, while all the following variables are set to 0.

After the introduction of the new XOR gate, the remainder of the circuit is duplicated and the overall new branch of the circuit is connected to the fault free circuit. This method allows for a reduction of the number of CNF clauses created, and thus a faster solving time, as opposed to fully duplicating the circuit with the new XOR gate. Moreover, this process can be repeated for as many fault as needed, in order to simulate multiple fault injections. In the example of figure 4.6, only the last two rounds are duplicated for each fault, instead of the full 10 rounds. In this step, called attack construction, the newly created implementation also has new POs, one for each fault, which correspond to faulty ciphertexts (C_1 and C_2 on the figure). As explained previously, during the solving step, those new POs, as well as the fault free PO are set to known or simulated values. Since the same key is used in each branch of the new circuit, this results in conflict clauses during solving, which restrict the key space, and thus the number of key candidates to be tested.

Additionally, the circuit can also be truncated to only consider the fault affected branches, and a partial fault free branch, which start at the same location as the faulty branches (for example, the three branches of Figure 4.6 without anything preceding). This way, the number of CNF clauses is further reduced, which can improve the solving times. However, this comes at the cost of not being able to use a known plaintext as PI of the circuit, since the truncated circuit's PI is now an intermediate value and is unknown. Consequently, less values on other lines are implied by the PIs, and less conflict clauses are created. Therefore the key space may be larger, which may also lead to an increase in solving times, as more key candidates need to be tested. Even though, in this work, truncating the circuit always led to an improvement in term of solving time (see Section 4.6), it is a trade-off between the overall number of CNF clauses and the key space restriction (overall number of key candidates). Both methods are valid, and truncating the circuit should be considered depending on the considered cipher.

In order to perform the overall time-frame expansion stage (including the attack construction), a series of Python scripts are used, corresponding to each cipher. A script takes as

input the starting round (in case the circuit is to be truncated), the last round, the fault injection round and location, the number of faults, as well as the encryption parameters in the case where a family of ciphers is considered (e.g. the SSAES). It should be noted that the fault injection round is not only specified by an actual round, but also by a specific operation within the round. For example, for an AES implementation, one may want to inject a fault after the eighth *ShiftRows* operation. In this case, the script could be modified such "round" $(X;Y)$ corresponds to round X and operation Y (e.g. $(8;2)$ for a fault before the eighth *ShiftRows* operation). Each of the previous parameters allow to create a new top module in HDL, which is unrolled and has the additional fault branches at the desired location. In addition, the scripts create a Synopsys Design Compiler configuration files, which is used to synthesise the model's gate-level netlist, and copy the required sub-modules (from the original implementation) to the same folder as the new top module.

The scripts then calls Synopsys Design Compiler with the newly created top module and the configuration file, in order to synthesise the circuit. The output of this synthesis tool is a new gate-level Verilog netlist, of the time-frame expanded original circuit, with the addition of the fault information. This new circuit is not a circuit which would be manufactured, but rather a model used for the analysis of the original circuit vulnerability to fault injection attacks. It can then be converted to CNF through the Tseitin transform.

During the creation of the new circuit, the names of the inputs and outputs are also changed for ease of recognition during the later stages of the attack. For instance, the plaintext input is renamed as "*intermediate*", as it may correspond to either the plaintext, in the case of a full circuit, or to an intermediate state, in the case of a truncated circuit. This renaming is important to ensure that the process remains automated and that no further user input is needed to identify each PIs and POs.

Finally, in addition to the new circuit to be attacked, another circuit is generated. This circuit, denoted as simulation circuit, is used during the preparation of the attack, in order to simulate a fault free encryption, as well as the several faulty encryption needed. More details on the CNF simulation are available in Section 4.4, but the principle is to have a circuit composed of all the rounds, and a single fault location, in order to be able to generate values for the attack, by giving chosen inputs to this circuit.

4.3.2 CNF Conversion

Once the time-frame expansion of the original cryptographic implementation has been performed, it is ready for the CNF conversion stage. In this stage, the newly generated hardware description undergoes the Tseitin transform. This operation is performed by PHAETON [SBP16], a tool developed by the chair of computer architecture of the University of Freiburg. The tool can apply the Tseitin transform directly to the circuit given a correct format of the netlist. The format is ensured during synthesis by the configuration file generated for Synopsys Design Compiler.

While the Tseitin transform is automatically performed by PHAETON, in order to be able to process the CNF clauses for an AFA, a header containing the information on the PIs and POs of the circuit is necessary. To this end, PHAETON was modified to add this functionality. A new application module was added to PHAETON. The new module first loads the circuit and all the sub-modules necessary for the CNF conversion already present in the tool. Then, the following naming scheme (from Section 4.3.1) is used to correctly identify each pin of all the PIs and POs automatically identified by the framework. The plaintext, or intermediate input in the case of truncated circuits, is denominated by "*intermediate_Y*" (where Y corresponds to the bit/pin of the PIs or POs), the key by "*key_Y*", the faults by "*faultX_Y*" (where X is the fault's number), the ciphertext by "*cipher_Y*", and the faulty ciphertexts by "*cipher_faultyX_Y*". Additionally, potential masks used for masking counter-measures (see Section 2.2.2.2) are denominated by "*maskX_Y*" and a generic "*helperX_Y*" name can be used for any other pin (at the discretion of the user). This allows to first assign a single letter to each PIs and POs, for easier identification in the later stages of `AutoFault`, while also allowing to automatically get the sizes (in terms of bits) and number of variables (e.g. number of faulty ciphertexts). Once the mapping is done, the module checks to which cipher the circuit corresponds, in order to generate an appropriate file name, as `AutoFault` later automatically gets the solving settings from the file names. If the cipher is not recognised as a supported cipher, the module selects a generic file name, which can still be used by `AutoFault`. The Tseitin transform is then finally applied and the CNF file, containing all the CNF clauses, is generated. Lastly, the header is generated from the previous mapping and appended to the CNF file.

```
c variable correspondence:
c l->keyPinLits->key_; Verilog input;
c f->faultPinLits->faultX_; Verilog input;
c p->intermediatePinLits->intermediate_; Verilog input;
c x->cipherPinLits->cipher_; Verilog output;
c y->cipherFaultyPinLits->cipher_faulty; Verilog output;
c m->maskPinLits->mask_; Verilog input;
c a->helperPinLits->helper_; Verilog input;
c 49 := l_0, 48 := l_1, ... , 34 := l_15, -972 := k_0, ... , 982 := k_15, -1301 := x_0, ... ,
    -1292 := x_15, 65 := p_0, ... , 50 := p_15, -1297 := y_0_0, ... , -1285 := y_0_15,
    -1295 := y_1_0, ... , -1283 := y_1_15, 33 := f_0_0, ... , 18 := f_0_15, 17 := f_1_0,
    ... , 2 := f_1_15,
p cnf 1309 4436
c --- START CLAUSE DB
1 0
-66 50 2 0
...
```

Listing 4.1: Shortened CNF Header of an SSAES

Listing 4.1 is a shortened example of a CNF header created for an $SR^*(10, 2, 2,)$ SSAES. The line starting with a "*c*" of the CNF file are comments ignored by the SAT solver, and correspond to the header. First all the PIs and POs of the cryptographic circuit, which follow the overall naming scheme of `AutoFault`, are listed with their corresponding letter, in order to be humanly readable. For instance, the letter "*f*" relates to the injected faults, and should be followed by the fault number. In this example, two faults are injected, therefore "*f_0_0*" corresponds to the bit 0 of the first injected fault, while "*f_1_0*" refers to the same bit of the second injected fault. Then, the variable correspondence to the the CNF variables is listed: a number refers to a CNF variable and is associated with the corresponding letter. For instance, in Listing 4.1, variable "*33*" is in fact the bit 0 of the first fault. After the header, the number of CNF clauses, as well as the number of variables is also given, and finally all the CNF clauses are listed until the end of the file.

Moreover, additional variables can be mapped for the evaluation of counter-measures or more intricate ciphers, which require different inputs. For instance, mask's pins are mapped to the letter "*m*", and dedicated helper pins (mapped to "*a*") can be used for any other generic inputs or outputs which may be needed. Helper pins are however at the discretion of the framework's user, and need to be properly handled later on.

A header such as the one of Listing 4.1 is necessary for the next stage of the attack, where specific CNF variables are set to some given values, and fed to a SAT solver, in order to mount an AFA.

4.3.3 CNF Processing and Mapping

After the CNF conversion, the attack can be mounted. This is handled by the main module of `AutoFault`, which is denoted as the AFA module in this chapter. The AFA module, written in C++, handles the CNF inputs as well as the diverse settings of the attack, and returns the secret key after a successful attack. It is composed of multiple C++ sub-modules, each of them dedicated to a specific task. Figure 4.7 lists all the major sub-modules, as well as the folder architecture which constitute the AFA module.

During the attack stage, when the AFA module is called, it can either be called by specifying every attack parameters manually or by specifying a setting file. Consequently, the first sub-module called is the *settings* sub-module. It parses the specified setting file (present in the correspondingly name folder), and sets all of the attack parameters, as well as the any parameters which can be derived from the setting file or the CNF inputs, to the correct values. Such parameters include the size of every variable, the number of each variable (in the case where several are available, e.g. the faults), every cipher specific value (e.g. number of rounds, fault injection round...) or the size of the fault. They are automatically derived from the CNF file, by following `AutoFault`'s naming scheme or parsing the file's name itself. Other settings related to file paths or solver parameters are also given in the setting file. The user can for example set the number of attacks to be performed, the SAT solver to be used, or the simulation mode in this file. Listing 4.2 shows an example of setting file.

The *CNFPreProcessor* is called throughout the attack phase in order to handle the CNF files, with their header. Several functions parse the circuit's files, starting with the header. The different letters associated with the PIs and POs are mapped to the corresponding `AutoFault` variables. This is done according to the previously defined settings. Once the header has been processed, the circuit's CNF clauses are loaded by the SAT logic via the *solver-proxy* sub-module. In the case where the simulation is performed via CNF clauses

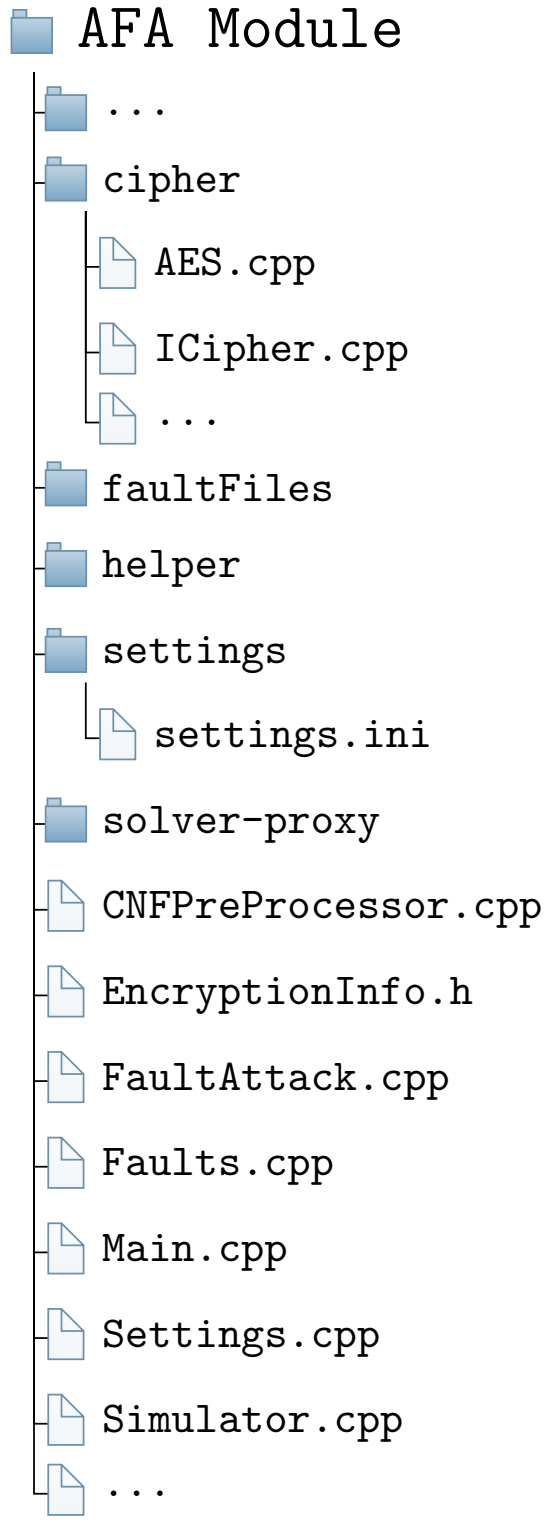

Figure 4.7: `AutoFault` AFA Module Structure

```
solver = cryptominisat ; Chosen SAT solver
noOfThreads = 8 ; Number of threads

cipher = "ssAES" ; Cipher to attack. If the file name contains it: not necessary
repetitions = 1 ; How often the circuit should be attacked
timeout = 86400 ; Timeout for one attack, in seconds
randomSeed = 1579630018 ; Initial seed for the random number generator
cnfSimulator = true ; CNF simulation or software simulation

; Path to the circuit file
circuitFile = "../../faultAttackCircuitsAsCNF/SSAES/circuit_ssAES_2-2-4_round_8
    -10_faultAt_8_skipMc_1_fault_injections_2_formula.cnf"

; Path to the simulator file
simulatorCircuitFile = "../../faultAttackCircuitsAsCNF/SSAES/circuit_ssAES_2-2-4
    _round_8-10_faultAt_8_skipMc_1_fault_injections_2_formula_sim.cnf"

; Random mode or manual fault injections
randomMode = true
;The fault file, if randomMode set to false
;faultFile = "../faultFiles/faultFile_ssAES_2-2-4_round_8-10
    _faultAt_8_skipMc_1_fault_injections_2"
```

Listing 4.2: Example of Setting File

(i.e. not through software implementations of the diverse ciphers), the *CNFPreProcessor* is called twice, once for each CNF file (corresponding to each circuit).

The *Simulator* sub-modules is used for the generation of the values needed for the attack. The main purpose of the *Simulator* is therefore to call the correction function with regards to the parameters. dependent on the simulation mode, it either calls a software implementation for the considered cipher, or a SAT solver in the case where a CNF simulation circuit is provided (more details in Section 4.4). In the case where the simulation is done fully in software, the *Simulator* calls the *ICipher* sub-module present in the *cipher* folder. This sub-module is used to correctly map the required functions to the correct cipher. For instance, if an AES encryption is required to generate a fault free ciphertext, the *ICipher* sub-module refers to the encryption function available in the *AES* corresponding sub-module. The *Simulator* is prepared right after the initialisation of the *CNFPreProcessor*.

Once both the *CNFPreProcessor* and the *Simulator* sub-modules have been initialised, the *Faults* sub-module is called in order to generate the necessary attack values. `AutoFault` can be used in two modes, either random, or manual. If `AutoFault` is in random mode,

then all the inputs necessary for the attack are randomly generated in the *Faults* sub-module, before being forwarded to the *Simulator* in order to derive the corresponding outputs. For instance, to mount an AFA, a fault free and several faulty ciphertexts are required. Once the input values (i.e. plaintext, key and faults) are randomly generated, the *Simulator* performs both fault free and faulty encryptions to generate the outputs. However, during the solving phase, the key is of course "forgotten" (i.e. not forwarded to the solver), in order to simulate an AFA. If the selected mode of `AutoFault` is the manual mode, the *Faults* sub-module parses a file available in the *faultFiles* folder, and specified in the settings. In this case, only the required values are present in the fault file (i.e. ciphertexts and fault position), and not the secret key. This mode is aimed at performing AFA from physically injected faults, instead of simulation. In either case, all the values are stored in a new data structure defined in the *EncryptionInfo* header. This data structure contains all the encryption's data (e.g. plaintext, ciphertext, faults...), both as vectors of integer and characters for usage with other functions of `AutoFault`.

With a fully filled *EncryptionInfo* data structure, the attack can be performed. The *FaultAttack* sub-module is consequently called, and sets all the necessary CNF variables to the correct values. For example, all the CNF variables of the fault free ciphertext are set to the corresponding values of each bit of the ciphertext member of the *EncryptionInfo* data structure, the key is left unknown and the plaintext can be set if desired. In the case where the fault model is a random nibble fault, the fault itself is not specified, but only the fault position, as an attacker would not be able to know which fault was injected, but would be able to control the fault position. To ensure this, and since the fault variable is as wide as the intermediate data it is XORed to (see Figure 4.6), a single nibble is selected randomly. With the position of the nibble known, the *FaultAttack* sub-module sets all the CNF variables corresponding to the bits unaffected by the fault to 0, and only the remaining variables stay unspecified. Every CNF variable is set to 0 or 1 by making a call to the SAT solver (through the *solver-proxy*), and adding a new unite clause to the set of CNF clauses. Once every necessary variable is set, a call to the SAT solver is performed, in order to solve for the secret key (i.e. find a satisfiable assignment for the given inputs and outputs).

Once a satisfiable assignment is found, this means a key candidate is available. The *FaultAttack* sub-module reads the SAT flag returned by the SAT solver, as well as the CNF vari-

ables corresponding to the key candidate. The *Simulator* is then called in order to perform an encryption with a known plaintext and the key candidate. If the resulting ciphertext is the same as a known correct ciphertext, then the key candidate is the correct secret key, and `AutoFault` returns it. If the two ciphertexts are different, then an incorrect key was found. In this case, a new CNF clause corresponding to the key is set to 0, and, without resetting the solver, a new call to the solver is made (incremental solving). The process is repeated until the correct key is found. In the event where the SAT solver would exhaust every key candidate, and return an UNSAT flag, either the design or the fault model is incorrect.

The overall AFA module repeats the previous steps for each attack, and returns either the secret key, an UNSAT flag or a timeout, if the solving took longer than the user specified timeout. Moreover, sub-modules available in the *helper* folder are used to record some statistics on the attack, for instance the solving time, and a log file is created with the information concerning the AFAs.

4.3.4 SAT Solving

A SAT solver is used to perform the AFA. However, in `AutoFault`, multiple SAT solvers are supported. This is thanks to the *solver-proxy* sub-module, which serves as a compatibility layer for the divers SAT solver supported by `AutoFault`. The *solver-proxy* sub-module was developed by the chair of computer architecture of the University of Freiburg, who are experts in SAT instances for circuit oriented problems. It is therefore not part of this thesis's work. Nonetheless, the *solver-proxy* is an important part of `AutoFault` and is briefly mentioned in this section for completeness.

Thanks to the *solver-proxy* sub-module, different SAT solver can be used in conjunction with `AutoFault`. Currently, the following SAT solvers are supported: antom [SLB10, SR16], CaDiCaL [Bie17], Glucose [AS18], MapleLCMDistChronoBT (MLBT) [RN18] and CryptoMiniSat (CMS) [SNC09]. When a call to the SAT solver is made in the AFA module, it is mapped to the correct functions depending on the selected solver. For instance, the addition of unit clauses for the each known variable is different for every SAT solver, but the *solver-proxy* allows for an easy and automated way to switch between solvers.

During SAT solving, the multiple ciphertexts being set to specific values will create several conflict clauses, which will reduce the number of possible assignment for the remaining unknown variables. The variable of interest in the context of AFAs is the secret key, however, a satisfiable assignment still needs to be found for other variables as well. Nevertheless, in order to recover the secret key, having an efficient SAT solver is essential. To this end, *solver-proxy* was used to compare different SAT solver and the results for an SSAES $SR^*(10, 4, 4, 4)$ are available in Table 4.1. From this table, it can be seen that CMS was overall the most efficient SAT solver for AFA problems, especially in multi-core mode. Results are similar for different ciphers, and thus CMS was chosen as the primary SAT solver.

Table 4.1: Comparison of SAT Solver Solving Time - SSAES $SR^*(10, 4, 4, 4)$

Time in s	antom	CaDiCaL	Glucose	MLBT	CMS - Single Core	CMS - 4 cores
Average	790.6	42.9	38.0	216.9	39.7	18.3
Median	553.6	33.9	33.8	207.2	32.3	16.3
Minimum	101.5	24.2	8.3	163.3	13.2	11.8
Maximum	2019.0	81.1	73.3	280.2	80.7	35.6

4.4 CNF Simulation

The simulation of values is an important step in `AutoFault`, as well as for the automation of AFAs. In the first implementation of `AutoFault` [BGE+17], the simulation was handled fully by the AFA module, and a software implementation of each cipher was therefore necessary. Each software implementation allowed for encryptions, with and without faults, to be performed, or for key scheduling. This way, the necessary fault free and faulty ciphertexts could be generated in order to mount an AFA (unless `AutoFault` was used in manual mode, with values resulting from physical fault injections). While having a software implementation of each cipher allows for fast generation of values, it requires manual work to add a new cipher to the framework, as a new implementation is needed. Similarly, if a counter-measure is added to the hardware implementation, it may be required to update the software implementation linked to the newly protected encryption scheme, or even completely re-implement it. Since one of the goals of `AutoFault` is to automate the

evaluation of hardware implementation of cryptographic primitives, this step needed to be automatised further.

During SAT solving of an AFA instance, the CNF clauses corresponding to the ciphertexts are set, as well as the one of the fault position, which allows for the SAT solver to find a satisfiable assignment for every unset value. The same principle can be used for the generation of values for the attack, and is denoted as CNF simulation. If a ciphertext is needed, then the CNF clauses corresponding to the hardware implementation of the circuit can be used, and by setting every PI to given values, only a single ciphertext satisfies the set of CNF clauses, and is returned by the SAT solver. However, since `AutoFault` supports the truncation of the fault affected circuit, the circuit generated during the time-frame expansion cannot always be used for this purpose, as in this case, the plaintext input is not present, but rather an intermediate state. Moreover, if the circuit to be attacked contains multiple faults, the number of CNF clauses is increased significantly by the duplication of the fault affected operations, which increases the solving time. Therefore, during the time-frame expansion, a second circuit, containing every operations (i.e. not truncated), and a single fault is generated automatically. This circuit can then be used in conjunction with the SAT solver to generate, first a fault free ciphertext given the plaintext and key as input, and then every faulty ciphertexts necessary, by varying the value of the single fault, and resetting the SAT solver after every encryption. Additionally, the same circuit is used during the key candidate checking phase, by setting the plaintext and the key candidate CNF variables to the specified values in order to generate a new ciphertext and compare it with a known correct one.

Using this approach for the simulation of values for an AFA, as opposed to a software implementation of each cipher, means that only a simulation circuit is necessary for this step, and essentially any cipher can be easily added to `AutoFault`. The generation of the simulation circuit leverages `AutoFault`'s functionality to produce a complete circuit with a single fault injection, meaning that the Python scripts from the time-frame expansion can handle this process without any modification, and only with a second call. Moreover, in the event where a counter-measure has been added to the considered circuit, the time-frame expansion needs to be redone in any case, and thus employing the CNF simulation is not only simpler than re-implementing the protected cipher in software, but also only a

single additional call to the time-frame expansion script, making the evaluation of counter-measures simpler.

It should however be noted that, for encryption schemes with a key scheduler, the reverse key schedule operation, needed in the case of truncation, is still processed in software for simulation. This is due to the fact that, while possible to do in CNF as well, this would require more modification to the time-frame expansion scripts, and `AutoFault` itself, for the processing of one additional input. Since a reverse key schedule is usually simple to implement in software, and often independent of the counter-measures which may be added, this choice was made for `AutoFault`.

Furthermore, attacks on multiple ciphers, and for several repetitions, were conducted with `AutoFault` in CNF simulation mode. The runtimes were comparable to the runtimes of `AutoFault` without the CNF simulator. Therefore, since no noticeable differences were observed, the CNF simulation can be used at no extra cost, or at least no noticeable cost, despite several calls to the SAT solver, reinforcing its usefulness.

4.5 AutoFault in the Design Flow

`AutoFault` can automatically generate AFAs given a hardware implementation of a cipher. Since `AutoFault` can handle multiple type of inputs and fault models, it can be used at diverse stages of the cryptographic hardware's design phase. This is especially useful for automated evaluation of such critical implementations. This section discusses how `AutoFault` can be used for this purpose, and Figure 4.8 summarises the overall use cases of the framework, which can be divided in two categories: pre-silicon and post-silicon analysis.

Pre-silicon analysis refers to the use of `AutoFault` the manufacturing of the cryptographic circuit. Consequently, no physical hardware is available, and no physical fault injections can be realised. In turn, this means that the fault injection needs to be simulated. However, the circuit designer should not blindly simulate any fault. While the designer can use `AutoFault` to check different fault locations deemed critical, a fault model needs to be chosen. The fault model should match the fault injection equipment available to an attacker, and his general fault injection capabilities. The assumptions for the fault model

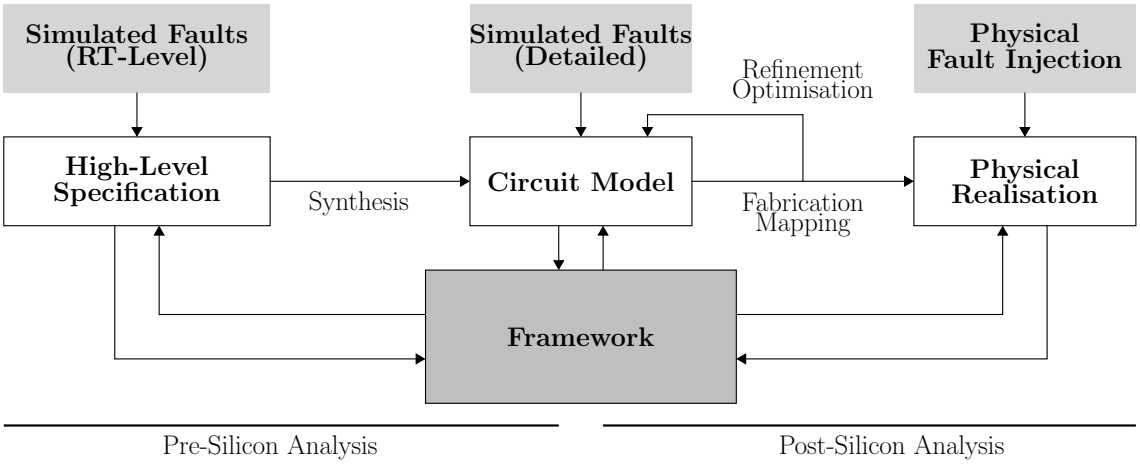

Figure 4.8: `AutoFault` Usage during the Design Flow & Fault Models

are both temporal and spatial (i.e. fault location and position). For example, an attacker who has access to a highly accurate, both spatially and temporally, fault injector could aim at injecting single bit flip faults. Even though his aim is to inject such faults, the reality of physical fault injection could mean that, sometimes, the neighbouring bits may be flipped as well. In this case, the fault model should be created accordingly, for instance by considering random nibble faults instead of only bit flips, or if the probability of flipping the neighbouring bits is known, then it can be mapped into the fault model as well. Nevertheless, and even if the designer should consider several fault models, matching the fault model in pre-silicon analysis is much simpler than in post-silicon analysis, where the physical fault injector has to be considered.

Once the fault model for simulation has been defined, the framework can be applied either in the early stage of the design phase, or before manufacturing. In the early stages, only the high-level specification is available. This corresponds to the register-transfer level, where only states are available, as well as the overall flow of operations between each state (e.g rounds, key schedule...). In this case, the time-frame expansion scripts (as explained in Section 4.3.1) are used before synthesising the design to mount the attack. If the design has already been synthesised, then a gate-level implementation is available. This means that, instead of the overall operation flow, each gate implementing each operation is available. In this case, logic gates can be a target for the fault injection, and, with the complete netlist, it is possible to represent such an attack directly. However, this is not the

case at the register transfer level, and the fault needs to be approximated, for instance by the addition of an XOR operation with an intermediate state. The difference between both cases is thus the available information about the fault model, and how it is mapped in the circuit implementation, before the conversion to CNF, and the AFA itself.

The opposite use case for `AutoFault` is the post-silicon synthesis. The framework can be used after manufacturing of a chip, which allows for further evaluations of the design with physical fault injection hardware. While a fault injection setup lets the designer evaluate its design against a real world scenario, the fault model specified in `AutoFault` should match the injected faults. Depending on the equipment available, this is not necessarily a simple task, but if both the fault model, and the physical effect of the faults are different, then the SAT instance will most likely be UNSAT during solving, or at least the attack will be inconsistent (i.e. produce incorrect key candidates). This is due to the fact that, if the fault location and position are not correctly specified, but the faulty ciphertexts resulting of the fault injection are manually forwarded to `AutoFault`, then the SAT solver will try to find a satisfiable assignment given the corresponding fault parameters, which is unlikely if the fault was actually injected at another location, or affected more bits (an example of this is discussed in Section 4.6). Practical attacks often have a large key space, meaning that the processing time can be rather long. If the fault model is incorrect, and the attack inconsistent, a large amount of time can be wasted on trying to solve the attack with the framework. One possibility to avoid this, is to use a more lax fault model (e.g. allow for more bit flips). Similarly to the pre-silicon analysis, a laxer fault model increases the number of key candidates. Consequently, the solving process will also take more time, and if the model is too lax, for example in the case of an unknown fault position, not enough restrictions will be placed on the key space, leading in the worst case scenario to an attack equivalent to a brute force attack. The user should be aware of this, and find the correct balance between a lax fault model, with too many key candidates, and strict one, not matching the physical fault injector behaviour.

No matter if `AutoFault` is used in pre- or post-silicon analysis, it can automatically mount an AFA, given a meaningful fault model, and by doing so a designer can identify design flaws and vulnerabilities. If such a vulnerability is discovered, the design can be modified to resolve the issue. For instance, a designer can decide to add a counter-measure at

a specific location in the circuit. One example of physical modification could be to add some shielding over a critical area of the circuit in order to prevent fault injections at this location. Another possibility could be to add an ECC counter-measure (such as the one presented in Chapter 3). In both cases, it is simple to use `AutoFault` again and evaluate the newly modified design before proceeding to the next design phase. Furthermore, at any stage, the design can be optimised (e.g. pipelined), and re-evaluated in the same way. This process can be repeated until no vulnerabilities are found, and for different fault models, in order to cover more possible fault injections.

4.6 Hardware-Oriented AFAs on SPN Ciphers

In this section, results on the application of the `AutoFault` framework on different ciphers are presented. The framework was applied to Substitution and Permutation Network (SPN) ciphers, as they are widely used, which holds in particular for the AES. The considered ciphers have known vulnerabilities against conventional fault injection attacks. However, and as motivated in Section 4.1, even a slight modification of the encryption scheme, such as a different variant of the SSAES, changes the fault propagation pattern, which may lead to either under- or over-propagation and an unsuccessful fault attack. Therefore, SPN ciphers were chosen as a baseline to be met, in the case of known successful fault models, and an evaluation of the capacity of the framework to automatically mount different attacks, either less optimised, or on different variants of the ciphers.

The experiments were conducted on a commercial grade AMD Ryzen 9 3950X processor with 16 cores and 128GB of RAM. Moreover, the CryptoMiniSat (CMS) [SNC09] solver was used, as it was found comparatively more efficient than other SAT solvers for AFA instances (see Section 4.3.4). The tables presented in this section showcase the runtimes of the AFA module of `AutoFault` (i.e. of the attack), as well as the number of key candidates considered until the secret key was found, denoted as #KC. While the number of key candidates until the attack is successful does not exactly indicate the size of the key space, it gives an indication about the attack and the fault model considered. Furthermore, all fault injected for SPN cipher are random nibble (or respectively byte) faults, of the respective size of the cipher's intermediate state's operations (usually the SBoxes).

4.6.1 AES (including Small Scale variants)

The AES is one of the most widely used cipher, and optimised DFA attacks, such as the one presented in Section 2.1.4.2 with a single fault injection at the beginning of the eighth round, are known. In this section, an AFA with the same fault model as the previous attack was considered for validation of `AutoFault` functionalities, as well as other fault locations. Moreover, SSAES variants were attacked as well, to showcase the scalability of the framework, and are a direct automation of the DFA presented in Section 4.1. Additionally, it should be noted that, in comparison to the DFA attacks, the framework does not necessarily proceed to do the second step of the attack using the key schedule equations. It is possible in `AutoFault` to omit the key schedule module, and instead have either a new input for each round key, or hard code the round keys directly into the circuit. For the following experiment, this is however not the case, and the key schedule was present. It is nevertheless an option in the framework, which can be used by designer to emulate a design were the keys have been precomputed.

First, it is important to find a suitable fault model for the attacks. To this end, and thanks to the scalability of the SSAES, smaller square (equal number of rows and columns) variant of the AES, can be attacked with the framework, before generalising to larger variants, or variants with unequal number of rows and columns. Based on the initial findings for the DFA on the SSAES $SR^*(10, 2, 2, 4)$ and $SR^*(10, 4, 4, 4)$, a fault injection at the beginning of the eighth round should lead to the most efficient attack. Table 4.2 shows experimental results for both variants of the SSAES, with the previous fault model and a complete circuit, over one hundred mounted attacks. For each cipher, both a single fault injection and two fault injections were considered. It can be seen that, in both cases, the injection of a second fault led to faster solving times. This is expected, as a second fault injection constraints the key space further, which can be observed in the table as well, by the extremely low number of key candidates. With two fault injections, almost always only a single key candidate remains, while the number of candidates for single fault attacks is much more important. The difference in term of runtime for the smallest variant is marginal, but this is due to the size of the cipher itself. The SSAES $SR^*(10, 2, 2, 4)$ is a 16-bit cipher, and therefore only very few key candidates remain with a single fault injection. This leads to barely slower solving times, and even a better one in the case

of the minimum, which is due to sheer luck during the attack, as the first key candidate was the correct one (also why the minimum number of key candidates is 1). However, the improvement for the $SR^*(10, 4, 4, 4)$ is very noticeable, with an improvement of more than two orders of magnitude. Since larger variants of the SSAES mean longer processing time, and two fault injections remains low, two faults are injected for the following attacks on the SSAES.

Table 4.2: Number of Faults Comparison for the SSAES - 100 Attacks

	SSAES $SR^*(10, 2, 2, 4)$		SSAES $SR^*(10, 2, 2, 4)$		SSAES $SR^*(10, 4, 4, 4)$		SSAES $SR^*(10, 4, 4, 4)$	
#Faults	1		2		1		2	
	Time in s	#KC	Time in s	#KC	Time in s	#KC	Time in s	#KC
Average	2.69	15	2.51	1	2714.12	1516	12.49	1
Median	1.98	12	1.84	1	558.53	393	10.27	1
Minimum	0.05	1	0.10	1	15.49	8	2.45	1
Maximum	10.81	97	9.09	2	50349.50	26541	39.81	1

Table 4.3 shows results for the same two SSAES variants with two fault injections, but using a truncated circuit to improve the solving times. Moreover, ten thousand attacks were ran to have a larger sample size, and thus less corner cases on average. In comparison to the previous table, runtimes are smaller, already showcasing the improvement of only considering the relevant part of the circuit (refer to Section 4.3.1). In this case, only rounds 8 to 10 are considered. Table 4.6 showcases this further more for the full scale AES. Overall, for small and square variant of the SSAES, solving times are in the order of seconds, even for the 64-bit SSAES $SR^*(10, 4, 4, 4)$.

Table 4.3: Runtimes for SSAES $r = c$ with 2 Fault Injections - 10000 Attacks

	SSAES $SR^*(10, 2, 2, 4)$		SSAES $SR^*(10, 4, 4, 4)$	
	Time in s	#KC	Time in s	#KC
Average	1.36	1	3.29	1
Median	0.83	1	2.84	1
Minimum	0.02	1	1.69	1
Maximum	11.61	4	14.53	2

Following the experiments on the square variants of the SSAES, and to confirm the results on the DFA attacks of the SSAES from Section 4.1, let us consider both the SSAES $SR^*(10, 2, 4, 4)$ and $SR^*(10, 4, 2, 4)$. Two faults were injected at the beginning of round

8 for both ciphers, and an additional attack at the beginning of round 9 for the SSAES $SR^*(10, 4, 2, 4)$ was performed as well, since it was the most optimised fault attack in the case of the DFA. Concerning the attack of the former cipher, the solving times range from less than a second to around three hundred seconds, as shown in Table 4.4, and similarly, the number of key candidates range from a single one, to more than 18000, over ten thousand attacks. If one recalls Section 4.1, these attacks correspond to the optimised DFA, since two faults were injected. However, the fault positions were chosen randomly in `AutoFault`, and therefore, a complete fault propagation (see Figure 4.2) was often not achieved. Since the fault propagation is not always optimal, the key space is not restricted well enough, which explains the large number of key candidates and long runtimes. In the worst case, two faults were injected in the same nibble, leading to a key space of approximately size 2^{16}, of which around a fourth of the key candidates were tested. The opposite case is also true. The attack with only a single key candidate and a runtime of less than a second is in fact an attack following the fault model defined for the DFA, which led to those results. In addition, manual fault injections were performed following the same optimised fault model, and the runtimes and number of key candidates were similar. This is an expected result, but, from a designer's perspective, studying in more details the patterns in the fault injections which led to the lowest runtimes would showcase the vulnerability of the cipher to especially dual faults of this type, without the need to perform a manual cryptanalysis of the cipher.

Table 4.4: Runtimes for an SSAES $SR^*(10, 2, 4, 4)$ with 2 Fault Injections - 10000 Attacks

	SSAES $SR^*(10, 2, 4, 4)$	
	Time in s	#KC
Average	18.93	791
Median	8.31	317
Minimum	0.18	1
Maximum	300.04	18050

In the case of the SSAES $SR^*(10, 4, 2, 4)$, the optimal fault model defined in Section 4.1 was two faults at the beginning of the ninth round, while fault injections at round 8 lead to over-propagation of the fault, and thus more complex equations. The experiments from Table 4.5 show that, on one hand and despite the increase in complexity, fault attacks at round 8 are still successful. Moreover, due to the high propagation of the fault, the key

space is constrained to the point were only a single key candidate remains. On the other hand, the more optimised attack at the next round has on average much worse results (and thus only a thousand attacks were ran, due to the increase in solving time). This is due to the random nature of the faults injected by the framework, instead of specifically chosen, and is similar to the previous case of the SSAES $SR^*(10, 2, 4, 4)$. However, if the fault model matches the two fault positions presented in Figure 4.4 (or equivalent positions), the solving time reduces dramatically, as well as the number of key candidates (which can be observed in the table, and was validated with manually selected faults). Once again, the attack on the SSAES $SR^*(10, 4, 2, 4)$ shows how `AutoFault` can be used to automatically find the most suitable fault locations and positions, especially in the case of multiple fault injections. Additionally, in the case of over-propagation of the fault, even if the runtimes are slower than more well targeted attacks, some attacks remain mountable and successful. As such, the framework highlights that, even if state-of-the-art attacks exist, other fault locations should not be discarded as potential risk, and counter-measures may be necessary at different locations.

Table 4.5: Fault Location Comparison for an SSAES $SR^*(10, 4, 2, 4)$ with 2 Fault Injections

	SSAES $SR^*(10, 4, 2, 4)$		SSAES $SR^*(10, 4, 2, 4)$	
#Attacks	10000		1000	
Fault Location	Round 8		Round 9	
	Time in s	#KC	Time in s	#KC
Average	8.31	1	149.75	6844
Median	4.82	1	85.59	3974
Minimum	1.05	1	0.20	1
Maximum	82.40	1	300.06	20320

The preceding experiments only considered smaller versions of the SSAES, and what can be considered as lightweight ciphers. In reality, larger cryptographic circuits are used as well, such as the full scale AES. Table 4.6 presents `AutoFault` data for attacks on the AES and the SSAES $SR^*(10, 4, 4, 8)$. Both ciphers only differ in their respective key schedule. Several attacks on both ciphers were conducted, and the runtimes were similar. Therefore, the marginal difference in key scheduling does not induce any additional difficulty for the framework, and both ciphers are equivalent (as claimed in [CMR05]). Since both ciphers are interchangeable, Table 4.6 compares the effect of the truncation of the circuit

between a complete AES implementation, and a truncated $SR^*(10, 4, 4, 8)$ (round 8 to 10), both with two faults injected before the eighth round. It is clear that the number of key candidates remains the same, and low, however the solving times are greatly reduced by the truncation of the circuit. Even though attacks are automated via `AutoFault`, a much lower runtime is essential for iterative evaluation of a cryptographic circuit, and consequently, results show the advantages of the truncation functionality.

Table 4.6: Effect of Truncation on Runtimes for an AES with 2 Fault Injections

	AES		SSAES $SR^*(10, 4, 4, 8)$	
#Attacks	100		1000	
Truncated Circuit	No		Yes	
	Time in s	#KC	Time in s	#KC
Average	2738.19	1	421.85	1
Median	1586.45	1	348.59	1
Minimum	388.38	1	113.29	1
Maximum	25598.40	1	3509.28	1

Finally, the previous attacks on the SSAES were all performed directly from the circuit description in HDL, corresponding to a pre-silicon analysis, but `AutoFault` can also be used for post-silicon analysis. To this end, the fault injection setup from Figure 3.10 was used to inject faults onto an SSAES $SR^*(10, 4, 4, 4)$ implemented on a SAKURA-G board. Due to the imprecise nature of the fault injector, only roughly 5.9% of the successful fault injections resulted in single nibble (i.e. 4 bits) faults at the beginning of the eighth round. The single faults were then selected, and given as input to the framework, with the corresponding faulty ciphertexts. `AutoFault` was able to recover the secret key for most successful single fault injections, in similar times to Table 4.2, and Table 4.3. However, a few fault injections resulted in faulty outputs which led to an UNSAT instance for the SAT solver. This is due to the imprecise behaviour of the fault injector, and after careful inspection of the injected fault, it was discovered that the fault injector affected the ninth round in addition to the eighth round, resulting in a fault model different from the fault model chosen in `AutoFault`. This last SSAES experiment validates the use of the framework for post-silicon analysis, and shows the importance of having a matching fault model between `AutoFault` and the physically injected faults.

4.6.2 LED

The LED cipher [GPPR11] is a lightweight block cipher commonly used in constrained devices. As such, it is of particular interest for the `AutoFault` framework. A well known location for a successful fault injection attack is at the beginning of the 30^{th} round. `AutoFault` was used to evaluate an hardware implementation of LED against fault injections at rounds 29, 30 or 31. The latter were not successful, but this is to be expected, as in this case the fault does not propagate enough to sufficiently constrain the key space. Similarly to the case of the AES, the injection of two faults, as well as the truncation of the circuit to only consider the remaining rounds from the fault injection round, significantly improved the runtimes. Table 4.7 shows the runtimes and numbers of key candidates for AFAs at rounds 29 and 30 (only ten attacks were ran for the former, due to the long runtimes).

Table 4.7: Fault Location Comparison for the LED Cipher with 2 Fault Injections

	LED64		LED64	
#Attacks	10		10000	
Fault Location	Round 29		Round 30	
	Time in s	#KC	Time in s	#KC
Average	213737.00	1	3.96	1
Median	92797.55	1	3.91	1
Minimum	12845.11	1	2.62	1
Maximum	684122.00	1	9.39	83

From Table 4.7, it can be seen that, as expected, the attack of round 30 runs flawlessly for `AutoFault`, with low solving times. One can note that the maximum number of key candidates is higher than for the SSAES $SR^*(10, 4, 4, 4)$, due to the different fault propagation pattern in LED. Despite this, the runtimes of both ciphers are similar for a fault injected at the beginning of the third to last round (respectively 30 and 8). The attack at round 29 however is significantly slower. Similarly to the SSAES, this is due to an over-propagation of the fault, which increases the complexity of solving, and further constrains the key space. The runtimes range from slightly less than four hours, to more than eight days. For a cipher of the size of LED (i.e. 64 bits), this is a rather long runtime. However, a designer who would want to evaluate his LED implementation against fault attacks could run attacks on both locations in parallel for a few days. He would then find that round 30 needs to be protected, but round 29 should not be discarded, even if the complexity of the

attack is higher. This once again showcases the capacity of `AutoFault` to automatically find attacks, which may have been discarded otherwise.

4.6.3 PRESENT

PRESENT [BKL+07] is another widely used and studied lightweight cipher. It differs from the AES and LED by not having *ShiftRows* and *MixColumns* operations for its diffusion layer, but rather a remapping of each bits constituting the internal state after the SBoxes. Another difference is the key size. While the internal state is 64-bit wide, the master key is 80-bit long, and 64-bit round keys are derived from it. This makes a fault attack slightly harder, as a faulty value may not depend on the complete master key. Therefore, conventional DFA require at least two fault injections at round 29 to recover the secret key.

Table 4.8: Number of Faults Comparison for the PRESENT Cipher

	PRESENT		PRESENT	
#Faults	5		10	
#Attacks	20		10000	
	Time in s	#KC	Time in s	#KC
Average	11184.08	117514	75.84	963
Median	1020.88	11022	1.64	14
Minimum	5.98	59	0.36	1
Maximum	78027	929926	56041.90	605168

Fault injections at round 29 with varying number of faults were considered with the framework. The results for five and ten faults are available in Table 4.8. Once again the circuit was truncated from round 29 to 32 in order to reduce the number of CNF clauses to process and improve the solving times. It can be seen that, for five fault injections, the runtimes widely vary and for this reason only twenty attacks were ran, and successful. A timeout of one day (i.e. 86400 seconds) was set in `AutoFault`, and a few more attacks ran into the timeout limit (roughly one third of the attacks), and results are only shown for successful attacks. To a smaller extent, the same high variance is true for the attacks with ten injected faults, and especially the maximum runtimes, and the numbers of key candidates, are high. This is due to the fault propagation pattern inside PRESENT, which can often overlap because of the bitwise remapping, but also due to the larger master key

size, as well as the random fault positions considered by `AutoFault`. This leads, in some cases, to a key space which is only slightly constrained, and that is why, compared to more conventional attacks, the framework needs more fault injections on average to mount a successful attack. If the fault model was restricted further, and faults were injected at specific positions, such that the fault propagation is maximal, with as little overlap as possible, then fewer faults would be required and the runtimes would be lower. This is for example the case in both minimum entries of Table 4.8, and thanks to the framework's logs, such cases can be once again automatically found by `AutoFault`.

4.6.4 Extension to other Types of Ciphers

The results for attacks on SPN ciphers show the versatility of the `AutoFault` framework for attack hardware implementation of cryptographic primitives, but also how it can be used to automatically find the most suitable location and position of faults, in order to mount efficient attacks. Moreover, the automated process can be leveraged by a designer to verify that, even if he is aware of a critical location and/or position to protect with countermeasures, thanks to an already well known attack, no other locations are vulnerable. One last example fo this, and another extension of the framework, is the attack on the Midori cipher [BBI+15] presented in Table 4.9, showing that fault injection attacks at the beginning of the seventeenth round leads to a successful attack, but fault injections a round later also lead to successful attacks in most cases (out of 1000 attacks, 10 lead to timeouts, due to the large size of the remaining key space for some fault positions).

Table 4.9: Fault Location Comparison for the Midori Cipher with 5 Fault Injections

	Midori128		Midori128	
#Attacks	1000		990	
Fault Location	Round 17		Round 18	
	Time in s	#KC	Time in s	#KC
Average	539.49	1	114.13	1554
Median	42.50	1	2.49	2
Minimum	2.55	1	0.98	1
Maximum	121880.00	1	20593.30	277608

`AutoFault` is however not limited to SPN ciphers. SPN ciphers were studied in more detail due to their usage in a wide variety of applications, but other types of ciphers can

be attacked as well. In general, `AutoFault` can process any cipher, as long as it is given in the proper format and with a suitable fault model. Other cipher types, such as Feistel networks or elliptic curve cryptosystems can be processed as well. In the case of Feistel networks (such as SIMON [BTCS+15]), a designer would have to modify the time-frame expansion script to add faults in two different rounds. This is due to the nature of Feistel networks, which process the data partitioned into two blocks (the right and left blocks) of half the size of the data. In each round, each block is either processed or directly forwarded to the next round, becoming the next right (or left respectively) block. This means that the secret key only affects one block at a time, and a fault does not propagate to the second block, leading to only half of the secret key being constrained. This can be circumvented by injected a second fault in the second block, or at a previous round, for better fault propagation, and can be realised by the time-frame expansion script.

Another example of extension of the framework could be elliptic curve cryptosystems. These ciphers are asymmetric block ciphers, meaning that the secret key is only used during the decryption process. Therefore, a designer should consider the decryption process and not the encryption, otherwise the attack in itself would be similar. However, elliptic curve cryptosystems have a large number of operations and are usually implemented sequentially for this reason. While a time-frame expansion can be realised, the time-frame expanded circuit may become extremely large and thus difficult to process for the SAT solver, due to the large number of CNF clauses, and the user should be aware of this.

4.7 Multiple Faults effect in AutoFault

Support for multiple fault injections was added to the `AutoFault` Framework in [GPU+19]. While multiple fault injections allowed new attacks to be mounted, the question emerged about how many fault injections should be performed. For more conventional fault attacks, such as DFA, more fault injections are often better runtime-wise, as each additional fault restricts the key space further. There are diminishing returns which come with more fault injections, but for fault injection attacks such as DFA, more faults injections generally either decrease the solving times, or do not impact them.

In the context of hardware-oriented AFAs, however, injecting more faults does not only add more inputs, but also a complete branch of the circuit is duplicated from the fault location and onwards (Section 4.3.1). In turn, this leads to more CNF clauses being generated and processed by the SAT solver. More faults still restrict the key space further, but in the case of an already constrained key space, an additional fault may not reduce significantly the remaining number of key candidates, while adding more CNF clauses to be processed, and consequently increasing the runtime necessary for a successful attack. This can be seen in Table 4.10, which presents results on attacks performed on the SSAES $SR^*(4,4,4,4)$ and the LED cipher, both truncated for optimisation purposes and over 10000 attacks.

In the case of the SSAES, the fault was injected at the beginning of the eighth round, as it was determined to be the most efficient fault location. Similarly, for the LED cipher, the fault was injected at the beginning of the thirtieth round, and for both ciphers, the truncation was performed at the same respective locations. Table 4.10 shows the significant improvement from injecting two faults instead of a single one. For the AES, the average solving time reduces by a factor 60, while the attack on LED is more than two hundred times faster. Similarly, the number of key candidates is almost always reduced to a single one, or at the least a low number of candidates for the maximum cases (such as 83 for the LED cipher). Even though the results are to be expected, it shows the importance of supporting multiple fault injections for the evaluation of hardware implementation of cryptographic schemes, as it drastically reduces the solving times, and thus allows for faster identification of critical fault locations. Moreover, for the full scale AES, a single fault injection does not restrict the key space enough to allow a attack to be mounted in a reasonable time with the framework. Over several experiments with a single fault injection on the AES, only a single instance was successful in approximately sixteen days of runtime, while attacks with two faults take on average around 400 seconds (as can be seen in Table 4.6). This fact strengthens further the need for multiple fault injections support for `AutoFault`, and AFA frameworks in general, especially for the evaluation of larger encryption schemes.

However, one can also see in the table that increasing the number of fault injection further than two, for those specific ciphers, leads to an increase in solving times. This is due to the fact that the number of remaining key candidates is already extremely low for two

Table 4.10: Impact of Multiple Fault Injections on Runtimes - 10000 Attacks

Cipher	#Faults	Average		Median		Minimum		Maximum		#CNF Clauses
		Time in s	#KC	Time in s	#KC	Time in s	#KC	Time in s	#KC	
SSAES $SR^*(4,4,4,4)$	1	205.82	2009	33.21	492	1.23	3	42786.31	136946	13306
	2	3.29	1	2.84	1	1.69	1	14.53	2	19550
	3	3.75	1	3.10	1	1.75	1	17.10	1	25783
	5	5.51	1	4.43	1	2.23	1	33.76	1	39610
	10	8.72	1	8.10	1	3.07	1	28.25	1	71851
LED64	1	552.23	8768	165.42	2861	2.94	4	62940.70	636238	15502
	2	3.96	1	3.91	1	2.62	1	9.39	83	23371
	3	4.31	1	4.26	1	2.96	1	8.99	6	31240
	5	5.47	1	5.34	1	3.59	1	12.94	1	46203
	10	8.08	1	7.70	1	4.88	1	16.09	1	84933

fault injections, meaning that adding further faults would only avoid the evaluation of less than a hundred of key candidates, and since the evaluation is performed quickly, this does not significantly reduce the overall runtimes. However, since more CNF clauses are added to the SAT instance for more fault injections, as can be seen in the table as well (from ∼15.5k clauses to almost 85k for the LED), the instance is harder for the SAT solver. Consequently, adding more faults beyond the point where the key space is reduced to only a few key candidates is not only ineffective attack-wise, but also detrimental solving time-wise.

Overall, more fault injections implies both a restriction of the key space and an increase in the number of CNF clauses. Therefore, the user should cautiously select the number of fault injections to be performed. A generic method for the automatic evaluation of cipher's hardware implementation could be to first automatically mount attacks with a varying number of fault injections, and then compare the numbers of key candidates considered by `AutoFault`. The number of faults to be considered should then be the lowest number for which there were few key candidates. The design can then be evaluated with this number of fault injections, but also with a slightly lower number of fault and for an extended period of time, to verify if the attack is still mountable in those cases. In any case, the design is vulnerable at this specific, but depending on the hardware constraints, and the desired security level, if too many fault injections are required, the attack can be deemed unpractical. Finally, from an attacker point of view, the lowest realistic number of fault injections which lead to a small key space should always be considered to reduce the overhead of CNF clauses.

4.8 Future work: Counter-Measure Validation

The `AutoFault` framework can be used to evaluate the effectiveness of counter-measures against fault attacks, or if counter-measures against other types of attacks (e.g. masking against side-channel analysis) have a negative effect on AFAs mounted against the cipher. This section discusses the possible applications of `AutoFault` for automated evaluation of counter-measure, as well as a few preliminary results.

The first counter-measures of interest for evaluation with the `AutoFault` framework are Error Detecting Codes (EDCs) and Error Correcting Codes (ECCs) (Section 2.2.1). Code based counter-measure, such as the one presented in Chapter 3 are aimed directly at preventing successful fault injection attacks. Consequently, their capabilities in that regard needs to be evaluated, and the automated capacity of `AutoFault` to mount attacks on cryptographic circuit can be leveraged for that purpose. Thanks to the versatility of the time-frame expansion step of the framework, circuits previously found vulnerable and which have been improved to include a code-based counter-measure can easily be tested further. In most cases, for HDL implementation, only the new ECC module needs to be included, as well as the new component protected by the code. The remainder of the steps are the same for an evaluation against random fault models. The RK code architecture presented in Section 3.3 was added to the Midori cipher and tested against random nibble faults injected at round seventeen. This resulted in timeouts of `AutoFault`, even when the timeout was set to a full day. The framework was able to try several hundreds of thousands of key candidates before timeout, however, since the RK architecture is always able to correct such faults, the ciphertexts after the fault injections were not faulty anymore, and the key space was not restricted, rendering the attack computationally infeasible.

This result suggests that most fault injection attacks are therefore indeed mitigated by the addition of an RK code architecture. They are fully eliminated for nibble faults, but faults of larger multiplicities can be detected, even if they can not be corrected. However, as discussed in Section 3.2, evaluating ECC against randomly generated faults is not sufficient, and worst case scenarios need to be considered as well. The `AutoFault` framework can also be used for this purpose by adding new CNF clauses to restrict the fault model to only masked faults. If the framework can automatically find such a fault in a reasonable amount of time, and subsequently mount a successful, then the cryptographic circuit is still vulnerable to fault attacks and the ECC counter-measure is not sufficient. Let us denote by $C = \{C_i\}$ and $C' = \{C'_i\}$ the sets of CNF variables corresponding to the fault free and faulty ciphertext respectively, where i refers to the related bit. Similarly, $E = \{E_i\}$ and $E' = \{E'_i\}$ are the sets of CNF variables relating to the output of intermediate ECC module (respectively fault free and faulty). Let us define c_i, such that $c_i = C_i$ when the bit of the fault free ciphertext is 1, and $c_i = \neg C_i$ when it is 0, and we define c'_i respectively

for the faulty ciphertext. Adding the set of CNF clauses from Equation 4.7 to the already available time-frame expanded circuit and fault description's sets of clauses during solving will allow for the evaluation of the ECC module in the worst case scenario of masked faults.

$$(c_i) \wedge (c_i') \wedge ((\neg E_i \vee E_i') \wedge (E_i \vee \neg E_i')) \tag{4.7}$$

In more details, by proceeding this way, `AutoFault` can find secret key assignments which are consistent with the simulated (or observed) ciphertexts, and fault assignments which are equal between both the fault free and faulty branch of the time-frame expanded circuit (i.e. masked faults). As of time of writing, this feature has not been added to `AutoFault` and is planned as future work.

Another use case for which `AutoFault` can be used, is the evaluation of nonce-based cryptosystem against fault attacks. In principle, such ciphers are resilient to fault injection attacks which require several fault injections with same input values (see Section 2.2.2.3). In the context of this thesis, the authenticated cipher ASCON [DEMS16] was considered. Attack were mounted on the last rounds of ASCON, first by truncating the cipher to only those specific rounds, as an optimisation step. Furthermore, as an initial step in attacking the encryption scheme, the nonce was considered known. Even so, and at the time of writing, no attack were successful on ASCON with known nonce, and only a reduced version of the cipher was successfully attacked.

Nevertheless, the framework can be used to evaluate nonce-based cipher such as ASCON by considering the nonce either fully known, partially known or unknown. In the former case, an attack on the nonce-based cipher would be equivalent to an attack on ciphers without any nonce. If the nonce is partially known or completely unknown, the SAT solver will have to (at least partially in the former case) find a satisfiable assignment for the nonce as well, which makes the attack more complex. If `AutoFault` still manages to mount a successful attack given such a nonce model, then the hardware implementation of the nonce-based cipher needs to be refined, or other counter-measures added. Another common consideration for fault attacks on nonce-based ciphers is the misuse of the nonce. For instance, an attacker may be able to encrypt multiple plaintexts with the same nonce reused for each encryption. This case can also be automatically investigated in `AutoFault`

by modifying slightly the time frame-expansion script to forward the nonce to each fault branch, instead of using a new nonce per branch. The remainder of the attack would be the same.

One last example of automated counter-measure evaluation with the `AutoFault` framework, is a masking scheme implemented as a counter-measure to side-channel attacks. While masking is not aimed at circumventing fault injection attacks, the implementation of a masking scheme may have a negative effect on the resilience of the cipher to this type of attacks. Usually, for masked implementations, the overall structure of the encryption remains the same, and only the operations are modified to account for the XOR with the randomness needed for masking (Section 2.2.2.2). Therefore, either no modification are required in `AutoFault` for the protected cipher to be evaluated, or only slight changes in the time-frame expansion will be required.

A Domain-Oriented Masking (DOM) [GMK16] AES implementation, with three shares, was considered for evaluation with `AutoFault`. However, at the time of writing, no complete automatic attack against this implementation have been realised. Nevertheless, the circuit was simulated with `AutoFault` and a few faulty ciphertexts were manually saved for evaluation with an unprotected AES circuit. In more detail, a fault was injected in one of the shares at the beginning of the eighth round, and the corresponding ciphertext were recorded, before being given as input to `AutoFault`, on an (unprotected) AES circuit. Such an attack was successful, which is unsurprising due to the nature of masking and how intermediate values are recombined to generate the final ciphertext (all shares are XORed together, and hence the fault propagates the same way as for an unmasked AES). This is a promising preliminary result on the attack of masked circuits with the `AutoFault` framework, as in theory the attack should be successful on the DOMAES circuit as well, given the similar fault propagation patterns. However, the larger size of the circuit (due to the sharing) increases the number of CNF clauses, making the instance harder for the SAT solver, and a complete attack of the DOMAES implementation should thus still be performed, which is planned as future work.

In order to automatically evaluate protected ciphers as discussed above, without the need for introducing new variables for each case, a set of generic *helper* variables was added to `AutoFault` (a full list of the currently used variables is given in Section 4.3.2). These

helper variables can be used for different purposes, depending on the designer needs and the evaluation parameters. For instance, the helper pins can be mapped to the randomness needed for the resharing step of a DOM scheme. The random masks can then be considered either known or unknown, or even forwarded to multiple fault branches in the time-frame expansion, to simulate a broken RNG. Similarly, the helper variables can be used for the nonce of an authenticated cipher, or the intermediate ECC module output (for the restriction to masked faults). While the generic *helper* pins should be set according to the cipher and chosen carefully by the user, they allow to automatically evaluate not only counter-measure, but even other modifications to the cryptographic hardware with ease.

4.9 Comparison to other State-of-the-Art Algebraic Fault Attack Frameworks

The advantages of using the `AutoFault` framework have been described throughout this chapter and more precisely the capacity of the framework to automate the evaluation of sensitive cryptographic hardware implementations against fault attacks. However, other AFA frameworks exist, and it is important to compare `AutoFault` to other state-of-the-art AFA frameworks in order to see the benefits of using `AutoFault`. The frameworks discussed in Section 2.1.5.2 are, to the best of the author's knowledge, and at the time of writing, the most recent AFA frameworks available. Table 4.11 gives an overall comparison between all the frameworks and their capabilities.

First and foremost, it should be noted that the XFC framework, as well as the one from [SKMD17], are not fully fledged AFA solvers in the sense that they can not automatically recover for the secret key, but they rather study the fault propagation patterns throughout an encryption, leading to the potential construction of attacks (as discussed in Section 2.1.5.2). Therefore, both first frameworks in the table, while allowing new attacks to be mounted, can not be directly compared with the remainder of the frameworks, which automatically mount attacks, given the correct inputs. Nonetheless, both frameworks can be used for the generation of AFA on several ciphers, and should as such not be discarded.

Table 4.11: Comparison of `AutoFault` to other State-of-the-Art AFA Frameworks

AFA Framework	AFA Solver	Cipher Description	Multiple Faults Support Reported	Evaluation of Counter-Measures	Supported Ciphers
XFC [KRH17]	No	Functional	No	No	AES, CLEFIA, SMS4
Saha et al. [SKMD17]	No	Functional	No	No	AES, PRESENT
Zhao et al. [ZGZ+13]	Yes	Functional	No	No	LED
Zhang et al. [ZZG+13] [ZGZ+16]	Yes	Functional	Yes	No	Piccolo, AES, DES, MIBS-64, LED, PRESENT, Twofish
`AutoFault`	Yes	Hardware-Oriented	Yes	Yes	SSAES & AES, LED, PRESENT, Midori, ASCON, SIMON, DOMAES (partial)

The third column of Table 4.11 shows the type of input supported by each framework. All frameworks, with the exception of `AutoFault`, take functional description of the ciphers as input. While functional descriptions allow for faster solving compared to the hardware-based description taken by `AutoFault`, they need to be manually generated for each individual cipher. Consequently, and even though conversion methods exist (for example from ANF to CNF), the user is still required to manually check and derive the input of the frameworks, making the automatic evaluation of different encryption schemes more difficult compared to using the `AutoFault` framework. It should also be noted that `AutoFault` can take functional descriptions of the ciphers, or any other additional equations, as input. The focus on hardware-based descriptions is a choice for the framework, as it allows for more flexibility on the input, as well as for the evaluation of sensitive implementations. Concerning the latter, the hardware-oriented nature of `AutoFault` allows for a utilisation of the framework at different stages of the design phase, which is impossible with only functional description of an encryption scheme, and thus only `AutoFault` can be used in an automated fashion for the evaluation of cryptographic hardware.

Another point of comparison for AFA frameworks is the support for multiple fault injections. As Section 4.7 showed, multiple fault injections are not only a major benefit in terms of solving times, but also a necessary functionality to enable attacks on more complex ciphers. In this regard, only `AutoFault` and the framework proposed by Zhang et al. [ZGZ+16] support this feature, which also manifests itself in a larger number of supported ciphers for both frameworks.

The last point of comparison between `AutoFault` and other frameworks is the possibility to evaluate counter-measures via the framework itself. Once again, due to the nature of the input of other AFA frameworks, the automated evaluation of counter-measures is difficult at best. One would need to add the functional description of the counter-measure to the cipher description, and while this is possible, any change in the counter-measure, or any additional counter-measure against other type of attacks, would mean regenerating, at least partially, the functional description. From a designer's perspective, this means a substantial effort needs to be invested to achieve this. `AutoFault` can however be used for this purpose with minimal, if any, modification of the framework (as discussed in Section 4.8), and is as such the only AFA framework capable of automatically evaluating counter-measures.

Overall, the main advantage of the `AutoFault` framework compared to other state-of-the-art AFA frameworks is the increased level of automation for the evaluation of ciphers, and more especially their hardware implementations. The framework is capable of attacking encryption schemes at multiple stages of the design phase, as opposed to only perform attacks at the theoretical level.

Chapter 5

Conclusion

Fault attacks are an ever growing threat for cryptographic circuits, with new attacks vectors discovered regularly for every cipher implementation. The work presented in this thesis proposes two different methods to automatically protect sensitive hardware implementations against fault injection attacks.

On one hand, code-based counter-measures were presented to circumvent the effect of fault injections. In a cryptographic context, protecting the hardware against precise fault injections which may go through counter-measures undetected is especially important, as a strongly capable attacker can take advantage of such vulnerabilities. In this regard, the proposed RK architectures are based on robust codes, which are more efficient in the worst case scenario of masked faults compared to linear codes. The architectures retain the detection and correction capabilities of the linear code they are based of, ensuring the detection of natural faults, and the correction of precise, low multiplicity, maliciously injected faults. They were constructed efficiently thanks to the ECLT decoding, as well as the overall architecture of the counter-measure. More precisely, the ECLT based decoding allows for a compact hardware implementation, significantly reducing hardware costs compared to other widely used decoding techniques. Moreover, by design, they are scalable to any cipher, as a different alphabet can be chosen and larger ciphers can be protected by smaller ECC modules, for more efficient operations. The inner-outer code architectures allow for a higher protection against miscorrection, while being scalable as well. This is especially true in the case where a CPC is used as the outer code. The CPC provides a greater granular control over the trade-off between the security level, and the

hardware cost of the counter-measure. Given an encryption scheme and a desired security level (for example in term of distance), an architecture can be automatically chosen, and then implemented. In addition, the experimental results on the proposed architecture showcased its overall effectiveness against fault injections, as well as its relatively low hardware cost in comparison to other ECC, which do not account for the critical case of masked faults.

On the other hand, a tool for the evaluation of hardware implementation of ciphers was presented, in the form of the `AutoFault` framework. The main idea behind the automation of fault attacks originates from the fact that fault equations are different for any cipher, and even for any slight difference in a cipher family, while being work intensive to produce. The case of the SSAES, for which DFA equations were given, perfectly illustrates the need for automation, and was the first step toward hardware-oriented AFAs. `AutoFault` is the resulting framework of this work on automated fault injection attacks. It is capable of automatically mounting AFAs at different stages of the design phase, which allows for the detection of the potential vulnerabilities of the encryption scheme even when the design has been optimised, or if counter-measures have been added. More precisely, the time-frame expansion can be performed for any cipher, and the support for multiple fault injections enables more complex attacks to be mounted. Adding only a second fault injection to the AFA against a full scale AES lead to a successful key recovery, and drastically reduced the solving times of attacks against other ciphers. The limitation of adding more faults, as well as the trade-off between the number of faults and the number of CNF clauses, was likewise described. Moreover, the introduction of CNF-based simulation further automates the evaluation process. The framework was capable to mount attacks for a wide variety of commonly used SPN ciphers, and can be easily extended to any other type of cryptographic scheme. Compared to other AFA frameworks, successful attacks may be slower in some cases, due to the hardware nature of the framework inputs, but remain reasonably low. `AutoFault` also features more functionalities compared to other frameworks, such as the evaluation of counter-measures, and is the only hardware-oriented AFA framework, which can be used during the design phases of a cryptographic circuit.

Future Work

Both of the previous methods can be used to automatically protect cryptographic hardware against fault injection attacks. However, further work is still needed on both ECC counter-measures and the `AutoFault` framework.

In the case of the RK architectures, while they can protect cryptographic hardware against fault injection attacks, the introduction of new components processing sensitive data, mainly the secret key, can induce other sources of leakage. The resilience of the RK architecture against passive side-channel attacks (e.g. DPA) needs to be considered as well, which was not the case so far in this work. Such attacks remain a threat even in the presence of other counter-measures, if proper care to design the cryptographic circuit is not taken. For instance, the cryptographic components of the circuit may be designed to be resilient against side-channel analysis, but the predictors used by the ECC architecture may leak sensitive information nonetheless, as they are implemented differently. One possibility to deal with potential leakage is the careful combination of ECC with masking, such as the IPM-RED architecture proposed in [KP21]. The IPM-RED architecture is not built on RK codes. However, it should be possible to use an RK architecture described in this thesis in conjunction with inner product masking. This would increase the security level provided by the RK architecture, but the implementation costs need to be considered and the implementation itself needs to be tested against leakage on physical hardware.

Concerning the `AutoFault` framework, while the evaluation of counter-measures has been discussed in this thesis (Section 4.8). It is still in its early stage and practical evaluations of encryption scheme's hardware implementations remain to be done, for example fully supporting the DOMAES. Furthermore, the framework can be extended to support a wider variety of fault models and attacks in general, to improve the capacity of the framework to automatically evaluate cryptographic hardware. For example, non SPN ciphers may require slightly different fault models for an attack to be successful in `AutoFault`. Feistel networks are one such example, where the two faults need to be injected at different rounds in order to mount a successful attack, which has not been tested in `AutoFault` yet. Finally, the framework could also be extended to support other type of hardware, such as neural networks. Recently, the vulnerability of neural networks to fault injection attacks has been considered, and a new attack aiming at reverse engineering them was proposed

[BJH+20]. The construction of attacks on neural networks shares high similarities with attacks on cryptographic hardware, and as such `AutoFault` could be used for this purpose as well. The unrolling and mapping scripts within `AutoFault` need to be adapted to achieve this, and other considerations may be needed to speed up the solving (e.g. on the sparsity of the weights), as the neural networks are usually larger than cryptographic circuits when time-frame expended.

Bibliography

[AARR03] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The em side-channel(s). In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 29–45, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN: 978-3-540-36400-9.

[ABC+17] Stéphanie Anceau, Pierre Bleuet, Jessy Clédière, Laurent Maingault, Jean-luc Rainard, and Rémi Tucoulou. Nanofocused x-ray beam to reprogram secure circuits. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 175–188, Cham, 2017. Springer International Publishing.

[ADM+10] Michel Agoyan, Jean-Max Dutertre, Amir-Pasha Mirbaha, David Naccache, Anne-Lise Ribotta, and Assia Tria. How to flip a bit? In *16th Int. On-Line Testing Symposium (IOLTS)*, pages 235–239, Piscataway, NJ, USA, 2010. IEEE. doi:10.1109/IOLTS.2010.5560194.

[ADY15] R. Altawy, O. Duman, and A. Youssef. Fault analysis of kuznyechik. *IACR Cryptol. ePrint Arch.*, 2015:347, 2015.

[AG01] Mehdi-Laurent Akkar and Christophe Giraud. An implementation of des and aes, secure against some attacks. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, pages 309–318, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[ALK12] N. Admaty, S. Litsyn, and O. Keren. Puncturing, expurgating and expanding the q-ary BCH based robust codes. In *IEEE Conv. of Elec. & Electronics Engineers in Israel*, pages 1–5, 2012. doi:10.1109/EEEI.2012.6376995.

[AS18] Gilles Audemard and Laurent Simon. On the glucose sat solver. *International Journal on Artificial Intelligence Tools*, 27(01):1840001, 2018.

[BBI+15] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A block cipher for low energy. In *ASIACRYPT (2)*, volume 9453 of *Lecture Notes in Computer Science*, pages 411–436. Springer, 2015.

[BBK+03] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri. Error analysis and detection procedures for a hardware implementation of the advanced encryption standard. *IEEE Transactions on Computers*, 52(4):492–505, 2003. doi:10.1109/TC.2003.1190590.

[BBKN12] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault injection attacks on cryptographic devices: Theory, practice and countermeasures. *Proc. IEEE*, pages 3056–3076, 2012.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, pages 16–29, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN: 978-3-540-28632-5.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 37–51, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[BDL01] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *J. Cryptology*, 14(2):101–119, 2001.

[BECN+06] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[BEG13] Nasour Bagheri, Reza Ebrahimpour, and Navid Ghaedi. New differential fault analysis on present. *EURASIP Journal on Advances in Signal Processing*, 2013(1):1–10, 2013.

[Ber84] E. R. Berlekamp. *Algebraic coding theory*. Aegean Park Press, Laguna Hills, CA, USA, 1984.

[BG13] Alberto Battistello and Christophe Giraud. Fault analysis of infective AES computations. In *FDTC*, pages 101–107. IEEE Computer Society, 2013.

[BGN+14] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Higher-order threshold implementations. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 326–343. Springer, 2014.

[Bie17] Armin Biere. Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018. *Proceedings of SAT Competition*, pages 14–15, 2017.

[BJH+20] Jakub Breier, Dirmanto Jap, Xiaolu Hou, Shivam Bhasin, and Yang Liu. Sniff: reverse engineering of neural networks with fault attacks. *arXiv preprint arXiv:2002.11021*, 2020.

[BKHL20] J. Breier, M. Khairallah, X. Hou, and Y. Liu. A countermeasure against statistical ineffective fault analysis. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(12):3322–3326, 2020. doi:10.1109/TCSII.2020.2989184.

[BKL+07] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: an ultra-lightweight block cipher. In *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[BRC60] R.C. Bose and D.K. Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 3(1):68–79, 1960. doi:https://doi.org/10.1016/S0019-9958(60)90287-4.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, pages 513–525, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[BS03] Johannes Blömer and Jean-Pierre Seifert. Fault based cryptanalysis of the advanced encryption standard (aes). In Rebecca N. Wright, editor, *Financial Cryptography*, pages 162–181, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[BTCS+15] Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith, and Louis Wingers. The simon and speck lightweight block ciphers. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015.

[Can05] David Canright. A very compact s-box for AES. In *CHES*, pages 441–455, 2005. doi:10.1007/11545262_32.

[CDF+08] Ronald Cramer, Yevgeniy Dodis, Serge Fehr, Carles Padró, and Daniel Wichs. Detection of algebraic manipulation with applications to robust secret sharing and fuzzy extractors. In Nigel Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, pages 471–488, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 398–412, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[CJW10] N. T. Courtois, K. Jackson, and D. Ware. Fault-algebraic attacks on inner rounds of DES. In *European Smart Card Security Conf.*, 2010.

[Cla07] Christophe Clavier. Secret external encodings do not prevent transient fault analysis. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 181–194, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[CMR05] C. Cid, S. Murphy, and M. J. B. Robshaw. *Small Scale Variants of the AES*, pages 145–162. Springer Berlin Heidelberg, 2005. ISBN: 978-3-540-31669-5.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 13–28, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN: 978-3-540-36400-9.

[CZS16] Wei Cheng, Yongbin Zhou, and Laurent Sauvage. Differential fault analysis on midori. In Kwok-Yan Lam, Chi-Hung Chi, and Sihan Qing, editors, *Information and Communications Security*, pages 307–317, Cham, 2016. Springer International Publishing.

[DCBR+15] Thomas De Cnudde, Begül Bilgin, Oscar Reparaz, Ventzislav Nikov, and Svetla Nikova. Higher-order threshold implementation of the aes s-box. In *International conference on smart card research and advanced applications*, pages 259–272. Springer, 2015.

[DEK+18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. Sifa: Exploiting ineffective fault inductions on symmetric cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):547–572, Aug. 2018. doi:10.13154/tches.v2018.i3.547-572.

[DEMS16] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. *Submission to the CAESAR Competition*, 2016.

[DFL11] Patrick Derbez, Pierre-Alain Fouque, and Delphine Leresteux. Meet-in-the-middle and impossible differential fault analysis on aes. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 274–291, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[DLV03] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential fault analysis on a.e.s. In Jianying Zhou, Moti Yung, and Yongfei Han, editors, *Applied Cryptography and Network Security*, pages 293–306, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[DMN+17] Debayan Das, Shovan Maity, Saad Bin Nasir, Santosh Ghosh, Arijit Raychowdhury, and Shreyas Sen. High efficiency power side-channel attack immunity using noise injection in attenuated signature domain. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 62–67, 2017. doi:10.1109/HST.2017.7951799.

[DMN+18] Debayan Das, Shovan Maity, Saad Bin Nasir, Santosh Ghosh, Arijit Raychowdhury, and Shreyas Sen. Asni: Attenuated signature noise injection for low-overhead power

side-channel attack immunity. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(10):3300–3311, 2018. doi:10.1109/TCSI.2018.2819499.

[DPW18] Stefan Dziembowski, Krzysztof Pietrzak, and Daniel Wichs. Non-malleable codes. *Journal of the ACM,*, 65(4), April 2018.

[Gir05] Christophe Giraud. Dfa on aes. In Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa, editors, *Advanced Encryption Standard – AES*, pages 27–41, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[GMK16] Hannes Gross, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In *Theory of Implementation Security*, TIS '16, page 3, New York, NY, USA, 2016. Association for Computing Machinery. ISBN: 9781450345750.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, pages 251–261, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN: 978-3-540-44709-2.

[GPPR11] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The led block cipher. In *CHES*, pages 326–341. Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-23950-2.

[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)*, pages 252–261, 2020. doi:10.1109/HOST45689.2020.9300260.

[GYS15] Nahid Farhady Ghalaty, Bilgiday Yuce, and Patrick Schaumont. Differential fault intensity analysis on present and led block ciphers. In Stefan Mangard and Axel Y. Poschmann, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 174–188, Cham, 2015. Springer International Publishing.

[GYTS14] Nahid Farhady Ghalaty, Bilgiday Yuce, Mostafa M. I. Taha, and Patrick Schaumont. Differential fault intensity analysis. In *FDTC*, pages 49–58. IEEE Computer Society, 2014.

[Ham50] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950.

[HHM+13] Y. Hayashi, N. Homma, T. Mizuki, T. Aoki, and H. Sone. Transient iemi threats for cryptographic devices. *IEEE Transactions on Electromagnetic Compatibility*, 55(1):140–148, 2013. doi:10.1109/TEMC.2012.2206393.

[HR08] Michal Hojsík and Bohuslav Rudolf. Differential fault analysis of trivium. In Kaisa Nyberg, editor, *Fast Software Encryption*, pages 158–172, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Jeo12] Ki-Tae Jeong. Differential fault analysis on block cipher piccolo-80. *The Journal of Advanced Navigation Technology*, 16(3):510–517, 2012.

[JKP12] Philipp Jovanovic, Martin Kreuzer, and Ilia Polian. A fault attack on the LED block cipher. In *COSADE*, volume 7275 of *Lecture Notes in Comp. Sc.*, pages 120–134. Springer, 2012.

[JSP20] Amit Jana, Dhiman Saha, and Goutam Paul. Differential fault analysis of norx. In *Workshop on Attacks and Solutions in Hardware Security - ASHES 2020*, ASHES'20, page 67–79, New York, NY, USA, 2020. Association for Computing Machinery. ISBN: 9781450380904.

[KDK+14] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014. doi:10.1109/ISCA.2014.6853210.

[KGT18] Jonas Krautter, Dennis R. E. Gnad, and Mehdi B. Tahoori. Fpgahammer: Remote voltage fault attacks on shared fpgas, suitable for dfa on aes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):44–68, Aug. 2018.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, pages 388–397. Springer, 1999.

[KKT04] M. Karpovsky, K. J. Kulikowski, and A. Taubin. Robust protection against fault-injection attacks on smart cards implementing the advanced encryption standard. In *International Conference on Dependable Systems and Networks, 2004*, pages 93–101, 2004. doi:10.1109/DSN.2004.1311880.

[KKW07] M. Karpovsky, K. Kulikowski, and Z. Wang. Robust error detection in communication and computational channels. In *Int'l Workshop Spectral Methods & Multirate Signal Proc.*, 2007.

[KM10] Lars R. Knudsen and Charlotte V. Miolane. Counting equations in algebraic attacks on block ciphers. *International Journal of Information Security*, 9(2):127–135, Apr 2010.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[KP21] Osnat Keren and Ilia Polian. Ipm-red: combining higher-order masking with robust error detection. *Journal of Cryptographic Engineering*, 11(2):147–160, 2021.

[KQ08] Chong Hee Kim and Jean-Jacques Quisquater. New differential fault analysis on aes key schedule: Two faults are enough. In Gilles Grimaud and François-Xavier Standaert, editors, *Smart Card Research and Advanced Applications*, pages 48–60, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[KRH17] P. Khanna, C. Rebeiro, and A. Hazra. Xfc: A framework for exploitable fault characterization in block ciphers. In *2017 IEEE Design Auto. Conf. (DAC)*, pages 1–6, June 2017. doi:10.1145/3061639.3062340.

[KT04] M. Karpovsky and A. Taubin. New class of nonlinear systematic error detecting codes. *IEEE Transactions on Information Theory*, 50(8):1818–1819, 2004. doi:10.1109/TIT.2004.831844.

[KW14] Mark G. Karpovsky and Zhen Wang. Design of strongly secure communication and computation channels by nonlinear error detecting codes. *IEEE Trans. Comp.*, 63(11):2716–2728, 2014.

[LM06] H Li and S Moore. Security evaluation at design time against optical fault injection attacks. *IEE Proc.-Info. Sec.*, 153(1):3–11, 2006.

[LOS12] Y. Li, K. Ohta, and K. Sakiyama. New fault-based side-channel attack using fault sensitivity. *IEEE Trans. Information Forensics and Security*, 7(1):88–97, 2012.

[LSG+10] Yang Li, Kazuo Sakiyama, Shigeto Gomisawa, Toshinori Fukunaga, Junko Takahashi, and Kazuo Ohta. Fault sensitivity analysis. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 320–334, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Mas69] J. Massey. Shift-register synthesis and bch decoding. *IEEE Transactions on Information Theory*, 15(1):122–127, 1969. doi:10.1109/TIT.1969.1054260.

[MDH+13] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 77–88, 2013. doi:10.1109/FDTC.2013.9.

[MMSS19] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. Glitch-resistant masking revisited: or why proofs in the robust probing model are needed. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):256–292, Feb. 2019. doi:10.13154/tches.v2019.i2.256-292.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007. ISBN: 978-0-387-30857-9.

[MPL+11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the limits: A very compact and a threshold implementation of aes. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 69–88. Springer, 2011.

[MSI16] M. Matsubayashi, A. Satoh, and J. Ishii. Clock glitch generator on sakura-g for fault injection attack against a cryptographic circuit. In *2016 IEEE 5th Global Conference on Consumer Electronics*, pages 1–4, 2016. doi:10.1109/GCCE.2016.7800490.

[Muk09] Debdeep Mukhopadhyay. An improved fault based attack of the advanced encryption standard. In Bart Preneel, editor, *Progress in Cryptology – AFRICACRYPT 2009*, pages 421–434, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[NBD+15] X. T. Ngo, S. Bhasin, J.-L. Danger, S. Guilley, and Z. Najm. Linear complementary dual code improvement to strengthen encoded circuit against hardware Trojan horses. In *IEEE Int'l Symp. on Hardware Oriented Security and Trust*, pages 82–87, 2015.

[NIoSTN01] National Institute of Standards and Technology (NIST). Advanced Encryption Standard. *NIST FIPS PUB 197*, 2001.

[NK14] Yaara Neumeier and Osnat Keren. Robust generalized punctured cubic codes. *IEEE Transactions on Information Theory*, 60(5):2813–2822, 2014.

[O'F16] Colin O'Flynn. Fault injection using crowbars on embedded systems. *IACR Cryptol. ePrint Arch.*, 2016:810, 2016.

[PDL18] Dmytro Petryk, Zoya Dyka, and Peter Langendoerfer. Optical fault injections: a setup comparison. *RESCUE - Interdependent Challenges of Reliability, Security and Quality in Nanoelectronic Systems Design*, 06 2018.

[Phe83] KT Phelps. A combinatorial construction of perfect codes. *SIAM Journal on Algebraic Discrete Methods*, 4(3):398–403, 1983.

[POTSC+20] FE Potestad-Ordóñez, Erica Tena-Sánchez, R Chaves, Manuel Valencia-Barrero, Antonio José Acosta-Jiménez, and Carlos Jesús Jiménez-Fernández. Hamming-code based fault detection design methodology for block ciphers. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2020.

[PQ03] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against spn structures, with application to the aes and khazad. In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, pages 77–88, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[QS01] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In Isabelle Attali and Thomas Jensen, editors, *Smart Card Programming and Security*, pages 200–210, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN: 978-3-540-45418-2.

[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)*, pages 41–50. IEEE, 2019.

[RBIK12] Francesco Regazzoni, Luca Breveglieri, Paolo Ienne, and Israel Koren. Interaction between fault attack countermeasures and the resistance against power analysis attacks. In *Fault Analysis in Cryptography*, pages 257–272. Springer, 2012.

[RE04] Wolfgang Rankl and Wolfgang Effing. *Smart card handbook*. John Wiley & Sons, 2004.

[RK17] Hila Rabii and Osnat Keren. A new construction of minimum distance robust codes. In Ángela I. Barbero, Vitaly Skachek, and Øyvind Ytrehus, editors, *Coding Theory and Applications - 5th International Castle Meeting, ICMCTA 2017, Vihula, Estonia, August 28-31, 2017, Proceedings*, volume 10495 of *Lecture Notes in Computer Science*, pages 272–282. Springer, 2017. doi:10.1007/978-3-319-66278-7_23.

[RN18] Vadim Ryvchin and Alexander Nadel. Maple_lcm_dist_chronobt: Featuring chronological backtracking. *SAT Comp.*, page 29, 2018.

[RNK19] H. Rabii, Y. Neumeier, and O. Keren. High rate robust codes with low implementation complexity. *IEEE Transactions on Dependable and Secure Computing*, 16(3):511–520, 2019. doi:10.1109/TDSC.2018.2816638.

[Rog04] Phillip Rogaway. Nonce-based symmetric encryption. In *International workshop on fast software encryption*, pages 348–358. Springer, 2004.

[RP17] Francesco Regazzoni and Ilia Polian. Counteracting malicious faults in cryptographic circuits. In *European Test Symp.* IEEE, 2017.

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[SBHS16] Bodo Selmke, Stefan Brummer, Johann Heyszl, and Georg Sigl. Precise laser fault injections into 90 nm and 45 nm sram-cells. In Naofumi Homma and Marcel Medwed, editors, *Smart Card Research and Advanced Applications*, pages 193–205, Cham, 2016. Springer International Publishing.

[SBP16] Matthias Sauer, Bernd Becker, and Ilia Polian. Phaeton: A sat-based framework for timing-aware path sensitization. *IEEE Transactions on Computers*, 65(6):1869–1881, 2016. doi:10.1109/TC.2015.2458869.

[Sch08] Werner Schindler. Evaluation criteria for physical random number generators. In *Cryptographic Engineering*, pages 25–54. Springer, 2008.

[SH13] Ling Song and Lei Hu. Differential fault attack on the prince block cipher. In Gildas Avoine and Orhun Kara, editors, *Lightweight Cryptography for Security and Privacy*, pages 43–54, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[SHS16] B. Selmke, J. Heyszl, and G. Sigl. Attack on a DFA protected AES by simultaneous laser fault injections. In *FDTC*, pages 36–46. IEEE Computer Society, 2016.

[SIH+11] Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. Piccolo: an ultra-lightweight blockcipher. In *International workshop on cryptographic hardware and embedded systems*, pages 342–357. Springer, 2011.

[Sil] Silicon Integration Initiative. Si2: Nangate freepdk45 generic open cell library, v1.3. http://www.si2.org/openeda.si2.org/projects/nangatelib.

[SJ05] William Stein and David Joyner. SAGE: System for algebra and geometry experimentation. *ACM SIGSAM Bulletin*, 39(2):61–64, 2005.

[SKMD17] Sayandeep Saha, Ujjawal Kumar, Debdeep Mukhopadhyay, and Pallab Dasgupta. Differential fault analysis automation. *IACR Cryptology ePrint Archive*, 2017:673, 2017.

[SLB10] Tobias Schubert, Matthew Lewis, and Bernd Becker. Antom—solver description. *SAT Race*, 2010.

[SMG16] Tobias Schneider, Amir Moradi, and Tim Güneysu. ParTI - towards combined hardware countermeasures against side-channel and fault-injection attacks. In *CRYPTO*, pages 302–332, 2016. doi:10.1007/978-3-662-53008-5_11.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Sat. Testing, 12th International Conference, SAT 2009, Swansea, UK, June 2009. Proceedings*, pages 244–257, 2009. doi:10.1007/978-3-642-02777-2_24.

[SR16] Tobias Schubert and Sven Reimer. antom. In *https://projects.informatik.uni-freiburg.de/projects/antom*, 2016.

[SSMC17] Akhilesh Siddhanti, Santanu Sarkar, S. Maitra, and A. Chattopadhyay. Differential fault attack on grain v1, acorn v3 and lizard. *IACR Cryptol. ePrint Arch.*, 2017:678, 2017.

[Tar10] Christopher Tarnovsky. Hacking the smartcard chip. *Blackhat DC*, 2010.

[TBM14] Harshal Tupsamudre, Shikha Bisht, and Debdeep Mukhopadhyay. Differential fault analysis on the families of simon and speck ciphers. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 40–48. IEEE, 2014.

[TFY07a] J. Takahashi, T. Fukunaga, and K. Yamakoshi. Dfa mechanism on the aes key schedule. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*, pages 62–74, 2007. doi:10.1109/FDTC.2007.13.

[TFY07b] Junko Takahashi, Toshinori Fukunaga, and Kimihiro Yamakoshi. DFA mechanism on the AES key schedule. In Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and ean-Pierre Seifert, editors, *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2007, FDTC 2007: Vienna, Austria, 10 September 2007*, pages 62–74. IEEE Computer Society, 2007. doi:10.1109/FDTC.2007.4318986.

[TMA11] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. *Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault*, pages 224–233. Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-21040-2.

[TNK+14] V. Tomashevich, Y. Neumeier, R. Kumar, O. Keren, and I. Polian. Protecting cryptographic hardware against malicious attacks by nonlinear robust codes. In *DFT*, pages 40–45, 2014.

[Tse68] G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 1968.

[Vas65] J.L. Vasil'ev. On nongroup close-packed codes, Probl. Kibern., 8 (1962), 337–339. *English translation in Probleme der Kybernetik*, 8:92–95, 1965.

[VN56] John Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies.(AM-34), Volume 34*, pages 43–98. Princeton university press, 1956.

[vWWM11] Jasper G. J. van Woudenberg, Marc F. Witteman, and Federico Menarini. Practical optical fault injection on secure microcontrollers. In *FDTC*, pages 91–99. IEEE Computer Society, 2011.

[WK11] Z. Wang and M. Karpovsky. Algebraic manipulation detection codes and their applications for design of secure cryptographic devices. In *2011 IEEE 17th International On-Line Testing Symposium*, pages 234–239, 2011. doi:10.1109/IOLTS.2011.5994535.

[Wu16] Hongjun Wu. Acorn v3. *Submission to the CAESAR Competition*, 2016.

[WZ11] Wenling Wu and Lei Zhang. Lblock: A lightweight block cipher. In *ACNS*, Lecture Notes in Comp. Sc., pages 327–344. Springer, 2011.

[XjZ11] Shi-ze Guo Xin-jie Zhao, Tao Wang. Fault-propagation pattern based dfa on spn structure block ciphers using bitwise permutation, with application to present and printcipher. Cryptology ePrint Archive, Report 2011/086, 2011. `https://eprint.iacr.org/2011/086`.

[ZGZ+13] Xinjie Zhao, Shize Guo, Fan Zhang, Zhijie Shi, Chujiao Ma, and Tao Wang. Improving and evaluating differential fault analysis on LED with algebraic techniques. In *FDTC*, pages 41–51. IEEE Computer Society, 2013.

[ZGZ+16] Fan Zhang, Shize Guo, Xinjie Zhao, Tao Wang, Jian Yang, François-Xavier Standaert, and Dawu Gu. A framework for the analysis and evaluation of algebraic fault attacks on lightweight block ciphers. *IEEE Trans. Info. Forensics and Sec.*, pages 1039–1054, 2016.

[ZZG+13] Fan Zhang, Xinjie Zhao, Shize Guo, Tao Wang, and Zhijie Shi. Improved algebraic fault analysis: A case study on piccolo and applications to other lightweight block ciphers. In *COSADE*, Lecture Notes in Comp. Sc., pages 62–79. Springer, 2013.

Appendix A

Small Scale AES Differential Fault Equations

Without last *MixColumns*

1. Case $(r, c) = (1, 1)$:

$$\delta_1 = S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) \tag{A.1}$$

2. Case $(r, c) = (1, 2)$:

$$\begin{aligned} \delta_1 &= S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) \\ \delta_2 &= S^{-1}(x_2 \oplus k_2) \oplus S^{-1}(x'_2 \oplus k_2) \end{aligned} \tag{A.2}$$

3. Case $(r, c) = (1, 4)$:

$$\begin{aligned} \delta_1 &= S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) \\ \delta_2 &= S^{-1}(x_2 \oplus k_2) \oplus S^{-1}(x'_2 \oplus k_2) \\ \delta_3 &= S^{-1}(x_3 \oplus k_3) \oplus S^{-1}(x'_3 \oplus k_3) \\ \delta_4 &= S^{-1}(x_4 \oplus k_4) \oplus S^{-1}(x'_4 \oplus k_4) \end{aligned} \tag{A.3}$$

4. Case $(r, c) = (2, 1)$:

$$\begin{aligned} 3\delta_1 &= S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) \\ 2\delta_1 &= S^{-1}(x_2 \oplus k_2) \oplus S^{-1}(x'_2 \oplus k_2) \end{aligned} \tag{A.4}$$

5. Case $(r, c) = (2, 2)$:

$$\begin{aligned} 3\delta_1 &= S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) \quad & 2\delta_2 &= S^{-1}(x_3 \oplus k_3) \oplus S^{-1}(x'_3 \oplus k_3) \\ 2\delta_1 &= S^{-1}(x_4 \oplus k_4) \oplus S^{-1}(x'_4 \oplus k_4) \quad & 3\delta_2 &= S^{-1}(x_2 \oplus k_2) \oplus S^{-1}(x'_2 \oplus k_2) \end{aligned} \tag{A.5}$$

6. Case $(r, c) = (2, 4)$:

   First fault injection in the first element of the state matrix.

$$\begin{aligned} 3\delta_1 &= S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) \quad & 2\delta_2 &= S^{-1}(x_7 \oplus k_7) \oplus S^{-1}(x'_7 \oplus k_7) \\ 2\delta_1 &= S^{-1}(x_8 \oplus k_8) \oplus S^{-1}(x'_8 \oplus k_8) \quad & 3\delta_2 &= S^{-1}(x_6 \oplus k_6) \oplus S^{-1}(x'_6 \oplus k_6) \end{aligned} \tag{A.6}$$

   Second fault injection in the fifth element of the state matrix.

$$\begin{aligned} 3\delta_3 &= S^{-1}(x_5 \oplus k_5) \oplus S^{-1}(x'_5 \oplus k_5) \quad & 2\delta_4 &= S^{-1}(x_3 \oplus k_3) \oplus S^{-1}(x'_3 \oplus k_3) \\ 2\delta_3 &= S^{-1}(x_4 \oplus k_4) \oplus S^{-1}(x'_4 \oplus k_4) \quad & 3\delta_4 &= S^{-1}(x_2 \oplus k_2) \oplus S^{-1}(x'_2 \oplus k_2) \end{aligned} \tag{A.7}$$

7. Case $(r, c) = (4, 1)$:

$$\begin{aligned} 2\delta_1 &= S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) \\ \delta_1 &= S^{-1}(x_2 \oplus k_2) \oplus S^{-1}(x'_2 \oplus k_2) \\ \delta_1 &= S^{-1}(x_3 \oplus k_3) \oplus S^{-1}(x'_3 \oplus k_3) \\ 3\delta_1 &= S^{-1}(x_4 \oplus k_4) \oplus S^{-1}(x'_4 \oplus k_4) \end{aligned} \tag{A.8}$$

8. Case $(r, c) = (4, 2)$:

   First fault injection in the first element of the state matrix.

$$\begin{aligned} 2\delta_1 &= S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) \\ \delta_1 &= S^{-1}(x_6 \oplus k_6) \oplus S^{-1}(x'_6 \oplus k_6) \\ \delta_1 &= S^{-1}(x_3 \oplus k_3) \oplus S^{-1}(x'_3 \oplus k_3) \\ 3\delta_1 &= S^{-1}(x_8 \oplus k_8) \oplus S^{-1}(x'_8 \oplus k_8) \end{aligned} \tag{A.9}$$

   Second fault injection in the fifth element of the state matrix.

$$\begin{aligned} 2\delta_2 &= S^{-1}(x_5 \oplus k_5) \oplus S^{-1}(x'_5 \oplus k_5) \\ \delta_2 &= S^{-1}(x_2 \oplus k_2) \oplus S^{-1}(x'_2 \oplus k_2) \\ \delta_2 &= S^{-1}(x_7 \oplus k_7) \oplus S^{-1}(x'_7 \oplus k_7) \\ 3\delta_2 &= S^{-1}(x_4 \oplus k_4) \oplus S^{-1}(x'_4 \oplus k_4) \end{aligned} \tag{A.10}$$

9. Case $(r, c) = (4, 4)$:

$$\begin{aligned} 2\delta_1 &= S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1) & \delta_2 &= S^{-1}(x_5 \oplus k_5) \oplus S^{-1}(x'_5 \oplus k_5) \\ \delta_1 &= S^{-1}(x_{14} \oplus k_{14}) \oplus S^{-1}(x'_{14} \oplus k_{14}) & \delta_2 &= S^{-1}(x_2 \oplus k_2) \oplus S^{-1}(x'_2 \oplus k_2) \\ \delta_1 &= S^{-1}(x_{11} \oplus k_{11}) \oplus S^{-1}(x'_{11} \oplus k_{11}) & 3\delta_2 &= S^{-1}(x_{15} \oplus k_{15}) \oplus S^{-1}(x'_{15} \oplus k_{15}) \\ 3\delta_1 &= S^{-1}(x_8 \oplus k_8) \oplus S^{-1}(x'_8 \oplus k_8) & 2\delta_2 &= S^{-1}(x_{12} \oplus k_{12}) \oplus S^{-1}(x'_{12} \oplus k_{12}) \\ \\ \delta_3 &= S^{-1}(x_9 \oplus k_9) \oplus S^{-1}(x'_9 \oplus k_9) & 3\delta_4 &= S^{-1}(x_{13} \oplus k_{13}) \oplus S^{-1}(x'_{13} \oplus k_{13}) \\ 3\delta_3 &= S^{-1}(x_6 \oplus k_6) \oplus S^{-1}(x'_6 \oplus k_6) & 2\delta_4 &= S^{-1}(x_{10} \oplus k_{10}) \oplus S^{-1}(x'_{10} \oplus k_{10}) \\ 2\delta_3 &= S^{-1}(x_3 \oplus k_3) \oplus S^{-1}(x'_3 \oplus k_3) & \delta_4 &= S^{-1}(x_7 \oplus k_7) \oplus S^{-1}(x'_7 \oplus k_7) \\ \delta_3 &= S^{-1}(x_{16} \oplus k_{16}) \oplus S^{-1}(x'_{16} \oplus k_{16}) & \delta_4 &= S^{-1}(x_4 \oplus k_4) \oplus S^{-1}(x'_4 \oplus k_4) \end{aligned} \tag{A.11}$$

With last *MixColumns*

1. Case $(r, c) = (1, 1)$, $(r, c) = (1, 2)$ and $(r, c) = (1, 4)$:
   Same equations as previously (no *MixColumns* operation)

2. Case $(r, c) = (2, 1)$:

$$\begin{aligned} 3\delta_1 &= S^{-1}(3(x_1 \oplus k_1) \oplus 2(x_2 \oplus k_2)) \oplus S^{-1}(3(x'_1 \oplus k_1) \oplus 2(x'_2 \oplus k_2)) \\ 2\delta_1 &= S^{-1}(2(x_1 \oplus k_1) \oplus 3(x_2 \oplus k_2)) \oplus S^{-1}(2(x'_1 \oplus k_1) \oplus 3(x'_2 \oplus k_2)) \end{aligned} \tag{A.12}$$

3. Case $(r, c) = (2, 2)$:

$$\begin{aligned} 3\delta_1 &= S^{-1}(3(x_1 \oplus k_1) \oplus 2(x_2 \oplus k_2)) \oplus S^{-1}(3(x'_1 \oplus k_1) \oplus 2(x'_2 \oplus k_2)) \\ 2\delta_1 &= S^{-1}(2(x_3 \oplus k_3) \oplus 3(x_4 \oplus k_4)) \oplus S^{-1}(2(x'_3 \oplus k_3) \oplus 3(x'_4 \oplus k_4)) \\ 2\delta_2 &= S^{-1}(3(x_3 \oplus k_3) \oplus 2(x_4 \oplus k_4)) \oplus S^{-1}(3(x'_3 \oplus k_3) \oplus 2(x'_4 \oplus k_4)) \\ 3\delta_2 &= S^{-1}(2(x_1 \oplus k_1) \oplus 3(x_2 \oplus k_2)) \oplus S^{-1}(2(x'_1 \oplus k_1) \oplus 3(x'_2 \oplus k_2)) \end{aligned} \tag{A.13}$$

4. Case $(r, c) = (2, 4)$:
   First fault injection in the first element of the state matrix.

$$\begin{aligned} 3\delta_1 &= S^{-1}(3(x_1 \oplus k_1) \oplus 2(x_2 \oplus k_2)) \oplus S^{-1}(3(x'_1 \oplus k_1) \oplus 2(x'_2 \oplus k_2)) \\ 2\delta_1 &= S^{-1}(2(x_7 \oplus k_7) \oplus 3(x_8 \oplus k_8)) \oplus S^{-1}(2(x'_7 \oplus k_7) \oplus 3(x'_8 \oplus k_8)) \\ 2\delta_2 &= S^{-1}(3(x_7 \oplus k_7) \oplus 2(x_8 \oplus k_8)) \oplus S^{-1}(3(x'_7 \oplus k_7) \oplus 2(x'_8 \oplus k_8)) \\ 3\delta_2 &= S^{-1}(2(x_5 \oplus k_5) \oplus 3(x_6 \oplus k_6)) \oplus S^{-1}(2(x'_5 \oplus k_5) \oplus 3(x'_6 \oplus k_6)) \end{aligned} \tag{A.14}$$

Second fault injection in the fifth element of the state matrix.

$$
\begin{aligned}
3\delta_3 &= S^{-1}(3(x_5 \oplus k_5) \oplus 2(x_6 \oplus k_6)) \oplus S^{-1}(3(x'_5 \oplus k_5) \oplus 2(x'_6 \oplus k_6)) \\
2\delta_3 &= S^{-1}(2(x_3 \oplus k_3) \oplus 3(x_4 \oplus k_4)) \oplus S^{-1}(2(x'_3 \oplus k_3) \oplus 3(x'_4 \oplus k_4)) \\
\\
2\delta_4 &= S^{-1}(3(x_3 \oplus k_3) \oplus 2(x_4 \oplus k_4)) \oplus S^{-1}(3(x'_3 \oplus k_3) \oplus 2(x'_4 \oplus k_4)) \\
3\delta_4 &= S^{-1}(2(x_1 \oplus k_1) \oplus 3(x_2 \oplus k_2)) \oplus S^{-1}(2(x'_1 \oplus k_1) \oplus 3(x'_2 \oplus k_2))
\end{aligned}
\tag{A.15}
$$

5. Case $(r, c) = (4, 1)$:

$$
\begin{aligned}
2\delta_1 = \oplus & \quad S^{-1}(14(x_1 \oplus k_1) \oplus 11(x_2 \oplus k_2) \oplus 13(x_3 \oplus k_3) \oplus 9(x_4 \oplus k_4)) \\
& \quad S^{-1}(14(x'_1 \oplus k_1) \oplus 11(x'_2 \oplus k_2) \oplus 13(x'_3 \oplus k_3) \oplus 9(x'_4 \oplus k_4)) \\
\delta_1 = \oplus & \quad S^{-1}(9(x_1 \oplus k_1) \oplus 14(x_2 \oplus k_2) \oplus 11(x_3 \oplus k_3) \oplus 13(x_4 \oplus k_4)) \\
& \quad S^{-1}(9(x'_1 \oplus k_1) \oplus 14(x'_2 \oplus k_2) \oplus 11(x'_3 \oplus k_3) \oplus 13(x'_4 \oplus k_4)) \\
\delta_1 = \oplus & \quad S^{-1}(13(x_1 \oplus k_1) \oplus 9(x_2 \oplus k_2) \oplus 14(x_3 \oplus k_3) \oplus 11(x_4 \oplus k_4)) \\
& \quad S^{-1}(13(x'_1 \oplus k_1) \oplus 9(x'_2 \oplus k_2) \oplus 14(x'_3 \oplus k_3) \oplus 11(x'_4 \oplus k_4)) \\
3\delta_1 = \oplus & \quad S^{-1}(11(x_1 \oplus k_1) \oplus 13(x_2 \oplus k_2) \oplus 9(x_3 \oplus k_3) \oplus 14(x_4 \oplus k_4)) \\
& \quad S^{-1}(11(x'_1 \oplus k_1) \oplus 13(x'_2 \oplus k_2) \oplus 9(x'_3 \oplus k_3) \oplus 14(x'_4 \oplus k_4))
\end{aligned}
\tag{A.16}
$$

6. Case $(r, c) = (4, 2)$:

First fault injection in the first element of the state matrix.

$$
\begin{aligned}
2\delta_1 = \oplus & \quad S^{-1}(14(x_1 \oplus k_1) \oplus 11(x_2 \oplus k_2) \oplus 13(x_3 \oplus k_3) \oplus 9(x_4 \oplus k_4)) \\
& \quad S^{-1}(14(x'_1 \oplus k_1) \oplus 11(x'_2 \oplus k_2) \oplus 13(x'_3 \oplus k_3) \oplus 9(x'_4 \oplus k_4)) \\
\delta_1 = \oplus & \quad S^{-1}(9(x_5 \oplus k_5) \oplus 14(x_6 \oplus k_6) \oplus 11(x_7 \oplus k_7) \oplus 13(x_8 \oplus k_8)) \\
& \quad S^{-1}(9(x'_5 \oplus k_5) \oplus 14(x'_6 \oplus k_6) \oplus 11(x'_7 \oplus k_7) \oplus 13(x'_8 \oplus k_8)) \\
\delta_1 = \oplus & \quad S^{-1}(13(x_1 \oplus k_1) \oplus 9(x_2 \oplus k_2) \oplus 14(x_3 \oplus k_3) \oplus 11(x_4 \oplus k_4)) \\
& \quad S^{-1}(13(x'_1 \oplus k_1) \oplus 9(x'_2 \oplus k_2) \oplus 14(x'_3 \oplus k_3) \oplus 11(x'_4 \oplus k_4)) \\
3\delta_1 = \oplus & \quad S^{-1}(11(x_5 \oplus k_5) \oplus 13(x_6 \oplus k_6) \oplus 9(x_7 \oplus k_7) \oplus 14(x_8 \oplus k_8)) \\
& \quad S^{-1}(11(x'_5 \oplus k_5) \oplus 13(x'_6 \oplus k_6) \oplus 9(x'_7 \oplus k_7) \oplus 14(x'_8 \oplus k_8))
\end{aligned}
\tag{A.17}
$$

Second fault injection in the fifth element of the state matrix.

$$
\begin{aligned}
2\delta_2 = \oplus \quad & S^{-1}(14(x_5 \oplus k_5) \oplus 11(x_6 \oplus k_6) \oplus 13(x_7 \oplus k_7) \oplus 9(x_8 \oplus k_8)) \\
& S^{-1}(14(x'_5 \oplus k_5) \oplus 11(x'_6 \oplus k_6) \oplus 13(x'_7 \oplus k_7) \oplus 9(x'_8 \oplus k_8)) \\
\delta_2 = \oplus \quad & S^{-1}(9(x_1 \oplus k_1) \oplus 14(x_2 \oplus k_2) \oplus 11(x_3 \oplus k_3) \oplus 13(x_4 \oplus k_4)) \\
& S^{-1}(9(x'_1 \oplus k_1) \oplus 14(x'_2 \oplus k_2) \oplus 11(x'_3 \oplus k_3) \oplus 13(x'_4 \oplus k_4)) \\
\delta_2 = \oplus \quad & S^{-1}(13(x_5 \oplus k_5) \oplus 9(x_6 \oplus k_6) \oplus 14(x_7 \oplus k_7) \oplus 11(x_8 \oplus k_8)) \\
& S^{-1}(13(x'_5 \oplus k_5) \oplus 9(x'_6 \oplus k_6) \oplus 14(x'_7 \oplus k_7) \oplus 11(x'_8 \oplus k_8)) \\
3\delta_2 = \oplus \quad & S^{-1}(11(x_1 \oplus k_1) \oplus 13(x_2 \oplus k_2) \oplus 9(x_3 \oplus k_3) \oplus 14(x_4 \oplus k_4)) \\
& S^{-1}(11(x'_1 \oplus k_1) \oplus 13(x'_2 \oplus k_2) \oplus 9(x'_3 \oplus k_3) \oplus 14(x'_4 \oplus k_4))
\end{aligned}
\tag{A.18}
$$

7. Case $(r, c) = (4, 4)$:

$$
\begin{aligned}
2\delta_1 = \oplus \quad & S^{-1}(14(x_1 \oplus k_1) \oplus 11(x_2 \oplus k_2) \oplus 13(x_3 \oplus k_3) \oplus 9(x_4 \oplus k_4)) \\
& S^{-1}(14(x'_1 \oplus k_1) \oplus 11(x'_2 \oplus k_2) \oplus 13(x'_3 \oplus k_3) \oplus 9(x'_4 \oplus k_4)) \\
\delta_1 = \oplus \quad & S^{-1}(9(x_{13} \oplus k_{13}) \oplus 14(x_{14} \oplus k_{14}) \oplus 11(x_{15} \oplus k_{15}) \oplus 13(x_{16} \oplus k_{16})) \\
& S^{-1}(9(x'_{13} \oplus k_{13}) \oplus 14(x'_{14} \oplus k_{14}) \oplus 11(x'_{15} \oplus k_{15}) \oplus 13(x'_{16} \oplus k_{16})) \\
\delta_1 = \oplus \quad & S^{-1}(13(x_9 \oplus k_9) \oplus 9(x_{10} \oplus k_{10}) \oplus 14(x_{11} \oplus k_{11}) \oplus 11(x_{12} \oplus k_{12})) \\
& S^{-1}(13(x'_9 \oplus k_9) \oplus 9(x'_{10} \oplus k_{10}) \oplus 14(x'_{11} \oplus k_{11}) \oplus 11(x'_{12} \oplus k_{12})) \\
3\delta_1 = \oplus \quad & S^{-1}(11(x_5 \oplus k_5) \oplus 13(x_6 \oplus k_6) \oplus 9(x_7 \oplus k_7) \oplus 14(x_8 \oplus k_8)) \\
& S^{-1}(11(x'_5 \oplus k_5) \oplus 13(x'_6 \oplus k_6) \oplus 9(x'_7 \oplus k_7) \oplus 14(x'_8 \oplus k_8))
\end{aligned}
$$

$$
\begin{aligned}
\delta_2 = \oplus \quad & S^{-1}(14(x_5 \oplus k_5) \oplus 11(x_6 \oplus k_6) \oplus 13(x_7 \oplus k_7) \oplus 9(x_8 \oplus k_8)) \\
& S^{-1}(14(x'_5 \oplus k_5) \oplus 11(x'_6 \oplus k_6) \oplus 13(x'_7 \oplus k_7) \oplus 9(x'_8 \oplus k_8)) \\
\delta_2 = \oplus \quad & S^{-1}(9(x_1 \oplus k_1) \oplus 14(x_2 \oplus k_2) \oplus 11(x_3 \oplus k_3) \oplus 13(x_4 \oplus k_4)) \\
& S^{-1}(9(x'_1 \oplus k_1) \oplus 14(x'_2 \oplus k_2) \oplus 11(x'_3 \oplus k_3) \oplus 13(x'_4 \oplus k_4)) \\
3\delta_2 = \oplus \quad & S^{-1}(13(x_{13} \oplus k_{13}) \oplus 9(x_{14} \oplus k_{14}) \oplus 14(x_{15} \oplus k_{15}) \oplus 11(x_{16} \oplus k_{16})) \\
& S^{-1}(13(x'_{13} \oplus k_{13}) \oplus 9(x'_{14} \oplus k_{14}) \oplus 14(x'_{15} \oplus k_{15}) \oplus 11(x'_{16} \oplus k_{16})) \\
2\delta_2 = \oplus \quad & S^{-1}(11(x_9 \oplus k_9) \oplus 13(x_{10} \oplus k_{10}) \oplus 9(x_{11} \oplus k_{11}) \oplus 14(x_{12} \oplus k_{12})) \\
& S^{-1}(11(x'_9 \oplus k_9) \oplus 13(x'_{10} \oplus k_{10}) \oplus 9(x'_{11} \oplus k_{11}) \oplus 14(x'_{12} \oplus k_{12})) \\
\delta_3 = \oplus \quad & S^{-1}(14(x_9 \oplus k_9) \oplus 11(x_{10} \oplus k_{10}) \oplus 13(x_{11} \oplus k_{11}) \oplus 9(x_{12} \oplus k_{12})) \\
& S^{-1}(14(x'_9 \oplus k_9) \oplus 11(x'_{10} \oplus k_{10}) \oplus 13(x'_{11} \oplus k_{11}) \oplus 9(x'_{12} \oplus k_{12})) \\
3\delta_3 = \oplus \quad & S^{-1}(9(x_5 \oplus k_5) \oplus 14(x_6 \oplus k_6) \oplus 11(x_7 \oplus k_7) \oplus 13(x_8 \oplus k_8)) \\
& S^{-1}(9(x'_5 \oplus k_5) \oplus 14(x'_6 \oplus k_6) \oplus 11(x'_7 \oplus k_7) \oplus 13(x'_8 \oplus k_8)) \\
2\delta_3 = \oplus \quad & S^{-1}(13(x_1 \oplus k_1) \oplus 9(x_2 \oplus k_2) \oplus 14(x_3 \oplus k_3) \oplus 11(x_4 \oplus k_4)) \\
& S^{-1}(13(x'_1 \oplus k_1) \oplus 9(x'_2 \oplus k_2) \oplus 14(x'_3 \oplus k_3) \oplus 11(x'_4 \oplus k_4)) \\
\delta_3 = \oplus \quad & S^{-1}(11(x_{13} \oplus k_{13}) \oplus 13(x_{14} \oplus k_{14}) \oplus 9(x_{15} \oplus k_{15}) \oplus 14(x_{16} \oplus k_{16})) \\
& S^{-1}(11(x'_{13} \oplus k_{13}) \oplus 13(x'_{14} \oplus k_{14}) \oplus 9(x'_{15} \oplus k_{15}) \oplus 14(x'_{16} \oplus k_{16})) \\
3\delta_4 = \oplus \quad & S^{-1}(14(x_{13} \oplus k_{13}) \oplus 11(x_{14} \oplus k_{14}) \oplus 13(x_{15} \oplus k_{15}) \oplus 9(x_{16} \oplus k_{16})) \\
& S^{-1}(14(x'_{13} \oplus k_{13}) \oplus 11(x'_{14} \oplus k_{14}) \oplus 13(x'_{15} \oplus k_{15}) \oplus 9(x'_{16} \oplus k_{16})) \\
2\delta_4 = \oplus \quad & S^{-1}(9(x_9 \oplus k_9) \oplus 14(x_{10} \oplus k_{10}) \oplus 11(x_{11} \oplus k_{11}) \oplus 13(x_{12} \oplus k_{12})) \\
& S^{-1}(9(x_9 \oplus k_9) \oplus 14(x_{10} \oplus k_{10}) \oplus 11(x_{11} \oplus k_{11}) \oplus 13(x_{12} \oplus k_{12})) \\
\delta_4 = \oplus \quad & S^{-1}(13(x_5 \oplus k_5) \oplus 9(x_6 \oplus k_6) \oplus 14(x_7 \oplus k_7) \oplus 11(x_8 \oplus k_8)) \\
& S^{-1}(13(x'_5 \oplus k_5) \oplus 9(x'_6 \oplus k_6) \oplus 14(x'_7 \oplus k_7) \oplus 11(x'_8 \oplus k_8)) \\
\delta_4 = \oplus \quad & S^{-1}(11(x_1 \oplus k_1) \oplus 13(x_2 \oplus k_2) \oplus 9(x_3 \oplus k_3) \oplus 14(x_4 \oplus k_4)) \\
& S^{-1}(11(x'_1 \oplus k_1) \oplus 13(x'_2 \oplus k_2) \oplus 9(x'_3 \oplus k_3) \oplus 14(x'_4 \oplus k_4))
\end{aligned}
\tag{A.19}
$$

Key schedule equations

Only the cases where $r > 1$ are considered.

1. Case $(r, c) = (2, 1)$:

$$\begin{aligned} k_1^{-1} &= S^{-1}(k_2) \\ k_2^{-1} &= S^{-1}(k_1 \oplus \kappa_i) \end{aligned} \tag{A.20}$$

2. Case $(r, c) = (2, 2)$:

$$\begin{aligned} k_1^{-1} &= S(k_4 \oplus k_2) \oplus k_1 \oplus \kappa_i & k_3^{-1} &= k_3 \oplus k_1 \\ k_2^{-1} &= S(k_3 \oplus k_1) \oplus k_2 & k_4^{-1} &= k_4 \oplus k_2 \end{aligned} \tag{A.21}$$

3. Case $(r, c) = (2, 4)$:

$$\begin{aligned} k_1^{-1} &= S(k_8 \oplus k_6) \oplus k_1 \oplus \kappa_i & k_5^{-1} &= k_5 \oplus k_3 \\ k_2^{-1} &= S(k_7 \oplus k_5) \oplus k_2 & k_6^{-1} &= k_6 \oplus k_4 \\ k_3^{-1} &= k_3 \oplus k_1 & k_7^{-1} &= k_7 \oplus k_5 \\ k_4^{-1} &= k_4 \oplus k_2 & k_8^{-1} &= k_8 \oplus k_6 \end{aligned} \tag{A.22}$$

4. Case $(r, c) = (4, 1)$:

$$\begin{aligned} k_1^{-1} &= S^{-1}(k_4) \\ k_2^{-1} &= S^{-1}(k_3) \\ k_3^{-1} &= S^{-1}(k_2) \\ k_4^{-1} &= S^{-1}(k_1 \oplus \kappa_i) \end{aligned} \tag{A.23}$$

5. Case $(r, c) = (4, 2)$:

$$
\begin{aligned}
k_1^{-1} &= S^{(}k_8 \oplus k_4) \oplus k_1 \oplus \kappa_i & \qquad k_5^{-1} &= k_5 \oplus k_1 \\
k_2^{-1} &= S^{(}k_7 \oplus k_3) \oplus k_2 & \qquad k_6^{-1} &= k_6 \oplus k_2 \\
k_3^{-1} &= S^{(}k_6 \oplus k_2) \oplus k_3 & \qquad k_7^{-1} &= k_7 \oplus k_3 \\
k_4^{-1} &= S^{(}k_5 \oplus k_1) \oplus k_4 & \qquad k_8^{-1} &= k_8 \oplus k_4
\end{aligned}
\tag{A.24}
$$

6. Case $(r, c) = (4, 4)$:

$$
\begin{aligned}
k_1^{-1} &= S^{(}k_{16} \oplus k_{12}) \oplus k_1 \oplus \kappa_i & \qquad k_9^{-1} &= k_9 \oplus k_5 \\
k_2^{-1} &= S^{(}k_{15} \oplus k_{11}) \oplus k_2 & \qquad k_{10}^{-1} &= k_{10} \oplus k_6 \\
k_3^{-1} &= S^{(}k_{14} \oplus k_{10}) \oplus k_3 & \qquad k_{11}^{-1} &= k_{11} \oplus k_7 \\
k_4^{-1} &= S^{(}k_{13} \oplus k_9) \oplus k_4 & \qquad k_{12}^{-1} &= k_{12} \oplus k_8 \\
k_5^{-1} &= k_5 \oplus k_1 & \qquad k_{13}^{-1} &= k_{13} \oplus k_9 \\
k_6^{-1} &= k_6 \oplus k_2 & \qquad k_{14}^{-1} &= k_{14} \oplus k_{10} \\
k_7^{-1} &= k_7 \oplus k_3 & \qquad k_{15}^{-1} &= k_{15} \oplus k_{11} \\
k_8^{-1} &= k_8 \oplus k_4 & \qquad k_{16}^{-1} &= k_{16} \oplus k_{12}
\end{aligned}
\tag{A.25}
$$

Publications of the Author

In the following, all former publications of the author are listed.

Book Chapters

[PGP+19] Ilia Polian, Mael Gay, Tobias Paxian, Matthias Sauer, and Bernd Becker. Automatic construction of fault attacks on cryptographic hardware implementations. In *Automated Methods in Cryptographic Fault Analysis*, pages 151–170. Springer, 2019.

Journal Publications

[GKKP20] Mael Gay, Batya Karp, Osnat Keren, and Ilia Polian. Error control scheme for malicious and natural faults in cryptographic modules. *Journal of Cryptographic Engineering*, 10, June 2020.

[GKKP19] M. Gay, B. Karp, O. Keren, and I. Polian. Toward error-correcting architectures for cryptographic circuits based on rabii–keren codes. *IEEE Embedded Systems Letters*, 11(4):115–118, 2019.

Conference Proceedings

[KGKP18] Batya Karp, Mael Gay, Osnat Keren, and Ilia Polian. Security-oriented code-based architectures for mitigating fault attacks. In *Conference on Design of Circuits and Integrated Systems, DCIS 2018, Lyon, France, November 14-16, 2018*, pages 1–6, 2018.

[EGP20] N. Elhamawy, M. Gay, and I. Polian. An open-source area-optimized eceg cryptosystem in hardware. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 120–125, 2020.

Workshop Contributions

[GPU+19] Mael Gay, Tobias Paxian, Devanshi Upadhyaya, Bernd Becker, and Ilia Polian. Hardware-oriented algebraic fault attack framework with multiple fault injection support. In *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–32, August 2019.

[BGE+17] J. Burchard, M. Gay, A. M. Ekossono, J. Horáček, B. Becker, T. Schubert, M. Kreuzer, and I. Polian. Autofault: Towards automatic construction of algebraic fault attacks. In *FDTC*, pages 65–72, Sep. 2017.

[GBH+16] M. Gay, J. Burchard, J. Horacek, A.-S. Messeng Ekossono, T. Schubert, B. Becker, I. Polian, and M. Kreuzer. Small scale AES toolbox: Algebraic and propositional formulas, circuit-implementations and fault equations. In *Trustworthy Manufacturing and Utilization of Secure Devices*, 2016.

[GKKP18] M. Gay, B. Karp, O. Keren, and I. Polian. On security metrics for evaluating fault-injection countermeasures. In *Trustworthy Manufacturing and Utilization of Secure Devices*, 2018.

[KGKP18] Batya Karp, Mael Gay, Osnat Keren, and Ilia Polian. Detection and correction of malicious and natural faults in cryptographic modules. In *PROOFS*, volume 7, pages 68–82. EasyChair, 2018.

Other Publications

[PG18] Ilia Polian and Mael Gay. Hardware-oriented security in a computer science curriculum. In *12th European Workshop on Microelectronics Education, EWME 2018, Braunschweig, Germany, September 24-26, 2018*, pages 59–62, 2018.

Declaration

All the work contained within this thesis,
except where otherwise acknowledged, was
solely the effort of the author.
At no stage was any collaboration entered into
with any other party.

Maël Gay