# Secure and Efficient Hardware Implementations for Modern Cryptography


Dissertation

---

*zur Erlangung des Grades eines Doktor-Ingenieurs*
*der Fakultät für Informatik*
*an der Ruhr-Universität Bochum*

---

*by Jan Richter-Brockmann*
*Bochum, January 2023*

To Annika.

Jan Richter-Brockmann
Place of birth: Datteln, Germany
Author's contact information:
jan.richter-brockmann@rub.de

| | |
|---|---|
| Thesis Advisor: | **Prof. Dr.-Ing. Tim Güneysu**<br>Ruhr-Universität Bochum, Germany |
| Secondary Referee: | **Prof. Dr. Peter Schwabe**<br>Max Planck Institute for Security and Privacy, Germany |
| Tertiary Referee: | **Prof. Dr.-Ing. Thomas Eisenbarth**<br>Universität zu Lübeck, Germany |
| Thesis submitted: | January 9, 2023 |
| Thesis defense: | February 10, 2023 |
| Last revision: | April 12, 2023 |

# Abstract

In our contemporary life, digital infrastructure plays a crucial role and it is almost impossible to imagine a modern world without the advantages provided by highly advanced technology. Due to the influence of this infrastructure on so many areas of our life, robust, reliable, and secure systems are necessary. Moreover, this includes performing any kind of communication encrypted avoiding misuse of information and protecting data integrity. Creating an environment of encrypted communication became even more important over the last years since many systems are connected including a huge amount of embedded devices. Therefore, underlying cryptographic algorithms need to be implemented on highly diverse platforms including microcontrollers, Field-Programmable Gate Arrays (FPGAs), and Application-Specific Integrated Circuits (ASICs).

In addition, due to extensive research in the field of quantum computers during the last years, it is more likely than ever that today's deployed public-key cryptography can be broken in the near future. As a consequence, the National Institute of Standards and Technology (NIST) announced a Post-Quantum Cryptography (PQC) standardization process in order to find suitable cryptographic algorithms that are secure against attacks mounted on both classical and quantum computers.

These two emerging fields of study lead to the requirement for secure and efficient hardware implementations for modern cryptography. More precisely, the emergence of post-quantum secure algorithms introduces new challenges with respect to efficient implementations targeting microcontrollers, FPGAs, and ASICs. At the same time, side-channel and fault-injection attacks pose a huge threat against any type of cryptographic implementations on embedded devices, where securing conventional symmetric cryptography is still raising unconsidered and challenging questions.

In this work, we first address the protection of symmetric cryptography against side-channel and fault-injection attacks. We start by investigating protection mechanisms that combine established countermeasures against side-channel attacks with instantiations of linear Error-Correcting Codes (ECCs). Exploiting the structure of linear ECCs and dynamically exchanging their underlying generator matrices, allow us to introduce additional noise into cryptographic operations achieving increased protection against side-channel adversaries.

In the second part, we revisit existing models abstracting fault injections and propose a new generic, simple, and consolidated fault adversary model. Here, we connect the physical behavior of different fault-injection mechanisms more closely with the theoretical abstracted adversary model. Additionally, we cover and introduce security notions for secure and composable gadgets protecting hardware implementations against fault-injection and combined attacks.

For our third part, we use the aforementioned theoretical and essential work to create formal verification frameworks parsing gate-level netlists of implemented countermeasures and analyzing their security. We first present the framework FIVER which incorporates our fault adversary model and evaluates – based on a data structure relying on binary decision diagrams – fault-

injection countermeasures. We continue to drive this work even further by presenting VERICA which is capable of verifying the security of gadgets and entire cryptographic functions in a setting considering combined attacks.

Eventually, we propose efficient implementations of the PQC scheme BIKE targeting reconfigurable hardware. Our first implementation presents an optimized polynomial multiplier, a core performing the polynomial inversion based on Fermat's little theorem, and the first hardware implementation of the black-gray flip decoder. We further improve these results by introducing a new multiplier design exploiting the sparseness of one of its input operands and an optimized polynomial inversion based on the extended Euclidean algorithm.

**Keywords.**

Fault-Injection Analysis, Side-Channel Analysis, Error-Correction Codes, LMDPL, Combined Countermeasures, Hiding, Reconfiguration, Fault Modeling, Adversary Model, Combined Gadgets, Fault Verification, Formal Verification, BDD, Symbolic Simulation, Combined Analysis, BIKE, QC-MDPC, FPGA

# Kurzfassung

## Sichere und effiziente Hardwareimplementierungen für moderne Kryptografie

In unserem heutigen Leben spielt die digitale Infrastruktur eine entscheidende Rolle, und es ist fast unmöglich, sich eine moderne Welt ohne die Vorteile unserer hochentwickelten Technologie vorzustellen. Aufgrund des Einflusses dieser Infrastruktur auf so viele Bereiche unseres Lebens sind robuste, zuverlässige und sichere Systeme unverzichtbar. Dazu gehört auch, dass jegliche Art von Kommunikation verschlüsselt durchgeführt wird, um den Missbrauch von Informationen zu vermeiden und Datenintegrität sicherzustellen. Die Schaffung einer Umgebung für verschlüsselte Kommunikation ist in den letzten Jahren noch wichtiger geworden, da viele Systeme über eine immer weiter wachsende Anzahl von eingebetteten Systemen miteinander verbunden sind. Somit müssen auch die zugrunde liegenden kryptografischen Algorithmen auf sehr unterschiedlichen Plattformen wie Mikrocontrollern, FPGAs und ASICs implementiert werden.

Darüber hinaus ist es aufgrund umfangreicher Forschungsarbeiten auf dem Gebiet der Quantencomputer in den letzten Jahren wahrscheinlicher geworden, dass die heute eingesetzte asymmetrische Kryptografie in naher Zukunft gebrochen werden kann. Infolgedessen hat das NIST einen Standardisierungsprozess angekündigt, um geeignete kryptografische Algorithmen zu finden, die gegen Angriffe auf klassischen Computern und auf Quantencomputern sicher sind.

Diese beiden sich abzeichnenden Forschungsbereiche erfordern das Erforschen von sicheren und effizienten Hardwareimplementierungen für moderne Kryprografie. In diesem Zuge bringt das Aufkommen von sicheren Post-Quantum Algorithmen neue Herausforderungen in Bezug auf effiziente Implementierungen auf Mikrocontrollern, FPGAs und ASICs mit sich. Gleichzeitig stellen Seitenkanal- und Fehlerinjektionsangriffe eine große Bedrohung für jede Art von kryptografischen Implementierungen auf eingebetteten Geräten dar, bei denen selbst die Sicherung symmetrischer Kryptografie immer noch unbeantwortete Fragen aufwirft.

In dieser Arbeit befassen wir uns zunächst mit dem Schutz von symmetrischer Kryptografie vor Seitenkanal- und Fehlerinjektionsangriffen. Zu diesem Zweck untersuchen wir als erstes Schutzmechanismen, die etablierte Gegenmaßnahmen gegen Seitenkanalangriffe mit Instanziierungen von linearen Fehlerkorrekturcodes kombinieren. Durch die Ausnutzung der Struktur von linearen fehlerkorrigierenden Codes und dem dynamischen Austausch der zugrunde liegenden Generatormatrizen können wir zusätzliches Rauschen in eine kryptografische Operation integrieren, was zu einem Design mit erhöhter Wiederstandfähigkeit gegen Seitenkanalangriffe führt.

Im zweiten Teil überprüfen wir bestehende Modelle zur Abstraktion von Fehlerinjektionen und stellen ein generisches, einfaches und vereinheitlichtes Fehlermodell vor. Hier verknüpfen wir das physikalische Verhalten verschiedener Fehlerinjektionsmechanismen enger mit unserem theoretischen Fehlermodell. Außerdem präsentieren wir Sicherheitsdefinitionen für sichere

und zusammensetzbare Gadgets, die Hardwareimplementierungen gegen Fehlerinjektionsangriffe und kombinierte Angriffe schützen.

Im dritten Teil verwenden wir die oben genannten theoretischen und grundlegenden Arbeiten, um formale Verifikationsframeworks zu erstellen, die Netzlisten von Gegenmaßnahmen auf Gatterebene zu analysieren und deren Sicherheit zu überprüfen. Wir stellen zunächst das Framework FIVER vor, welches unser Fehlermodell als Grundlage einbindet und – basierend auf einer Datenstruktur, die auf binären Entscheidungsdiagrammen beruht – Gegenmaßnahmen gegen Fehlerinjektionen evaluiert. Wir treiben diesen Ansatz weiter voran, indem wir mit VERICA die Sicherheit von Gadgets und ganzen kryptografischen Funktionen in einer Umgebung mit kombinierten Angriffen verifizieren.

Schließlich präsentieren wir effiziente Implementierungen des PQC Verfahrens BIKE für rekonfigurierbare Hardware. Unsere erste Implementierung beinhaltet eine optimierten Polynommultiplikation, einen Kern, der die Polynominversion basierend auf dem kleinen fermatischem Satz durchführt, und die erste Hardwareimplementierung des Black-Gray-Flip Decoders. Anschließend verbessern wir diese Ergebnisse, indem wir einen neuen Multiplizierer entwickeln, der die Spärlichkeit eines seiner Eingangsoperanden für eine bessere Leistung ausnutzt, und eine optimierte polynomielle Inversion, die auf dem erweiterten euklidischen Algorithmus basiert.

**Schlagworte.**

Fehlerinjektionsangriffe, Seitenkanalangriffe, Fehlerkorrigierende Codes, LMDPL, Kombinierte Gegenmaßnahmen, Verschleierung, Rekonfiguration, Fehlermodellierung, Angriffsmodel, Kombinierte Gadgets, Fehlerverifikation, Formale Verifikation, Binäre Entscheidungsbäume, Symbolische Simulation, BIKE, QC-MDPC, FPGA

# Acknowledgements

Writing this thesis would not have been possible without the support of many people. First of all, I would like to thank Annika for her unconditional support during my time as graduate student. Next, I thank all my friends for always understanding that I had to work on pending submissions.

I am very grateful to Tim Güneysu for giving me the opportunity to work and learn as a PhD student in his group in Bochum. Thank you for supervising my thesis, giving me advice for my career, and always allowing me to work on projects I was excited about. I also would like to thank Peter Schwabe and Thomas Eisenbarth for taking their time reading this thesis and attending my defense.

Furthermore, I would especially like to thank Pascal Sasdrich who has been a great mentor for me and has taught me many things. I also would like to thank Jakob Feldtkeller for the successful collaborations, especially for enjoying to construct theorems and proofs for our works. Many thanks also go to Ming-Shing Chen who introduced me to many mathematical approaches improving our hardware implementations. Additionally, I would like to thank Florian Bache for sharing an office with me and for supporting me with his exceptional engineering knowledge. I also thank all my coauthors for the great collaborations without them this thesis could not have been written. Eventually, I would like to thank all members of the EmSec, ImpSec, and SecEng groups for creating a wonderful working environment which I always enjoyed being a part of. For each question, you could find an expert who was happy to help.

A special thanks goes to Irmgard Kühn, Horst Edelmann, and Jenny Knothe for taking care of all the administrative work which made my work much easier.

Finally, I thank all the members of the SecRec project, the team behind the project that accelerated the elliptic curve method on GPUs, and all the members of the collaboration with Intel. Especially, I thank Matthias Schunter and Santosh Ghosh for giving me the opportunity to work as an intern for Intel during my PhD.

# Table of Contents

# Part I

# Preliminaries

# Chapter 1

# Introduction

*This chapter briefly introduces the necessity of efficient implementations of cryptographic algorithms on embedded devices. Thereby, threats through physical attacks and their corresponding countermeasures are reviewed. On this basis, we highlight the advantages of formal verification in the design process of those countermeasures. Eventually, we discuss new challenges arising from post-quantum cryptographic algorithms and their implementations. All of these parts, including the design of countermeasures against physical attacks, their formal verification, and post-quantum cryptography form the foundation and contributions of this thesis.*

**Contents of this Chapter**



## 1.1 Motivation

In our contemporary life, digital infrastructure plays a crucial role and it is almost impossible to imagine a modern world without the advantages provided by our highly advanced technology. These innovations can nowadays be found in many sectors including economy, education, healthcare, public transportation, and the majority of critical infrastructure. For example, our economy would not work without a secure and reliable digital banking system. Especially the last few years have shown that a modern educational system needs to support functional homeschooling. Our healthcare system would be inefficient without robust digital monitoring systems supporting hospital attendants. Just recently, a presumptive attack on the German rail networks has demonstrated that the infrastructure of public transportation highly depends on reliable technology. Eventually, critical infrastructure does not only include public transportation but also the energy supply controlled by advanced, digital systems.

Due to these influences of the digital infrastructure on so many areas of our life, robust, reliable, and secure systems are necessary. This includes to perform any kind of communication encrypted in order to avoid misuse of data and maintain data integrity. Encrypted communication became even more important over the last years since many devices ranging from servers to personal computers to a huge amount of embedded devices (including mobile phones, Internet of

Things (IoT), home automatization, etc.) are now connected. To this end, the underlying cryptographic algorithms need to be implemented on highly diverse platforms like microcontrollers, FPGAs, and ASICs.

However, all cryptographic implementations on these devices have in common that they usually are vulnerable to physical attacks. Even though the algorithms are mathematically secure, an attacker can exploit physical side channels to gain information about the secret key material used in an encryption or decryption process on the target device. For example, information can be leaked through timing differences in the algorithm's execution time if it depends on secret key material or sensitive data. Furthermore, critical side channels are also present in the power consumption of target hardware devices. In general and without any countermeasure in place, the dynamic power consumption of common hardware depends on the processed data due to switching activities of transistors in modern Complementary Metal-Oxide-Semiconductor (CMOS) technologies. Therefore, an attacker who has physical access to the target device can measure the power consumption – while the cryptographic algorithm performs an encryption or decryption – and extract sensitive information from the power traces. This cannot only be achieved by directly measuring the power consumption at the power supply but also by acquiring electromagnetic emanation of the Integrated Circuit (IC). Both of these attacks are *passive attacks*, i.e., attacks that do not actively influence the computation.

As opposed to this, fault-injection attacks are classified as *active attacks* since ongoing computations are actively disturbed by using one of many different fault-injection mechanisms. The most common mechanisms are clock or voltage glitches, electromagnetic pulses, and focused laser beams. More precisely, the purpose of clock and voltage glitches is to increase the propagation time of a digital signal so that memory elements sample wrong intermediate results. In contrast, electromagnetic pulses directly target the sampling process of registers by reducing the electrical potential between the ground and supply voltage. Eventually, faults injected by focused laser beams are caused by currents charging or discharging internal nodes of the target circuit so that transitional faults occur which could eventually lead to wrong intermediate results sampled by the registers.

Due to the threat arising from physical attacks, a lot of research focused on investigating and developing countermeasures over the last two decades. Countermeasures against Side-Channel Analysis (SCA) are divided into *hiding*- and *masking*-based approaches. While hiding-based countermeasures aim to hide secret information within noise, masking-based countermeasures apply the concept of *secret sharing* to prevent information leakage. Countermeasures against Fault-Injection Analysis (FIA) are often based on some kind of redundancy. Common approaches use redundancy in time, spatial redundancy, or redundancy in information. Even though the basic principles of these countermeasures are well understood, applying them to a target cryptographic algorithm is still a time-consuming and error-prone task requiring experienced designers and developers.

To this end, incorporating formal verification into the workflow of a hardware designer implementing countermeasures can reduce complex and expensive development cycles. For example, formal hardware verification can avoid to create physical test environments and expensive prototyping. However, such formal verification frameworks require precisely defined models to abstract side-channel and fault-injection attacks. On the one hand, these models need to be simple enough to design efficient verification approaches while describing physical effects as accurately as possible. On the other hand, excessive abstractions could lead to oversimplified

models that do not reflect the actual processes occurring in the hardware. Therefore, adversary models for formal verification tools need to be selected by balancing between accuracy and efficiency.

Besides the well-known threats emerging through physical side channels and their corresponding research areas of designing and verifying countermeasures, extensive advances in the development of quantum computers raise new challenges for modern cryptography. More precisely, established schemes for Public-Key Cryptography (PKC) like RSA and elliptic-curve cryptography can be broken in polynomial time by large-scale quantum computers as presented by Peter Shor in 1999. Therefore, over the last two decades, extensive research focused on finding new schemes which are secure against attacks mounted on powerful quantum computers. The research in this area was even more boosted by the announcement of the NIST searching for new standardized PQC schemes in 2017.

The most important selection criteria for submitted schemes is to provide the required security against both classical and quantum attackers, i.e., attacks mounted on classical computers and quantum computers. However, the NIST published additional selection criteria including the cost and performance of the implemented algorithm on various platforms. This includes optimized implementations for servers or personal computers Central Processing Units (CPUs) as well as embedded devices like microcontrollers, FPGAs, and ASICs.

## 1.2 Summary of Research Contributions

The research of this thesis contributes to all of the aforementioned areas and is partitioned into four parts. More precisely, we contribute to countermeasures against physical attacks, their theoretical models abstracting the underlying physical behavior, and frameworks verifying countermeasures on their gate-level netlist. Additionally, we investigate efficient hardware implementations for post-quantum secure algorithms at the example of Bit Flipping Key Encapsulation (BIKE). All contributions are published in peer-reviewed journals or at international conferences [RSBG20, RG20, RBSG22, FRSG22, RSS$^+$21, RFSG22, RMG22, RCGG22]. These publications form the basis for the chapters of this thesis. Please note that the content is slightly rearranged for the sake of readability. In addition, the author contributed to other works which are out of scope for this thesis and therefore excluded [WRS$^+$20, HFL$^+$20, KLRG22b, LMRG22, KLRG22a, FGG$^+$22]. In the following, we summarize the research contributions in more detail.

### 1.2.1 Protecting Hardware Implementations against Physical Attacks

The first part of this work addresses the design of countermeasures preventing SCA and FIA. Here, we mainly target hardware implementations of symmetric cryptographic algorithms.

**Orthogonal Concurrent Error Correction [RSBG20].** The first work in this part revises the layout of linear ECCs to protect hardware implementations of the Advanced Encryption Standard (AES) against FIAs. Therefore, we exploit the concept of concurrent error detection and elaborate important properties of linear ECCs with respect to their applications as protection mechanisms in the context of fault injections. Based on this concept, we develop a new layout of the linear ECCs which is arranged orthogonal to the state matrix of AES. This novel code layout allows us to achieve more compact hardware implementations for FPGA and ASIC designs.

Moreover, we combine our novel approach with the side-channel countermeasure LUT-Masked Dual-rail with Precharge Logic (LMDPL) to achieve additional protection against side-channel adversaries. To confirm the resistance against first-order SCA, we perform a Test Vector Leakage Assessment (TVLA) using 200 million power traces acquired from a FPGA measurement board.

**Dynamic Fault-Injection Countermeasures [RG20].** In the second work, we combine a first-order secure Threshold Implementation (TI) with linear ECCs since the combination of both countermeasures promises implementations with reasonable overhead. However, combining higher-order TI with linear ECCs would result in considerably higher implementation costs. For the first time, we employ the inherent structure of non-systematic linear ECCs as a countermeasure against FIA and additionally mutate the underlying generator matrices in a dynamic way to achieve protection against higher-order SCAs. In order to demonstrate the effectiveness of our approach, we apply our scheme to the PRESENT cipher and perform a TVLA with 150 million power traces that confirm protection against attacks processing information from up to the third statistical moment.

### 1.2.2 Adversary Models and Security Notions for Physical Attacks

After the practical design of countermeasures against physical attacks on hardware platforms, we address their theoretical security models and abstractions of the corresponding physical behavior. Therefore, we first contribute to this research area by introducing a novel fault-injection adversary model and afterwards defining new security notions for FIA and combined attacks, i.e., attacks combining SCA and FIA.

**Revisiting Fault Adversary Models - Hardware Faults in Theory and Practice [RBSG22].** FIAs are known as a critical attack vector for cryptographic algorithms implemented on embedded devices for a long time. Therefore, researchers from academia and industry proposed a long list of countermeasures. However, the security of most approaches is validated on custom adversary models which are not closely coupled to the actual physical behavior of the deployed fault-injection mechanism. Additionally, custom adversary models complicate to compare the security of different countermeasures. To this end, we present a simple, generic, and consolidated fault-injection adversary model that represents and abstracts the physical behavior of common fault-injection mechanisms. To demonstrate the advantages, we apply our model to cryptographic primitives and extend existing verification tools in order to support our novel adversary model.

**Security Notions for Secure Hardware Gadgets [FRSG22, RFSG22].** Security models abstracting and describing SCAs were extensively researched over the last two decades. This does not only include models for proving security of entire cryptographic primitives but also composability notions allowing to build smaller circuits – so called *gadgets* – that can be used to construct larger designs and still provide the desired security. We use this as a basis to push the research further to composability notions covering combined attacks. Therefore, we first revise existing security notions and slightly redefine them. Afterwards, we define new composability notions for combined attacks using the Probe-Isolating Non-Interference (PINI) notion

as inspiration. Eventually, we present gadgets for hardware implementations that fulfill these security notions.

### 1.2.3 Novel Frameworks for Formal Hardware Verification

Given the results from the previous part, we now present two verification frameworks incorporating the presented adversary model and security notions.

**Formal Verification of Countermeasures against Fault-Injection Attacks [RSS+21].** The validation of countermeasures against FIA is often done by empirical testing in a late stage of the design process. To avoid this and receive early feedback, a designer can deploy formal verification tools. However, existing tools to test hardware implementations of countermeasures against FIA can report false positive verification results, which could be devastating. In this chapter, we present a new formal verification framework called FIVER that parses gate-level netlists and converts them into a data structure based on Binary Decision Diagram (BDD). Utilizing BDDs for this task allows us to perform extensive analyses by considering all input assignments. To demonstrate the effectiveness of our framework, we present several case studies on protected designs of the lightweight ciphers CRAFT and LED-64 as well as of AES.

**Verification of Combined Attacks [FRSG22, RFSG22].** Next, we present the first formal verification framework for combined attacks. Therefore, we combine strategies from the SCA verification tool SILVER [KSM20] with FIVER which we introduced in the previous chapter. The new tool – VERICA – can execute all strategies and analyses which could be performed by SILVER and FIVER. Additionally, VERICA is able to check combined security as well as combined composability notions. Furthermore, we extend the capabilities introduced by FIVER with a strategy checking protection against Statistical Ineffective Fault Analysis (SIFA). Again, we perform several case studies on existing designs from the literature and revealed implementation flaws in gadgets promising protection against combined attacks.

### 1.2.4 Efficient Hardware Implementations of BIKE

In this part, we explore efficient hardware implementations of the PQC scheme BIKE mainly tailored to FPGAs as targeted platform.

**Folding BIKE – Scalable Hardware Implementation for FPGAs [RMG22].** We present the first complete hardware implementation for BIKE. One of the challenges for efficient implementations is the polynomial inversion required in the key generation of BIKE. We explore different strategies to realize the inversion in hardware based on Fermat's little theorem. Moreover, we improve already existing polynomial multipliers and present a design that outperforms previous implementations. Eventually, we implement for the first time the Black-Gray-Flip (BGF) decoder on hardware. As an additional feature, our implementation is fully scalable and generic with respect to BIKE specific parameters. Altogether, our fastest design requires 2.69 ms for the key generation, 0.1 ms for the encapsulation, and 1.89 ms for the decapsulation considering the lowest security level.

**Racing BIKE – Optimized Hardware Design [RCGG22].** The second chapter improves the hardware implementation of BIKE. Therefore, we optimize the implementations of the two key arithmetic operations – the polynomial multiplication and polynomial inversion. First, for the polynomial multiplier, we exploit that each multiplication in BIKE involves at least one sparse polynomial. As part of this, we propose a design that is able to deal with the indefinite Hamming weight in BIKE's encapsulation and still finishes in constant time. Second, our inversion core is based on the extended Euclidean algorithm improving the latency by 5.5 times compared to our previous approach. Besides these two main contributions, we additionally present a united hardware design of BIKE with shared resources and shared sub-modules among the Key Encapsulation Mechanism (KEM) functionalities. On Xilinx Artix-7 FPGAs, our lightweight implementation consumes only 3 777 slices and performs a key generation, encapsulation, and decapsulation in 3 797 µs, 443 µs, and 6 896 µs, respectively.

## 1.3 Structure of this Thesis

The main part of this thesis is divided into four topics presenting the contributions as introduced above. The first part presents two novel countermeasures protecting hardware implementations of symmetric cryptography against physical attacks considering side-channel attacks and fault-injection attacks. The second part defines formal models describing physical attacks used as a foundation for formal verification. These definitions and models are used in the third part introducing verification frameworks for hardware implementations. Eventually, the fourth part introduces efficient hardware implementations of BIKE.

**Part I: Preliminaries.** This part covers important preliminaries used throughout this work and introduces important definitions. First, we provide background about physical attacks covering SCAs and FIAs separately in Chapter 2. We finish the section by discussing countermeasures against FIAs including linear ECCs which serves as transition to Chapter 3 covering important background and definitions from coding theory. Here, Section 3.2 introduced a special family of linear ECCs called Quasi-Cyclic Moderate-Density Parity-Check codes. These codes are used in the PQC scheme BIKE introduced in Chapter 4. Eventually, the last chapter of this part covers models for digital logic circuits and theoretic background of BDDs.

**Part II: Protecting Hardware Implementations against Physical Attacks.** The second part of this thesis covers the two works introducing novel protection mechanisms based on linear ECCs that are combined with existing countermeasures thwarting SCA. Chapter 6 presents the application of linear ECCs that are applied orthogonal the state matrix of AES instead of using a byte-oriented approach as done in previous works.

In Chapter 7, we discuss the countermeasure based on dynamic reconfiguration techniques exchanging linear ECCs of a protected design. Combined with a first-order protected SCA implementation, we show that a higher-order protection is achieved due to the additionally introduced noise of the reconfiguration approach.

**Part III: Adversary Models and Security Notions for Physical Attacks.** Part III is also structured into two chapters. The first chapter (Chapter 8) deals with the introduction and

definition of our formal model describing fault injections. To this end, we first cover common fault-injection mechanisms and explain the physical behavior occurring in the hardware. Afterwards, we introduce our novel model with all its parameters required to accurately abstract fault-injection methods. Eventually, we apply the model to cryptographic examples.

Chapter 9 introduces the new security notions for analyzing and designing hardware gadgets protected against combined attacks, i.e., attacks exploiting information from SCAs and FIAs. Here, we redefine security notions originally introduced in [DN20] and provide new notions inspired by the PINI notion.

**Part IV: Novel Frameworks for Formal Hardware Verification.** Part IV takes up the theoretical models and definitions from Part III and incorporates them into formal verification frameworks. Therefore, Chapter 10 introduces FIVER which formally verifies hardware countermeasures against FIAs. FIVER uses BDDs as underlying data structure allowing it to perform a verification without any false positives as it could happen in frameworks presented in the literature.

Based on the side-channel verification tool SILVER [KSM20] and FIVER, Chapter 11 presents a framework that combines techniques from both tools into a novel tool (called VERICA) that verifies digital logic circuits for protection against combined attacks. Therefore, VERICA preserves all functionalities from SILVER and FIVER but additionally incorporates the theoretical notions for combined security from Chapter 9.

**Part V: Efficient Hardware Implementations of BIKE.** While previous parts mostly considered physical attacks and the verification of symmetric cryptography, this part investigates efficient hardware implementations of BIKE. Chapter 12 presents the first hardware implementation of BIKE including a polynomial inversion, polynomial multiplication, and a decoder. The design is generically implemented so that the internally applied data width can be controlled by a bandwidth parameter allowing to adapt the implementation to the required environment.

Based on that work, Chapter 13 presents several optimizations to achieve more efficient designs. First, we show the advantages of exchanging the symmetric cores of the random oracles of BIKE. Second, a novel multiplier design is introduced that has a considerably lower footprint and latency. Third, the work investigates the efficiency of using the extended Euclidean algorithm as inversion approach outperforming the previous approach based on Fermat's little theorem.

**Part VI: Conclusion.** The last part concludes this thesis by summarizing the individual results and contributions. Additionally, future research directions are presented and discussed.

# Chapter 2

# Physical Attacks and Countermeasures

*Physical and implementation attacks, such as passive Side-Channel Analysis and active Fault-Injection Analysis, have become a major threat to cryptographic algorithms implemented on embedded devices. In this chapter, we discuss both attack vectors by introducing different adversary models, evaluation methods, and countermeasures.*

**Contents of this Chapter**



## 2.1 Side-Channel Analysis

In this section, we discuss passive side-channel attacks by introducing common sources of information leakage used to attack embedded devices, presenting adversary models to abstract the physical behavior of power side-channel attacks, and evaluation methods to assess designs under test. Furthermore, we present existing countermeasures from literature preventing SCA.

### 2.1.1 Physical Side Channels

Over the last 25 years, SCA has become a well-known threat to implementations of cryptographic algorithms. These threats range from analyses of timing differences, over measurements of the power consumption of a target device, to the evaluation of the electromagnetic emanation of an IC. In the following, we briefly describe these three methods in more detail since they are frequently used to attack and evaluate cryptographic implementations.

**Timing Analysis.** Runtime dependencies in implementations of cryptographic algorithms can be exploited to recover sensitive data or secret key material. More precisely, whenever the runtime of a target algorithm depends on secret data (e.g., branching or cache accesses), an attacker can carefully measure differences in the execution time and gather information about the processed sensitive data. This attack vector was presented for the first time by Paul C. Kocher in 1996 [Koc96].

**Power Consumption.** The foundation for modern cryptography is the fast development of ICs resulting in high-efficient CPUs and ASICs supporting dedicated accelerators for cryptographic operations. However, due to data-dependent switching activities of the underlying CMOS structures, plain implementations (in software and hardware) can leak sensitive information via the power consumption of the target device. Therefore, measuring the dynamic power consumption, allows an attacker to gain information about the processed data which eventually could reveal information about secret key material. The discovery of this side channel in 1999 by Kocher et *al.* [KJJ99] was the beginning of an entirely new research direction investigating new attack methods and their corresponding countermeasures.

**Electromagnetic Emanation.** Related to the previously described side channel, an attacker can measure the electromagnetic emanation generated by an active hardware device. Again, the electromagnetic emanation directly depends on the processed data and the switching activities of the transistors. Therefore, an attacker can exploit the dependencies between the electromagnetic emanation and the processed data to gain information about secret data processed by the target device.

### 2.1.2 Adversary Model

In this thesis, we only consider side-channel adversaries using information gained from the power consumption or the electromagnetic emanation of a target hardware device. In order to develop and assess countermeasures preventing SCA, a proper definition of the adversary model is inevitable. A commonly used adversary model is based on the work from Ishai, Sahai, and Wagner [ISW03]. In general, an adversary is assumed to learn intermediate values from a hardware implementation which can be modeled by having access to the exact values of wires carrying the internal signals of the circuit. Since it is infeasible to construct countermeasures where an adversary has access to the values of all wires at the same time without learning anything about the processed data, the power of the adversary is limited by the number of wires she can probe and, hence, learns the corresponding values [ISW03]. We call this model *d-probing model* where an adversary has access to up to $d$ wires which can freely be chosen from all available wires. The set of selected wires is also called *probes*. Hence, if an adversary is not able to learn anything about the processed data using any possible probe combination of up the $d$ probes, a target circuit is assumed to be secure against a $d$-th order attacker.

However, hardware designs suffer from additional sources leading to exploitable side-channel leakage. This includes physical defaults like *glitches*, *transitions*, and *couplings* [FGP$^+$18]. In an ideal digital circuit, each gate – and therefore each transistor – is only evaluated once for each new input. However, each single transistor introduces a propagation delay (depending on physical properties and the previous data) which implies that some gates are evaluated more than once. The short time where the gate evaluates to the wrong value is called *glitch*. While glitches only occur in combinatorial logic, *transitions* are caused by consecutive values written to memory cells. Eventually, *couplings* are caused by adjacent wires such that information from one wire can couple into another wire. The result of each of these physical defaults is that an adversary probing one wire can gain information about values of more than the probed wire. Therefore, Faust et *al.* introduced an $(g, t, c)$*-robust d-probing model* covering these additional effects [FGP$^+$18]. The parameters $g$, $t$, and $c$ indicate whether the effect of glitches, transitions,

and couplings, respectively, are considered. In this work, we focus our analyses on the *glitch-extended d-probing model*, i.e., the $(1, 0, 0)$-robust $d$-probing model.

Recently, the *random probing model* gained more attention in the community [BCP+20]. In this model, each wire is assumed to leak its value with probability $p$. However, as mentioned above, we do not conduct evaluations in the random probing model but rather model a side-channel adversary by the glitch-extended $d$-probing.

### 2.1.3 Practical Evaluation Methods

To assess the resistance of a design implemented on an embedded device against power side-channel attacks, the applied evaluation method should be independent of any theoretical models, intermediate values of the executed algorithm, and on any specific attacks. Therefore, Goodwill et *al.* proposed to use Welsh's $t$-test which is an extension of the Student's $t$-test [GGJR+11]. Before we describe the corresponding leakage assessment, we recap important statistical preliminaries mainly taken from [SM15].

**Statistical Preliminaries.** The evaluation of acquired power traces relies on the application of statistical properties. We start by defining the *raw statistical moment* in Definition 1.

**Definition 1** (Raw Statistical Moment)**.** *The d-th order raw statistical moment of a random variable $X$ is defined by $M_d = \mathsf{E}\left(X^d\right)$.*

For the first order (i.e., $d = 1$), we denote $\mu = M_1$ as the *mean*. Additionally, we define the $d$-th order *central moment* for $d > 1$ by Definition 2.

**Definition 2** (Central Moment)**.** *The d-th order central moment for $d > 1$ is defined by $CM_d = \mathsf{E}\left((X - \mu)^d\right)$.*

For $d = 2$ the central moment is also called *variance* denoted by $\sigma^2$. Eventually, we define the *standardized moment* by Definition 3.

**Definition 3** (Standardized Momement)**.** *The d-th order standardized moment for $d > 2$ is defined by $SM_2 = \mathsf{E}\left(\left(\frac{X-\mu}{\sigma}\right)^d\right)$.*

Here, $SM_3$ is called the *skewness* and $SM_4$ the *kurtosis*.

In general, Welsh's $t$-test is used to validate the null hypothesis that samples from two populations have equal means which implies they are indistinguishable from each other. Therefore, let $Q_0$ and $Q_1$ be two sets analyzed by the $t$-test. Further, let $\mu_0$ and $\mu_1$ the means of the two sets and $\sigma_0^2$ and $\sigma_1^2$ the sample variance of $Q_0$ and $Q_1$. The cardinality of $Q_0$ and $Q_1$ is denoted by $n_0$ and $n_1$, respectively. Then, the $t$-value for the two sets $Q_0$ and $Q_1$ is computed by

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{\sigma_0^2}{n_0} + \frac{\sigma_1^2}{n_1}}}. \tag{2.1}$$

**Test Vector Leakage Assessment.** TVLA is used to evaluate a cryptographic implementation with respect to side-channel leakage and was proposed in [GGJR+11]. As already mentioned above, the advantage of TVLA is that it can be applied without any knowledge about the origin of possible side-channel leakage. Therefore, TVLA utilizes Welsh's $t$-test (cf. Equation 2.1) where the two sets consist of power traces acquired from the target device. More precisely, the power traces $T_{i\in\{1,...,n\}}$ belong to some data $D_{i\in\{1,...,n\}}$ where $n$ denotes the total amount of queries to the target. Each power trace $T_i$ consists of $m$ samples points such that $T_i = \{t_1, ..., t_m\}$ and each sample point is evaluated individually based on Equation 2.1. In a cryptographic context, the associated data $D_i$ are mostly plaintexts or ciphertexts while the underlying secret of the target algorithm is fixed for the analysis. Eventually, the two sets $Q_0$ and $Q_1$ can be constructed from the power traces $T_i$ by applying one of the following two approaches.

**Specific:** In the *specific* $t$-test the two sets $Q_0$ and $Q_1$ are constructed based on intermediate results of the target algorithm. All traces that belong to an execution of the algorithm where a specific intermediate value (e.g., one single bit or one byte) is equal to $x$ are collected in $Q_0$ while all remaining ones are collected in $Q_1$. However, this strategy suffers from the specific selection of $x$ which does not allow a comprehensive evaluation of the target design.

**Non-Specific:** In order to overcome these limitations, the *non-specific* $t$-test is used. The two sets $Q_0$ and $Q_1$ are constructed by traces acquired from a fixed input (e.g., plaintext or ciphertext) and a random input, respectively. Due to this procedure, the test is often called *fixed vs. random* $t$-test.

After constructing the two sets, the means $\mu_0$, $\mu_1$ and the variances $\sigma_0^2$, $\sigma_1^2$ can be estimated to compute the first-order $t$-values for each sample point individually based on Equation 2.1. In order to determine the $t$-values for higher orders, the traces need to be preprocessed first. Afterwards, the mean and variance of the preprocessed traces are computed allowing to apply Equation 2.1. For second-order tests, the mean-free squared traces $Y = (X - \mu)^2$ are used for the $t$-value computation. Hence, the mean is actually the variance $CM_2$ of the original traces. For third and higher orders, the standardized moment $SM_d$ (cf. Definition 3) describes the estimated mean. To compute the entire $t$-values for higher orders, the estimates of the variances of the preprocessed traces are required as well. For the second order, the variance $\sigma_Y^2$ of the mean-free squared values $Y = (X - \mu)^2$ is estimated as $\sigma_Y^2 = CM_4 - CM_2^2$ (for a detailed derivation, please see [SM15]). In case $d \geq 3$, the variance $\sigma_{Z_d}^2$ of the preprocessed traces $Z_d$ for order $d$ is given by

$$\sigma_{Z_d}^2 = SM_{2d} - SM_d^2 = \frac{CM_{2d} - CM_d^2}{CM_2^d}.$$

Eventually, based on the computed $t$-values, the null hypothesis is accepted or rejected. Usually, a threshold of $|t| > 4.5$ is defined to reject the hypothesis, i.e., a design is considered to be insecure for the given order $d$. This simplification allows to reject the null hypothesis with a confidence of more than 0.999 99.

Note, in 2017 Ding et *al.* showed that the threshold of $\pm 4.5$ needs to be adapted for measurements with many sample points in order to avoid false positives in the evaluation phase [DZD+17].

**Measurement Setup.** In many chapters of this thesis, we perform practical side-channel evaluations to validate the desired protection level against SCA. Therefore, we utilize the side-channel measurement setup presented by Bache et *al.* in [BPW+19]. The setup consists of a software part controlling the measurement and a hardware part executing the design under test. More precisely, the software controls the entire hardware including the oscilloscope, amplifier, and the target FPGA. Additionally, all statistical computations (i.e., the $t$-test as introduced above) are executed by the software framework. The hardware part is divided into a control and target FPGA. The control FPGA generates randomness for masks and decides whether a fixed or random input is processed. The target FPGA executes the design under test while its power supply is connected via an amplifier with the oscilloscope. In this work, we use different types of oscilloscopes, amplifiers, and target FPGA boards which we briefly summarize in the following.

**Oscilloscope:** The first oscilloscope that is used for practical evaluations is the 6404D PicoScope. The second oscilloscope (Spectrum M4 oscilloscope) is a more performant device achieving a higher throughput since it is integrated into a PCI express card. Both oscilloscopes have a resolution of 8 bit.

**Amplifier:** The used amplifiers are Low Noise Amplifiers (LNAs) from MiniCircuits. The ZFL-1000LN+ amplifier has a gain of 24 dB while the gain of the ZFL-2000GH+ can be controlled by an additional input voltage.

**Targets:** Both target boards are explicitly designed for side-channel measurements. The Sakura-G board is equipped with a Spartan-6 FPGA to implement the target design while the Sakura-X board is equipped with a more recent Kintex-7 FPGA.

### 2.1.4 Countermeasures

In this section, we discuss important approaches to design countermeasures against SCA. In the first part, we briefly describe *hiding* methods and especially focus on randomization techniques. In the second part, we introduce *masking* with a focus on Boolean masking and TI.

#### Hiding

As introduced above, power SCAs exploit dependencies between the processed secret data and the power consumption of the target device. Hence, an approach to decrease these dependencies is to *hide* the secret data in the overall power consumption of the device. This can be achieved by increasing or adding noise such that the signal cannot easily be identified in the power traces. Vice versa, a designer could try to decrease the signal of the secret data such that it is indistinguishable from the noise. In both cases, the objective is to decrease the Signal-to-Noise Ratio (SNR) of the running implementation on the target device.

**Randomization.** Hiding mechanisms can also be realized by randomization, e.g., randomizing the clock frequency [HDL+20] to hamper the alignment of traces which is an essential step to conduct a side-channel attack. Another strategy relies on reconfiguration techniques that randomize the dynamic power consumption of the target device. These kinds of hiding methods are especially effective when they are combined with provable secure protection mechanisms

(e.g., with masking which is introduced in the next section) to achieve protection against higher-order attacks [SMG15, SMG17].

### Masking

*Masking* is a common countermeasure to protect cryptographic hardware implementations on embedded devices against SCAs and is studied for more than twenty years [CJRR99]. The basic principle is to split up a sensitive value $x$ into multiple *shares* $x_i$ with $0 \leq i < s$ leading to

$$x = x_0 \circ x_1 \circ \cdots \circ x_{s-1}. \tag{2.2}$$

Here, $s$ defines the number of shares and $\circ$ denotes the group operator of the underlying masking scheme. In order to achieve a secure masking, $s-1$ shares are chosen uniformly at random and the remaining share is determined such that Equation 2.2 is satisfied. Since all sensitive values are split up into shares, any function $f$ processing $x$ is also transferred to a shared representation $f = f_0 \circ f_1 \circ \cdots \circ f_{s-1}$.

We call a masking scheme instantiated with group operator $\oplus$ a *Boolean masking* and an *arithmetic masking* when an addition or multiplication is applied. In this thesis, we only use Boolean masking which is explained in more detail in the following paragraph.

**Boolean Masking.** Boolean masking is a common countermeasure to protect implementations of symmetric cryptographic schemes against SCA. As mentioned above, it is realized by substituting the group operator of Equation 2.2 with an Exclusive OR (XOR) operation. To this end, we formally define a Boolean sharing by Definition 4.

**Definition 4** (Boolean Sharing)**.** *A* Boolean sharing *of a value $x \in \mathbb{F}_2^m$ is a vector $\boldsymbol{x} = \langle x_0, \ldots, x_{s-1} \rangle$ such that $x = \bigoplus_{i=0}^{s-1} x_i$, with $x_i \in \mathbb{F}_2^m$ uniform random and for all $\mathcal{X} \subsetneq \{x_0, ..., x_{s-1}\}$ the values $x_i \in \mathcal{X}$ are independent. Further, let $\mathsf{S} : \mathbb{F}_2^m \mapsto \mathbb{F}_2^{m \cdot s}$ be a probabilistic* share function *that outputs a valid Boolean sharing $\langle x_0, \ldots, x_{s-1} \rangle$ for some $x$. Similarly, $\mathsf{U} : \mathbb{F}_2^{m \cdot s} \mapsto \mathbb{F}_2^m$ is a deterministic* unshare *function that computes the original value $x$ for a share vector $\langle x_0, \ldots, x_{s-1} \rangle$.*

When using $\mathsf{S}(x)$ for a $x \in \mathbb{F}_2^{m \cdot \ell}$, we apply the sharing function element-wise on $\ell$ different values $\mathbb{F}_2^m$. Similarly, we write $\mathsf{U}(x)$ for $x \in \mathbb{F}_2^{m \cdot s \cdot \ell}$ when unsharing $\ell$ different share vectors element-wise.

Further, we define a *shared circuit* as a digital logic circuit that operates on Boolean shares. As usually done in literature, we consider the sharing and unsharing functions outside of the scope of the adversary [AIS18].

**Definition 5** (Shared Circuit)**.** *A* shared circuit $\mathbf{C}_{\mathsf{F}}^{s}$ *for a function $\mathsf{F} : \mathbb{F}_2^{m \cdot \ell} \mapsto \mathbb{F}_2^{m \cdot \ell'}$ and a Boolean sharing scheme with $s$ shares is a probabilistic circuit realizing a function $\mathsf{F}^{\mathbf{C}} : \mathbb{F}_2^{m \cdot s \cdot \ell} \mapsto \mathbb{F}_2^{m \cdot s \cdot \ell'}$, such that $\forall x \in \mathbb{F}_2^{m \cdot \ell}$ it holds that $\mathsf{F}(x) = \mathsf{U}(\mathsf{F}^{\mathbf{C}}(\mathsf{S}(x)))$ (*functional correctness*).*

However, the challenge of all masking schemes is to share non-linear functions. In the following, we discuss Threshold Implementations and constructions from gadgets which are both approaches based on Boolean masking.

**Threshold Implementations.** TI was originally proposed by Nikova, Rechberger, and Rijmen and is known as a provable secure and widely used masking scheme to protect digital circuits against SCA [NRR06]. To provide the desired security, the target implementation has to fulfill the following properties.

**Correctness:** Given a function $y = \mathsf{F}(x)$ from $\mathbb{F}_2^{m \cdot \ell}$ to $\mathbb{F}_2^{m \cdot \ell'}$, the TI realization of $\mathsf{F}$ requires a shared representation $\mathsf{F}^{\mathbf{C}} = (\mathsf{F}^0, ..., \mathsf{F}^{t-1})$ where the $\mathsf{F}^i$ are called component functions. *Correctness* is ensured if $\boldsymbol{y} = \mathsf{F}^{\mathbf{C}}(\boldsymbol{x})$ satisfies $y = \bigoplus_{i=0}^{t-1} \mathsf{F}^i(\boldsymbol{x})$ for $x = \bigoplus_{i=0}^{s-1} x_i$ and $\boldsymbol{x} = \langle x_0, \ldots, x_{s-1} \rangle$.

**Non-Completeness:** To ensure a secure TI implementation in the presence of glitches, each function $\mathsf{F}^{\mathbf{C}}$ has to be *non-complete*. Particularly, for a first-order secure implementation of a function $\mathsf{F}$ each component function $\mathsf{F}^{i \in \{0,\ldots,t-1\}}$ must be independent of at least one input share $x_{j \in \{0,\ldots,s-1\}}$.

**Uniformity:** Since the security of TI is based on Boolean masking, a *uniform* distribution of the shared representation is essential. However, the results of a shared function $\mathsf{F}^{\mathbf{C}}$ are used as input to subsequent functions such that *uniformly* distributed outputs of $\mathsf{F}^{\mathbf{C}}$ are required. In other words, the set of all possible output sharings $\mathcal{F} = \{\mathsf{F}^0, ..., \mathsf{F}^{t-1} \mid \boldsymbol{x} \in \mathcal{X}\}$ must be uniformly drawn from the set $\mathcal{Y} = \{\boldsymbol{y} \mid \bigoplus_{i=0}^{t-1} y_i = y\}$ assuming a given set of all possible input sharings $\mathcal{X} = \{\boldsymbol{x} \mid \bigoplus_{i=0}^{s-1} x_i = x\}$. Violating the *uniformity* property would lead to a biased sharing.

Again, the complexity to find TIs for non-linear functions increases with the complexity of the underlying function and its algebraic degree. Nevertheless, constructing efficient TIs for arbitrary non-linear functions can be challenging.

**Composability Notions.** To overcome the challenges of finding suitable TIs, unprotected circuits can be transformed to secure implementations by replacing all Boolean gates with atomic and secure building blocks called *gadgets*. Since the theoretical foundations of the $d$-probing model introduced in Section 2.1.2 are not sufficient to argue about the *composability* of gadgets ensuring secure composed circuits, additional security notions are required.

A common technique of defining security notions is *simulation* [Can01, Mau11]. For this, a *real* and an *ideal* game are defined. The ideal game is assumed to be trivially secure with respect to a given adversary model. Furthermore, the ideal game is modeled by a probabilistic polynomial-time simulator determining the view of the adversary without using information of the secrets. To this end, a circuit in the real game is secure iff the view of the adversary is indistinguishable from the ideal game's simulator. Hence, the adversary is not able to distinguish the two games with a probability higher than 1/2. As a consequence, the view of the adversary is independent of the circuit's secrets.

Given that, the first security notion to analyze gadgets with respect to their composability has been proposed by Barthe et *al.* [BBD+15] and is known as Non-Interference (NI). In this thesis, we use the term Probe Non-Interference (PNI) in order to prevent any confusion with other non-interference definitions introduced later.

**Definition 6** (Probe Non-Interference [BBD+15])**.** *A shared gadget $\mathsf{G}$ is $d$-*PNI *iff any set of $d' \leq d$ probes can be perfectly simulated with at most $d'$ shares of each input.*

PNI limits the amount of leakage for shared intermediate variables in a gadget. However, the composition of PNI gadgets does not guarantee probing secure circuits. Hence, Barthe et *al.* proposed Probe Strong Non-Interference (PSNI) as an extension to PNI introducing more restrictions [BBD+16].

**Definition 7** (Probe Strong Non-Interference [BBD+16])**.** *A shared gadget $\mathsf{G}$ realizing a function $\mathsf{F} : \mathbb{F}_2^{m \cdot s \cdot \ell} \mapsto \mathbb{F}_2^{m \cdot s}$ is* $d$-PSNI *iff for any set of probes, of which $d_1$ are internal probes and $d_2$ are output probes such that $d_1 + d_2 \leq d$, the probes can be perfectly simulated by $d_1$ shares of each input.*

A circuit composed of PSNI gadgets is again PSNI and therefore also probing secure. Since a circuit constructed of PSNI gadgets can produce a huge overhead in terms of area and latency, more efficient compositions can be achieved by combining PNI and PSNI gadgets following rules presented in [BBD+16, BGR18]. Another strategy to reduce the overhead has been proposed by Cassiers and Standaert introducing the PINI security notion [CS20].

**Definition 8** (Probe-Isolating Non-Interference [CS20])**.** *A gadget $\mathsf{G}$ is $d$-PINI iff for any set of $d_1$ internal probes and any set $\mathcal{S}_2$ of $d_2$ share domains, such that $d_1 + d_2 \leq d$, there exists a set $\mathcal{S}_1$ of at most $d_1$ share domains such that the outputs of the share domains in $\mathcal{S}_2$ and the probes can be simulated with the inputs of the share domains in $\mathcal{S}_1 \cup \mathcal{S}_2$.*

Hence, PINI ensures that information cannot leak from one *share domain* to another. Therefore, the composition of circuits can easily be done since all share domains are isolated. In the next section, we briefly introduce three types of gadgets that are frequently used in this thesis.

### Gadgets

Given the composability notions from the previous section, we now describe common gadgets that fulfill some of these security assertions.

**Domain-Oriented Masking.** Domain-Oriented Masking (DOM) has been presented in 2016 by Gross, Mangard, and Korak [GMK16]. The approach associates each share of a variable $x$ with a specific domain. More precisely, assuming that $x$ is divided into $s = d + 1$ shares denoted by $x = x_0 \oplus x_1 \oplus ... \oplus x_d$, the indices of the shares indicate the corresponding domain. To this end, the main idea is to keep all shares independent from shares of other domains [GMK16]. Based on this idea, the authors propose an algorithm to construct shared Boolean multiplications for arbitrary security orders. The formal description of this gadget is given in Algorithm 1 (by Reg[$v$] we indicate that the value $v$ has been sampled and stored in a register).

The DOM gadget fulfills the PNI security property in the standard and glitch-extended model. However, it is only PSNI secure in the standard probing model and does not provide security under the PINI security notion.

**Hardware Private Circuit 1.** In 2021, Cassiers et *al.* presented a paper introducing two new gadgets for hardware that fulfill the PINI security notion [CGLS21]. These Hardware Private Circuits (HPCs) are specifically designed to provide security in the glitch-extended $d$-probing model for arbitrary orders while being trivially composable. The first gadget is called HPC1 and is depicted in Figure 2.1a. HPC1 is composed of a DOM gadget (see previous paragraph) and

**Algorithm 1** Multiplication Gadget using Domain-Oriented Masking.

**Require:** $a = \sum_{i=0}^{d} a_i$, $b = \sum_{i=0}^{d} b_i$.
**Ensure:** $c = a \cdot b = \sum_{i=0}^{d} c_i$
1: **procedure** DOM($a_0, ..., a_d, b_0, ..., b_d$)
2: **for** $i = 0$ **to** $d$ **do**
3: $u_{i,i} \leftarrow a_i \cdot b_i$
4: **for** $j = i + 1$ **to** $d$ **do**
5: $u_{i,j} \leftarrow a_i \cdot b_j \oplus r_{i+j(j-1)/2}$
6: $u_{j,i} \leftarrow a_j \cdot b_i \oplus r_{i+j(j-1)/2}$
7: **end for**
8: **end for**
9: **for** $i = 0$ **to** $d$ **do**
10: $c_i \leftarrow \mathrm{Reg}[u_{i,0}]$
11: **for** $j = 1$ **to** $d$ **do**
12: $c_i \leftarrow c_i \oplus \mathrm{Reg}[u_{i,j}]$
13: **end for**
14: **end for**
15: **end procedure**

a refresh gate at one of its inputs. Hence, the total amount of required bits of fresh randomness is given by $d \cdot (d + 1)$.

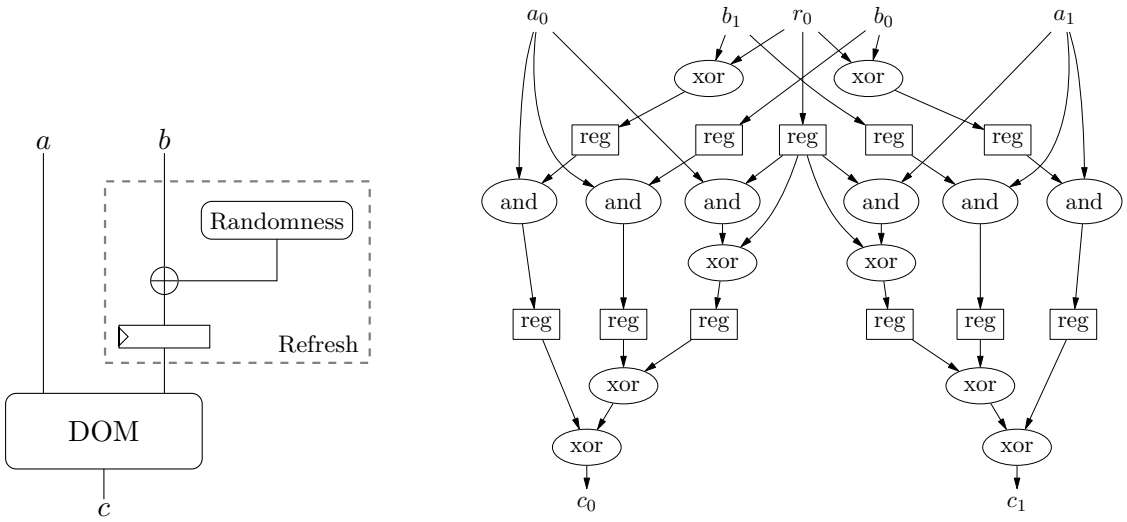
**Hardware Private Circuit 2.** The second gadget that has been introduced by Cassiers et *al.* is named HPC2 and is sketched for $d = 1$ in Figure 2.1b. The main motivation for introducing HPC2 is to reduce the number of random bits required for refreshing inside the gadget. Hence, HPC2 gadgets have the same randomness costs as DOM gadgets, i.e., $(d \cdot (d + 1)/2)$ random bits. For a generic and algorithmic description of HPC2, we refer the interested reader to [CGLS21].

## 2.2 Fault-Injection Attacks

The second class of physical attacks we discuss in this thesis falls in the area of active attacks consolidating all mechanisms to inject faults in an ongoing cryptographic operation. In this section, we briefly introduce different attack mechanisms and adversary models. Afterwards, we describe different approaches to counteract FIAs.

### 2.2.1 Fault-Injection Mechanisms

An attacker using active fault-injection attacks aims to alter an intermediate state of an ongoing cryptographic process (e.g., an encryption or decryption). The erroneous intermediate values eventually lead to a wrong output of the faulted algorithm. In case the fault was precisely placed, the attacker is able to use the faulted (and often also the correct) outputs to receive information about the sensitive key material.




(a) HPC1 Gadget.

(b) First-order HPC2 Gadget.

Figure 2.1: Gadgets from Hardware Private Circuits [CGLS21].

Besides the theory of where and when a fault needs to be placed to gain information about the internal secrets of an algorithm, many research focused on injection mechanisms, i.e., which methods can be used to successfully inject faults in an IC. These attack vectors include clock or voltage glitches [ADN+10, ZDCT13], electromagnetic pulses [DDRT12, DLM21, OGM15, OGM17], or focused photon injection using laser beams [SA02, RSDT13, CLFT14, SBHS15]. A detailed description of these mechanisms and the physical background of their functionality is given in Chapter 8 since it is closely connected to the contribution of the chapter.

### 2.2.2 Adversary Model

Since we consider physical attacks, we assume that an attacker has physical access to the target device. Additionally, for most of the aforementioned injection mechanisms, the attacker needs to modify the Printed Circuit Board (PCB) or target chip in order to successfully launch an attack (e.g., removing an oscillator or decapsulating the chip).

The faults occurring in the hardware are often modeled by *stuck-at* and *toggle* (bit-flip) faults as they are often used in IC testing and verification [RU96]. The stuck-at model is further divided into *stuck-at-0* and *stuck-at-1* faults where a wire is assumed to be faulted such that it is tied to logical zero or logical one, respectively. The fault described by the toggle model relies on the original value where a logical zero is turned into a logical one and vice versa. However, as we show in Chapter 8, these basic models are often too inaccurate to precisely describe the underlying physical behavior of the fault injection mechanisms.

### 2.2.3 Analysis Techniques

In order to exploit injected faults in a hardware implementation of a cryptographic algorithm, researchers proposed a plethora of analysis techniques over the last two decades. These analysis techniques range from the seminal Differential Fault Analysis (DFA) over ineffective and

statistical methods like Ineffective Fault Analysis (IFA) and Statistical Fault Attack (SFA) to a combination of both.

**Differential Fault Analysis.** With the seminal work of Biham and Shamir presenting the first DFA on the Data Encryption Standard (DES) [BS97], a foundation for further research with respect to fault analysis of the AES was laid. In 2002, Giraud presented the first DFA for AES by inducing faults in the ninth round and requiring around 250 faulty encryptions [Gir04]. This attack was later improved by Blömer and Seifert such that 128 to 256 faulty ciphertexts are sufficient to recover the secret key [BS03]. By injecting faults between the eighth and ninths round, Dusart et *al.* were able to reduce the number of required faulty ciphertexts to 40 [DLV03]. Eventually, Ali and Mukhopadhyay pushed DFA on AES to its limit requiring just one faulty encryption [AM11].

However, all approaches have in common that pairs of correct and faulty ciphertexts are required for a successful attack. As a basic intuition, DFA computes the difference between a correct and faulty ciphertext and instantiates a counter for each possible key candidate. By estimating the corresponding key bytes, the attacker recomputes the intermediate state and searches for correlations to the difference between the correct and faulty ciphertext. In case a correlation is found for one estimated key candidate, the corresponding counter is increased. Eventually, the key candidate that belongs to the highest counter is assumed to be the correct key used in the underlying cryptographic implementation.

**Ineffective Fault Analysis.** In 2007, Clavier presented IFA and demonstrated its effectiveness on DES [Cla07]. In order to successfully mount an attack based on IFA, the attacker is assumed to precisely inject a known fault (e.g., stuck-at-1 or stuck-at-0) into a known gate. Given that, random data is fed into the implemented algorithm and the attacker observes the output. In case the output is correct, the fault was *ineffective* and the attacker learns the intermediate value where the fault has been injected. However, this attack assumes a powerful attacker who is able to precisely inject controllable faults while requiring a huge amount of encryptions or decryptions.

**Statistical Fault Attack.** SFA was introduced by Fuhr et *al.* in 2013 [FJLT13]. SFA assumes that faults are injected in a specific byte or nibble of a cipher and that the faults do not occur randomly but rather with a biased distribution (e.g., stuck-at faults). Based on this, an attacker collects $N$ faulty ciphertexts and guesses the involved parts of the key $\hat{K}$. Using all key guesses, the attacker calculates back the intermediate result $v$ of the encryption or decryption up to the point where the fault was injected. The most likely key candidate is determined by applying the Squared Euclidean Imbalance (SEI) to all intermediate results for each key guess. The key candidate with the highest SEI value is assumed to be the correct key candidate. Assuming a cipher state divided into bytes, the SEI is defined by Equation 2.3 where $|\{v \mid v = n\}|$ denotes the number of intermediate values $v$ that is equal to $n$.

$$\mathsf{SEI}\left(\hat{K}\right) = \sum_{n=0}^{255} \left( \frac{|\{v \mid v = n\}|}{N} - \frac{1}{256} \right)^2 \tag{2.3}$$

**Statistical Ineffective Fault Analysis.** Eventually, Dobraunig et *al.* proposed a powerful attack combining IFA and SFA to SIFA [DEK+18]. This method is seen to be very powerful since it can be used to circumvent almost all detection-based countermeasures (for more details about countermeasures see Section 2.2.4). This is possible since SIFA only requires correct ciphertexts to recover the secret key. Hence, all ciphertexts that are detected and marked as faulty are not required to mount a successful attack. However, the same assumptions about a biased distribution of the injected faults as for SFA are supposed. But instead of knowing the exact fault type and location as for IFA, the attacker does not need to know the fault type which is a more realistic scenario for real-world attacks. To this end, SIFA has relaxed assumptions about the attacker's capabilities and can be used to circumvent many well-established countermeasures.

### 2.2.4 Countermeasures

Countermeasures against FIA commonly rely on *redundancy* with respect to area, time, or information. Another approach tries to *infect* the intermediate state of a cryptographic algorithm such that the values are independent of sensitive key material.

**Redundancy in Time.** One trivial approach to detect faults is to recompute an algorithm's output twice or several times [MSY06]. The outputs are compared allowing to detect or correct errors. In general, $k+1$ re-computations are required to *detect* up to $k$ faults or $2k+1$ re-computations to *correct* up to $k$ faults. However, in case error detection is applied and a fault is detected, the output is held back and not released.

**Infection.** In contrast to fault-detection-based countermeasures, *infection*-based countermeasures always return an output. Therefore, in case an effective fault is injected in an ongoing encryption or decryption process, the intermediate result is *infected* by randomness such that the final output is useless to the attacker. Examples for infection-based countermeasures can be found in [GST12, TBM14]

**Redundancy in Area.** Spatial redundancy pursues the same purpose as redundancy in time. However, instead of computing an algorithm's output several times sequentially, the algorithm is instantiated in parallel and executed simultaneously [MSY06]. Again, this approach can be combined with simple detection or correction schemes instantiating the algorithm $k+1$ or $2k+1$ times, respectively.

**Redundancy in Information.** More advanced techniques consider code-based approaches to protect a target algorithm against FIA. Especially linear ECCs have been extensively investigated in the literature [AMR+20, SRM20, RSM21]. The first approach [AMR+20] utilizes linear ECCs for detection purposes only. Later, [SRM20] extended the capabilities to construct protection mechanisms allowing to correct occurring faults. Eventually, in [RSM21] both approaches have been combined and the detection, as well as correction capabilities of the underlying code, are used to create fault-resistant designs. Since linear ECCs are not only used as countermeasures against FIA but also play a crucial role in other cryptographic areas, we cover them separately in Chapter 3.

# Chapter 3

# Coding Theory

*In this chapter, we cover the theoretical background of linear error codes which are used in many cryptographic applications. The detection and correction capabilities of linear error codes can be used as countermeasure against physical attacks as highlighted in the previous chapter. Additionally, they serve as foundation for entire cryptographic schemes. After an introduction of linear ECCs, we cover a specific family of linear error codes called Quasi-Cyclic Moderate-Density Parity-Check (QC-MDPC) codes. They particularly got attention in the last years due to the post-quantum standardization process hosted by the NIST.*

**Contents of this Chapter**



## 3.1 Theory of Linear Codes

Linear ECCs are well-known from communication theory and are an established technique to *detect* or *correct* errors occurring on communication channels. However, as mentioned in the introduction of this chapter, they also serve as foundations for many applications in cryptography. Therefore, we start this section by defining linear codes and stating their corresponding detection and correction capabilities. For this, we closely follow the notations presented in [vT93].

**Definition 9** (Linear Code)**.** *A linear $(n, k)$ code $\mathcal{C}$ of length $n$, dimension $k$, and co-dimension $r = (n - k)$ is a $k$-dimensional subspace of $\mathbb{F}_q^n$.*

Each linear code is defined by a matrix $G \in \mathbb{F}_q^{k \times n}$ called *generator matrix*. The counterpart of a linear code's generator matrix is the *parity-check matrix* $H \in \mathbb{F}_q^{(n-k) \times n}$. Both matrices are formally defined by Definition 10 and Definition 11.

**Definition 10** (Generator Matrix)**.** *A matrix $G \in \mathbb{F}_q^{k \times n}$ is called a generator matrix of a $(n, k)$-linear code $\mathcal{C}$ if $\mathcal{C} = \left\{ mG \mid m \in \mathbb{F}_q^k \right\}$.*

**Definition 11** (Parity-check Matrix)**.** *A matrix $H \in \mathbb{F}_q^{(n-k)\times n}$ is called parity-check matrix of $\mathcal{C}$ if $\mathcal{C} = \left\{c \in \mathbb{F}_q^n \mid Hc^T = 0\right\}$.*

In this thesis, we only consider linear codes defined over $\mathbb{F}_2^n$. On the one hand, binary linear codes are best suited to protect symmetric ciphers against fault injections. On the other hand, this thesis investigates optimized hardware implementations of a cryptosystem that is based on binary linear codes. To this end, given a valid codeword $c \in \mathcal{C}$ of a binary linear code and a vector $e \in \mathbb{F}_2^n$ such that $c' = c \oplus e$, the vector $s^{\mathrm{T}} = Hc'^{\mathrm{T}}$ is called *syndrome*. Due to Definition 11, it holds that $s^{\mathrm{T}} = Hc^{\mathrm{T}} \oplus He^{\mathrm{T}} = He^{\mathrm{T}}$.

In order to determine the detection and correction capabilities of a linear error code, we define its minimum Hamming distance.

**Definition 12** (Minimum Hamming Distance)**.** *The minimum distance $d_{min}$ of a linear code $\mathcal{C}$ is the smallest Hamming Distance (HD) between all codewords and is defined as*

$$d_{min} = min\left(\{HD\left(c_1, c_2\right) \mid c_1, c_2 \in \mathcal{C}, c_1 \neq c_2\}\right). \tag{3.1}$$

Since the minimum distance $d_{\min}$ is an essential property of a linear code, they are commonly called $(n, k, d_{\min})$-codes. The detection and correction capabilities are given by Corollary 1.

**Corollary 1.** *A code $\mathcal{C}$ with minimum distance $d_{min}$ can detect $u = d_{min} - 1$ errors and correct $v = \left\lfloor \frac{d_{min}-1}{2} \right\rfloor$ errors. If $d_{min}$ is even, this implies $\mathcal{C}$ can simultaneously detect $u = \frac{d_{min}}{2}$ errors and correct $v = \frac{d_{min}-2}{2}$ errors.*

Hence, a faulted codeword $c' = c \oplus e$, where $e \in \mathbb{F}_2^n$ denotes an error vector, can be detected by an $(n, k, d_{\min})$-code as long as $\mathrm{HW}(e) \leq u$ and corrected as long as $\mathrm{HW}(e) \leq v$.

Furthermore, Definition 13 defines additional important properties of linear error codes.

**Definition 13** (Equivalent Codes)**.** *Two linear codes over $\mathbb{F}_2$ are equivalent if one can be obtained from the other by a combination of operations of the following two types:*

*(a) an arbitrary permutation of its coordinate positions*

*(b) multiplication of symbols appearing in a fixed position by a non-zero scalar.*

Hence, equivalent codes have the same properties, i.e., the same minimum distance $d_{\min}$.

**Definition 14** (Systematic Codes)**.** *A code $\mathcal{C}$ is called systematic code if and only if $G = [I_k \mid P]$ where $I_k$ denotes the identity matrix of size $k$.*

Note that every generator matrix $G$ of a non-systematic code $\mathcal{C}$ can be transformed to another generator matrix $\tilde{G}$ of a systematic code based on Definition 13 [Bla03].

## 3.2 Quasi-Cyclic Moderate-Density Parity-Check Codes

A special family of linear ECCs is described by QC-MDPC codes which we briefly introduce in this section. Therefore, we start by defining *circulant matrices*, Quasi-Cyclic (QC) codes, and important properties of such codes. For the corresponding definitions, we closely follow the notations of [ABB+20b].

**Definition 15** (Circulant Matrix)**.** *A binary square matrix $A$ is called circulant matrix if each row is the rotation of one element to the right of the preceding row.*

As a result, a circulant matrix is completely defined by its first row. Additionally, a block-circulant matrix is composed of circulant square blocks of identical size called *order*. The number of circulant blocks in a row is called *index*. A formal definition is given below.

**Definition 16** (Quasi-Cyclic Code)**.** *A (binary) QC code of index $n_0$ and order $r$ is a linear code which admits as generator matrix a block-circulant matrix of order $r$ and index $n_0$. A $(n_0, k_0)$-QC code is a quasi-cyclic code of index $n_0$, length $n_0 r$ and dimension $k_0 r$.*

A binary $r \times r$ matrix $A$ can be expressed by an element from a quotient polynomial ring $\mathcal{R} = \mathbb{F}_2[X]/(X^r - 1)$. The mapping between $A$ and $\mathcal{R}$ is described by a natural ring isomorphism $\varphi$ which maps the first row of $A$, represented by $(a_0, a_1, ..., a_{r-1})$, to the polynomial $\varphi(A) = a_0 + a_1 X + ... + a_{r-1} X^{r-1}$. Therefore, all matrix operations can be seen as polynomial operations.

**Definition 17** (Transposition)**.** *The transposition of a polynomial $a_0 + a_1 X + ... + a_{r-1} X^{r-1} = a \in \mathcal{R}$ is defined as $a^T = a_0 + a_{r-1} X + ... + a_1 X^{r-1}$.*

This definition ensures $\varphi(A^{\mathrm{T}}) = \varphi(A)^{\mathrm{T}}$. Furthermore, the isomorphism $\varphi$ can be extended to any binary vector $(v_0, v_1, ..., v_{r-1}) = \mathbf{v} \in \mathbb{F}_2^r$ such that $\varphi(\mathbf{v}) = v_0 + v_1 X + ... + v_{r-1} X^{r-1}$. To stay consistent with Definition 17, the transposition of $\varphi(\mathbf{v})$ is defined as $\varphi(\mathbf{v}^{\mathrm{T}}) = v_0 + v_{r-1} X + ... + v_1 X^{r-1}$ resulting in $\varphi(\mathbf{v}A) = \varphi(\mathbf{v})\varphi(A)$ and $\varphi(A\mathbf{v}^{\mathrm{T}}) = \varphi(A)\varphi(\mathbf{v})^{\mathrm{T}}$.

**Definition 18** (Quasi-Cyclic Moderate-Density Parity-Check Code)**.** *A quasi-cyclic code of length $n = n_0 r$, dimension $k = k_0 r$, order $r$ and a parity-check matrix with constant row weight $w = \mathcal{O}(\sqrt{n})$ is called an $(n_0, k_0, r, w)$-QC-MDPC code.*

In the next section, we introduce the basic principles behind decoders used in QC-MDPC codes.

## 3.3 Iterative Decoding for QC-MDPC Codes

The decoding process, i.e., recovering the correct message from a faulty codeword, for Low-Density Parity-Check (LDPC) and Moderate-Density Parity-Check (MDPC) codes is a complex proceeding. In 1962, Gallager presented the first decoding algorithms for LDPC codes [Gal62]. The principle behind these iterative algorithms can also be transferred to decoding algorithms used for QC-MDPC codes. More precisely, the algorithms determine bits of the codewords that most probably contain an error and flip the corresponding bits according to some predefined and code-specific threshold. In the following, we introduce these *bit-flipping* algorithms more formally.

Given an erroneous codeword $c' \in \mathbb{F}_2^n$ and a parity-check matrix $H \in \mathbb{F}_q^{(n-k)\times n}$ of an underlying code $\mathcal{C}$, we first compute the syndrome $s^{\mathrm{T}} = Hc'^{\mathrm{T}}$ as introduced in Section 3.1. Next, we compute the number of Unsatisfied-Parity-Check (UPC) equations for each codeword bit. The number of UPC equations for bit $j$ of $c'$ is determined by performing an element-wise multiplication of the syndrome $s$ with $H_j$ indicating the $j$-th column of $H$. We denote the result

by $upc_j = s * H_j$ where $*$ represents an element-wise multiplication. To this end, the number of violated parity-check equations for a single bit of $c'$ is given by $|upc_j|$. Based on these values, all bits in $c'$ for which $|upc_j| > T$ are flipped. The threshold $T$ is a predefined parameter determined for each decoder and code individually.

The intuition behind the bit-flipping algorithm is to identify the columns in the parity-check matrix $H$ that most likely lead to a non-zero entry in the syndrome $s$. Since a column of $H$ directly corresponds to a bit in $c'$, and therefore to the error vector included in $c'$, a higher $|upc_j|$ indicates that position $j$ is erroneous with a higher probability. In the following, we explain the algorithm by reference to an example.

**Example 1.** *As an example, we assume a given generator matrix $G \in \mathbb{F}_2^{8\times 14}$ as defined in Equation 3.2.*

$$G = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

*Furthermore, we encode the message $m = [0\ 1\ 1\ 1\ 0\ 0\ 1\ 0]$ with $G$ as introduced in Definition 10. Hence, the error-free codeword $c$ is given by*

$$c = m \cdot G = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

*Now, let us assume that a single-bit error occurs during transmission over a noisy communication channel. Therefore, we assume the following error vector*

$$e = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

*with $|e| = 1$ which leads to the faulty codeword*

$$c' = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

*The parity-check matrix $H \in \mathbb{F}_2^{6\times 14}$ associated with the defined generator matrix $G$ is given by*

$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}. \tag{3.3}$$

*Given $H$ and $c'$, we now compute the syndrome $s \in \mathbb{F}_2^6$ which results in*

$$s = \left(Hc'^T\right)^T = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

*Eventually, we determine the UPC equations $|upc_j|$ by performing an element-wise multiplication of the columns of $H$ with the transposed syndrome $s^T$ followed by a summation of the resulting elements. The vector $U$ containing all added violated parity-check equations $|upc_j|$ for each bit position $j$ of $c'$ is given by*

$$U = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 2 & 0 & 1 & 1 \end{bmatrix}.$$

*Hence, in this example, $|upc_{10}|$ is the largest value in $U$ such that we can assume that the corresponding bit in $c'$ is erroneous and need to be flipped.*

This example visualizes the computation of UPC equations and the corresponding bit flipping. However, over the years, many different algorithms for efficient bit-flipping decoders have been proposed [HvMG13, DGK20c]. In the next chapter, we present a specific decoder used in BIKE.

# Chapter 4

# Bit Flipping Key Encapsulation

*In 2017, the NIST announced a standardization process to find new asymmetric cryptographic schemes that are secure against attacks mounted on large-scale quantum computers. One important category of these PQC schemes are Key Encapsulation Mechanisms which we introduce in the first part of this chapter. Afterwards, we describe Bit Flipping Key Encapsulation – one candidate of NIST's standardization process that proceeded to the fourth round. Since BIKE is based on linear ECCs, a decoder recovering an intentionally added error is required. We explain the functionality of BIKE's decoder in the last section of this chapter.*

**Contents of this Chapter**



## 4.1 Key Encapsulation Mechanisms

In general, key encryption schemes can be divided into symmetric and asymmetric cryptography. Symmetric key encryption schemes use the same key for the encryption and decryption and are perfectly suited to encrypt large data. In contrast, asymmetric cryptography uses a key pair consisting of a private and public key. Such schemes are often impractical to use for encrypting large data because their underlying algorithms are usually more complex than symmetric algorithms. However, asymmetric cryptography is often used to implement KEMs allowing to secretly exchange a shared private key applied in symmetric encryption schemes. KEMs consist of three algorithms, the *key generation*, *encapsulation*, and *decapsulation*. The key generation generates the public and private key using some random input. The public key is transmitted to the communication partner and is utilized in the encapsulation to derive the shared secret key with the help of additional randomness (often called message). Additionally, the encapsulation computes a cryptogram which is sent back. Eventually, in the decapsulation this cryptogram is used together with the private key to compute the same shared secret key as in the encapsulation. An important property of each asymmetric encryption scheme and KEM is that it is only possible to compute the same shared secret from the cryptogram if the private key is known.

In the next section, we introduce the KEM BIKE which is a candidate of NIST's PQC standardization process.

Table 4.1: BIKE parameters.

| | **BIKE Specific** | | | **Decoder Specific** | | | | |
|---|---|---|---|---|---|---|---|---|
| Security | $r$ | $w$ | $t$ | $f_0$ | $f_1$ | $c$ | $NBIter$ | $\tau$ |
| Level 1 | 12 323 | 142 | 134 | 0.0069722 | 13.530 | 36 | 5 | 3 |
| Level 3 | 24 659 | 206 | 199 | 0.005265 | 15.2588 | 52 | 5 | 3 |
| Level 5 | 40 973 | 274 | 264 | 0.00402312 | 17.8785 | 69 | 5 | 3 |

## 4.2 Specifications

BIKE is a code-based scheme and was initially defined as a suite of three different algorithms. However, with the submission to the third round to the NIST standardization process, only one algorithm survived which is presented in this section. BIKE uses QC-MDPC codes (cf. Section 3.2) as underlying structure. Before we introduce the exact definitions of the key generation, encapsulation, and decapsulation, we discuss important parameters specifying BIKE.

**Parameters.** Besides the security level $\lambda$[1], BIKE is specified by three parameters $r$, $w$, and $t$. The parameter $r$ defines the block length and needs to be prime such that $(X^r - 1)/(X - 1) \in \mathbb{F}_2[X]$ is irreducible. The row weight $w$ defines the number of bits set in the private key and is chosen such that $w/2$ is odd. The parameter $t$ is a positive integer and determines the decoding radius, i.e., the Hamming weight of an error vector $e = (e_0, e_1)$. As an additional parameter, the shared secret size $\ell$ is defined as a positive integer and fixed to 256 for all parameter sets. Please note that the code length $n$ is set to $n = 2r$, i.e., $n_0 = 2$ as introduced in Definition 16. All parameters are summarized in Table 4.1 with their corresponding values for the different security levels. The parameters given in the last five columns are required in the decapsulation described below.

**Special Functions.** Besides the parameters introduced above, BIKE defines a set of three functions $\mathbf{H}$, $\mathbf{K}$, and $\mathbf{L}$ modeled as random oracles. The functions are defined over the following domains and ranges.

$$\begin{aligned}\mathbf{H} &: \{0,1\}^{\ell} \rightarrow \{0,1\}^{2r}_{[t]} \\ \mathbf{K} &: \{0,1\}^{r+2\ell} \rightarrow \{0,1\}^{\ell} \\ \mathbf{L} &: \{0,1\}^{2r} \rightarrow \{0,1\}^{\ell}\end{aligned}$$

Here, $\{0,1\}^{2r}_{[t]}$ describes a polynomial of length $2r$ with exactly $t$ non-zero coefficients. Over the time period of NIST's standardization process, the exact definition of the three random oracles changed. In the beginning, $\mathbf{H}$ was realized by an AES256 implementation while $\mathbf{K}$ and $\mathbf{L}$ were realized by a SHA2-384 instantiation. However, as we demonstrate in Chapter 13, this choice is not optimal for hardware implementations. Therefore, $\mathbf{H}$ has been replaced by a SHAKE256 core while $\mathbf{K}$ and $\mathbf{L}$ have been replaced by an instantiation of SHA3-384. The

[1]The security level $\lambda = 1$ corresponds to an equivalent security of AES-128, $\lambda = 3$ to AES-192, and $\lambda = 5$ to AES-256.

**Algorithm 2** Key Generation.

**Require:** BIKE parameters $n, w, t, \ell$.
**Ensure:** Private key $(h_0, h_1, \sigma)$ and public key $h$.
1: Generate $(h_0, h_1) \xleftarrow{\$} \mathcal{R}^2$ both of odd weight $|h_0| = |h_1| = w/2$.
2: Generate $\sigma \xleftarrow{\$} \{0,1\}^\ell$ uniformly at random.
3: Compute $h \leftarrow h_1 h_0^{-1}$.
4: **return** $(h_0, h_1, \sigma)$ and $h$.

advantage of this choice is that both algorithms work on the same structure, i.e., KECCAK. Hence, in order to realize **H**, the SHAKE256 core generates random strings which are used to determine the non-zero bit positions of the target polynomial. In case the sampled randomness is larger than $r$, it is discarded to avoid any kind of bias. Additionally, if a position of the target polynomial is sampled multiple times, the corresponding randomness is also rejected. This procedure is called *rejection sampling*. For **K** and **L**, the SHA3-384 is used as hash-function whereas the 384-bit hash value is truncated to 256 bits.

Just recently, Guo et *al.* presented an attack on the procedure of the rejection sampling used for **H** [GHJ+22]. Since the input to **H** is a secret string $m$ ($m$ is used as seed for SHAKE256), the attacker is able to gain information about $m$ due to runtime differences. Therefore, the exact definition of **H** was adapted again in order to achieve a constant-time implementation. More precisely, the Fisher-Yates algorithm is used as proposed by Sendrier in [Sen21]. Since these latest changes are not relevant for this thesis, we refer the interested reader to the full specification of BIKE containing all recent modifications [ABB+22].

**Key Generation.** The formal definition of BIKE's key generation is given in Algorithm 2. The first step is to sample two secret polynomials $h_0$ and $h_1$ such that both consist of exactly $w/2$ non-zero coefficients. Additionally, a random bit string $\sigma$ of $\ell$ bits is sampled. Together, both polynomials and $\sigma$ represent the secret key of BIKE. The public key is computed by multiplying $h_1$ with the inverse of $h_0$.

**Encapsulation.** The encapsulation (cf. Algorithm 3) starts by sampling a random bit string $m$ of $\ell$ bits. The string is called message and is used as input to **H**, i.e., it is used as seed for SHAKE256 as explained above. However, the output of **H** are two error polynomials $e_0$ and $e_1$ which are required to have in sum a Hamming weight of $t$. The error polynomials are used together with the public key $h$ to compute the first part of the cryptogram $C = (c_0, c_1)$. More precisely, $e_1$ is multiplied by $h$ and added to $e_0$. The second part of the cryptogram is computed by adding (exclusive or) the message $m$ to a hash derived by hashing $e_0, e_1$ using **L**. Eventually, the shared secret key is derived by **K** using $m$ and $C$ as input.

**Decapsulation.** The core functionality of the decapsulation is to recover the intentionally added error vector from the cryptogram. Therefore, the syndrome $s$ is computed by multiplying $c_0$ with $h_0$. Together with the secret key polynomials $h_0$ and $h_1$, all three polynomials are used as input for a decoder which is described in the next section in more detail. However, the decoder either returns two error polynomials $e'_0$ and $e'_1$ or indicates a failed decoding by $\perp$. In case the decoding succeeds, the determined error polynomials are hashed by **L** and the result is

**Algorithm 3** Encapsulation.

**Require:** Public key $h$.
**Ensure:** Encapsulated key $K$ and ciphertext $C = (c_0, c_1)$.
1: Generate $m \xleftarrow{\$} \{0,1\}^{\ell}$ uniformly at random.
2: Compute $(e_0, e_1) \leftarrow \mathbf{H}(m)$.
3: Compute $C = (c_0, c_1) \leftarrow (e_0 + e_1 h, m \oplus \mathbf{L}(e_0, e_1))$.
4: Compute $K \leftarrow \mathbf{K}(m, C)$.
5: **return** $(C, K)$.

**Algorithm 4** Decapsulation.

**Require:** Private key $(h_0, h_1, \sigma)$ and ciphertext $C = (c_0, c_1)$.
**Ensure:** Decapsulated key $K$.
1: Compute syndrome $s \leftarrow c_0 h_0$.
2: Compute $\{(e'_0, e'_1), \perp\} \leftarrow \texttt{decoder}(s, h_0, h_1)$.
3: Compute $m' \leftarrow c_1 \oplus \mathbf{L}(e'_0, e'_1)$.
4: **if** $\mathbf{H}(m') \neq (e'_0, e'_1)$ **then**
5: Compute $K \leftarrow \mathbf{K}(\sigma, C)$.
6: **else**
7: Compute $K \leftarrow \mathbf{K}(m', C)$.
8: **end if**
9: **return** $K$.

added to $c_1$ obtaining $m'$. In order to verify the correctness of the recovered error polynomials, $m'$ is given to $\mathbf{H}$ and the resulting polynomials are compared to $e'_0$ and $e'_1$. If the polynomials match, the same shared secret key as in the encapsulation can be derived using $\mathbf{K}$ with $m'$ and $C$ as input. If the polynomials do not match, $\mathbf{K}$ is invoked with $\sigma$ and $C$ as inputs.

## 4.3 Decoder

As briefly introduced above, the decapsulation of BIKE invokes a `decoder` (cf. Algorithm 4) trying to recover the error vector sampled in the encapsulation process in order to recover the message $m$. An efficient algorithm for this task has been presented in [DGK20c] and is called Black-Gray-Flip decoder (cf. Algorithm 5). Since the submission to the third round of the NIST PQC competition, the BGF decoder is included in the BIKE specifications. The decoder is an iterative algorithm, running for $NBIter$ iterations, taking $(s, h_0, h_1)$ as input, and returning an error vector $e' = (e'_0, e'_1)$ in case of a successful decoding or $\perp$ when the decoding fails. Based on the Hamming weight of the sum $s + eH^{\mathrm{T}}$, a threshold $T$ is computed by

$$\texttt{threshold}(x) = \max(\lfloor f_0 \cdot x + f_1 \rfloor, c) \tag{4.1}$$

where $f_0, f_1$, and $c$ are constants associated with the security level $\lambda$. The procedure `BFIter` counts the UPC equations by invoking `ctr` (i.e., the Hamming weight of $H_j \cdot s$ where $H_j$ is the $j$-th column of the matrix $H$) and flips all bits in the error vector that were indicated by counter values exceeding the threshold $T$ (for more details about this procedure, please see Section 3.3). Additionally, `BFIter` generates two lists – *black* and *gray* – which mark all positions where the

**Algorithm 5** Black-Gray-Flip Decoder [DGK20c, ABB+20b].

**Require:** $H \in \mathbb{F}_2^{r \times n}$, $s \in \mathbb{F}_2^r$

$e \leftarrow 0^n$
**for** $i = 1$ **to** $NBIter$ **do**
  $T \leftarrow \texttt{threshold}\left(\left|s + eH^{\mathrm{T}}\right|\right)$
  $e, black, gray \leftarrow \text{BFIter}\left(s + eH^{\mathrm{T}}, e, T, H\right)$
  **if** $i = 1$ **then**
    $e \leftarrow \text{BFMIter}\left(s + eH^{\mathrm{T}}, e, black, (d+1)/2+1, H\right)$
    $e \leftarrow \text{BFMIter}\left(s + eH^{\mathrm{T}}, e, gray, (d+1)/2+1, H\right)$
  **end if**
**end for**
**if** $s = eH^{\mathrm{T}}$ **then**
  **return** $e$
**else**
  **return** $\perp$
**end if**

**procedure** BFIter$(s, e, T, H)$
  **for** $j = 0$ **to** $n - 1$ **do**
    **if** $\texttt{ctr}(H, s, j) \geq T$ **then**
      $e_j \leftarrow e_j \oplus 1$
      $black_j \leftarrow 1$
    **else if** $\texttt{ctr}(H, s, j) \geq T - \tau$ **then**
      $gray_j \leftarrow 1$
    **end if**
  **end for**
  **return** $e, black, gray$
**end procedure**

**procedure** BFMIter$(s, e, mask, T, H)$
  **for** $j = 0$ **to** $n - 1$ **do**
    **if** $\texttt{ctr}(H, s, j) \geq T$ **then**
      $e_j \leftarrow e_j \oplus mask_j$
    **end if**
  **end for**
  **return** $e$
**end procedure**

counter exceeds $T$ or $T - \tau$, respectively. In the first iteration of the decoder, these two lists are used to adjust the error vector by applying the procedure `BFMIter`. All parameters used to define the decoder are summarized in Table 4.1 for all three security levels.

# Chapter 5

# Hardware Verification

*In the previous chapters, we covered physical attacks and common countermeasures for hardware implementations of cryptographic algorithms. Additionally, we introduced practical evaluation techniques to quantify the security of side-channel countermeasures. However, these methods are often very time-consuming and error-prone. As a solution, formal verification can support a designer in the development cycle of countermeasures against physical attacks. This requires to define appropriate abstractions to model a digital hardware circuit in computer-aided verification tools. These abstractions are covered in the first part of this chapter. Since we especially use BDDs to perform verification of a target protection mechanism in this thesis, we introduce their background in the second part of this chapter.*

**Contents of this Chapter**



## 5.1 Circuit Abstraction

The formal verification of security compliances of countermeasures against physical attacks implemented on hardware platforms requires to transfer the underlying digital logic circuit into an abstract model. To this end, a gate-level netlist describing a target digital logic circuit $\mathbf{C}$ is the perfect basis to construct such a model. The circuit $\mathbf{C}$ realizes an arbitrary Boolean function $\mathsf{F} : \mathbb{F}_2^p \rightarrow \mathbb{F}_2^q$ with input size $p \geq 1$ and output size $q \geq 1$. At the lowest level, we decompose the circuit $\mathbf{C}$ into atomic components, called *gates*, which can be further divided into purely *combinational gates* and *memory gates*.

**Definition 19** (Combinational Gate)**.** *A* combinational gate $g_c$ *is a physical component in a digital logic circuit that evaluates its output as a pure (Boolean) function of the present inputs only (without any dependency on the history of inputs).*

In this work, we limit the set of Boolean functions implemented as combinational gates by $\mathcal{G}_\mathrm{c} = \{\mathsf{not}, \mathsf{and}, \mathsf{nand}, \mathsf{or}, \mathsf{nor}, \mathsf{xor}, \mathsf{xnor}\}$. We further distinguish between gates with fan-in size one (unary gates) and size two (binary gates) leading to the sets $\mathcal{G}_\mathrm{u} = \{\mathsf{not}\}$ and $\mathcal{G}_\mathrm{b} = \{\mathsf{and}, \mathsf{nand}, \mathsf{or}, \mathsf{nor}, \mathsf{xor}, \mathsf{xnor}\}$ such that $\mathcal{G}_\mathrm{c} = \mathcal{G}_\mathrm{u} \cup \mathcal{G}_\mathrm{b}$.

**Definition 20** (Memory Gate)**.** *A* memory gate $g_m \in \mathcal{G}_m$ *is a physical, clock-synchronized component in a digital logic circuit for which the output depends not only on the present inputs but also on the history of previous inputs.*

Memory gates store a single Boolean variable $x \in \mathbb{F}_2$ while we model them as clock-dependent synchronization points. We suppose that a memory gate has only one output. Some standard libraries make use of flip-flops with two complementary outputs. In such cases, the gate is further decomposed to a sequential gate followed by a combinational gate $\mathsf{not}$. Analogously to the definition of combinational gates, we define the set $\mathcal{G}_\mathrm{m} = \{\mathsf{reg}\}$.

**Definition 21** (Randomness Gate)**.** *A randomness gate* $g \in \mathcal{G}_{\mathsf{rand}}$ *is a physical, clock-synchronized component in a digital logic circuit* $\mathbf{C}$ *without inputs. For each clock cycle the output is an independently and uniformly chosen random value.*

Given that, we unite all valid gates in one set $\mathcal{G} = \mathcal{G}_\mathrm{c} \cup \mathcal{G}_\mathrm{m} \cup \mathcal{G}_{\mathsf{rand}}$ to properly model a digital logic circuit $\mathbf{C}$ as defined in Definition 22.

**Definition 22** (Circuit Representation)**.** *A digital logic circuit* $\mathbf{C}$ *is modeled by a Direct Acyclic Graph (DAG) formally described by* $\mathbf{D} = \{\mathcal{V}, \mathcal{E}\}$*, with* $\mathcal{V}$ *the set of vertices and* $\mathcal{E}$ *the set of edges. A single vertex* $v \in \mathcal{V}$ *represents a combinational or memory gate* $g \in \mathcal{G}$ *and a single edge* $e \in \mathcal{E}$ *represents a wire connecting two vertices* $v_1, v_2 \in \mathcal{V}$ *and carrying a digital signal, modeled as an element from the finite field* $\mathbb{F}_2$*.*

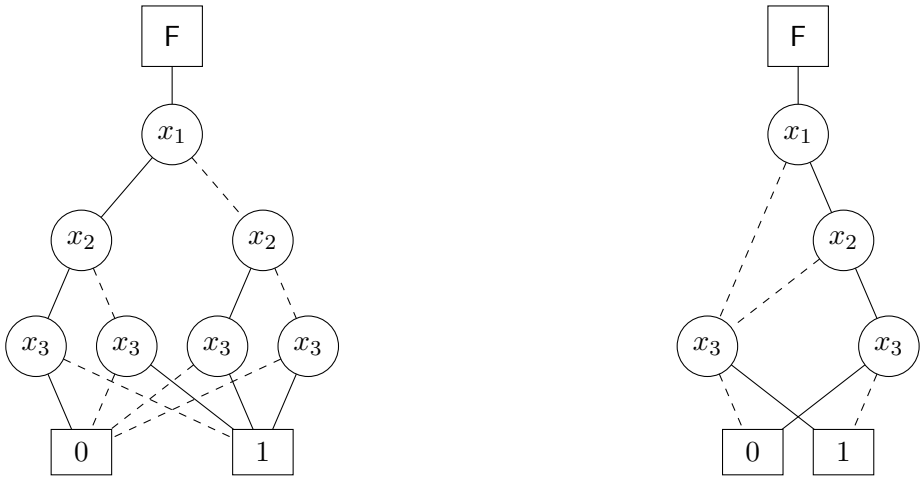
Based on this definition, the model cannot handle circuits with loops, which is a common practice in digital logic circuit designs[1]. Hence, to allow our model to handle such cases, the circuit needs to be unrolled before it is translated to a DAG. By unrolling, we describe the process of removing any cyclic loops and replacing them with acyclic structures. For example, a cryptographic round-based implementation of an arbitrary block-cipher can be unrolled by instantiating the round logic $r$-times connecting them separated by registers where $r$ denotes the number of rounds.

## 5.2 Binary Decision Diagrams

As introduced in the previous section, we model digital logic circuits representing Boolean functions as DAGs. However, this model cannot be used efficiently to evaluate the exact output values for specific – or even all – input assignments. For this purpose, Akers proposed Binary Decision Diagrams in 1978 [Jr.78]. Later, Bryant showed that reduced and ordered BDDs have a canonical representation [Bry86] which restored the attention of BDDs in the research community.

**Reduced Ordered Binary Decision Diagram.** In general, BDDs can be represented by binary decision trees as shown in Figure 5.1. More precisely, Figure 5.1 depicts the BDD for the function $\mathsf{F}(x_1, x_2, x_3) = x_1 \cdot x_2 \oplus x_3$ where the top node is called *function* node. The round nodes labeled with the variable names $x_1$, $x_2$, and $x_3$ are called *internal* nodes. The rectangular nodes at the bottom are the *terminal* nodes of the BDD and can either be 1 or 0.

[1] By this, we do not refer to combinatorial loops which are not considered as a usual synchronous design architecture.




(a) Non-reduced BDD. (b) Reduced BDD.

Figure 5.1: BDDs for the function $\mathsf{F} = x_1 \cdot x_2 \oplus x_3$.

Figure 5.1a shows a non-reduced BDD of the target function F which is evaluated by following the solid arcs (also called *then* arc) if the corresponding variable is set to 1 and the dotted arcs (also called *else* arc) if the variable is set to 0. The *order* of the BDD is given by the sequence the nodes appear in the graph from top to bottom, i.e., for the example the order is given by $x_1 < x_2 < x_3$. For each level $l$ with $l \geq 0$, the BDD consists of $2^l$ internal nodes.

The number of internal nodes can be reduced by generating the *reduced* BDD depicted in Figure 5.1b. Note, we use the term BDD to refer to Reduced Ordered Binary Decision Diagrams (ROBDDs).

**ROBDD with Attributed Arcs.** Implementations of BDD libraries can be optimized (e.g., reducing the memory requirements) by introducing *complement arcs* realized by attaching attributes to the BDD's arcs [Som99]. All arcs without any attached attributes are called *regular arcs*. Hence, assuming we have given a function F, we can compute and represent the inverse function $\mathsf{G} = \bar{\mathsf{F}}$ by annotating the corresponding arc indicating that G is the complement of F. To this end, using attributed arcs in BDDs allows to reduce the constant outputs (i.e., terminal nodes) to just one single output (e.g., 1).

**Formal Definition of BDDs.** Given this more visualized description, we present a more formal definition of BDDs in this paragraph based on the notations of Somenzi [Som99]. Therefore, we start by recalling important definitions and theorems from Boolean algebra $B$ with $B = \{0, 1\}$. We denote conjunctions by $\cdot$, disjunctions by $+$ and negations by $\bar{x}$. We first define the cofactors of a Boolean function F in Definition 23.

**Definition 23** (Cofactors)**.** *Lef* $F(x_1, ..., x_n)$ *be a Boolean function. Then*

$$F_{x_i} = F(x_1, ..., x_{i-1}, 1, x_{i+1}, ...., x_n)$$
$$F_{\bar{x_i}} = F(x_1, ..., x_{i-1}, 0, x_{i+1}, ...., x_n)$$

*are the* positive *and* negative *cofactors of* F *with respect to* $x_i$*.*

The cofactors of a Boolean function $\mathsf{F}$ can be used for Shannon's expansion theorem as defined in Theorem 1.

**Theorem 1** (Shannon's Expansion)**.** *Let $\mathsf{F}(x_1, ..., x_n)$ be a Boolean function. Then*

$$\mathsf{F}(x_1, ..., x_n) = \bar{x_i} \cdot \mathsf{F}_{\bar{x_i}} + x_i \cdot \mathsf{F}_{x_i}.$$

Shannon's expansion theorem is one fundamental building block for defining BDDs which is given below.

**Definition 24** (Reduced Ordered Binary Decision Diagram [Som99])**.** *A ROBDD represents a vector of Boolean functions $\hat{\mathsf{F}}$ and is modeled as a DAG $(\mathcal{V} \cup \mathcal{F} \cup \{1\}, \mathcal{E})$. The nodes of the DAG are partitioned into three subsets. Internal nodes are described by $\mathcal{V}$ where each node $v \in \mathcal{V}$ has an outdegree of two. Every node $v$ has a label $l(v)$ in the support of $\hat{\mathsf{F}}$. The function nodes are denoted by $\mathcal{F}$ where each function $\mathsf{F} \in \mathcal{F}$ has an outdegree of one and an indegree of zero. Eventually, the set $\{1\}$ denotes the terminal node 1 with outdegree zero. The function nodes are in one-to-one correspondence with the components of $\hat{\mathsf{F}}$. The outgoing arcs of function nodes $\mathsf{F} \in \mathcal{F}$ may have the complement attribute. The two outgoing arcs for a node $v \in \mathcal{V}$ are labeled $T$ and $E$ where the $E$ arc may have the complement attribute. The notation $(l(v), T(v), E(v))$ is used to access an internal node $v \in \mathcal{V}$ and its two outgoing arcs. The variables in the support of $\hat{\mathsf{F}}$ are ordered, i.e., if $v_j$ is a descendant of $v_i$ with $v_j, v_i \in \mathcal{V}$, then $l(v_i) < l(v_j)$. The function vector $\hat{\mathsf{F}}$ represented by the BDD is defined by the following properties.*

*(1) The function of the terminal node is the constant function* 1*.*

*(2) The function of a regular arc is the function of the head node. The function of a complement arc is the complement of the function of the head node.*

*(3) The function of a node $v \in \mathcal{V}$ is given by $\bar{l}(v)\mathsf{F}_E + l(v)\mathsf{F}_T$ where $\mathsf{F}_T$ and $\mathsf{F}_E$ are the functions for $T(v)$ and $E(v)$, respectively.*

*(4) The function of $\mathsf{F} \in \mathcal{F}$ is the function of its outgoing arc.*

As already mentioned above, BDDs are canonical in their representation.

**Theorem 2** (Canonicity of BDDs)**.** *BDDs are canonical, i.e., the representation of a vector of Boolean functions $\hat{\mathsf{F}}$ is unique for a given variable ordering, if:*

*(1) There are no distinct internal nodes $v_1$ and $v_2$ such that $l(v_1) = l(v_2)$, $T(v_1) = T(v_2)$, and $E(v_1) = E(v_2)$.*

*(2) For every node, $\mathsf{F}_T \neq \mathsf{F}_E$.*

*(3) All internal nodes are descendants of some node in $\mathcal{F}$.*

The proof for Theorem 2 is given in [Som99]. However, all these properties allow efficient manipulations of BDDs (applying Boolean operations) and to check a given Boolean function $\mathsf{F}$ for satisfiability.

**Satisfiability Problem.** If a variable assignment of a target Boolean function $\mathsf{F}$ evaluates to 1, the assignment is called *satisfying assignment*. Checking a function for satisfiability is trivial and can be accomplished in constant time on BDDs since it is sufficient to check the BDD for constant 0. Another interesting problem is to count the number of satisfying assignments which is especially important for this thesis in Part IV. For the example given in Figure 5.1, $\mathsf{F}(x_1, x_2, x_3)$ has four satisfying assignments. However, this result depends on the number of variables influencing the function. For example, $\mathsf{F}(x_1, x_2, x_3, x_4) = x_1 \cdot x_2 \oplus x_3$ has eight valid satisfying assignments.

We briefly outline an algorithm for computing the number of satisfying assignments based on the description from [Som99]. For each node $v \in \mathcal{V}$, the algorithm computes the number $\alpha_v$ of satisfying assignments for the Boolean function represented by $v$. The same holds for each arc $e \in \mathcal{E}$ denoted by $\alpha_e$. Additionally, we assume that the number of variables is given by $n$. The index of the function node is 0 while the index of the terminal node is $n + 1$. If the node $v$ is the terminal node, $\alpha_v = 2^n$. When determining $\alpha_e$ for an arc, we have to distinguish between regular arcs and complement arcs. Assuming two nodes $v, w \in \mathcal{V}$ connected by a regular arc $e$, $\alpha_e = \alpha_w$. If $v$ and $w$ are connected by a complement arc, $\alpha_e = 2^n - \alpha_w$. To this end, for an internal node $v \in \mathcal{V}$ holds

$$\alpha_v = \frac{\alpha_T + \alpha_E}{2}.$$

Hence, the algorithm can be realized by a post-order traversal through the given BDD.

# Part II

# Protecting Hardware Implementations against Physical Attacks

# Chapter 6

# Orthogonal Concurrent Error Correction

*Countermeasures against FIA and SCA attacks are often of different nature and follow design principles that are not necessarily compatible, hence combining countermeasures to resist both threats simultaneously is a challenging task and requires a careful design to preclude mutual levering due to unexpected interplay.*
*The contribution in this chapter is twofold and based on joint work with Florian Bache, Pascal Sadrich, and Tim Güneysu [RSBG20]. First, we present a novel orthogonal layout of linear ECCs to adjust classical Concurrent Error Detection (CED) to an adversary model that assumes precisely induced single-bit faults which, with a certain non-negligible probability, will affect adjacent bits. Second, we combine our orthogonal error correction technique with a state-of-the-art SCA protection mechanism to demonstrate resistance against both threats.*
*Eventually, using AES as a case study, our approach can correct entirely faulted bytes while it does not exhibit detectable first-order side-channel leakage using* 200 *million power traces and TVLA as state-of-the-art leakage assessment methodology. Furthermore, our hardware implementations reduce the area and resource consumption by* 14.9 % – 18.3 % *for recent technology nodes (compared to a conventional CED scheme).*

**Contents of this Chapter**



## 6.1 Introduction

Due to the advances in more cost-efficient equipment and more experienced adversaries, protection against FIA eventually started to gain more attention in recent years. Common approaches

to increase protection against FIA mainly follow the concepts of (concurrent) error detection, error correction, or recently proposed, infective computation [GST12]. In particular, CED based on redundancy in time or area (often in terms of error detecting codes) has been researched extensively and established as the main principle to design and implement FIA countermeasures.

In contrast, passive SCA analyzes the dependencies of processed sensitive information in observable physical characteristics, e.g., power consumption [KJJ99] of the device that runs the cryptographic implementation. As introduced in Section 2.1, an adversary might be able to extract sensitive information from the device through statistical analysis of side-channel measurements. Hence, due to the complexity of this approach, many attacks with different capabilities and complexities have been proposed and as a consequence, a similar wide range of countermeasure variations have been developed. In particular, masking (based on secret sharing concepts) is a promising approach due to its sound theoretical foundation and examination.

However, since techniques to thwart SCA and FIA usually follow design strategies that often are incompatible, combining both countermeasures in the same design is a challenging task. In particular, ensuring that the interplay of both approaches does not reduce the security of the device, requires careful implementations of the different mechanisms.

### 6.1.1 Contribution

Our contribution in this chapter is twofold: In a first step, we present an alternative approach to enhance the correction capabilities of classical Concurrent Error Correction (CEC). In particular, we refrain from applying a traditional, word-oriented encoding to a cryptographic design but choose an orthogonal pattern for the linear ECC instead. Benefiting from the fact that each bit of one data word will be encoded by different code words, we can correct multiple and adjacent bits of a single word which is not possible using a traditional layout. Assuming that decreasing technology sizes pose a huge challenge to adversaries precisely injecting *single-bit faults* and faulting adjacent bits becomes more likely, our approach will be at a clear advantage over classical schemes demonstrated in a dedicated case study.

As a second step, we extend our design to validate our claims with respect to resistance against SCA by modifying the basic architecture following the principles of Boolean masking. This second case study then demonstrates the effectiveness of our new scheme in combination with LMDPL as the SCA countermeasure of choice to resist both FIA and SCA. In particular, we use state-of-the-art leakage assessment strategies based on TVLA to demonstrate that our design does not exhibit detectable leakage using up to 200 million power measurements while providing error correction capabilities at the same time.

### 6.1.2 Related Work

Since the introduction of DFA in 1997 [BS97], researchers aimed to improve fault-injection attacks and countermeasures likewise [Gir04, AM11]. Using statistical methods under the assumption of a non-uniform fault distribution, SFA [FJLT13] became a powerful tool to break many cryptographic implementations. Dobraunig et *al.* recently proposed SIFA [DEK+18], a combination of SFA with IFAs [Cla07] which has been presented as an attack that is capable to break implementations even in the presence of sophisticated countermeasures against FIA.

With advancing attack techniques, new countermeasures were proposed at the same pace. In [BCN+06], various methods reaching from simple duplication to more advanced schemes

applying Multi Duplication with Comparison (MDC) structures have been presented. Since simple duplication schemes are limited in security, recent publications deal with the more sophisticated application of linear ECCs [SMG16, AMR+20, SRM20] or even non-linear robust codes [RNK19] to address more advanced adversary models. In addition, only a few approaches [BKHL20, SJR+19, SRM20] are known to prevent successfully SIFA by changing the fault distribution.

Research investigating the properties of combined countermeasures (against SCA and FIA) gained more interest in recent years. To address this problem, Schneider et *al.* [SMG16] first applied ECCs to TIs. Later, Reparaz et *al.* [RMB+18] used techniques from Multi-Party Computation (MPC) to simultaneously implement protection against SCA and FIA. Recently, De Meyer et *al.* [MAN+19] combined masking with Message Authentication Code (MAC) tags originating from information theory to extend the protection against FIA.

## 6.2 Preliminaries

In the following section, we briefly introduce the considered adversary model and justify the capabilities and limitations of this adversary. Afterwards, we recap the concept of CED.

### 6.2.1 Adversary Model

As introduced in Chapter 2, cryptographic implementations can be broken using various types of *side channels* or *precise fault injections* [KJJ99, ADN+10, ZDCT13, SA02, RSDT13, CLFT14, SBHS15]. Hence, in the case of side-channel attacks, we assume an adversary based on the $d$-probing model (cf. Section 2.1.2).

Considering FIAs, not only *precisely injected single-bit faults* can be used to recover secret information but instead many results also confirm that even *arbitrarily injected faults* can succeed to exploit such vulnerabilities [AM11, DEK+18, SJR+19]. Furthermore, common fault injection techniques, e.g., using electromagnetic pulses or clock glitches, cause sampling faults [DLM19, ADN+10]. Hence, such attack vectors do not tamper the combinatorial logic of a digital circuit but rather disturb the sampling process of a flip-flop or decrease the clock period leading to wrong values stored in the register cell. Additionally, considering continuously shrinking geometry sizes for ICs, [SBHS15] investigates the precision of single-bit faults based on laser fault injection. The findings show that precisely injecting faults into advanced technology nodes is becoming more challenging, the smaller the geometry gets. In particular, the injection of faults in register cells becomes highly dependent on adjacent cells and their current value. To this end, we assume an adversary that is capable to precisely inject single-bit faults but with a certain, non-negligible probability will affect and change adjacent and related cells (belonging to the same part of the internal state). Moreover, since attack vectors like laser [SHS16] or electromagnetic fault injections are not limited to a single laser or coil, we also consider cases where an adversary has two synchronized lasers or coils available for precise fault injections at two different locations or points in time. For these reasons, we follow the biased fault model defined in [SMG16] using the biased distribution $\mathcal{E}_{B_b}$ with the subsets

$$\begin{aligned} E_1 &= \{e \mid e \in E \wedge \mathrm{HW}(e) \leq b\} \text{ with } Pr[e \in E_1] = 1 \\ E_2 &= \{e \mid e \in E \wedge \mathrm{HW}(e) > b\} \text{ with } Pr[e \in E_2] = 0. \end{aligned} \tag{6.1}$$

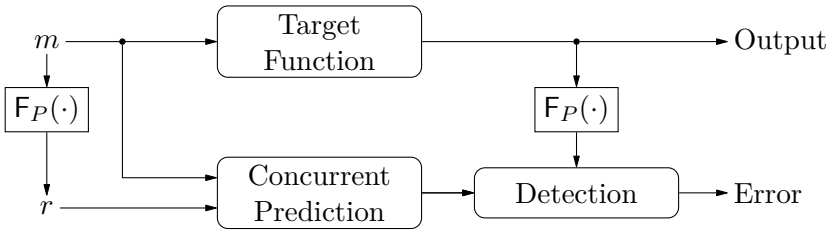



Figure 6.1: Schematic concept of Concurrent Error Detection.

Here, $E$ denotes the error variable and $b$ the maximum number of faults injected in a defined area represented by an error vector $e$. All possible error vectors in $E_1$ are assumed to be equally distributed.

Finally, we distinguish two different scenarios: (1) the adversary is able to inject $b$ faults into one single nibble or byte (e.g., using a single laser, an electromagnetic pulse, or clock glitches) based on the fault distribution $\mathcal{E}_{B_b}$ and (2) the adversary is able to inject $b$ faults in up to two different nibbles or bytes also following $\mathcal{E}_{B_b}$.

### 6.2.2 Principles of Concurrent Error Detection

Concurrent Error Detection (CED) is a generic concept that can be applied to any *target functions* to achieve fault tolerance (i.e., a correct functionality even under fault occurrence). In particular, CED allows to continuously monitor the execution of the *target function* and detects all covered faults during operation.

Figure 6.1 outlines the main principle of CED based on spatial redundancy. A naive instantiation of this principle uses duplication of the *target function* for the *concurrent prediction* with $\mathsf{F}_P$ instantiated as the identity function and the message passed unmodified to the *prediction*. Comparing the output of *target function* and *prediction* then allows to detect any fault occurrence as a mismatch. Depending on the targeted security level, this scheme can be extended beyond duplication by instantiating multiple functions in parallel or computing more complex functions on $\mathsf{F}_P$, e.g., parity bits for parts or even the entire state [KKG03, BBK$^+$03]. To this end, a more sophisticated approach applies systematic linear codes still using an unmodified *target function* but instead of duplication just processes the encoded part of the code for *concurrent prediction*. Also, if the code is chosen such that the minimum distance is $d_{\min} \geq 3$, the capabilities of the code will be extended from *detection* to *correction* allowing to correct one (or more) faults per code word and continue operation successfully (cf. Section 3.1).

Eventually, the complexity of the *concurrent prediction* depends on the *target function* and the chosen code. In general, using linear ECCs, linear functions are easier to encode than non-linear functions and in case choosing $n = i \cdot k$ with $i \in \mathbb{N}$ and $i \geq 2$, it will be possible to predict the output without knowledge of $m$. For more details, we refer the interested reader to [AMR$^+$20, SRM20].

## 6.3 Design Concept

In this section, we first outline our basic design considerations and argue for our final code selection before we discuss the application of linear ECCs and challenges when combining our concept with SCA countermeasures.

### 6.3.1 Design Considerations

In general, our concept addresses efficiency (i.e., area and performance) and security of hardware platforms while mostly disregarding software implementations.

Given our adversary model, we implement a correction mechanism for actively injected faults instead of simple fault detection, since reliability and fault tolerance are important features of modern cryptographic implementations. However, adequate coverage of expected faults (based on the distribution in Equation 6.1) is only possible when revisiting the conventional application of ECCs.

For classical encoding schemes, the correction within a single data word is limited by the Hamming distance of the ECC and the parameter $v$ introduced in Corollary 1. An adversary injecting faults that affect adjacent bits (e.g., due to shrinking geometry sizes) rapidly exceeds the correction capabilities of classical schemes. Since increasing the correction capabilities of the ECC easily raises the costs to an unacceptable level, our novel orthogonal design approach allows to keep the area footprint as small as possible.

Additionally, we do not only consider the protection against FIA but also address combined resistance against fault injections and (power) side channels. Selecting appropriate masking schemes for protection against SCA, in particular the transformation of non-linear sub-functions becomes more challenging and more expensive. For this reason, we decided to focus on linear and systematic codes instead of robust codes, as robust codes introduce additional non-linearity which hampers the efficient implementation of mask-based protection mechanisms against SCA.

### 6.3.2 Code Criteria

The detection and correction capabilities eventually depends on the choice of the parameters $n$, $k$, and $d_{\min}$, while improved capabilities directly increase the area overhead in hardware. Hence, it is essential to carefully consider the following steps for the final choice of the parameters to balance capabilities and expenses:

1. **Selection of $k$:** For traditional schemes, $k$ is often deduced from the word size within the *target function* but following the orthogonal encoding scheme (cf. Section 6.4) also allows other choices for $k$. In fact, $k$ is only required to be a *divisor of the state size* to ensure efficient, non-overlapping encodings using $P$, i.e., the encoding matrix of a systematic code as defined in Definition 14.

2. **Selection of $n$:** To allow an independent data path for the redundancy $r$ according to the CED principle (cf. Section 6.2.2), the codeword length $n$ should be selected such that $n = i \cdot k$ with $i \in \mathbb{N}$ and $i \geq 2$ where $i$ determines the desired security level and the corresponding implementation overhead.

3. **Selection of $d_{\min}$:** Given $n, k$, the selection of the generator matrix $G$ should be made with regard to maximize the minimal distance $d_{\min}$.

4. **Construction of $G$:** Given $n, k, d_{\min}$, the construction of the final generator matrix $G$ should be based on *systematic codes* with a non-singular part $P$. This ensures the efficient inversion of internals at any point in time.

### 6.3.3 Correction

This section briefly describes the correction mechanism when using linear ECCs. We denote a faulty message by $m' = m \oplus e_1$ and a faulty redundant part by $r' = r \oplus e_2$, where $e_1$ and $e_2$ represent error vectors with $\mathrm{HW}(e_1) + \mathrm{HW}(e_2) \leq v$ to allow correction of the occurred faults. To calculate the syndrome $s$, the faulty message and redundancy are concatenated to $c' = [m' \mid r']$ inherently providing a concatenation of the error vectors as $e = [e_1 \mid e_2]$. Eventually, the transposed syndrome is derived by

$$s^{\mathrm{T}} = H \cdot c'^{\mathrm{T}} = H \cdot (c \oplus e)^{\mathrm{T}} = H \cdot e^{\mathrm{T}}. \tag{6.2}$$

This formula points out that all possible errors are encoded by the parity check matrix $H$. Additionally, since we are limited by the number of correctable errors due to the code parameter $d_{\min}$, the fault-free case and all syndromes with the corresponding faults can be perfectly stored in a Look-Up Table (LUT) with $S$ entries where

$$S = 1 + \sum_{i=1}^{v} \binom{n}{i} \tag{6.3}$$

Generally, for a valid code word $c$ holds $0 = H \cdot c^{\mathrm{T}} = H \cdot G^{\mathrm{T}} \cdot m^{\mathrm{T}}$ which implies $H \cdot G^{\mathrm{T}} \stackrel{!}{=} 0$. Since we specified to use systematic codes, $H$ can be simply set to $H = \left[P^{\mathrm{T}} \mid I_{n-k}\right]$. It turns out that the calculation of the syndrome simplifies to an encoding of $m'$ by $P^{\mathrm{T}}$ and a subsequent addition (modulo two) with $r'$.

Additionally, the position of the correction modules in the target cipher should be well-considered. First, a correction module should be placed before every *non-linear function*. If an erroneous word is fed into a non-linearity, the error propagation will be hardly predictable and could eventually prevent the correction. Second, the inputs of any function that combines code words to produce a new code word should be fault free. Otherwise, faults from different sources can be accumulated and a correction might become unfeasible.

### 6.3.4 Combination with SCA Countermeasures

In this section, we identify challenges and requirements that should be considered when combining our technique with protection mechanisms against SCA.

**Sequential logic:** We assume faults occur with a higher probability in sequential logic (since effective faults eventually manifest in registers). Hence, our approach focuses on the protection of the data-dependent flip-flops whose number should be minimized for better efficiency of the correction modules.

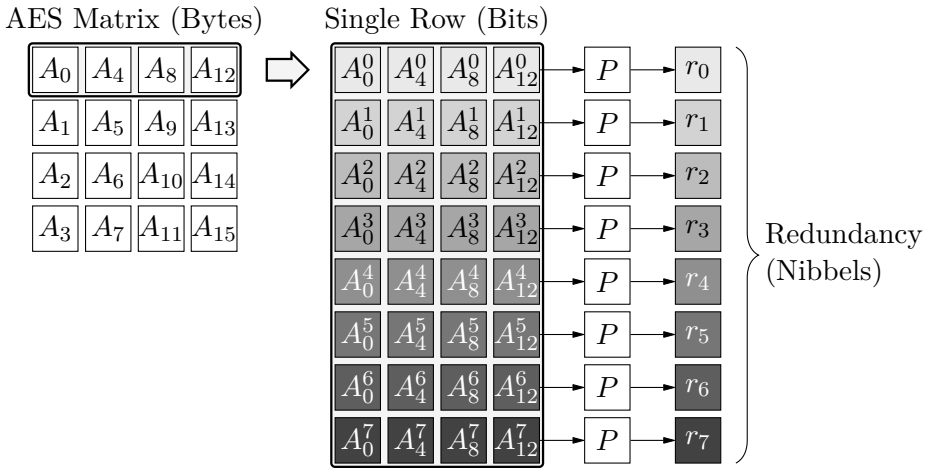



Figure 6.2: Orthogonal encoding scheme showed exemplary for the first row.

**Non-linear functions:** Due to our rearranged scheme, all code words are generated from $k$ bits of different words and each non-linear function $\mathsf{F} : \mathbb{F}^w \mapsto \mathbb{F}^w$ has to be combined with a preceding decoding $\mathsf{F}_P^{-1}$ and a succeeding encoding $\mathsf{F}_P$ which introduces overhead with every additional non-linear function between register stages.

**Hardware primitives:** Since our approach protects all data dependent registers, we cannot apply countermeasures based on hardware primitives or macros without direct access to register values (e.g., given for BRAM modules in modern FPGAs).

## 6.4 Case Study A: FIA Countermeasure

This section presents a practical case study using AES and discusses important features of our hardware design.

### 6.4.1 Encoding Procedure

As mentioned in Section 6.3.2, our implementation is not restricted to the underlying functions of AES, hence we choose $k = 4$ and encode the AES state matrix in an orthogonal orientation to the state bytes. The encoding is performed on $\begin{bmatrix} A_i^j & A_{i+4}^j & A_{i+8}^j & A_{i+12}^j \end{bmatrix}$ where $A_l$ denotes a single byte and $A_l^j$ a single bit of the corresponding byte with $0 \leq i \leq 3$ and $0 \leq j \leq 7$. As an example, the encoding of the first row of the AES state matrix is shown in Figure 6.2. For each row, the encoding scheme produces eight redundant nibbles $r_j$ which are processed by a concurrent predictor. To this end, the approach requires $4 \cdot 8 = 32$ generators $P$ in total to encode the entire 128-bit state in parallel.

In contrast to our chosen encoding scheme applied to single bits out of the same row, it is also conceivable to apply it to single bits out of the same column. Generally, this approach can be realized with the same error coverage as for the introduced scheme based on a row-encoding. However, since `ShiftRows` works on the AES state matrix's rows, a column aligned design would increase complexity and hence result in an increased area consumption.

### 6.4.2 Code Selection

Given $k = 4$ and the fact that we opted for $n = 2k$, we performed an exhaustive search over all potential matrices $P$ that provide a minimum Hamming distance $d = 4$ and are invertible[1]. It turns out that the matrix

$$P = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

and all $4! = 24$ possible arrangements of these four rows and columns are the only matrices that fulfill our requirements. However, because the above-stated matrix $P$ is also self-inverse, we decide to use it for our countermeasure as it simplifies the implementation process.

A direct consequence of choosing a self-inverse matrix $P$ is that instead of using the parity check matrix $H = \left[P^{\mathrm{T}} \mid I_k\right]$, we can apply the inverse of the transposed $P$ to the redundant part such that $H$ can be adapted to $\tilde{H} = \left[I_k \middle| \left(P^{\mathrm{T}}\right)^{-1}\right] = [I_k \mid P] = G$ while the syndrome $s$ is still zero $\forall c \in \mathcal{C}$. As explained below, this method allows particular optimizations when implementing the predicting module for `SubBytes`. Note, however, that a different LUT is required to correct the corresponding faults although the fault coverage is left unaltered.

### 6.4.3 Concurrent Prediction

For concurrent prediction of the AES encryption using the orthogonal encoding scheme, all sub-functions have to be reconsidered before application to the redundant part of the internal state. In particular for the byte-oriented, non-linear sub-functions, application of the orthogonal encoding within the concurrent prediction becomes more challenging and requires careful consideration.

**Key Addition.** Predicting the output of the key addition is straightforward and requires no modification of the addition module, assuming that the round key encoding follows the same orthogonal encoding principle as the state. Due to the linearity of the chosen code and the key addition operation, the addition prediction can be performed concurrently on the redundancy using the same addition operation (XOR).

**Shifting of Rows.** Fortunately, the predictor for `ShiftRows` also comes without any additional costs mainly due to the inherent structure of the chosen code. All left-shifting of each particular row can be applied independently of the remaining rows and performed bit-wise such that each word $\begin{bmatrix} A_i^j & A_{i+4}^j & A_{i+8}^j & A_{i+12}^j \end{bmatrix}$ can be considered independently. Hence, the successive

[1] We decided to choose a code with a minimum distance $d_{\min} = 4$, since, in addition to the 1-bit error correction, this also provides capabilities for 2-bit error detection.

operations of *decoding*, *left-shifting*, and *encoding* given as $P \circ LS \circ P^{-1}$ can be combined into a single operation. In particular, given the previously chosen $P$ eventually leads to

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

which can be implemented as simple *right-shifting* of each particular row (instead of the original *left-shifting*).

**Byte Substitution.** Applying a *conventional* encoding scheme to `SubBytes` could be realized by merging the preceding decoding step with the actual execution of the substitution and a subsequent encoding into an adapted S-box. Assuming an implementation of `SubBytes` as LUT, both realizations would require the same hardware resources. However, in the case of our applied *orthogonal* encoding scheme, we are not able to merge these three steps into one recomputed LUT as every S-box input depends on eight different codewords. Instead, we can optimize our implementation by adapting the preceding correction module using $\tilde{H}$ (cf. Section 6.4.2) such that the inputs to our modified substitution layer are already decoded during the correction process. Hence, no dedicated initial decoding is required and the adapted module just contains the original S-box and a successive orthogonal encoding.

**Mixing of Columns.** For a *conventional* encoding scheme $\mathsf{G}_P$ the predictor of `MixColumn` would be realized by a preceding decoding of the redundancy $r$ and re-encoding after executing the actual operation as depicted in Equation 6.4.

$$MC' = \mathsf{G}\left(MC\left(\mathsf{G}_P^{-1}(r)\right)\right) \tag{6.4}$$

Obviously, these three operations do not have to be performed separately but rather can be merged into one modified `MixColumn` $MC'$. This would reduce additional hardware costs but still exceed the resources compared to an unmodified implementation.

However, our proposed encoding scheme does not require any modifications of `MixColumn` to predict the output and can be used unaltered. This benefit comes from the fact that $\mathsf{F}_P\left(MC\left(A\right)\right) = MC\left(F_P\left(A\right)\right)$ holds for the applied orthogonal encoding function $\mathsf{F}_P$ and any arbitrary $P$. Each bit of a single byte is multiplied by the same bit of the encoding matrix $P$ and it does not matter if the multiplication happens before or after a multiplication in GF $(2^8)$.

### 6.4.4 Implementation Overview

Figure 6.3 shows a schematic overview of our proposed implementation. The left part represents the encryption path and is left unaltered compared to a parallel, round-based implementation of AES (except for the inserted correction module). We decided to use a two-stage pipeline version to correct faults before `SubBytes` and `MixColumn` to prevent undesired fault propagation (cf. Section 6.3.3).

The right part represents the prediction circuit where the plaintext is initially transformed by $P$ to provide the redundant parts of the code words. As mentioned before, the execution of

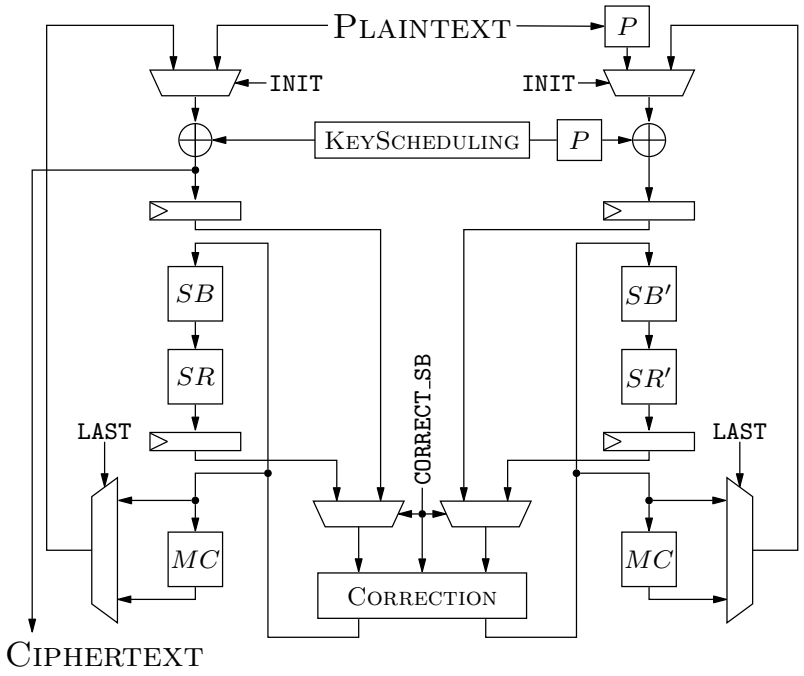



Figure 6.3: Schematic overview of our proposed scheme.

the key addition requires another encoding module to encode the corresponding round key prior to the addition. The predicting sub-function of `SubBytes` $SB'$ contains the original AES S-box followed by an encoding matrix $P$ as explained in Section 6.4.3. As a first step of the diffusion layer, the $SR'$ module predicts the `ShiftRows` operation performing *right-shifting* instead of *left-shifting* as explained in Section 6.4.3. Before the second step of the diffusion layer prediction is performed using an unaltered `MixColumns` module, the intermediate result of the `ShiftRows` prediction is buffered in a register stage and used for the correction inside the shared correction module.

Also, we would like to emphasize that Figure 6.3 only shows an implementation with a single and shared correction module in the data path. In general, this strategy provides a realization with a smaller area overhead but reduced throughput. A doubling of the throughput can be achieved when spending more area and implementing two independent correction modules, one per register stage (located at the outputs of the registers) as the design allows pipelining.

### 6.4.5 Correction Module

Above, we mentioned that the decoding step before applying `SubBytes` can be merged with the operations in the correction module by modifying the correction process from Section 6.3.3 using $\tilde{H}$ as parity check matrix. In case of implementing our approach using only one correction module, it would be desirable to apply the same LUT for correcting faults after `AddKey` and `ShiftRows` in order to keep the footprint low. To this end, the decoding is done by multiplying the redundancy $r_{\text{in}}$ with $P$ as explained above and depicted in Figure 6.4. To determine the corresponding error vector $e = [e_1 \mid e_2]$, the input data $m_{\text{in}}$ and decoded redundancy $\tilde{r}_{\text{in}} = P \cdot r_{\text{in}}$ are added and afterwards fed into the LUT. The correction in the encryption path can be

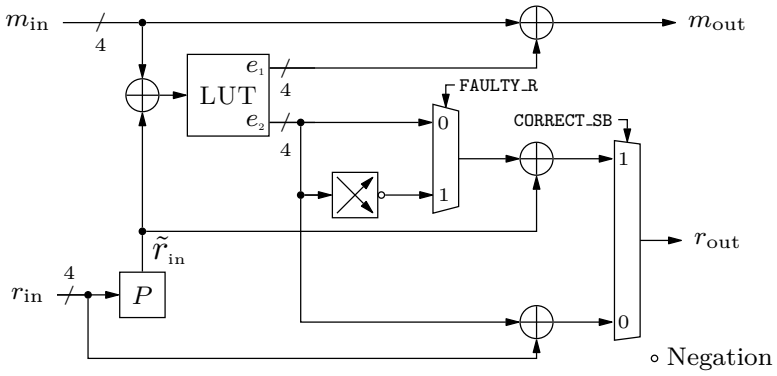



Figure 6.4: Improved correction module to lower costs for $SB$.

accomplished by adding the first part of the error vector $e_1$ to $m_{\text{in}}$. For the redundancy we consider three cases:

(1) The correction is done after `AddKey` such that the outputs $m_{\text{out}}$ and $r_{\text{out}}$ serve as inputs to `SubBytes` and a fault has not occurred in the redundant path (`FAULTY_R` $= 0$ and `CORRECT_SB` $= 1$)

(2) The correction is done after `AddKey` such that the outputs $m_{\text{out}}$ and $r_{\text{out}}$ serve as inputs to `SubBytes` and a fault occurred in the redundant path (`FAULTY_R` $= 1$ and `CORRECT_SB` $= 1$)

(3) The correction is done after `ShiftRows` such that the outputs $m_{\text{out}}$ and $r_{\text{out}}$ serve as inputs to `MixColumns` (`FAULTY_R` $=$ X and `CORRECT_SB` $= 0$)

While `CORRECT_SB` can be easily determined by tracking the current position of valid data in the algorithm flow, `FAULTY_R` is only true if the error vector $e_2$ is non-zero. In case, both control signals are in a true state, a fault occurred in the redundant path and the corrected output should be returned for processing in `SubBytes`. Hence, the already decoded redundant input needs to be corrected and fed to the output to get rid of the decoding module before applying the substitution. As the LUT is generated for correcting faults on the input data $m_{\text{in}}$ and $r_{\text{in}}$, the actual error vector $e_2$ has to be transformed so that it can be used to correct faults on the decoded redundant data. Due to the advantageously chosen matrix $P$, this transformation can be realized by reversing and subsequently negating $e_2$ as for all one-bit error-vectors $e_{\text{one}}$ the matrix $P'$ describes a flip of every bit (cf. Equation 6.5).

$$\bar{e}_{\text{one}} = P^{-1} \cdot e_{\text{one}} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \cdot e_{\text{one}} = \underbrace{\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}}_{P'} \cdot \underbrace{\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}}_{P''} \cdot e_{\text{one}} \tag{6.5}$$

Finally, the reversed and negated error vector $\bar{e}_2$ can be added to the decoded redundancy $\tilde{r}_{\text{in}}$ and afterwards can be used in `SubBytes` in the redundant path.

In case no fault occurred in the redundant path, $e_2$ is the zero-vector and the decoded redundancy is released to the output $r_{\text{out}}$. The case where `CORRECT_SB` is zero can be easily covered

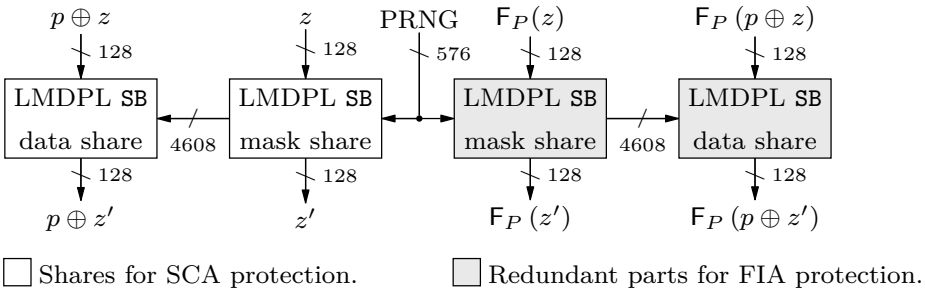



Figure 6.5: Combined protection of SB against SCA and FIA.

without using the output of the multiplexer controlled by FAULTY_R as shown in Figure 6.4. The error vector $e_2$ is directly routed to an XOR-operation adding a zero-vector in case no fault occurred or removing an error in $r_{\text{in}}$ in case $e_2$ is non-zero.

## 6.5 Case Study B: Combined Protection

We now extend our approach by adding a countermeasure against SCA. Before we describe the combined design, we briefly justify the selection of the chosen masking scheme.

### 6.5.1 Countermeasure Selection

In Section 6.3.4 we defined three requirements for an efficient combination of our proposed orthogonal encoding scheme with a countermeasure against SCA. Based on this list, we decided to implement the LMDPL scheme from [LMW14, SBHM20]. LMDPL is a masking technique which was specially designed to avoid glitches and early propagation. The implementation is realized on gate-level such that each elementary gate of a target function can be replaced by a protected gate. Therefore, the structure of a cipher can be left unaltered and non-linear functions do not have to be separated by several pipelining stages. This perfectly matches our first and second requirements avoiding additional register stages and additional non-linearities. Furthermore, LMDPL requires registers around a non-linear function F to allow precharging its combinatorial logic. Since our design already includes a pipeline stage before and after SubBytes, LMDPL seems to be a natural fit. Finally, LMDPL is a generic approach and does not require any platform-specific primitives (e.g., BRAM in FPGAs) which meets our last requirement. All register outputs are directly accessible and can be routed through a correction module introduced in Section 6.4.5. Even though LMDPL does not allow to use pipelining, we decided to avoid the routing to the correction module through the multiplexers (cf. Figure 6.3) and instantiated two correction modules right after both register stages.

### 6.5.2 Combined Approach

Since LMDPL is based on masking, both the encryption and the redundant path have to be modified and split into masked shares to protect our proposed FIA countermeasure against SCA. Figure 6.5 shows the realization of SubBytes as it is the most interesting part and the only part equipped with dual-rail logic. All in all, we get a design with four different data

paths. The leftmost one processes the intermediate results of the data share while the path right beside generates new table values from $z$. To generate updated table values for all 16 S-boxes in parallel, the design requires 576 bit fresh randomness every two clock cycles provided by a Pseudorandom Number Generator (PRNG) based on KECCAK [BDPA13]. Like the mask $z$ this randomness is reused in the redundant part containing our proposed scheme. Both the data and the mask shares are protected against FIA by the rightmost module and the second rightmost, respectively. To avoid any demasking in the correction process, each share is corrected separately on the masked data.

One important key feature of LMDPL is the precharge phase of combinatorial logic placed in the S-boxes. In the original work, this precharge phase is realized by resetting the input and table registers such that all gates and signals will be forced to zero. However, our orthogonal encoding scheme includes corrections modules right after every data-dependent register. Even if our design would reset these registers, this would still result in glitches due to the optimized correction module. Hence, we realized the precharge phase by a multiplexer which either precharges the LMDPL logic with zeros or forwards the current state. To avoid any glitches and ensure that all input signals to the multiplexer are settled, we triggered the precharge signal by the falling edge of the clock. Furthermore, this modification allows us to remove the high amount of registers storing the table values. The mask share is now synchronized with the data share and the computed table values can be used in the same clock cycle at the falling edge of the clock. For a more optimized implementation regarding the critical path a second clock domain (e.g., a faster or phase-shifted clock) could be used instead.

## 6.6 Evaluation

In this section, we present and discuss the implementation results for FPGAs and ASICs. Moreover, we compare our approach to related work.

### 6.6.1 FPGA Implementation

For comparison reasons, we first implemented a countermeasure using linear ECCs arranged in a *conventional* way as we are not aware of any works that applied linear codes to AES (using a 20 nm Xilinx Kintex UltraScale FPGA). We opted for a $[16, 8, 5]$-code[2] to align the word size of $k = 8$ in AES with the generator input width. For both, the conventional and orthogonal CED, we implemented two variants optimized for either area (single correction module) or throughput (two correction modules). Note that we considered the *Independence Property* from [AMR+20, SRM20] for all our implementations.

Using one correction module, our approach outperforms the conventional countermeasure by 18.3 % in terms of CLB resources (as shown in Table 6.1) since the orthogonal arrangement allows to reuse `MixColumn` and the area for the LUTs to correct occurring faults is considerably smaller. Similarly, when adding a second correction module, the required CLBs for our novel design is decreased by 35.3 % compared to the conventional approach.

[2]Such a code can be found in the appendix of [BCC+14].

Table 6.1: FPGA implementation results (xcku035).

| | Resources | | | Performance | |
|---|---|---|---|---|---|
| | Logic | Memory | Area | Frequency | Throughput |
| | LUT | FF | CLB | MHz | MB/s |
| *Reference AES Implementation* | | | | | |
| Round-Based | 1 388 | 271 | 253 | 454.5 | 559.4 |
| *Classical CED with Correction* | | | | | |
| One Correction Module | 4 533 | 659 | 714 | 259.7 | 180.7 |
| Two Correction Modules | 5 756 | 659 | 911 | 303 | 404 |
| *Orthogonal CED with Correction (OCEC)* | | | | | |
| One Correction Module | 3 532 | 659 | 583 | 277.8 | 193.2 |
| Two Correction Modules | 3 717 | 659 | 589 | 339 | 452 |
| *Combined Approach* | | | | | |
| LMDPL + OCEC | 31 141 | 1 177 | 4 290 | 46.1 | 32.1 |

### 6.6.2 ASIC Implementation

Table 6.2 provides ASIC results for three different libraries. We started our synthesis with the Open Cell Nangate 15 nm library[3], but we also provide implementation results for Nangate 45 nm and UMC 90 nm libraries.

Again, our approach outperforms the conventional CED scheme by 14.9 % using the NGate15 library and one correction module. For two correction modules, we highlight two interesting points. First, in the 45 nm technology the area decreases slightly compared to the design using one correction module (due to better optimization during synthesis) while providing higher throughput due to pipelining. Second, the gap between the footprints of a conventional encoding and our proposed orthogonal scheme significantly increases.

### 6.6.3 Comparison to Previous Work

In Table 6.3 we compare our approach to already existing countermeasures. Starting in 2002, Karri et *al.* proposed a countermeasure against FIA for AES which is based on the inverse operations of the encryption algorithm [KWMK02]. In 2004, Bertoni et *al.* suggested to use parity schemes to detect faults [BBKM04]. Although these countermeasures come with small additional implementation costs regarding the area overhead, they do not provide an appropriate security level.

Compared to [SMG16] and [MAN+19] our design comes with roughly the same area overhead on a modern Xilinx UltraScale FPGA. In the 45 nm technology, our approach requires slightly less hardware resources. However, both approaches rely on fault detection and withhold the faulty ciphertext or perform an infective computation, respectively. Hence, all aforementioned types of countermeasures can be broken by SIFA as shown in [DEK+18].

[3] www.silvaco.com/products/nangate/FreePDK15_Open_Cell_Library/index.html

Table 6.2: ASIC implementation results.

| | NGate15 | | | NGate45 | | | UMC90 | | |
|---|---|---|---|---|---|---|---|---|---|
| | Area | Freq. | T'put | Area | Freq. | T'put | Area | Freq. | T'put |
| | kGE | MHz | MB/s | kGE | MHz | MB/s | kGE | MHz | MB/s |
| *Reference AES Implementation* | | | | | | | | | |
| Round-Based | 15.5 | 4 016 | 4 943 | 14.2 | 870 | 1 070 | 14.7 | 290 | 357 |
| *Classical CED with Correction* | | | | | | | | | |
| One Corr. Module | 39.9 | 2 360 | 1 642 | 36.2 | 490 | 341 | 37.1 | 137 | 96 |
| Two Corr. Modules | 46.7 | 3 416 | 4 555 | 42.4 | 625 | 833 | 44.2 | 191 | 255 |
| *Orthogonal CED with Correction (OCEC)* | | | | | | | | | |
| One Corr. Module | 33.9 | 1 634 | 1 137 | 31.4 | 543 | 378 | 31.8 | 160 | 111 |
| Two Corr. Modules | 34.2 | 3 417 | 4 556 | 31.2 | 787 | 1 050 | 32.5 | 260 | 346 |
| *Combined Approach* | | | | | | | | | |
| LMDPL + OCEC | 112.8 | 180 | 125 | 101.7 | 128 | 89 | 104.1 | 100 | 70 |

Only [BKHL20] and [SJR+19] provide security against SIFA. The former approach relies on modular redundancy and comes with a huge area overhead which was, however, not explicitly evaluated. The later work considers a combination of a transform-and-encode strategy while the encoding is realized by linear ECCs. Unfortunately, no target platform and no overhead were provided.

## 6.7 Security Evaluation

In this section, we first discuss the fault coverage of our proposed scheme and compare it with the coverage of conventional encoding (i.e., applying a $[16, 8, 5]$-code aligned to bytes) and Triple Modular Redundancy (TMR). Afterwards, we evaluate the resistance of our combined countermeasures against SCA.

### 6.7.1 Fault Coverage

To evaluate the resilience against FIA, we consider the adversary model defined in Section 6.2.1. Generally, the fault coverage of an arbitrary protection scheme is determined by Equation 6.6 where $F_{\text{not}}$ denotes the number of faults that cannot be corrected by a target countermeasure and $F_{\text{tot}}$ the total number of possible faults.

$$C(b) = 1 - \frac{F_{\text{not}}}{F_{\text{tot}}} \tag{6.6}$$

Furthermore, we do not distinguish between faults occurring in the encryption part or in the redundant part.

Table 6.3: Comparison to other countermeasures against FIA.

| Approach | Cipher | Target Platform | Area Overhead | Correction |
|---|---|---|---|---|
| Karri et *al.* [KWMK02] | AES[a] | Xilinx XCV1000BG | 20.97 % | NO |
| | AES[b] | Xilinx XCV1000BG | 18.90 % | NO |
| | AES[c] | Xilinx XCV1000BG | 38.08 % | NO |
| Bertoni et *al.* [BBKM04] | AES | STMicro 0.18 $\mu$m | 18.00 % | NO |
| Schneider et *al.* [SMG16] | LED | UMC 0.18 $\mu$m | 156 % | NO |
| De Meyer et *al.* [MAN+19] | AES | NGate 45 nm | 153 % | NO |
| Breier et *al.* [BKHL20] | GIFT-64[i] | – | > 200 % | YES |
| | GIFT-64[ii] | – | > 400 % | YES |
| Saha et *al.* [SJR+19] | Present | – | N/A | YES |
| **This Work** | AES | UltraScale xcku035 | 154 % | YES |
| **This Work** | AES | NGate 45 nm | 119 % | YES |

[a] Algorithm level. [b] Round level. [c] Operation level. [i] Single-Bit faults. [ii] Double-Bit faults.

#### Fault Coverage assuming Faults in one Byte

Initially, we discuss the first scenario where an attacker injects faults into one single byte following a biased distribution $\mathcal{E}_{B_b}$ (see Figure 6.6a). Our approach and a TMR based scheme can correct every possible fault. Our approach uses a $[8, 4, 4]$-code and is able to recover one entire faulty byte due to the orthogonal layout. TMR corrects any faults that occur in just one of the copies such that a coverage of 100 % is achieved (i.e., $F_{\text{not}} = 0$). Implementing a conventional encoding scheme for AES, leads to a decreased coverage for $b \geq 3$ since the underlying $[16, 8, 5]$-code is only capable of correcting up to two faults. Following Equation 6.6, the fault coverage $C_{\text{conv}}(b)$ is defined by

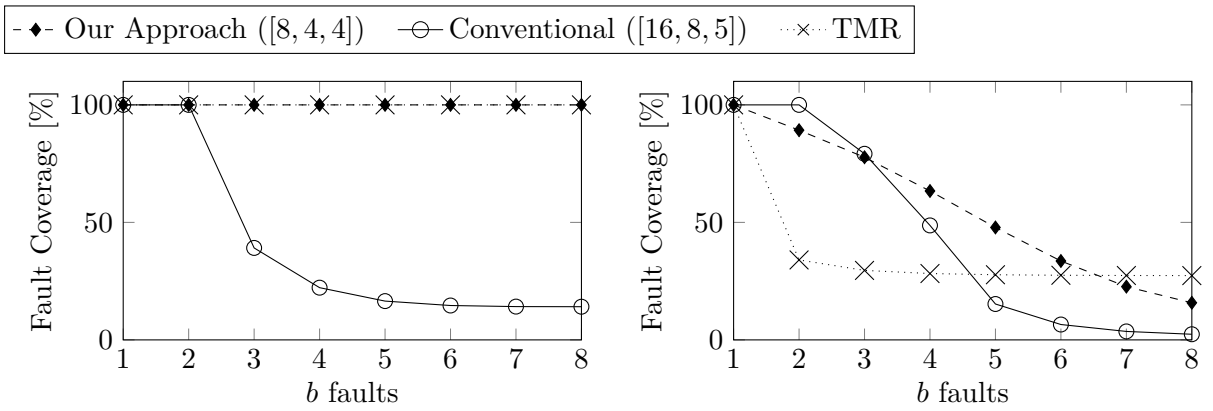
$$C_{\text{conv}}(b) = 1 - \frac{\sum_{i=3}^{b} \binom{8}{i}}{\sum_{i=1}^{b} \binom{8}{i}}.$$

#### Fault Coverage assuming Faults in two Bytes

Considering the second scenario of our adversary model, where an attacker is assumed to inject faults into up to two different bytes following $\mathcal{E}_{B_b}$, we again make use of Equation 6.6. To determine the total number of possible faults $F_{\text{tot}}(b)$, we used Equation 6.7 where $R$ denotes the number of bytes that can be targeted by an attacker.

$$F_{\text{tot}}(b, R) = \sum_{i=3}^{b} \left[ \left( \binom{2 \cdot 8}{b} - 2\binom{8}{b} \right) \cdot \binom{R}{2} + R\binom{8}{b} \right] + \sum_{i=1}^{2} \binom{8 \cdot R}{i} \tag{6.7}$$

Without loss of generality, we considered only one row of the AES state matrix and its corresponding redundancy. To this end, the number of attackable bytes $R$ results in $R = 8$ for our approach and the conventional encoding scheme and in $R = 12$ for TMR as the scheme requires




(a) Faults in a single byte.

(b) Faults in two bytes.

Figure 6.6: Fault coverage compared to a conventional encoding and to TMR.

two copies of the original design. If $b \leq 2$, the number of faults can be determined by the second summand as only two bytes can be targeted and the faults are uniformly distributed over the attackable area. In case $b > 2$ the first summand has to be taken into account as well. By using the first binomial coefficient $\binom{2 \cdot 8}{b}$, we determine the number of different faults with $b$ flipped bits within two bytes. As we would like to calculate all combinations of such faults in an entire row, we multiply it by $\binom{R}{2}$. However, before multiplying, we subtract $2\binom{8}{b}$ possible faults to exclude all cases where all bit flips $b$ occur in one single byte which otherwise would be counted several times. These cases are added just once by $R\binom{8}{b}$.

The number of uncorrectable faults $F_{\text{not}}$ for all three cases was determined by a simulation. We iterated over all possible error vectors combined by two bytes $x$ and $y$ and bounded by $b$ which results in $\binom{16}{b}$ possibilities. Since all approaches are based on different correction capabilities, we distinguished between these cases.

(1) **Our Approach** For each possibility where $\mathrm{HW}(x \odot y) > 0$ we found a fault that cannot be corrected since two bit flips occurred in the same codeword ($\odot$ denotes a *bitwise and*). This method gives us all possible faults that cannot be corrected considering just two bytes. To cover all cases for the entire row, we multiply the result by $\binom{8}{2}$.

(2) **Conventional Encoding** For each possibility where $\mathrm{HW}(x) > 2 \vee \mathrm{HW}(y) > 2$, we found a fault that cannot be corrected since more than two bit flips occurred in one codeword. To cover all cases for the entire row, we multiply the result by $\binom{8}{2}$ as above.

(3) **TMR** For each possibility where $\mathrm{HW}(x) \geq 1 \wedge \mathrm{HW}(y) \geq 1$ we found $4 \cdot 8 + 4 \cdot 4 = 48$ faults that cannot be corrected since they occurred in more than one instantiation. No final multiplication is required.

The resulting fault coverages are given in Figure 6.6b. In case $b = 1$, all three approaches achieve 100 % fault coverage. For $b = 2$, the coverage for TMR rapidly drops below 35 % as faults only can be corrected when they occur in the same instantiation. While the conventional encoding scheme still provides 100 % security (since $d_{\min} = 5$), the correction capability of our design decreases to 89.23 %. Assuming up to three faults, our approach and the conventional

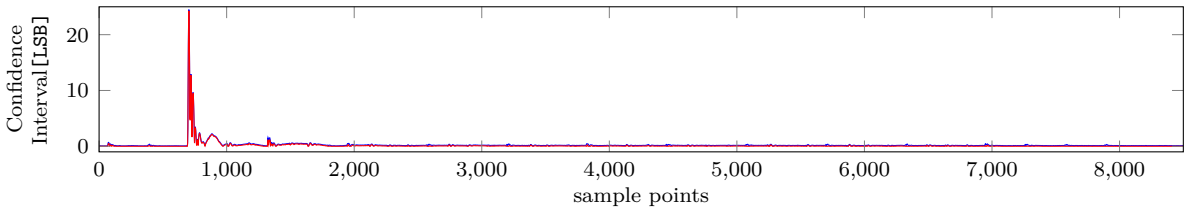



Figure 6.7: 1st-Order results of the unprotected design (10 000 traces).

encoding provide similar results whereas our approach performs slightly worse. From there on, our approach outperforms the conventional encoding. TMR performs worse than the orthogonal for $b \leq 6$ method but has a better fault coverage for $b \geq 7$.

### 6.7.2 Side-Channel Analysis Evaluation

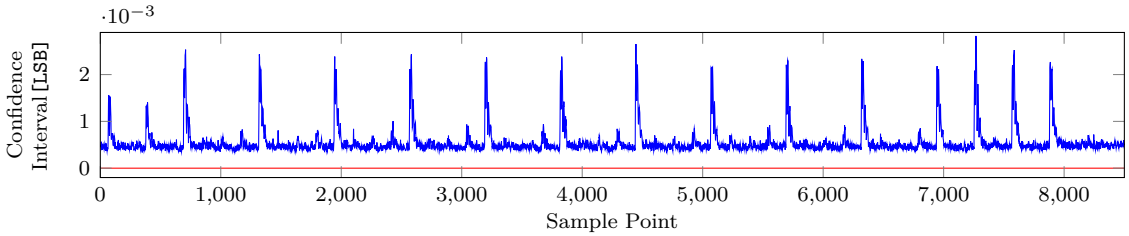
We applied the non-specific fixed-vs-random approach presented in [SM15] to evaluate the side-channel security of our proposal. To identify quantifiable lower and upper bounds for the leakage, we further employed the confidence-interval framework from [BPG18] where all confidence intervals are calculated for a family-wise error rate corrected significance level $\alpha = 0.01$.

Our implementation was instantiated on a Sakura-X side-channel evaluation board where the target was supplied with a 4 MHz clock and the current was measured indirectly via the voltage drop over a shunt in the supply path (with a 20 dB low-noise amplifier) using a 8 bit oscilloscope at a sample rate of 1.25 GS/s.

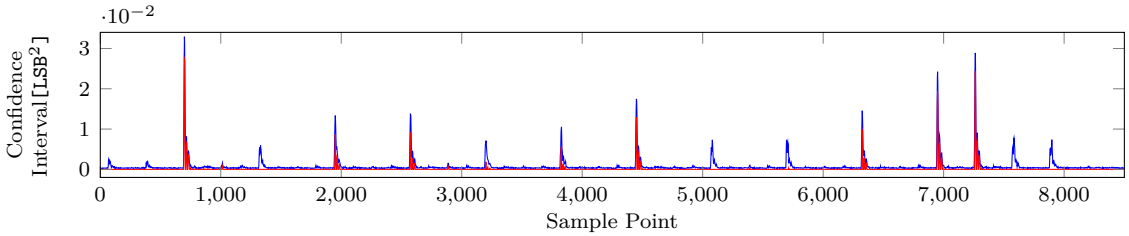
For reference, the result of the first-order evaluation of an unprotected version is shown in Figure 6.7 using only 10 000 traces. As expected, the non-zero lower bound clearly indicates detectable leakage.

The first-order evaluation of the protected design is depicted in Figure 6.8a and the lower bound of zero for every sample indicates that our evaluation was not able to detect first-order leakage. If there is some (undetectable) leakage, the absolute difference in means for the measurements is lower than 0.003 LSB. The second-order evaluation is shown in Figure 6.8b and confirms the detection of expected (albeit small) second-order leakage. For at least one sample point the absolute difference in variances is 0.027 $\text{LSB}^2$ while there is no difference above 0.033 $\text{LSB}^2$ for any sample. The third-order evaluation in Figure 6.8c shows results similar to the second-order case, but the influence of noise increases exponentially in the attack and evaluation order [CJRR99], hence the confidence intervals are not as tight as in the previous case.

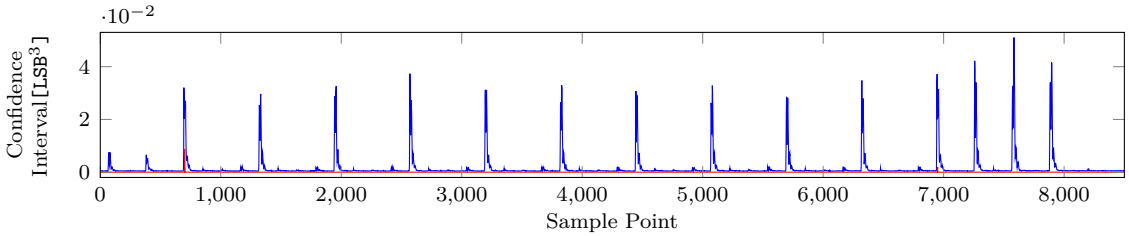
For reference, Figure 6.9a shows a sample trace of the power consumption of our implementation with enabled SCA countermeasures. Furthermore, Figure 6.9 also provides the evaluation results of the same collected data as analyzed before using Welsh's $t$-test. The threshold of $t_{\text{th}} = 4.93$ is adjusted for family-wise error rate at a total significance of $\alpha = 0.01$. An unprotected design shows clear leakage using only 10 000 traces as shown in Figure 6.9b. Our protected implementation exhibits no first-order leakage even when 200 million traces are used in the evaluation. Figure 6.9d and Figure 6.9e show the second- and third-order $t$-test results respectively. As expected, some leakage can be detected.




(a) 1st-Order results of the SCA protected design.




(b) 2nd-Order results of the SCA protected design.

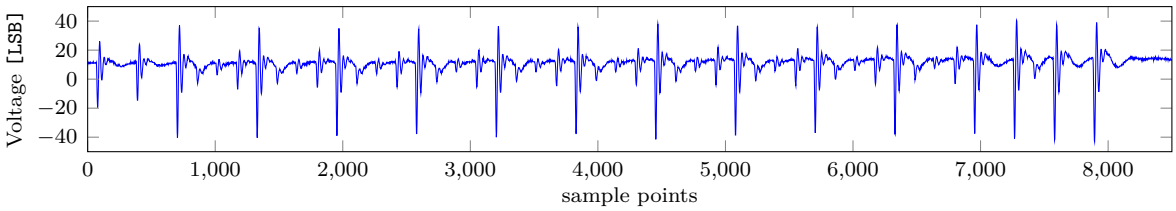



(c) 3rd-Order results of the SCA protected design.

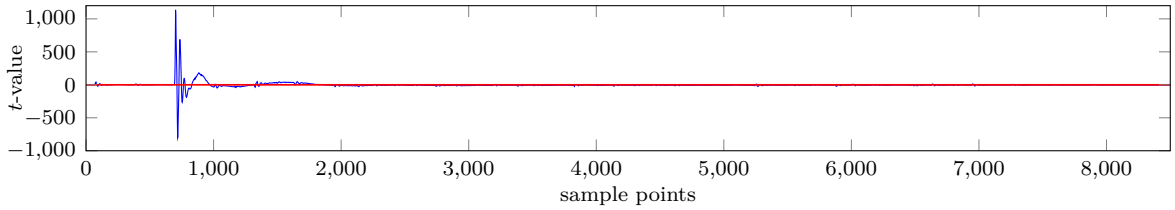
Figure 6.8: Confidence intervals for $\alpha = 0.01$ and 200 million traces. Lower bounds in red, upper bounds in blue.

## 6.8 Conclusion

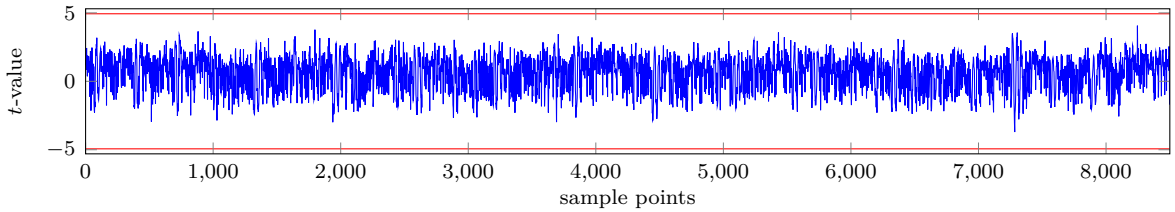
Revisiting Concurrent Error Detection, we presented a novel approach to use ECC for protection against active FIA. Arranging ECCs in an orthogonal pattern, we are able to correct adjacent bits of internal values using simple linear codes. Using a case study based on AES, we show that our approach is up to 35.3 % smaller in terms of area compared to classical arrangements while providing better fault coverage for the considered adversary model. Our second case study shows that our approach combines efficiently with state-of-the-art masking countermeasures to extend the protection even against passive side-channel analysis. Through practical evaluation using 200 million power traces, we validated the security of our design against first-order SCA using a state-of-the-art leakage assessment methodology.



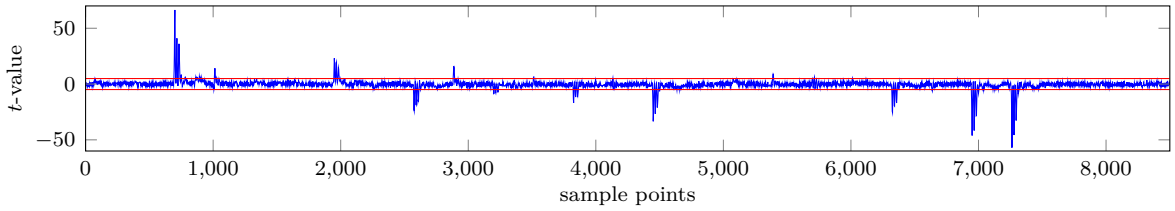

(a) Example trace of the protected design.



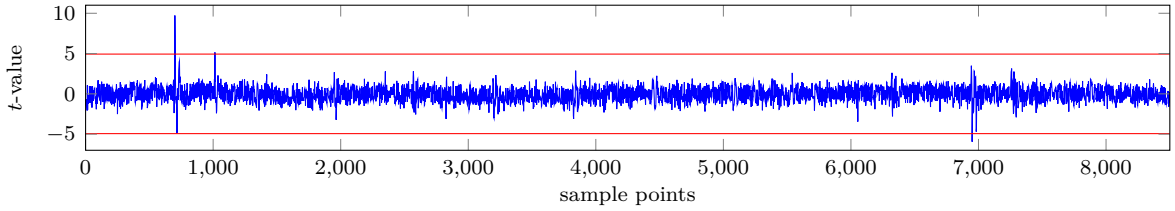

(b) 1st-Order $t$-test of the unprotected design (10 000 traces).




(c) 1st-Order $t$-test of the SCA protected design (200 million traces).




(d) 2nd-Order $t$-test of the SCA protected design (200 million traces).




(e) 3rd-Order $t$-test of the SCA protected design (200 million traces).

Figure 6.9: Non-specific $t$-test results for $\alpha = 0.01$. $t$-values are solid blue, the significance threshold is red.

# Chapter 7

# Improved Side-Channel Resistance by Dynamic Fault-Injection Countermeasures

*The combined protection against SCA and FIA is currently an open line of research. A promising countermeasure with acceptable implementation overhead appears to be a mix of first-order secure Threshold Implementations and linear Error-Correcting Codes.*
*In this chapter, we employ for the first time the inherent structure of non-systematic codes as fault countermeasure which dynamically mutates the applied generator matrices to achieve a higher-order side-channel and fault-protected design. As a case study, we apply our scheme to the PRESENT block cipher that does not show any higher-order side-channel leakage after measuring* 150 *million power traces. All contributions presented in this chapter are part of a joint work with Tim Güneysu [RG20].*

**Contents of this Chapter**



## 7.1 Introduction

Over the last years, a plethora of countermeasures has been proposed against SCA and FIA. Promising techniques to specifically counteract SCA can be divided into *hiding* and *masking*. While countermeasures based on *hiding* try to decrease the SNR in order to harden the extraction of usable information (e.g., from power traces), *masking* is based on secret sharing and multi-party computations. TI belongs to this type of SCA countermeasure and was originally designed to provide provable first-order security [NRR06]. However, the principle of TI can be extended ensuring also higher-order protection [RBN+15, CBR+15] but with a drawback of an unacceptable implementation overhead [MW15].

Countermeasures designed to resist FIA are often based on detection schemes that either withhold a faulty computation [KKG03, AMR+20] or perform an infective computation hampering an attacker to obtain any exploitable information from the outputs [GST12, MAN+19]. However, recently Dobraunig et *al.* demonstrated that these kinds of countermeasures can be broken by using a statistical analysis method called SIFA [DEK+18]. Therefore, linear ECCs seem to be a promising method to provide resilient protection against fault injections as they can also be used to correct occurred faults which would thwart SIFA based attacks [SJR+19, SRM20].

### 7.1.1 Realated Work

Despite the wealth of countermeasures treating SCA and FIA as a separate problem, only few works target the combined setting. In 2016, Schneider et *al.* [SMG16] used two already existing countermeasures (TI and linear ECCs), which separately resist SCA and FIA respectively, and combined the two techniques into one protected design. The resulting implementation provides first-order security against SCA including protection against fault injections. However, extending the TI to resist higher-order SCA would increase the implementation costs significantly and would be impracticable in a real-world environment. In the following years, Reparaz et *al.* [RMB+18] proposed a concept inspired by MPC protocols that achieved protection against SCA and FIA. De Meyer et *al.* [MAN+19] discuss a technique based on masking schemes while the resistance against FIA is achieved by adding MAC tags incorporating an information theoretic approach to the design. All proposals, however, share the unfavorable property of excessive costs in time and/or area in case protection against higher-order attacks should be considered as well.

### 7.1.2 Contribution

In this work, we present an alternative strategy to design a combined countermeasure against SCA and FIA that is suitable to achieve higher-order protection at reasonable cost. Therefore, we revisit existing solutions that successfully combine first-order secure masking schemes with hiding techniques, such as [SMG15, SMG17]. One strategy in this regard is to exploit the composition of small S-boxes into affine equivalences in order to replace the affine functions on the fly. This reconfiguration technique introduces additional randomness into a running encryption process and hides higher-order leakage. This can be further improved by encoding the cipher's state with randomly selected functions resulting in hiding the higher-order leakage in the introduced noise of the encoding scheme. The latter approach is inspired by the idea behind White-Box Cryptography.

Based on the observations from previous works, we now come up with the following original strategy: we compose a first-order secure TI with a randomization technique based on linear ECCs that augments our fault-injection protection with additional noise. In contrast to previous works, we do not rely on systematic codes here but rather explicitly pick generators producing non-systematic codes. These generators are dynamically evolved during runtime in order to hide higher-order leakage as a hiding countermeasure. As shown in our work, we finally achieve a combined hardware countermeasure that successfully resists higher-order side-channel and fault-injection attacks at very reasonable implementation costs.

## 7.2 Preliminaries

As motivated in the introduction, the presented approach is based on a first-order secure masking technique, more precisely on TI, and on linear ECCs to achieve resilience against FIA. In Section 2.1.4 we introduce the concept of TI while Section 3.1 presents background about linear ECCs. In this section, we briefly cover important notations from linear algebra by following definitions from [BV18].

**Lemma 1.** *The determinant of a quadratic matrix $A$ is non-zero if and only if $A^{-1}$ exists.*

**Lemma 2.** *If two matrices $A$ and $B$ are invertible the product $A \cdot B$ is invertible as well and the product's inverse is calculated by*

$$(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}.$$

**Definition 25.** *A quadratic $k \times k$ matrix $Q$ is called an orthogonal matrix if and only if*

$$Q^{\mathrm{T}} = Q^{-1}$$

*and the columns are unit vectors.*

This includes that $Q \cdot Q^{\mathrm{T}} = Q^{\mathrm{T}} \cdot Q = I_k$ holds for all orthogonal matrices.

**Definition 26.** *A quadratic $k \times k$ matrix $P$ is called a permutation matrix if and only if one entry per row and column is one and the rest is zero.*

Thus, due to Definition 25, each permutation matrix is also an orthogonal matrix. Note, however, that not every orthogonal matrix is a permutation matrix.

## 7.3 Methodology

This section describes our general design considerations and defines our adversary model. Based on this information, we introduce our generic principle and describe our implementation strategy. Eventually, we deduce suitable codes for lightweight ciphers.

### 7.3.1 General Considerations

We now introduce our combined countermeasure that aims to resist both side-channel attacks and fault-injection attacks. The fundamental (first-order-only) concept is inspired by [SMG16] and relies on a design which combines TI and linear ECCs. As evaluated in [AMR+20, SRM20], linear ECCs provide terrific properties protecting cryptographic implementations on hardware against fault-injection attacks. However, instantiating first-order secure TI as only countermeasure against SCA, higher-order attacks can be still successfully applied to the combined countermeasure. Note that the ideas of TI can generally be extended to higher orders at – unfortunately – significant costs [MW15]. To provide higher-order protection against SCA without excessive cost overhead, we therefore utilize the existing properties provided by linear ECCs in a continuous randomized update process as hiding countermeasure.

We select FPGAs as target platforms for our case-study. They inherently provide a perfect environment for implementing reconfigurable systems realizing the dynamic exchange of the ECCs.

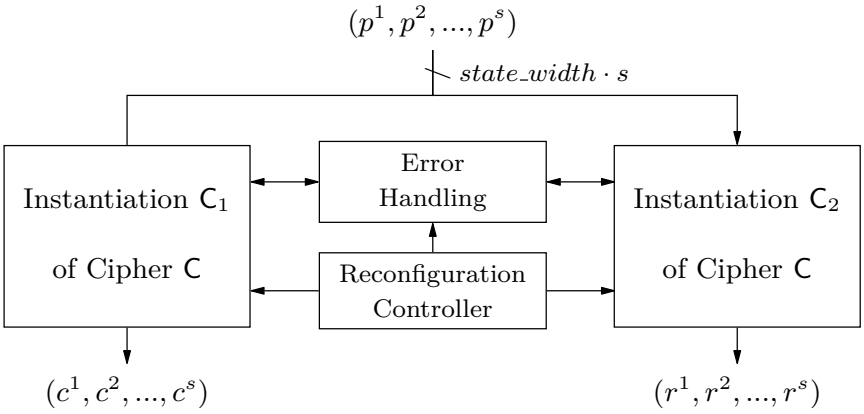



Figure 7.1: Generic principle of protecting a target cipher $\mathsf{C}$.

### 7.3.2 Adversary Model

First, we assume an attacker that characterizes a target device by acquiring the power consumption or the electro-magnetic radiation. Here, we follow the well-known $d$-probing model where an implementation is assumed to be secure under a $d$-order attack [ISW03]. Furthermore, our adversary model includes glitches which will be considered in our security evaluation. For more information, please see Section 2.1.2.

Second, we assume an adversary that additionally is able to inject faults into the target implementation. We model occurring faults by additive errors and instead of assuming an uniform error distribution, we follow the biased fault distribution $\mathcal{E}_{B_b}$ from [SMG16] where the attacker can inject up to $b$ faults into a target codeword $c$. This approach considers biased fault injections which seems to be more realistic considering an attacker trying to recover a cryptographic secret. In this work, we only consider faults occurring in the data depended path of the design since this kind of faults are more important regarding the design of countermeasures thwarting physical attacks.

### 7.3.3 Design Strategy

As introduced before, we employ a first-order secure TI combined with linear ECCs as fundamental building block. Inspired by previous work [SMG16], we decided to choose $n = 2 \cdot k$ such that each word of $k$ bits is separately encoded by a generator matrix $G$. However, instead of relying on systematic linear error codes – as it was mainly done in the past (e.g., in [AMR+20]) – we explicitly want to apply *non-systematic* codes. By dynamically exchanging the applied linear codes, we generate additional algorithmic noise in order to hide any exploitable higher-order side-channel leakage given the presence of a provably-secure first-order secure masking scheme. This way, we achieve an increased security level exploiting the already existing properties of the FIA countermeasure, i.e., of the underlying ECC.

The fundamental principle of our approach is depicted in Figure 7.1 and expects a shared input $p$ with $s$-shares. Due to the selected parameters for the linear ECCs ($n = 2 \cdot k$), the target cipher $\mathsf{C}$ is duplicated, and a redundancy is created processing the same input data as the original cipher. However, since we apply non-systematic codes to the target cipher $\mathsf{C}$, both

instantiations have to be adapted in order to process the encoded states. Generally, each word $m$ of $k$ bits of the cipher's state is encoded by a generator matrix $G = [G_1 \mid G_2]$. To ensure the maximum possible security level against FIA, the selection of a code should be made with regard to maximizing the minimum distance $d_{\min}$. Given a subfunction $\mathsf{F}$ of the target cipher $\mathsf{C}$ and $i \in \{1, 2\}$, each subfunction $\mathsf{F}_i$ of the cipher's instantiation $\mathsf{C}_i$ has to be adapted so that

$$\mathsf{F}_i = G_i \circ \mathsf{F} \circ G_i^{-1} \tag{7.1}$$

holds. Equation 7.1 also reveals another important requirement to the generator matrix $G$: the sub-matrices $G_1$ and $G_2$ have to be *non-singular* in order to calculate their inverses $G_1^{-1}$ and $G_2^{-1}$, respectively.

The protection against higher-order side-channel attacks should be achieved by dynamically exchanging the linear ECC, i.e., the generator matrix $G$ and therewith the sub-matrices $G_1$ and $G_2$. This task is accomplished by a *reconfiguration controller* which adapts on the one hand the cipher's subfunctions and on the other hand the module being responsible for the *error handling*. Depending on the properties of the applied ECCs, the error handling module can either be implemented to detect or to correct occurring faults.

### 7.3.4 Suitable Codes for Lightweight Ciphers

In this section, we describe the procedure of finding suitable codes implementing our approach for lightweight ciphers. Additionally, we investigate the total number of different variations that can be generated using dynamic ECCs. For lightweight symmetric ciphers, we assume a nibble-oriented state such that $k = 4$ and $n = 8$. Selecting these parameters, the maximum minimum distance $d_{\min}$ that can be achieved is $d_{\min} = 4$. We performed an exhaustive search over all possibilities of $[8, 4, 4]$ linear ECCs and identified the set

$$\mathcal{K}_1 = \left\{ G \in \mathbb{F}_2^{4\times 8} \,\middle|\, d_{\min} = 4 \text{ for } \mathcal{C} = \left\{ m \cdot G \mid m \in \mathbb{F}_2^4 \right\} \right\}$$

which contains 596 736 different generators. Since we need to split up $G$ into the sub-matrices $G_1$ and $G_2$ in order to allow separate processing of the data in $\mathsf{C}_1$ and $\mathsf{C}_2$, we tested the sub-matrices $G_1$ and $G_2$ of each $G \in \mathcal{K}_1$ for invertibility. This classification leaves us with a slightly narrowed set

$$\mathcal{K}_2 = \{ G = [G_1 \mid G_2] \in \mathcal{K}_1 | \det(G_1), \det(G_2) \neq 0 \}$$

including 483 840 different generators. However, since each generator is represented by 32 bit, storing all possible generators would require $483\,840 \cdot 32\,\text{bit} \approx 2\,\text{MByte}$ which consequently would result in exploding implementation costs.

To reduce these costs, we further minimized the size of $\mathcal{K}_2$ being able to randomly generate new generator matrices on the fly. Therefore, we defined a set $\mathcal{P}$ including all permutation matrices of size $4 \times 4$ leading to $|\mathcal{P}| = 4!$. Given that and a valid generator matrix $G$, we can construct another valid generator matrix $\tilde{G}$ by randomly choosing $P_{i\in\{1,2\}} \in \mathcal{P}$ and permuting the columns within the sub-matrices $G_{i\in\{1,2\}}$ which results in

$$\tilde{G} = [G_1 \cdot P_1 \mid G_2 \cdot P_2] \,. \tag{7.2}$$

This operation does not change the capability of the underlying code and the resulting sub-matrices are still invertible due to Lemma 2. Subsequently, given one arbitrary generator matrix

from the set $\mathcal{K}_2$, we can generate $4! \cdot 4! = 576$ different variants from it applying Equation 7.2. Hence, we do not have to store the entire 483 840 generators but rather can reduce the size of $\mathcal{K}_2$ creating a new set $\mathcal{K}_3$ with $483\,840/576 = 840$ different generators which we call *basis generators* in the following. In summary, we can generate all $\tilde{G} \in \mathcal{K}_2$ on the fly using the defined 840 basis generators and permutations from $\mathcal{P}$.

As shown in Equation 7.1, the dynamic exchange of the linear ECCs does not only require the sub-matrices $G_{i\in\{1,2\}}$ but also their inverses. Assuming we have pre-calculated and stored all the inverses of the basis generator's sub-matrices in a set $\bar{\mathcal{K}}_3$, we can compute the inverses of the permuted matrices $G_i \cdot P_i$ on the fly by

$$(G_i \cdot P_i)^{-1} = P_i^{-1} \cdot G_i^{-1} = P_i^{\mathrm{T}} \cdot G_i^{-1} \tag{7.3}$$

where $P_i \in \mathcal{P}$ and $i \in \{1, 2\}$. To simplify the implementation processes for hardware devices, we define an additional set $\bar{\mathcal{P}}$ which contains all transposed permutations from $\mathcal{P}$.

## 7.4 Case Study

As we concentrated our investigations mainly on lightweight ciphers, we apply our approach in a practical case study to the PRESENT block cipher [BKL+07]. As a target platform we selected FPGAs as already mentioned in Section 7.3.1.

### 7.4.1 PRESENT

PRESENT is a block cipher consisting of a 64-bit state and supporting key lengths of 80 bits and 128 bits. Independent of the chosen key length, the cipher executes 31 rounds where each round includes a key addition, a linear layer, and a non-linear substitution. The key addition adds (XOR) a round key $K_i$ for $1 \leq i \leq 32$ to the current state where the last round key $K_{32}$ is used for post-whitening. The linear layer is realized by a bit-wise permutation of the state. In order to perform the non-linear substitution, the state is divided into nibbles which are used as inputs to 16 parallel 4-bit to 4-bit S-boxes $S(x)$. Note that we refer to the PRESENT version using an 80-bit key in the following.

### 7.4.2 Reconfiguration Controller

One important part of our design is the *reconfiguration controller* as depicted in Figure 7.1. To generate all possible variations from $\mathcal{K}_2$ on the fly, we instantiated two 36 KB Block-RAM (BRAM) modules (cf. Figure 7.2) storing $\mathcal{K}_3$ and $\bar{\mathcal{K}}_3$, respectively. Using the random bits $r_{\mathrm{B}}$, we can read one of the stored basis generators and the corresponding inverse into one clock cycle setting the data width to 32 bits. The outputs $G$ and $G^{-1}$ are separated into two 16-bit words representing $G_{i\in\{1,2\}}$ and $G^{-1}_{i\in\{1,2\}}$, respectively. Using additional randomness $r_{\mathrm{EN}}$ and $r_{\mathrm{RE}}$, the permutation matrices $P_i \in \mathcal{P}$ and $\bar{P}_i \in \bar{\mathcal{P}}$ with $i \in \{1, 2\}$ are selected in order to permute the prior determined basis generators $G_i \in \mathcal{K}_3$ and $G_i^{-1} \in \bar{\mathcal{K}}_3$ applying Equation 7.3. The outputs $\tilde{G}_{i\in\{1,2\}}$ and $\tilde{G}^{-1}_{i\in\{1,2\}}$ are then used to reconfigure the TI S-boxes, the *error handling* module, and modules being responsible for encoding the state and round keys $K_i$.

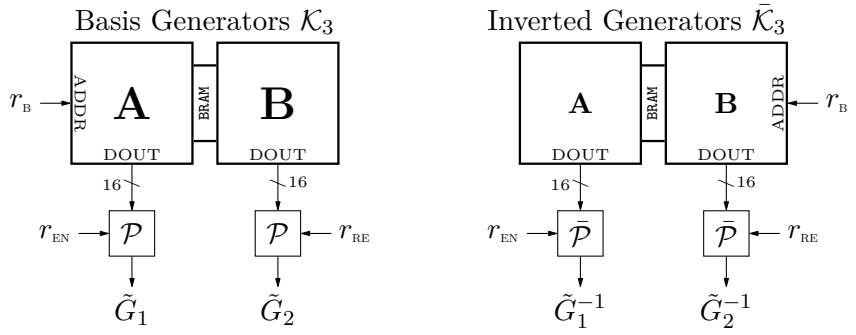



Figure 7.2: Randomized generation of $G$ and $G^{-1}$.

### 7.4.3 Cryptographic Instantiations

The non-linear S-boxes of the instantiations $\mathsf{C}_1$ and $\mathsf{C}_2$ are realized using BRAMs. Since we implement a first-order secure TI, we decompose the cubic-non-linearity $S(x)$ into two quadratic functions $S(x) = T(R(x))$ using affine equivalences. We decide to apply the same shared decomposed functions as the authors in [SMG16] where $R^1 = R^2 = R^3 = R^{\mathrm{d}}$ and $T^1 = T^2 = T^3 = T^{\mathrm{d}}$. Each 8-bit to 4-bit LUT is stored in an own dual-port BRAM module as exemplary shown in Figure 7.3 for two realizations of $R^{\mathrm{d}}$ where the above BRAM is placed in the instantiation of $\mathsf{C}_1$ (encryption path) and the lower BRAM in $\mathsf{C}_2$ (redundancy).

This implementation strategy reduces the amount of sequential logic and logic cells and allows a concurrent reconfiguration of the non-linear functions due to the dual-port memories. A reconfiguration is conducted by an eight-bit counter which on the one hand reads out the values of the shared decomposed functions $R^{\mathrm{d}}$ and $T^{\mathrm{d}}$ and on the other hand serves as foundation to determine new addresses. To complete the computation of the addresses, the counter values are split into two nibbles and are separately encoded by the corresponding generator matrices $\tilde{G}_{i\in\{1,2\}}$. The new S-box values are determined based on the original values of $R^{\mathrm{d}}$ and $T^{\mathrm{d}}$ and a subsequent encoding by $\tilde{G}_{i\in\{1,2\}}$. During reconfiguration, the second BRAM port is used for processing the data of the encryption and redundancy such that the input values are forwarded to the address ports and the outputs are used as inputs to the subsequent subfunction. After a reconfiguration is completed, a context switch is performed and the freshly reconfigured LUTs are used.

### 7.4.4 Error Handling

The realization of the *error handling* module follows the design of [SMG16] and is used to detect occurring faults within an encryption. After every key addition, the states of the encryption path and of the redundant path are decoded and compared in order to prevent faulty encryptions within the detection capability of the used ECC. As the applied ECCs change over time, the detection module has to be reconfigured as well which requires the inverse $\tilde{G}^{-1}$ of the used generator matrix $\tilde{G}$. Here we just rely on combinatorial logic and do not utilize BRAM in order to avoid additional delays when performing the error check.

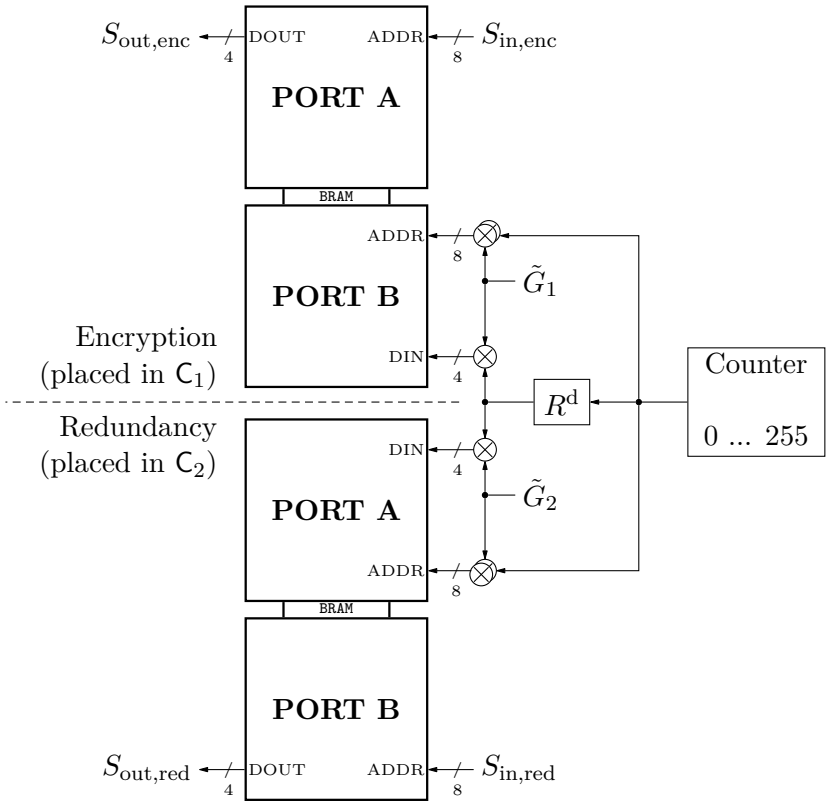



Figure 7.3: Reconfiguration of the TI S-boxes exemplary shown for the quadratic decomposed function $R^{\text{d}}$.

### 7.4.5 Overall Implementation

Figure 7.4 shows a schematic of the overall implementation composed of the aforementioned building blocks. Note that all data paths are realized in shares to implement a correct TI. Each plaintext that should be encrypted is first encoded by $G_1$ forwarded to the cipher's instantiation $\mathsf{C}_1$ (left data flow) and encoded by $G_2$ forwarded to the cipher's instantiation $\mathsf{C}_2$ (right data flow). Besides the shared plaintext, every round key $K_i$ needs to be encoded as well so that additional encoding modules (implemented in combinatorial logic) are placed right before every key addition. The following register stage is included in the BRAM modules and is used to prevent glitches. The LUTs $R_1$ and $R_2$ represent the encoded quadratic function $R$ generated by the reconfiguration technique described in Figure 7.3. Again, the next register stage is included in the BRAM modules. However, to generate the values of $T$, only the inputs get encoded by the reconfiguration controller. The outputs of $T$ are returned in a non-encoded form in order to allow a straightforward application of the permutation layer. Afterwards both states of $\mathsf{C}_1$ and $\mathsf{C}_2$ are encoded again by $G_1$ and $G_2$, respectively. As described above, the error handling module compares the states of $\mathsf{C}_1$ and $\mathsf{C}_2$ after the key addition. In case a fault is detected, the Error Flag is raised and an ongoing encryption is directly interrupted.

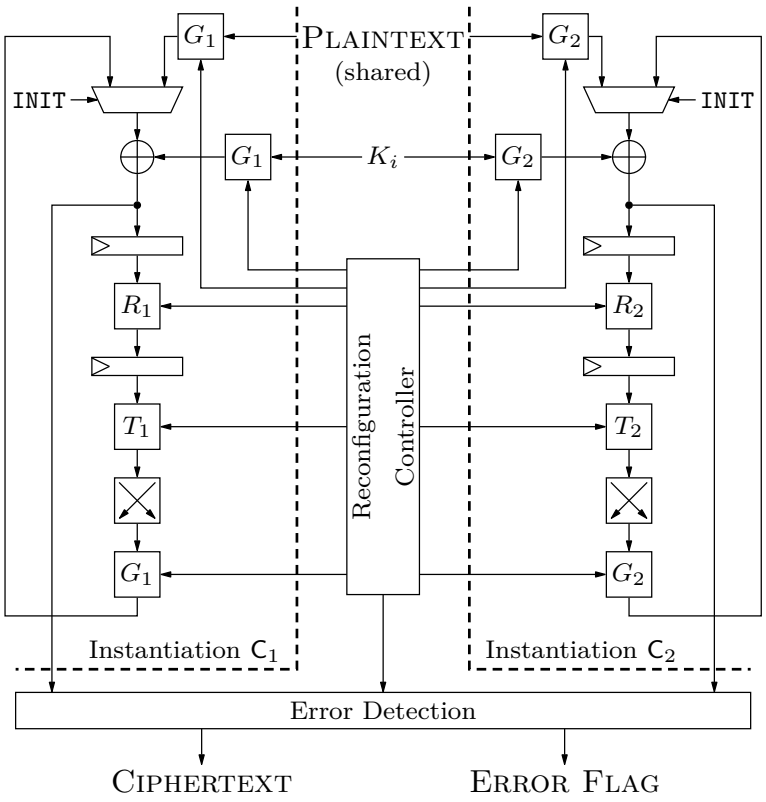



Figure 7.4: Schematic of the overall implementation.

### 7.4.6 Reconfiguration Performance

In Section 7.4.3 we already mentioned that a reconfiguration can be performed in parallel to an encryption due to the dual-port BRAMs. One reconfiguration takes $2^8 = 256$ clock cycles (one clock cycle for each value of the TI S-boxes). The latency of one encryption adds up to 64 clock cycles which perfectly fits the 256 clock cycles required for the reconfiguration. Hence, four encryptions are performed with the same encoding (i.e., the same generator matrices) before a context switch between the two parts of the BRAMs is induced and the freshly reconfigured LUTs can be used for upcoming encryptions. Furthermore, due to this technique, no additional latency is introduced and a continuous encryption process can be ensured.

## 7.5 Analysis

This section presents the implementation results as well as the security analysis. After we compare our approach to already existing implementations, we focus our evaluation on a theoretical discussion about the achieved fault coverage. Afterwards, we apply a state-of-the-art leakage assessment methodology based on TVLA validating higher-order security.

Table 7.1: Implementation results compared to related work.

| Approach | Logic | Memory | | | Latency | Freq. | Through. | Power |
|---|---|---|---|---|---|---|---|---|
| | LUT | FF | LUTRAM | BRAM[i] | cycles | MHz | MBit/s | mW |
| 1st-order TI [MW15] | 808 | 384 | – | 0 | 64 | 207 | 413 | N/A |
| 2nd-order TI [MW15] | 2 245 | 1 680 | – | 0 | 128 | 204 | 406 | N/A |
| Affine Equivalences [SMG15] | 1 834 | 742 | – | 1 | 64 | 112 | 224 | N/A |
| Glitch-Free Duplication [MW15] | 5 442 | 12 672 | – | 0 | 704 | 459 | 458 | N/A |
| Dynamic Hardware Mod. [SMG17] | 3 236 | 3 246 | 1 952 | 192 | 124 | 153 | 315 | N/A |
| Dynamic ECCs [**this work**][ii] | 3 955 | 219 | 0 | 196 | 64 | 135 | 266 | 423[iii] |

[i] 18 KB tiles. [ii] Only work that includes a countermeasure against SCA and FIA. [iii] Dynamic power.

### 7.5.1 Implementation Results

Since our approach uses reconfiguration techniques, FPGAs seem to be a perfect platform for implementing and evaluating our design. As a target platform, we selected a Xilinx Kintex-7 XC7K160T FPGA. Table 7.1 shows the implementation results divided into area utilization, speed, and power. Comparing our approach to designs reported in the literature, the area overhead regarding the required amount of LUTs is reasonable considering that our approach has implemented resistance against FIA which is missing in all other designs. The decreased number of registers used in our implementation originates from the instantiated BRAM modules realizing the non-linearities. Each BRAM module contains a non-configurable input register which is in case of TI needed anyway to avoid glitches. However, our design requires a total amount of 196 BRAM tiles since each instantiation $\mathsf{C}_{i\in\{1,2\}}$ requires 96 tiles realizing the S-boxes and the four additional tiles are required to hold the basis generators and the corresponding inverses.

The achieved throughput of 266 MB/s is comparable to the designs by Sasdrich et *al.* [SMG15, SMG17] and slightly lower than the approach by Moradi and Wild [MW15].

Even though no work from the considered references provides any results regarding the power consumption, we decided to include it within our evaluation table as an additional important metric, especially for devices relying on power provided by battery. We determined the power consumption using Vivado leaving all settings at their default values. Eventually, we obtained a power consumption with a high confidence of 423 mW for our target device (excluding static power).

### 7.5.2 Resistance against Fault Injections

The resistance against FIA is determined by the underlying ECCs. In our case study, we only used linear codes with the maximum minimal distance being available for the selected parameters, i.e., $n = 8$, $k = 4$, $d_{\min} = 4$. Since we rely on the same structure and codes with the same capabilities as the authors in [SMG16], our design achieves the same fault coverage.

Table 7.2: Fault coverage of the applied linear ECCs.

| $\mathcal{E}_{B_1}$ | $\mathcal{E}_{B_2}$ | $\mathcal{E}_{B_3}$ | $\mathcal{E}_{B_4}$ | $\mathcal{E}_{B_5}$ | $\mathcal{E}_{B_6}$ | $\mathcal{E}_{B_7}$ |
|---|---|---|---|---|---|---|
| 100 % | 100 % | 100 % | 91.36 % | 93.58 % | 94.31 % | 94.49 % |

Using the biased fault distribution $\mathcal{E}_{B_b}$ introduced in Section 7.3.2, where an attacker is able to inject up to $b$ faults into a single codeword, the fault coverage $C_{\text{cov}}$ follows

$$C_{\text{cov}} = 1 - \frac{F_{\text{not}}}{F_{\text{tot}}}$$

where $F_{\text{not}}$ represents the number of undetectable errors and $F_{\text{tot}}$ the total number of errors that can occur in the defined fault model. The corresponding fault distribution is given by Table 7.2. Note that the used codes have a 100 % fault coverage as long as $b \leq 3$ due to a minimum distance of $d_{\text{min}} = 4$ (cf. Corollary 1). However, this does not mean that $F_{\text{not}} = F_{\text{tot}}$ when $b \geq 4$ as only faults that are equal to valid codewords will not be detected. For detailed information, we refer the interested reader to the original work from Schneider et *al.* [SMG16].

### 7.5.3 Resistance against Side-Channel Analysis

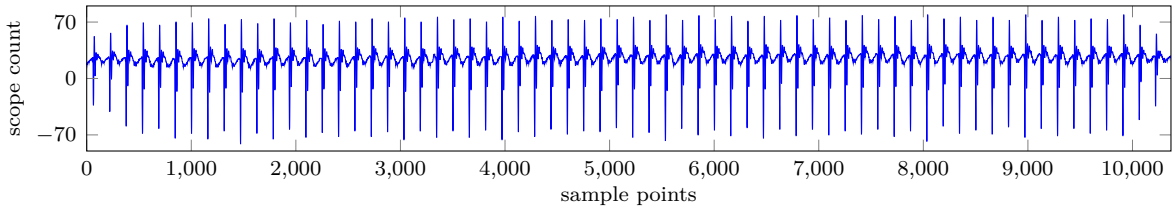
For evaluating the resistance against SCA, we used the side-channel measurement board Sakura-X equipped with a Kintex-7 FPGA holding the cryptographic implementation and a Spartan-6 FPGA controlling the measurement which includes the generation of a stable clock of 4 MHz. The voltage drop was measured using a 1 Ω shunt resistor while amplifying the signal with a ZFL-1000 LN+ amplifier (24 dB gain). The analog signal was converted into an 8-bit digital word using a 6404D PicoScope and a sampling rate of 625 MS/s. Furthermore, we used the PicoScope's low-pass filter with a cut-off frequency of 25 MHz. As leakage assessment methodology, we applied an univariate Welch's $t$-test (cf. Section 2.1.3) since it can be extended to higher-order statistical moments [SM15].

To validate our implementation and measurement setup, we first conduct measurements by setting all masks to zero and disabling the additional randomness required for the dynamic reconfiguration of the ECCs. The corresponding results are shown in Figure 7.5. As expected, after acquiring 1 million power traces, we clearly see leakage in all considered statistical orders.

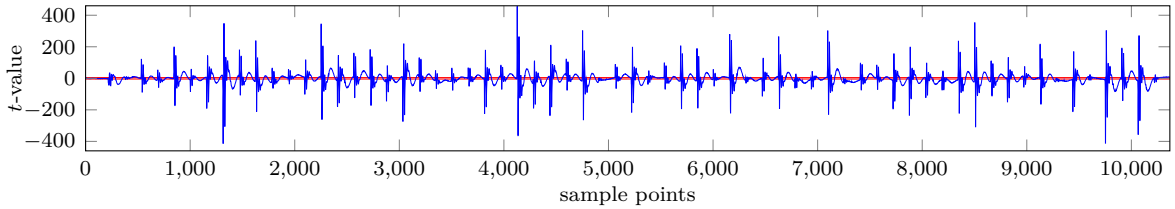
In our second experiment, we use random masks and enable the Linear Feedback Shift Register (LFSR) providing the required randomness for the reconfiguration of the dynamic ECCs. The $t$-test results for the first three statistical moments after acquiring 150 million power traces are shown in Figure 7.6. Within the considered confidence threshold of ±4.5, we do not detect any $t$-value that falls outside the interval. Hence, we do not detect any noticeable leakage in the considered statistical moments.

## 7.6 Discussion

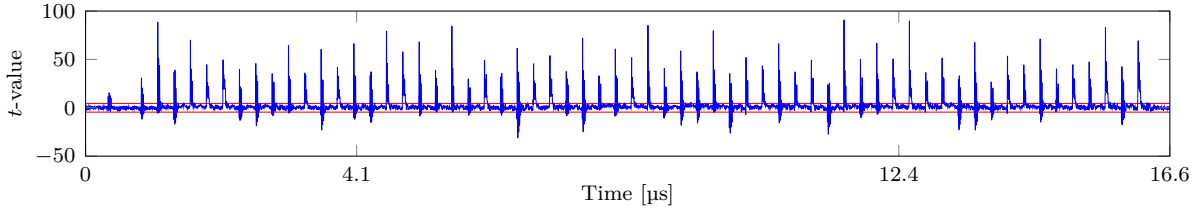
Implementing the *error handling* module just as detection module (cf. Figure 7.1), offers weak points against SIFA-based fault attacks. However, the application of storing the S-box values in BRAMs does not allow to implement correction modules as the input registers of BRAMs



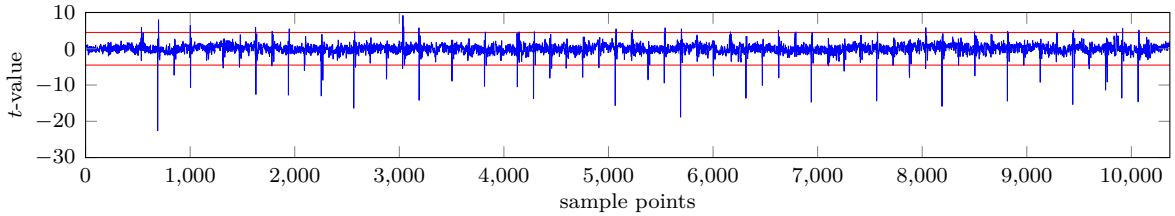

(a) Sample trace.




(b) First-order $t$-test results.




(c) Second-order $t$-test results.

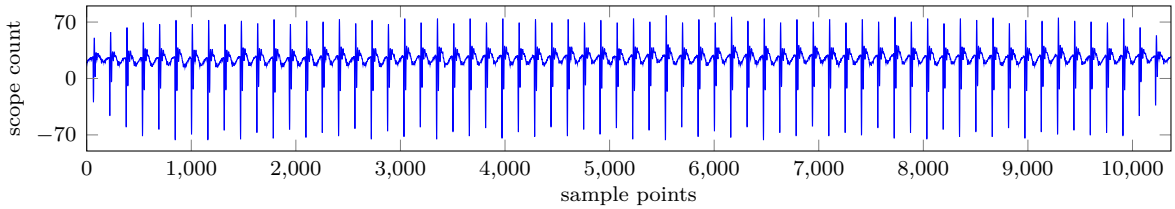



(d) Third-order $t$-test results.

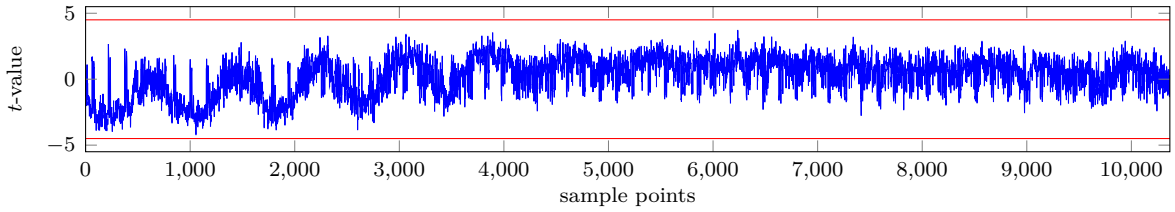
Figure 7.5: Measurement results using a static code, zero masks and 1 million traces.

are not accessible by the user so that faults occurring in these registers cannot be corrected before processed by the S-boxes. The non-linearities would uncontrollably spread a fault over an entire codeword such that the correction capabilities would be exceeded. To this end, our approach could be still implemented using distributed memory instead of BRAM as the designer can place input registers combined with correction modules before each non-linearity. This procedure would allow to apply our combined protection mechanism and to successfully thwart SIFA based attacks.

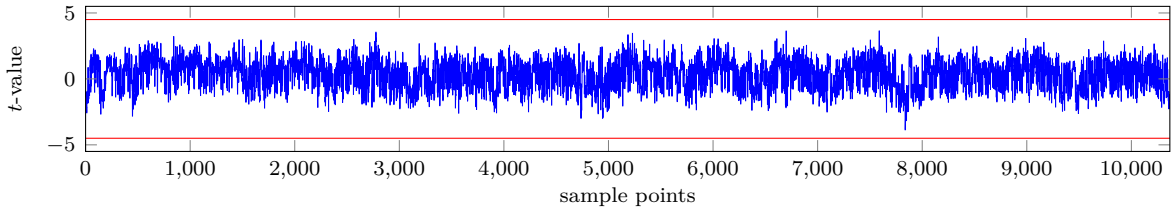
Section 7.3.4 deals with suitable codes for lightweight ciphers which does not include larger algorithms like the AES. Performing an exhaustive search over all $[16, 8]$-codes (each byte of the AES state matrix is encoded separately), would not be possible as there are $2^{16 \cdot 8}$ possibilities. However, picking already existing codes like the $[16, 8, 5]$-code described in the appendix of [BCC+14] and using the same permutation process as described in Section 7.3.4, would produce



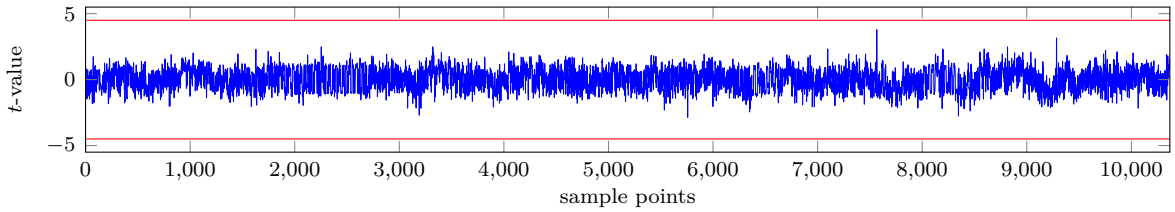

(a) Sample trace.




(b) First-order $t$-test results.




(c) Second-order $t$-test results.




(d) Third-order $t$-test result.

Figure 7.6: Measurement results using dynamic codes, random masks and 150 million traces.

$8! \cdot 8!$ different generator matrices given just one basis generator. Hence, an application to AES would be conceivable but not all existing basis generator matrices could be exploited.

## 7.7 Conclusion

In this work, we present a combined countermeasure against SCA and FIA based on a combination of a first-order secure TI and linear ECCs. Using the underlying structure of the linear codes as an opportunity to introduce additional noise by randomizing the used generators, we achieve a higher-order protected design against SCA. We narrowed down the size of the required generator matrices resulting in a reconfiguration controller which is able to generate 483 840 different variations on the fly while achieving acceptable implementation overhead. Eventually,

a case study on PRESENT, including power measurements with 150 million traces, shows protection up to the third statistical order while providing resistance against FIA.

# Part III

# Adversary Models and Security Notions for Physical Attacks

# Chapter 8

# Revisiting Fault Adversary Models - Hardware Faults in Theory and Practice

*Security validations of proposed countermeasures against FIA are mostly performed on custom adversary models that are often not tightly coupled with the actual physical behavior of available fault-injection mechanisms and, hence, fail to model the reality accurately. Furthermore, using custom models complicates comparisons between different designs and evaluation results.*
*In this chapter, we aim to close this gap by proposing a simple, generic, and consolidated fault-injection adversary model that can be perfectly tailored to existing fault-injection mechanisms and their physical behavior in hardware. To demonstrate the advantages, we apply it to a cryptographic primitive and evaluate it based on different attack vectors. We further show that our proposed adversary model can be integrated into the state-of-the-art fault verification tool VerFI. Finally, we provide a discussion on the benefits and differences of our approach compared to already existing evaluation methods. Our generic fault-injection adversary model was originally presented in [RBSG22] in cooperation with Pascal Sasdrich and Tim Güneysu.*

**Contents of this Chapter**



## 8.1 Introduction

Over the last two decades, a plethora of different fault-injection mechanisms has been proposed, e.g., clock or voltage glitches [ADN+10, ZDCT13], electromagnetic pulses [DDRT12, DLM21, OGM15, OGM17], or focused photon injection using laser beams [SA02, RSDT13, CLFT14, SBHS15]. Likewise, many different analysis techniques ranging from DFA [BS97], over IFA [Cla07] and SFA [FJLT13], to SIFA [DEK+18, DEG+18] has been presented to exploit injected

faults. Naturally, different approaches to increase protection against FIA have been proposed at similar pace, mainly following the concepts of redundancy and (concurrent) error detection [SMG16, AMR+20], error correction [SRM20, RSBG20], or infective computation [GST12].

However, checking and verifying that an implementation is successfully protected against FIA is a manual, downstream, test-driven, and error-prone process. Further, the quality of analysis and verification results comprehensively depends on the accuracy of underlying adversary models. If the adversary models fail to reflect the practical realities and capabilities of an adversary, countermeasures and protection mechanisms might be inappropriate, can fail to provide the desired level of security, and ultimately the physical implementation is still vulnerable to FIA.

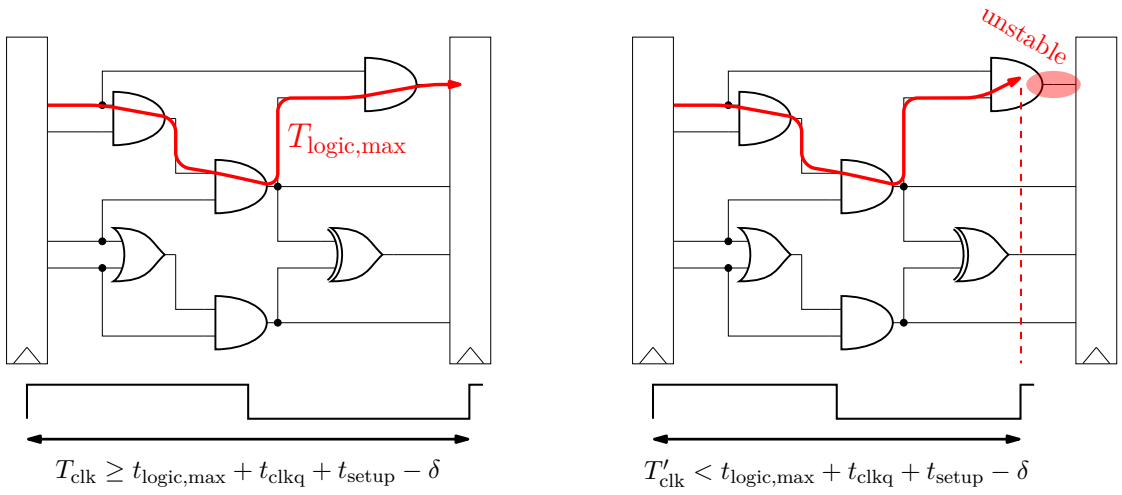
Given these observations and challenges, security should be considered during the entire development and life cycle of the implementation. More precisely, continuous analysis, evaluation, and verification of the design, even before deployment, can assist the designer to choose and implement countermeasures correctly. In addition, accurate description and modeling of the capabilities and limitations of the physical adversary and environment will ensure appropriate protection of the implementation after deployment.

Currently, a wide range of custom adversary models is used for evaluation and verification of protection mechanisms and often, with new countermeasures, new adversary models are proposed at the same time. Unfortunately, most of the adversary models are hardly compatible and do not allow fair and meaningful comparisons between different approaches and implementations. Ideally, a standardized model that is simple and generic, but allows customization would help to analyze, verify, and compare different implementations and countermeasures. Ultimately, designers would be able to choose countermeasures and protection mechanisms appropriately, and easily evaluate and verify the security level for the targeted practical environment and circumstances with minimal effort using the standardized adversary model tweaked for the given realities.

### 8.1.1 Contribution

In this work, we review existing approaches and methods to inject faults into cryptographic implementations in order to consolidate existing adversary models and extract an unified adversary model for standardized fault analysis and verification. In particular, we introduce a generic and abstract adversary model that can be parameterized and instantiated to model different adversaries with varying capabilities and limitations. More precisely, we show how the generic adversary model can be customized to reflect and model common fault injection approaches, including (but not limited to) *clock* or *voltage* glitches, *electromagnetic pulses*, and focused *laser beams*, and apply each model to a practical example emphasizing similarities and differences of the different adversary model instances.

Eventually, our consolidated and unified adversary model can be used to establish a standardized evaluation metric for FIA countermeasures that allows fair comparison in adversary capabilities and limitations as well as vulnerabilities of different protection mechanisms. In particular, our proposed adversary model facilitates application for design and verification through a simple, adaptable, and intuitive design. We demonstrate these features by integrating our fault model into the fault verification tool VerFI [AWMN20] and providing a case study on the lightweight block cipher LED-64 [GPPR11].




(a) Normal condition.

(b) Clock glitch occurrence.

Figure 8.1: Physical effects of clock glitches on digital circuits.

## 8.2 Fault-Injection Mechanisms

Over the last two decades, many different fault-injection mechanisms were proposed and successfully established to attack hardware implementations of cryptographic algorithms. We survey the most common techniques and explain the fundamental physical mechanisms.

### 8.2.1 Clock Glitches

Faulting digital circuits through generation and injection of *clock glitches* is considered as a rather inexpensive technique for FIA. However, before we examine the physical fundamentals and mechanisms of intentional fault injection through clock glitches, we briefly review state-of-the-art literature with respect to FIA based on clock glitch generation.

**State of the Art.** In an early work on the general principles of fault injection via clock glitches, Agoyan et *al.* [ADN+10] demonstrated its effectiveness using the example of an AES hardware implementation. Soon thereafter, Endo et *al.* [ESH+11] presented an on-chip clock glitch generator composed of Delay Locked Loops (DLLs) to test and validate the effectiveness of newly developed countermeasures addressing the threat of clock glitch insertion. In 2014, Korak et *al.* [KHEB14] increased the success rates of clock glitches in combination with heating of the device under attack. Although it was assumed that internal application of Phase-Locked Loops (PLLs) can easily defeat the threat of fault injection through clock glitches, Selmke et *al.* [SHO19] recently presented successful fault injections using clock glitches on a microcontroller internally equipped with a PLL. Note, however, that this attack is still limited and only possible if an ongoing computation is not interrupted by the `LOCKED` signal of the PLL, as also noted by the authors.

**Physical Mechanism.** At a first glance, clock glitches may have limited relevance in real-world scenarios (since they can be prevented by using the `LOCKED` signal of PLLs as described above) when compared to other fault-injection mechanisms covered later in this section. However, since clock glitch generators can be instantiated fairly easily in many common FPGAs, allowing to create cost-efficient test setups for countermeasure validation, we opt to cover this mechanism in more detail in the following paragraph.

For this, Figure 8.1 schematically depicts the physical effects of clock glitches on the behavior and operation of digital circuits. Under normal operation conditions (Figure 8.1a), all signals can propagate through the combinational logic and settle to a stable state before the rising edge of the clock signal triggers the sampling process of the subsequent register. As a consequence, the (maximum) clock period $T_{\text{clk}}$ of a digital circuit is usually determined under the following conditions and assumptions:

$$T_{\text{clk}} + \delta \geq t_{\text{logic,max}} + t_{\text{clkq}} + t_{\text{setup}} \tag{8.1}$$

Here, $\delta$ denotes the clock skew, $t_{\text{logic,max}}$ the maximum propagation delay of the combinational logic, $t_{\text{clkq}}$ the delay of the register, and $t_{\text{setup}}$ the setup time for the input of the register.

Given that an adversary now can generate a clock glitch for an effective fault injection, the clock period $T'_{\text{clk}}$ is instantaneously decreased such that the inequality in Equation 8.1 is violated (but will hold again afterwards). Hence, for some primary input combinations the clock period might be too short to allow full propagation of the signals through the entire combinational logic and a stabilization of the correct result at the input of the register is not guaranteed. Figure 8.1b visualizes this behavior, eventually leading to the fact that the output of the considered gate is still independent of the current primary inputs and might lead to a faulty value sampled by the register at the arrival of the rising edge of the clock glitch.

### 8.2.2 Underpowering and Voltage Glitches

Similar to fault injection through clock glitches, *underpowering* and *voltage glitches* are also considered as rather inexpensive but effective methods for FIA. While underpowering considers the scenario of lowering the supply voltage of the target device throughout the entire computation process, voltage glitches only lower the supply voltage for a very limited period of time during the execution. Again, we briefly summarize state-of-the-art literature, before we discuss the physical fundamentals and mechanics of fault injection through underpowering and voltage glitching.

**State of the Art.** The first successful fault injection using the mechanisms of *underpowering* was presented in 2008 by Selmane et *al.* [SGD08]. Using a 130 nm ASIC embedding an AES engine on a smart card target device, the authors report a successful recovery of the secret key processed inside the AES encryption engine. Their evaluations further demonstrate the dependency between voltage level and success rate of fault injection through underpowering. However, since underpowering naturally effects the entire execution of a cryptographic algorithm, precisely injecting faults, e.g., in a specific iteration of the algorithm, is very difficult and hardly achievable. As a consequence, Zussa et *al.* [ZDCT13] focused their investigations on the fault-injection mechanism of temporary *voltage glitches* to disturb the execution of cryptographic algorithms. More precisely, the authors prove that the physical mechanisms of voltage

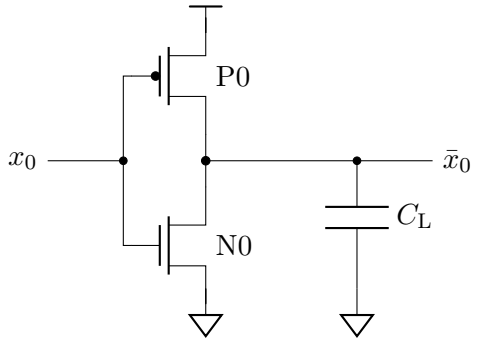



Figure 8.2: Transistor-level schematic of a CMOS inverter.

glitches and underpowering can be traced back to timing violations, as explained in the following paragraph.

**Physical Mechanism.** Considering the example of simple CMOS inverter at transistor level, as given in Figure 8.2, we briefly summarize the findings of [ZDCT13] with respect to timing violations caused through voltage glitches (and underpowering). Assuming that each CMOS gate introduces some propagation delay upon signal switching, the propagation delay in case of a simple CMOS inverter can be explained through the switching process in the transistors. Exemplary, we assume a switching activity from *low* to *high* at the output of the P-type Metal-Oxide Semiconductor (PMOS) transistor P0 in Figure 8.2. In this case, the propagation delay $t_\mathrm{pLH}$, as derived in [Raz08], is given by the following equation:

$$t_\mathrm{pLH} = \frac{C_\mathrm{L}\left[\frac{2|V_\mathrm{th,p}|}{V_\mathrm{DD}-|V_\mathrm{th,p}|} + \ln\left(3 - 4\frac{|V_\mathrm{th,p}|}{V_\mathrm{DD}}\right)\right]}{K_p\left(V_\mathrm{DD} - |V_\mathrm{th,p}|\right)}. \tag{8.2}$$

Here, $C_\mathrm{L}$ models the load of connected gates, $V_\mathrm{th,p}$ the threshold voltage of the transistor, and $K_p = \mu_\mathrm{p} C_\mathrm{ox}\frac{W_\mathrm{p}}{L_\mathrm{p}}$ the gain of the PMOS transistor.

Obviously, under a lower supply voltage $V_\mathrm{DD}$, the propagation delay of the inverter $t_\mathrm{pLH}$ increases. Further, similar equations can be derived for the N-type Metal-Oxide Semiconductor (NMOS) transistor and even for more complex gates than a simple inverter, resulting in the same effect and impact. Eventually, as the variation of the supply voltage affects all transistors and gates between two register stages, lowering the supply voltage through voltage glitches or underpowering will increase the maximum propagation delay of the combinational logic. As a consequence, the inequality in Equation 8.1 might be violated. Hence, as for clock glitches, the final result might not be stable at the input of the register resulting in the sampling of a faulty value.

### 8.2.3 Electromagnetic Pulsess

Another approach for fault injection into embedded devices, having higher precision than clock or voltage glitches but still at reasonable equipment and expertise requirement [BKH+19], uses *electromagnetic pulses*. Again, we briefly summarize related state-of-the-art literature and discuss the physical mechanisms responsible for the manifestation of faults.


(a) D Flip-Flop. (b) Simplified diagram.

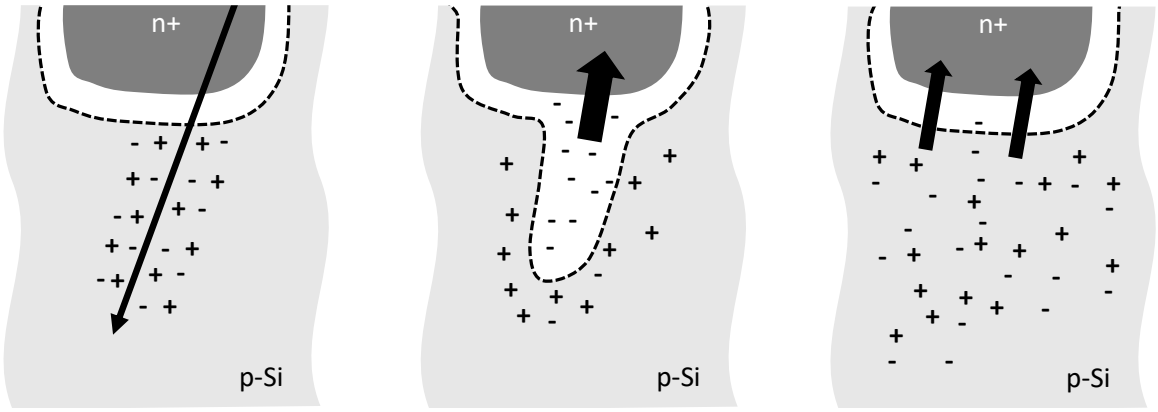
Figure 8.3: Physical effects due to faults caused by EMPs [DLM19, DLM21].

**State of the Art.** Over the last years, the understanding of the underlying mechanism of faults caused by Electromagnetic Pulses (EMPs) changed. While in 2012, Dehbaoui et *al.* [DDRT12] performed some experiments on microcontrollers and FPGAs leading to the conclusion that Electromagnetic Fault Injection (EMFI) can be explained by timing violations of the critical path (as for clock glitches), two years later, Ordas et *al.* [OGT+14] demonstrated that timing faults cannot capture and describe the complete behavior of EMFI. In the following years, they performed further experiments and eventually deduced a *sampling fault model* [OGM15, OGM17]. Most recently, Dumont et *al.* [DLM19, DLM21] were eventually able to explain the physical behavior for the sampling fault model and confirmed its correctness by conducting several simulations and additional practical experiments. For this, we will summarize the latest findings and explain the underlying physical mechanism responsible for fault injections caused by EMP in the following paragraph.

**Physical Mechanism.** Any EMFI setup usually consists of a ferrite core, a coil, and a voltage pulse generator to establish a magnetic field used to induce a current in any wire loop based on the theory developed by M. Faraday. Particularly in ICs, those wire loops are the power and ground networks, where the induced current leads to a voltage swing $S$ between the power and ground grid (cf. Figure 8.3b for the effects of an undershoot).

However, in the following, we limit our explanations on D Flip-Flops (DFFs) (see Figure 8.3a), as they are the main elements in digital ICs susceptible to EMFI. The aforementioned voltage drop caused by the EMP consequently pulls the potential of the clock signal and the input signal $D$ down, as visualized in a simplified diagram in Figure 8.3b. More precisely, this behavior is caused by the falling edge of the swing $S$ and can therefore be associated with the first EMP generated by the rising edge of the voltage pulse generator supplying the EM probe. With the rising edge of $S$ – caused by the second EMP generated through the falling edge of the pulse generator supplying the EM probe – the circuit recovers the original state.

Here, the authors of [DLM21] describe the recovering phase as a race between the clock signal and the input signal $D$. A successful fault injection is performed only if the clock signal wins the race, meaning that the clock recovers faster than the input signal $D$, and therefore the DFF stores a faulted value (cf. Figure 8.3b). Note, however, that not only a negative swing can be induced, but also a positive swing, then leading to an overshoot instead of undershoot. While




(a) Ionizing. (b) $I_{\text{drift}}$. (c) $I_{\text{diff}}$.

Figure 8.4: Physical effect of Laser Fault Injection as introduced in [Bau04].

the negative polarity often leads to bit-resets, the positive overshoot induces bit-set faults with higher probability. For more details, we refer the interested reader to [DLM19, DLM21].

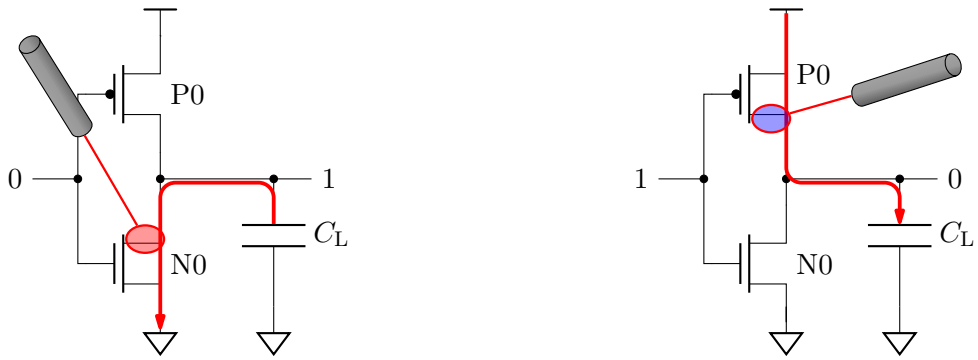
In summary, Dumont et *al.* showed that EMFI causes sampling faults which can also be modeled as set or reset faults in memory elements such as DFFs. Additionally, their most recent work in [DLM21] demonstrates a very fine spatial resolution of EMFI, surprisingly independent of the electromagnetic probe geometry.

### 8.2.4 Laser Fault Injection

Laser fault injection using focused laser beams was initially presented in 2002, in the seminal work of Skorobogatov and Anderson [SA02]. Since then, many follow-up works have been presented and improved the potential of laser-assisted fault injection. Before we summarize and explain the physical effects of laser-induced faults, we dedicate the next paragraph to the current progress and state-of-the-art research with respect to optical fault-injection methods.

**State of the Art.** The first case study of laser fault injections, presented in the seminal work [SA02] of Skorobogatov and Anderson, was designed for a target platform built in a quite large 1 200 nm technology. However, in the following years, several other works studied the influence of laser beams to the operation of ICs and improved the application for lasers as a fault-injection mechanism. For instance, in 2013, Roscian et *al.* [RSDT13] already targeted a 250 nm technology and performed investigations on the underlying fault model. In the following year, Courbon et *al.* [CLFT14] demonstrated the tremendous accuracy of laser fault injection and used it to characterize registers instantiated in a 90 nm technology. Similarly, Selmke et *al.* [SBHS15] investigated the accuracy of laser-induced faults for a 45 nm technology, but concluded that precise fault injections into memory cells become more difficult for smaller technologies.

However, not only memory cells, but also any combinational gate of a digital IC is susceptible to laser-induced faults, as was shown in 2016 by Schellenberg et *al.* [SFG+16]. In this work, the authors used successful injections of faults to perform a *fault sensitivity analysis*, also possible for smaller target technologies. Most recently, Dutertre et *al.* [DBC+18] successfully performed fault




(a) NMOS. (b) PMOS.

Figure 8.5: Sensitive drain regions for laser fault injection [RSDT13].

injections on an AES implemented on a very small 28 nm technology. However, although the hardness of laser fault injection varies with the targeted geometry size, the basic fundamentals and physical effects can be traced back to the same phenomena.

**Physical Mechanism.** Figure 8.4 exemplary shows the fundamental physical effect when a focused laser beam hits and affects an NMOS transistor. More precisely, the laser beam starts an ionizing process in a PN-junction while along the laser injection path a dense distribution of electron-hole pairs is produced (cf. Figure 8.4a). Afterwards, the carriers are rapidly collected by the electric field and the charge is compensated, resulting in a reduced voltage on that node while eventually, a temporary drift current arises as visualized in Figure 8.4b. However, shortly afterwards (usually at a magnitude of a few picoseconds) the funnel collapses and a small diffusion current dominates the collection process, which again is shown in Figure 8.4c. For more details, we refer the interested reader to [Bau04, WA08].

As a consequence, the effect of producing a temporary drift current $I_{\text{drift}}$ in a PN-junction of a transistor can be used to alter the state of a gate. For the sake of simplicity, we consider the CMOS inverter given in Figure 8.5 as a minimal example, where subsequent connected gates are simplified and modeled by a load capacity $C_L$. As a first step, we assume the input of the inverter to be zero and the output to be one, as visualized in Figure 8.5a. Once an adversary hits the drain region of the NMOS transistor with the help of a focused laser beam, the output state of the inverter may change. In particular, the high drift current through the transistor forces a discharge process of the output node, i.e., the electrical charge from $C_L$ is moved such that the output changes from one to zero. Note, this effect can only occur if the temporary drift current is larger than the current flowing through the PMOS transistor, which still conducts correctly. Hence, if the drift current $I_{\text{drift}}$ collapses, the output node will eventually switch back to its former high level. This results in a temporary injected fault which is called Single Event Transient (SET) (or *transient* Single Event Upset (SEU) [Pet11]). A similar effect occurs when the input of the inverter is one and the output is zero, but in this case the laser beam has to hit the drain region of the PMOS transistor (instead of the NMOS transistor) in order to switch the output node from zero to one, i.e., to load $C_L$ [RSDT13].

In summary, we can state that this fault-injection mechanism either causes bit-set or bit-reset faults considering the given inverter. Further, the bit-set or bit-reset faults can occur

Table 8.1: Functions included in $\mathcal{U}$ and in $\mathcal{B}$.

| **Inputs** | | $\mathsf{u}_i(x_0) \in \mathcal{U}$ | | | | $\mathsf{b}_i(x_0, x_1) \in \mathcal{B}$ | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_0$ | $x_1$ | $\mathsf{u}_0$ | $\mathsf{u}_1$ | $\mathsf{u}_2$ | $\mathsf{u}_3$ | $\mathsf{b}_0$ | $\mathsf{b}_1$ | $\mathsf{b}_2$ | $\mathsf{b}_3$ | $\mathsf{b}_4$ | $\mathsf{b}_5$ | $\mathsf{b}_6$ | $\mathsf{b}_7$ | $\mathsf{b}_8$ | $\mathsf{b}_9$ | $\mathsf{b}_{10}$ | $\mathsf{b}_{11}$ | $\mathsf{b}_{12}$ | $\mathsf{b}_{13}$ | $\mathsf{b}_{14}$ | $\mathsf{b}_{15}$ |
| 0 | 0 | – | | | | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | – | | | | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |

as temporary faults in both, combinational logic (i.e., logic gates) or in memory gates (e.g., registers). However, in case the attacker targets memory gates, the stored value will be altered, which is called a *static* SEU [Pet11], as this transient fault cannot be recovered while transient faults in combinational gates may be recovered by sufficient long clock periods (in comparison to the duration of the fault).

### 8.2.5 Miscellaneous Mechanisms

Besides the mechanisms introduced above, a few more techniques can be found in the literature, such as body biasing [MTOL12, O'F20], overpowering [CML+11], temperature [Sko09, HS13], and X-Ray beams [ABC+17]. However, they only have a minor or auxiliary role in practice in comparison to the mechanisms described above and therefore we decided to exclude them from following considerations in our work, although our versatile concept still allows modeling these mechanisms.

## 8.3 Concept

Given the broad range of physical fault-injection mechanisms that we surveyed in the previous section, our efforts in this section focus on a consolidated and unified model for fault-injection adversaries, ideally covering all previously introduced concepts. Since we focus on fault-injection techniques and adversaries for physical hardware and digital ICs, we utilize the circuit model introduced in Section 5.1. In this section, we introduce formal definitions of fundamental concepts as well as initial assumptions and limitations. Then, based on those definitions and assumptions, we propose and describe our generalized fault-injection adversary model in more detail.

### 8.3.1 Fault Model

For the description of faults and fault propagation in digital logic circuits, we first introduce the two sets of *unary* and *binary* Boolean functions. More precisely, the set of *unary* functions is given as $\mathcal{U} = \{ \mathsf{u} \mid \mathsf{u} : \mathbb{F}_2 \to \mathbb{F}_2 \}$ while the set of *binary* function is defined as $\mathcal{B} = \{ \mathsf{b} \mid \mathsf{b} : \mathbb{F}_2^2 \to \mathbb{F}_2 \}$.

In general, for a Boolean function $\mathsf{F} : \mathbb{F}_2^p \to \mathbb{F}_2^q$, we can construct $2^{q \times 2^p}$ distinct Boolean functions in $p$ variables and $q$ output bits, i.e., we have $|\mathcal{U}| = 4$ unary and $|\mathcal{B}| = 16$ binary functions. Further, the specific assignments of all possible functions for $\mathcal{U}$ and $\mathcal{B}$ are presented in Table 8.1.

As described in Section 5.1, we consider a limited set $\mathcal{G}_\mathrm{c}$ of combinational gates. More specifically, we consider a single unary gate $\{\mathsf{not}\} \in \mathcal{G}_\mathrm{u}$ that executes the unary Boolean function $u_0$ on its input. Further, the binary gates $\{\mathsf{and}, \mathsf{nand}, \mathsf{or}, \mathsf{nor}, \mathsf{xor}, \mathsf{xnor}\} \in \mathcal{G}_\mathrm{b}$ execute the Boolean functions $\mathsf{b}_i$ with $0 \leq i \leq 5$ on their inputs, respectively.

Then, given a DAG $\mathbf{D}$ modeling a digital circuit $\mathbf{C}$ as defined in Definition 22, we can associate each gate in the physical circuit, with a vertex $v \in \mathcal{V}$ of the graph, representing a combinational or memory gate by a Boolean function from the sets $\mathcal{U}$ or $\mathcal{B}$. For this, we define the following golden mapping $\tau_{golden}$ from gate to vertex and associated Boolean function $\mathsf{F}_g$:

$$\tau_{golden}\colon \begin{array}{lcllcl} \{\mathsf{not}\} & \mapsto & \{\mathsf{u}_0\} & \{\mathsf{or}\} & \mapsto & \{\mathsf{b}_2\} \\ \{\mathsf{reg}\} & \mapsto & \{\mathsf{u}_1\} & \{\mathsf{nor}\} & \mapsto & \{\mathsf{b}_3\} \\ \{\mathsf{and}\} & \mapsto & \{\mathsf{b}_0\} & \{\mathsf{xor}\} & \mapsto & \{\mathsf{b}_4\} \\ \{\mathsf{nand}\} & \mapsto & \{\mathsf{b}_1\} & \{\mathsf{xnor}\} & \mapsto & \{\mathsf{b}_5\} \end{array}$$

Given the gate-function mapping and the abstract representation of a digital circuit in terms of a DAG, we can formally describe the effects and propagation of an injected fault.

**Definition 27** (Fault)**.** *A* fault *can occur in a digital circuit* $\mathbf{C}$ *if and only if a gate* $g \in \mathcal{G}$ *within the circuit does not evaluate according to its associated Boolean function* $\mathsf{F}_g$*.*

**Definition 28** (Error)**.** *In a digital circuit* $\mathbf{C}$ *an* error *occurs if a wrong value is visible at the output of the circuit. An error is always caused by a fault.*

Considering our limited set of combinational and memory gates, a fault in a combinational gate occurs immediately if the considered $g \in \mathcal{G}_\mathrm{c}$ evaluates to an incorrect result $z'$ with $z' \neq \mathsf{F}_g(\mathbf{x})$. For registers $g_\mathrm{r} \in \mathcal{G}_\mathrm{m}$, faults will only manifest in synchronization to the provided clock signal such that $z' \neq \mathsf{F}_r(x)$.

Moreover, if a fault occurs in a gate of a digital circuit, i.e., the faulted gate evaluates incorrectly, this fault may also have an impact on subsequent gates. More specifically, if a faulty signal $z'$ is input to further gates, these gates may evaluate correctly according to their associated function but still provide wrong results due to incorrect inputs. In general, this effect is called *fault propagation*. Two different scenarios of fault propagation are given in Example 2.

**Example 2.** *In this example we assume a gate* $g \in \mathcal{G}_c$ *producing a faulty output* $z' \in \mathbb{F}_2$*. The faulty output* $z' = 1$ *is the first input* $x_0$ *to a gate* $g_2 = \{\mathsf{or}\}$ *while the second input* $x_1 = 1$*. In this case, fault propagation will stop immediately, as the output of* $g_2$ *will be* 1 *regardless of the first input. However, given that* $x_1 = 0$*, then, upon correct inputs,* $g_2$ *would evaluate to* 0*, however, due to* $x_0 = z' = 1$*, the fault will propagate through* $g_2$ *and may affect further gates in the circuit.*

**Definition 29** (Fault Scenario)**.** *We define a* fault scenario *as the occurrence of a fault in a target gate* $g \in \mathcal{G}$ *under a given input* $\mathbf{x} \in \mathbb{F}_2^p$ *to the circuit* $\mathbf{C}$*.*

Hence, each specific fault in a target gate $g \in \mathcal{G}$ creates a unique fault scenario for each valid input $\mathbf{x} \in \mathbb{F}_2^p$. Therefore, the input size $p$, the amount of considered gates, and the number of valid faults for each gate (more details will be given in Section 8.3.2) determine the total amount of fault scenarios $N_\mathrm{scenario}$ for a given circuit $\mathbf{C}$.

### 8.3.2 Consolidated Adversary Model

Introducing two abstraction levels for a digital logic circuit, we can explain and introduce our generic and consolidated fault-injection adversary model. As a consequence, this allows us to create a dedicated fault-injection adversary model that can be adjusted by a set of parameters introduced afterwards.

#### Initial Assumptions

For this, let us define and list some initial assumptions in order to provide a reasonably complex fault-injection model for digital logic circuits. First, we assume that all primary inputs to a target circuit $\mathbf{C}$ are fault-free since inputs that are already faulted can never be recognized by any fault model framework or by countermeasures that should be evaluated. Second, we do not consider any routing information of the circuit in our fault model since these undermine our attempt to create a generic model. We work on a netlist level which can be perfectly mapped to DAGs as described in Section 5.1. Nevertheless, this abstraction level allows us to attach timing information, i.e., propagation delays of the gates, to each node in the DAG representing the physical logic gates. Third, we do not consider fault probabilities, hence, purely focusing on a quantitative rather than qualitative analysis. Finally, we do not specifically consider *persistent* faults in our fault adversary model. If persistent faults should be modeled, it can still be accomplished within our assumption by triggering a specific fault on each evaluation. This, however, is not part of a fault model but rather part of the utilized framework integrating the fault adversary model.

#### Abstraction Levels

The description of a circuit $\mathbf{C}$ as a DAG $\mathbf{D}$ allows us to separate the fault modeling into two abstraction levels – a *structural level* and a *functional level*. On the structural level, we consider the edges and vertices of the DAG, i.e., the wires in $\mathbf{C}$ connecting the circuit gates. This gives us the possibility to model, describe, and track the propagation of faults through the entire circuit. Additionally, the structural level provides information about the placement of synchronization points, i.e., the memory gates. This information is important since faults ultimately will manifest in register stages which we demonstrate in Section 8.4. However, the actual faults are injected directly in combinational gates $g_\mathrm{c} \in \mathcal{G}_\mathrm{c}$ or in memory gates $g_\mathrm{r} \in \mathcal{G}_\mathrm{m}$ where both types of gates describe the functional level of $\mathbf{C}$ through the associated Boolean functions given in $\tau_{golden}$ (cf. Section 8.3.1).

#### Modeling Faults

On a very abstract (and simplified) level, we model a single fault by altering the associate function of the target gate to an arbitrary function within the same domain, i.e., defined over the same number of inputs and outputs. In particular, faults injected into a gate $g_\mathrm{u} \in \mathcal{G}_\mathrm{u}$ or in a memory gate $g_\mathrm{r} \in \mathcal{G}_\mathrm{m}$ are modeled by exchanging the associated function with a function $\mathsf{u} \in \mathcal{U}$. Similarly, faults injected into a gate $g_\mathrm{b} \in \mathcal{G}_\mathrm{b}$ are modeled by exchanging the associated function of $g_\mathrm{b}$ with a function $\mathsf{b} \in \mathcal{B}$.

In this sense, for each fault scenario, the DAG of the circuit is re-evaluated and updated, such that for each vertex $v \in \mathcal{V}$ of the graph, the associated functions are selected from $\tau_{golden}$ or a

fault type is chosen from a *fault model* $\tau_{faulty}$, depending on whether the fault event occurred in the corresponding gate or not, such that:

$$v_g = \begin{cases} \tau_{golden}(g) & g \text{ is fault-free} \\ \tau_{faulty}(g) & g \text{ is faulted} \end{cases}, \forall g \in \mathbf{C}$$

Notably, this model provides a generic approach to map various fault types to a circuit implemented in hardware. To this end, we further define a notation which allows us to denote mappings where several gates are mapped to the same Boolean function. For example, given a subset of gates $\mathcal{G}_{\text{sub}} \subset \mathcal{G}_{\text{b}}$ and each of the gates $g \in \mathcal{G}_{\text{sub}}$ should be mapped to the functions $\mathsf{b}_{i_0}$ and $\mathsf{b}_{i_1}$ in case a corresponding gate in $\mathbf{C}$ is faulty, we denote the underlying mapping $\tau_j$ as $\tau_j : \mathcal{G}_{\text{sub}} \mapsto \{\mathsf{b}_{i_0}, \mathsf{b}_{i_1}\}$ for a specific fault model $j$. However, to meet realistic scenarios for an actual attacker, we further introduce a set of parameters which allows us to constrain the generic model and customize it depending on given circumstances.

### Parametric Adversary Model

To this end, we introduce the following three parameters to describe the limitations of an adversary: $n$, $t$, and $l$. While the first parameter $n$ defines the power of the attacker in terms of how many faults can be injected at the same time, the second parameter $t$ defines the *type* of the faults. Finally, $l$ limits the circuit *locations* where the faults can occur, i.e., the gates of the circuit that can be targeted by the adversary. In the following, we present more details about the three parameters and their design rationales.

**Number of Fault Events $n$:** The parameter $n$ sets the total number of faults that can occur at the same time and therefore it constrains the power of an attacker in terms of simultaneously injected faults. Hence, when modeling adversarial fault injections in a digital circuit $\mathbf{C}$, $n$ can be selected from $\mathcal{N} = \{1, 2, ..., N\}$ with $N = |\mathcal{V}|$, i.e., $N$ is equal to the total number of combinational and memory gates that are available in $\mathbf{C}$.

By selecting $n \in \mathcal{N}$, we assume that an attacker is able to inject up to $n$ faults, meaning that we consider all possible fault scenarios with $n' \leq n$ faults. This assumption is well established in literature when evaluating countermeasures against fault-injection attacks [SMG16, RSBG20]. However, even if we select $n$ as an upper bound, we still might observe more than $n$ faults manifesting in a register stage or primary output due to fault propagation. We further explain this phenomenon in Example 3.

**Example 3.** *For this example, we assume that we model an attacker with $n = 1$ and that a fault is injected into the* nand *gate in Figure 8.6. Although $n$ is constrained by 1, the fault can propagate through the* and *and* xor *gate such that three errors would eventually manifest at the primary output register stage. Hence, $n$ only indicates the number of faults an attacker is able to inject but it does not give any information about the total number of errors that will occur at the output of the circuit. This behavior was also mentioned by Aghaie et al. in [AMR+20].*

**Fault Type $t$:** The fault type $t$ can be selected from a set $\mathcal{T} = \{\tau_{sr}, \tau_s, \tau_r, \tau_{bf}, \tau_{fm}\}$ which contains different fault models $\tau_j$. Each of these fault models describes how a gate from a target

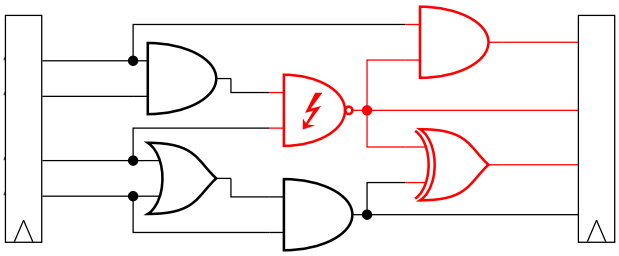


Figure 8.6: Influence of a single fault on subsequent gates.

circuit $\mathbf{C}$ is mapped to a function $\mathsf{u} \in \mathcal{U}$ or $\mathsf{b} \in \mathcal{B}$ in the resulting DAG $\mathbf{D}$. In this paragraph, we introduce common fault models used to describe different fault-injection mechanisms.

For this, we define $\tau_{sr}$ as a fault model where each gate from $\mathcal{G}$ is mapped to the *set* or *reset* function. We decided to use the terms *set* and *reset* instead of *stuck-at-one* and *stuck-at-zero* since the physical fault mechanisms of electromagnetic pulses and laser fault injections cause bit-sets or bit-resets by charging or discharging nodes in the digital circuit. Even if in the presence of clock glitches the fault mechanism could be described by a stuck-at behavior, we stay with the terms *set* and *reset* in the remainder of this work to highlight that the faulty behavior is caused by an active attacker. Note, however, that the choice of terminology does not affect the underlying modeling procedure but both physical mechanisms can be modeled similarly. Particularly, the faulty behavior of a gate $g_{\mathrm{u}} \in \mathcal{G}_{\mathrm{u}}$ or a memory gate $g_{\mathrm{r}} \in \mathcal{G}_{\mathrm{m}}$ is modeled by the function $\mathsf{u}_2(x) = 0$ or by $\mathsf{u}_3(x) = 1$ with $x \in \mathbb{F}_2$ (cf. Table 8.1). Hence, we apply a mapping that is described by $\{\mathsf{not}, \mathsf{reg}\} \mapsto \{\mathsf{u}_2, \mathsf{u}_3\}$. Similarly, a faulty gate $g_{\mathrm{b}} \in \mathcal{G}_{\mathrm{b}}$ is modeled by one of the functions $\mathsf{b}_6(\mathbf{x}) = 0$ or $\mathsf{b}_7(\mathbf{x}) = 1$ with $\mathbf{x} \in \mathbb{F}_2^2$, describing a reset or set fault, respectively. In this case, the mapping is formally described by $\{\mathcal{G}_{\mathrm{b}}\} \mapsto \{b_6, b_7\}$. In essence, the mapping $\tau_{sr}$ serves as a baseline for most of the fault-injection mechanisms introduced in Section 8.2, however, more details about the connection between the physical behavior and the proposed parameter selections are given in Section 8.4.

To allow more fine-grained evaluations, we additionally define the mappings $\tau_s$ and $\tau_r$ which describe only set or only reset fault, respectively. Hence, the mapping $\tau_r$ defines $\{\mathsf{not}, \mathsf{reg}\} \mapsto \{\mathsf{u}_2\}$ for unary gates and $\mathcal{G}_{\mathrm{b}} \mapsto \{\mathsf{b}_6\}$ for binary gates. Similarly, $\tau_s$ defines $\{\mathsf{not}, \mathsf{reg}\} \mapsto \{\mathsf{u}_3\}$ for unary gates and $\mathcal{G}_{\mathrm{b}} \mapsto \{\mathsf{b}_7\}$ for binary gates. This distinction can be useful for specific primitives or technologies where a fault injection can either cause set or reset faults only. Examples of such primitives are NOR flash memories where only bit-set faults occur as shown and explained in [CMD+19].

Another common fault model that can be found in the literature is based on bit-flips which we describe by the mapping $\tau_{bf}$. In this case, we map each gate from $\mathcal{G}$ to its inverse gate resulting in the following fault model:

$$\tau_{bf}\colon \quad \begin{array}{lcllcl} \{\mathsf{not}\} & \mapsto & \{\mathsf{u}_1\} & \{\mathsf{and}\} & \mapsto & \{\mathsf{b}_1\} \\ \{\mathsf{reg}\} & \mapsto & \{\mathsf{u}_0\} & \{\mathsf{nand}\} & \mapsto & \{\mathsf{b}_0\} \\ \{\mathsf{or}\} & \mapsto & \{\mathsf{b}_3\} & \{\mathsf{xor}\} & \mapsto & \{\mathsf{b}_5\} \\ \{\mathsf{nor}\} & \mapsto & \{\mathsf{b}_2\} & \{\mathsf{xnor}\} & \mapsto & \{\mathsf{b}_4\} \end{array}$$

Each gate is modeled by a function returning the inverse of the values that would be returned by the original gate.

Eventually, we intentionally leave space for custom definitions of fault models $\tau_{fm}$ in $\mathcal{T}$ to provide an adversary model that is as generic as possible while at the same time already covering common fault types and models. For this, we introduce one custom mapping $\tau_{nang15}$ in Section 8.4 and guide the reader through the process of precisely defining and modeling an attacker that uses a laser to inject fault events in a Nangate 15 nm technology.

**Fault Location $l$:** The fault location $l$ is the third parameter which is necessary to properly describe fault injections in our generic adversary model. We define the set $\mathcal{L} = \{\mathrm{c}_i, \mathrm{m}, \mathrm{mc}_i\}$ in order to distinguish between different areas on the structural level of a circuit $\mathbf{C}$. The first choice covers and models fault injections that solely affect combinational logic gates, i.e., gates from $\mathcal{G}_\mathrm{c}$. Here, $\mathrm{c}_\infty$ considers all combinational gates available in the circuit under test as targets for fault injections. A more fine-grained separation of combinational gates is also possible and denoted by the index $i$ while the separation is performed based on the gates' *propagation delays*. More precisely, we exploit that each gate has a specific maximum propagation delay which can be extracted from a CMOS library or for the sake of simplicity be assumed to be the same for each gate. Based on the individual propagation delays of the single gates, we approximately compute the Data Arrival Time (DAT) for each combinational gate $g \in \mathcal{G}_\mathrm{c}$ and access the corresponding value by $\mathtt{t}(g)$. Another approach could incorporate the notion of slack (i.e., the difference between the DAT and the Data Required Time (DRT)). However, we decided to rely our model just on the DAT since it is independent of the used clock frequency and can be better applied to actual circuits under test (see Example 4 and Section 8.4.1). In addition to the computation of the DAT for each gate $g \in \mathcal{G}_\mathrm{c}$, we define a set $\mathcal{G}_\mathrm{regin}$ that contains all combinational gates driving registers. The DAT of gates $g \in \mathcal{G}_\mathrm{regin}$ are used to create an ordered set $\mathcal{P} = \{t_0, t_1, ...t_{T-1}\}$ where $t_0 > t_1 > ... > t_{T-1}$ and $T \leq |\mathcal{G}_\mathrm{regin}|$. This allows us to create clusters of gates defined by

$$\mathcal{G}_{\mathrm{cluster},i} = \{\, g \in \mathcal{G}_\mathrm{regin} \mid \mathtt{t}(g) \geq t_i, t_i \in \mathcal{P} \,\}\,. \tag{8.3}$$

Finally, setting the location parameter $l = \mathrm{c}_i$ corresponds to fault injections that are performed in the subset $\mathcal{G}_{\mathrm{cluster},i}$ with $i < T$. The following example describes the process of generating $\mathcal{G}_{\mathrm{cluster},i}$ based on the circuit given in Figure 8.7.

**Example 4.** *First, for the sake of simplicity, we assume that each gate in Figure 8.7 has the same propagation delay $t_{gate}$. Second, we determine the set of gates whose outputs are connected to registers which is in our example $\mathcal{G}_{regin} = \{g_0, g_3, g_5, g_6, g_9\}$. The corresponding propagation delays are given by $\mathcal{P} = \{t_0, t_1, t_2\}$ with $t_0 = 4t_{gate}$, $t_1 = 3t_{gate}$, and $t_2 = t_{gate}$. Given this information, we construct the different subsets of $\mathcal{G}_{regin}$ representing the clusters and containing the following gates*

$$\mathcal{G}_{cluster,0} = \{g_9\} \qquad \mathcal{G}_{cluster,1} = \{g_3, g_5, g_6\} \cup \{g_9\}$$
$$\mathcal{G}_{cluster,2} = \{g_0\} \cup \{g_3, g_5, g_6\} \cup \{g_9\}.$$

Please note, by $c_\infty$, we describe fault injections targeting all combinatorial gates of the circuit under test.

Besides, setting $l = \mathrm{m}$, specifies fault injections where an attacker targets memory gates $g_r \in \mathcal{G}_\mathrm{m}$ only. The location parameter $l = \mathrm{mc}_i$ models faults that can occur in both types of gates where combinational gates can still be separated into subgroups.

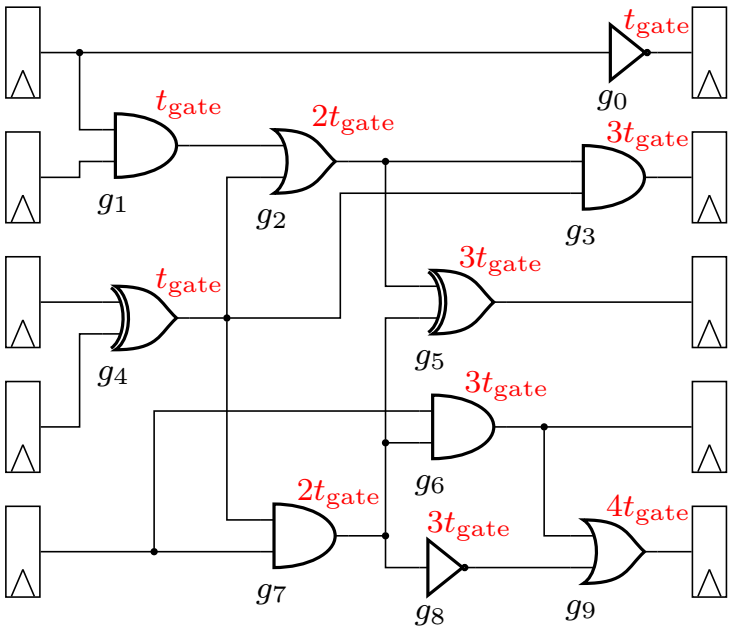



Figure 8.7: Propagation delays of different data paths in an exemplary circuit.

For reasons of clarity, Table 8.2 summarizes the available parameters with the corresponding options which are shortly described in the last column.

#### Instantiating Adversary Models

To bring together and connect the three introduced parameters $n$, $t$, and $l$, we define the function $\zeta(n, t, l)$. This allows us to instantiate different types of attackers and model the behavior of fault injections based on the committed parameter list. For example, we can regulate the strength by changing the fault type $t$, or determine the accuracy of the fault injections setting $n$ as a powerful attacker may be able to precisely inject single bit faults. However, one of the main advantages of introducing $\zeta$ is that we create a basis to allow comparability between different designs which should be evaluated regarding their protection against fault-injection attacks (under a given adversary model). In Section 8.4, we evaluate a cryptographic circuit using our generic adversary model to transfer our definitions to a practical instantiation.

## 8.4 Practical Instantiation

After we introduced our generic adversary model expressed through the corresponding function $\zeta$ to model adversaries with different capabilities, we show in this section how to map our theoretical considerations to real-world fault injection mechanisms and how to model associated adversaries. Therefore, we establish a connection between available fault injection mechanisms introduced in Section 8.2 and our findings from Section 8.3.

Further, to demonstrate the practical application of $\zeta$, we consider the ASCON S-box [DEMS16] as an example of a cryptographic primitive and potential target of FIA. The cor-

Table 8.2: Parameters to accurately model fault injections.

| Parameter | Options | Description |
|---|---|---|
| $n$ | $\mathcal{N} = \{1, 2, ..., N\}$ | Maximum number of fault events, $N$ depends on the application |
| $t$ | $\mathcal{T} = \{\tau_{sr}, \tau_s, \tau_r, \tau_{bf}, \tau_{fm}\}$ | $\tau_{sr}$: Fault model for set/reset fault<br>$\tau_s$: Fault model for set faults<br>$\tau_r$: Fault model for reset faults<br>$\tau_{bf}$: Fault model for bit-flips faults<br>$\tau_{fm}$: User-specified fault model |
| $l$ | $\mathcal{L} = \{\mathrm{c}_i, \mathrm{m}, \mathrm{mc}_i\}$ | $\mathrm{c}_i$: Faults in comb. gates only<br>m: Faults in memory gates only<br>$\mathrm{mc}_i$: Faults in all gates |

responding S-box circuit is depicted in Figure 8.8, exhibiting some interesting properties that provide a good starting point for the application and discussion of our concept. First, this circuit already consists of both gate types, i.e., combinational gates from $\mathcal{G}_\mathrm{c}$ and memory gates from $\mathcal{G}_\mathrm{m}$. Second, although the circuit is constructed on an almost regular pattern, fault propagation can be observed. More specifically, at the deepest logic level, the primary output $y_4$ is an input to the xor-gate which determines the output $y_0$. Hence, the structure of the ASCON S-box is perfectly suited to demonstrate and discuss different practical instantiations of our generic adversary model.

However, to facilitate a comparison of different fault injection mechanisms and associated adversary models, we first define the total number of effective faults $N_\mathrm{eff}$ as (single-bit) faults that eventually manifest in a primary output stage of a circuit $\mathbf{C}$. In our example, $N_\mathrm{eff}$ can be at a maximum five for each given fault scenario since the number of output registers is $q = 5$. Given that, the maximum number of possible faults $N_\mathrm{max}$ is limited by

$$N_\mathrm{max} = q \cdot N_\mathrm{scenario} = q \cdot 2^p \cdot \sum_{g \in \mathbf{C}} |\tau_j(g)| \tag{8.4}$$

under a considered fault model $j$.

As indicated in Equation 8.4, we consider each input combination to determine the maximum number of possible faults $N_\mathrm{max}$ because each valid input creates an independent fault scenario. We explicitly decided to follow this approach[1] for the considered example to allow a fairer comparison between the different instantiations of $\zeta$ at the end of this section. Note that for real-world applications (e.g., analyzing entire protected block-ciphers) with inputs $p \geq 64$ such an analysis is currently not possible. However, this is not a limitation of our adversary model (since our model targets the description of fault occurrences in hardware gates) but a limitation of exhaustively simulating fault injections in all available gates for all valid inputs which is not part of this work and still an open research question.

Please be aware, that the following analysis results only hold for an instantiation of the ASCON S-box as shown in Figure 8.8. Hence, in case the registers of the S-box are removed or any

[1]The corresponding source-code can be accessed at `https://nextcloud.seceng.rub.de/index.php/s/7ZmqynqYNnfPLw8`

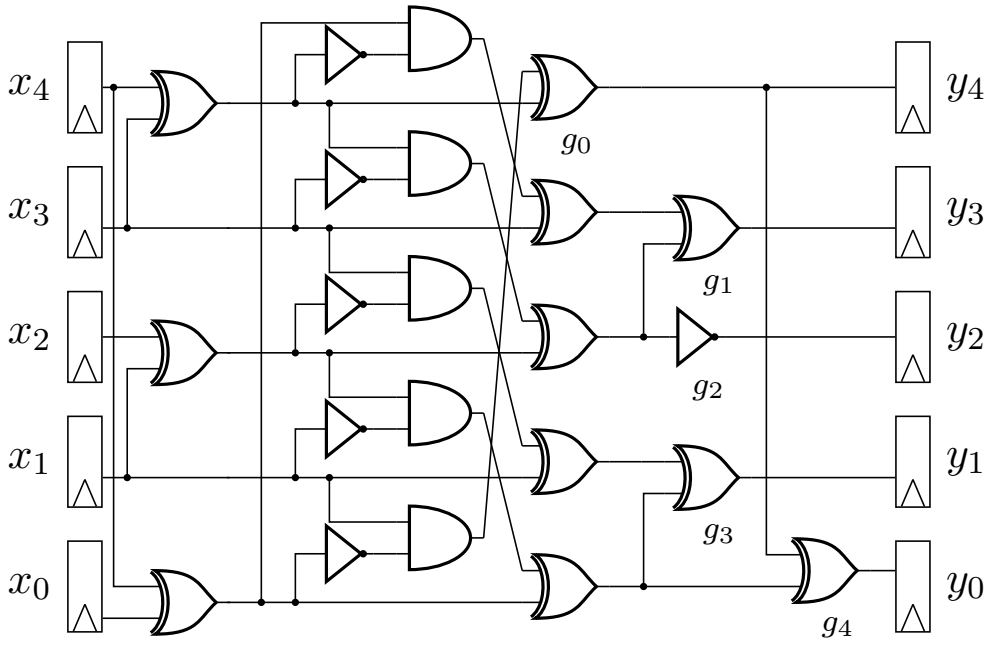



Figure 8.8: ASCON S-box [DEMS16].

combinational logic is added to the circuit, the analysis has to be re-performed. Consequently, the following examples only demonstrate the functionality of our proposed fault model and the mapping between theory and practice without any claims for generality.

### 8.4.1 Clock Glitches

As first step, in order to model clock glitches and the underlying fault mechanism explained in Section 8.2.1, we instantiate our adversary model as $\zeta(n, \tau_{sr}, \mathrm{c}_i)$ with $n \in \mathcal{N} = \{1, 2, ..., |\mathcal{G}_{\mathrm{cluster},i}|\}$. By setting $t = \tau_{sr}$ and $l = \mathrm{c}_i$, we consider *set* and *reset* faults occurring in combinational gates $g$ of the cluster $\mathcal{G}_{\mathrm{cluster},i}$ with $0 \leq i \leq T-1$.

This choice can be justified when looking closer at the origin of faults injected by clock glitches. Particularly, if the attacker manages to decrease the clock period of a clock cycle, this makes registers sample their input signals early. Then, a fault occurs if the internal logic cannot propagate the correct signals timely (negative slack) and the current input to a register differs from the correct input. As a consequence, there are several possibilities for how false inputs can occur, depending on the duration of the clock glitch. For short clock glitches[2] with $T_{\mathrm{clk}} + \delta < t_{T-1}$ the propagation of the data could be stopped before it stabilizes at any of the outputs of the gate $g \in \mathcal{G}_{\mathrm{regin}}$ such that in a worst-case scenario all registers connected to these gates sample wrong results. This corresponds to a fault model instantiated as $\zeta(n, \tau_{sr}, c_{T-1})$ with $n \in \mathcal{N} = \{1, 2, ..., |\mathcal{G}_{\mathrm{cluster},T-1}|\}$. We decided to allow $n$ to be an element from $\mathcal{N}$ and not to fix $n = |\mathcal{G}_{\mathrm{cluster},T-1}|$ in order to provide more fine-grained analysis possibilities. Note, however, that the number of faults occurring in one cluster cannot be controlled by an attacker using

[2]By *short clock glitches* we refer to a clock cycle with a drastically decreased duration (i.e., the time between two rising edges) compared to the cycle duration of the undisturbed clock.

clock glitches since they solely depend on the previous data processed by the circuit. Due to this behavior, the faults can be modeled best by set and reset faults and assuming that the previous state of each bit was either one or zero. However, the attacker can precisely control the clusters where the faults should occur by adjusting the duration of the clock glitch. This capability is modeled by selecting different $\mathrm{c}_i$ which corresponds to fault injections in combinational gates that are included in $\mathcal{G}_{\mathrm{cluster},i}$ with $0 \leq i \leq T-1$. The least number of combinational gates is affected by setting $l = \mathrm{c}_0$ while the maximum number of affected combinational gates is given by $l = \mathrm{c}_\infty$.

For clarification of these decisions, we transfer this model to the ASCON S-box depicted in Figure 8.8. Instead of assuming the same propagation delay for each gate, we now extract the propagation delays from the Open-Cell Library[3]. Here, we use the slowest propagation delays for an input transition of 0.198 5 ns and a load capacity of 26.016 2 fF for the gates with a driving strength of one. In our example, we need the propagation delays of xor-gates, not-gates, and and-gates which are 0.24 ns, 0.26 ns, and 0.20 ns, respectively. Applying these propagation delays to the ASCON S-box, results in three subsets $\mathcal{G}_0 = \{g_1, g_3, g_4\}$, $\mathcal{G}_1 = \{g_1, g_2, g_3, g_4\}$, and $\mathcal{G}_2 = \{g_0, g_1, g_2, g_3, g_4\} = \mathcal{G}_{\mathrm{regin}}$ with corresponding DATs of $t_0 = 0.118$ ns, $t_1 = 0.96$ ns, and $t_2 = 0.94$ ns that can be considered as targets for fault injections. Hence, considering $l = \mathrm{c}_0$ could lead to wrongly sampled bits in the registers $y_0$, $y_1$, and $y_3$ while $l = c_2$ could lead to a total of five effective faults manifesting in the registers.

Note, however, that any designer should use these values with caution since $t_1$ and $t_2$ are very close to each other. Due to process variations in chip manufacturing processes, these values could easily change such that faults in all corresponding gates could occur with the same likelihood. Hence, our model just considers an abstraction of the circuit under test.

Applying the specific adversary model $\zeta(1, \tau_{sr}, \mathrm{c}_0)$ to the ASCON S-box, we consider $2^5$ input combinations, three available gates to inject a set or reset fault, and five output registers to observe an error, resulting in a maximum number of $N_{\max} = 960$ faults. For each fault, we compare the resulting output $y_i'$ to the correct S-box output $y_i$ and for each output bit that is different from the correct one, we increase a fault counter which eventually results in $N_{\mathrm{eff}} = 96$ effective faults appearing at the output. Additionally, we instantiate our adversary model as $\zeta(5, \mathrm{sr}, \mathrm{c}_2)$ modeling a worst-case scenario where the clock glitch prevents data propagation in all gates $g \in \mathcal{G}_{\mathrm{regin}}$. In this case, there are $N_{\max} = 38\,720$ faults where $N_{\mathrm{eff}} = 13\,824$ are effective.

### 8.4.2 Voltage Glitches

As mentioned in Section 8.2.2, the fundamental physical behavior of voltage glitches (or under-powering) is very similar to clock glitches and is caused through the violation of Equation 8.1. By lowering the voltage, the right side of this inequality is increased such that the memory gates are triggered before all signals can propagate through the logic. As a result, this phenomenon can be modeled by the same adversary model as clock glitches. Hence, we model fault injections caused by voltage glitches also by a $\zeta(n, \tau_{sr}, \mathrm{c}_i)$ adversary with $n \in \mathcal{N} = \{1, 2, ..., |\mathcal{G}_{\mathrm{cluster},i}|\}$.

Obviously, applying the voltage glitch adversary model to our example generates the same results as described in Section 8.4.1.

[3] https://www.cs.upc.edu/~jpetit/CellRouting/nangate/Front_End/Doc/Databook/CornerList.html

### 8.4.3 Electromagnetic Pulses

The modeling of fault injections caused by EMPs can be conducted by instantiating the adversary model function as $\zeta(n, \tau_{sr}, \mathrm{m})$ with $n \in \mathcal{N}$. In Section 8.2.3, we explained that EMPs induce a positive or negative voltage pulse in the target circuit. These pulses produce a set or reset fault respectively. Hence, setting $t = \tau_{sr}$ perfectly models the physical mechanism of EMFI since the target gate functions are mapped to the set or reset function. Additionally, the choice for selecting memory gates as fault location $l$, matches the recently published results by Dumont et *al.* [DLM19, DLM21] who refined and confirmed the model of sampling faults caused by EMFI.

Applying the fault model function to the considered example depicted in Figure 8.8, leaves us with $2^5$ input combinations, 5 gates for the set and reset faults (ignoring primary inputs as we assume inputs to be correct), and five output registers, which eventually results in $N_{\max} = 1\,600$ possible faults at the primary output. All together, there are $N_{\text{eff}} = 160$ effective faults resulting in a fault rate of $r_{\text{fault}} = 10\,\%$ for the ASCON S-box under the given $\zeta(1, \tau_{sr}, \mathrm{m})$ adversary model.

### 8.4.4 Laser Fault Injection

At last, considering the mechanism of optical fault injections, an appropriate modeling is possible by defining the adversary model function as $\zeta(n, \tau_{fm}, \mathrm{mc}_\infty)$. This selection covers fault injections into combinational and memory gates likewise. As described in Section 8.2.4, a focused laser beam on a digital circuit charges or discharges specific nodes on transistor level. Hence, targeting memory gates, the value of a register can either be set or reset which needs to be covered in the custom fault mapping $\tau_{fm}$ defining $\{\mathsf{reg}\} \mapsto \{\mathsf{u}_2, \mathsf{u}_3\}$.

Additionally, the instantiation of the adversary function covers faults that directly occur in the combinational logic. Here, we define specified mappings for $\tau_{fm}$ between the instantiated gates $g \in \mathcal{G}_\mathrm{c}$ and the defined functions in $\mathcal{U}$ and $\mathcal{B}$. The simplest example – faults occurring in a CMOS inverter – was already discussed in Section 8.2.4 where the mapping $\{\mathsf{not}\} \mapsto \{\mathsf{u}_2, \mathsf{u}_3\}$ is applied. For the remaining gates from $\mathcal{G}_\mathrm{b}$, we now exemplary derive the mapping of an and gate designed in the 15 nm Open-Cell Library[4] which is depicted in Figure 8.9. Therefore, we will call the custom-defined mapping $\tau_{fm}$ in the following $\tau_{nang15}$ as it is tailored to the given example.

The and gate consists of six transistors where three transistors are NMOS and three are PMOS transistors. In our model, we assume that an adversary can affect any number of transistors available in a target gate. Hence, our parameter $n$ only describes the number of faults on gate level but does not distinguish the number of charged or discharged nodes. However, in case of the considered and gate, the attacker can easily change the function to a set or reset behavior by affecting the inverter stage. Additionally, it is possible to simultaneously inject a drift current $I_{\text{drift}}$ into the transistors N0 and N1 to force the gate to behave as an or gate. This is possible if $I_{\text{drift}}$ is larger than the current delivered by one of the PMOS transistors P0 and P1 such that the input node to the inverter can be discharged if either P0 or P1 conducts. In case both PMOS transistors conduct, the injected drift current would be too small to discharge the input

[4] `https://si2.org/open-cell-library/`

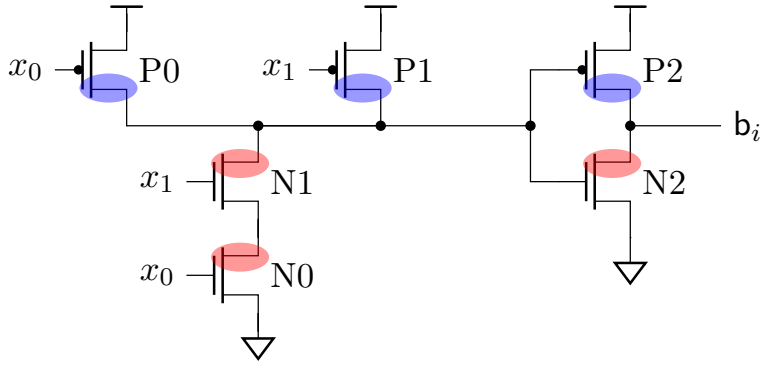



Figure 8.9: AND gate from the 15 nm Open-Cell Library. Blue areas mark drain regions of PMOS transistors, red areas mark drain regions of NMOS transistors.

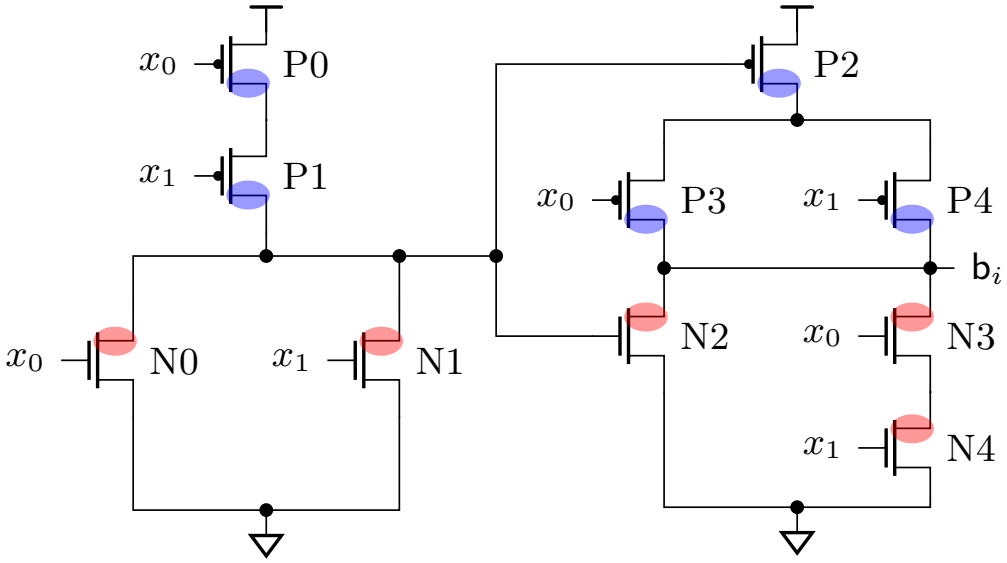



Figure 8.10: XOR gate from the Open Nangate 15 technology.

node to the inverter so that the output of the gate would be zero. These observations lead us to the mapping $\{\mathsf{and}\} \mapsto \{\mathsf{b}_2, \mathsf{b}_6, \mathsf{b}_7\}$ which is added to $\tau_{nang15}$.

Fault mappings for all remaining gates from $\mathcal{G}$ can be derived in a very similar way. As we require a specific mapping for a xor gate in order to evaluate the example from Figure 8.8, the corresponding schematic is shown in Figure 8.10. Again, the attacker can easily generate set and reset fault events by charging or discharging the output node. However, there are other possible modifications which allow the attacker to force the gate to behave as a nand gate or as an or gate. The former change can be achieved by discharging the input node to the second stage, i.e., by shooting on N0 or N1. To force the gate to behave as an or gate, the attacker has to hit one of the PMOS transistors P3 or P4. All together, we end up with the mapping $\{\mathsf{xor}\} \mapsto \{\mathsf{b}_1, \mathsf{b}_2, \mathsf{b}_6, \mathsf{b}_7\}$.

For the sake of completeness, we listed the corresponding mappings for all remaining gates from $\mathcal{G}_c$, describing all fault types in presence of Laser Fault Injection (LFI) for the Nangate

Table 8.3: Fault types for LFI on a Nangate 15 technology.

| | Mapped Functions | |
|---|---|---|
| **Gate** | *Functions from $\mathcal{U}$ and $\mathcal{B}$* | *Description* |
| not | $\{u_2, u_3\}$ | {set, reset} |
| and | $\{b_2, b_6, b_7\}$ | {or, set, reset} |
| nand | $\{b_3, b_6, b_7\}$ | {nor, set, reset} |
| or | $\{b_0, b_6, b_7\}$ | {and, set, reset} |
| nor | $\{b_1, b_6, b_7\}$ | {nand, set, reset} |
| xor | $\{b_1, b_2, b_6, b_7\}$ | {nand, or, set, reset} |
| xnor | $\{b_1, b_2, b_6, b_7\}$ | {nand, or, set, reset} |

15 nm technology, in Table 8.3. The first column shows the available gates of the technology. The second column indicates the mappings to the functions from $\mathcal{U}$ and $\mathcal{B}$ while the last column states the corresponding Boolean functions.

Evaluating the ASCON S-box, given the described adversary model instantiation $\zeta(1, \tau_{nang15}, \mathrm{mc}_\infty)$, can be accomplished by distinguishing between faults that are injected into registers and combinational gates. The former case reveals the same fault rate as the evaluation under faults caused by EMPs ($N_{\max} = 1\,600$ and $N_{\text{eff}} = 160$). The evaluation considering faults in combinational gates results in $N_{\max} = 11\,360$ possible faults while there are $N_{\text{eff}} = 1\,480$ effective faults. Thus, adding these numbers, results in $N_{\text{eff}} = 1\,640$ effective faults and $N_{\max} = 12\,960$ possible faults under the given $\zeta(1, \tau_{nang15}, \mathrm{mc}_\infty)$ adversary model.

### 8.4.5 Comparison of Fault-Injection Mechanisms

The previous sections illustrate that the introduced adversary models can describe the different fault injection mechanisms in a finer-grained fashion. A distinct differentiation between the available fault injection mechanisms based on the instantiated $\zeta$ is easily possible. It further demonstrates that the laser fault model $\zeta(n, \tau_{nang15}, \mathrm{mc}_\infty)$ is the most precise but also the most complex instantiation. Assuming that the same $n$ is selected, a design which is evaluated in $\zeta(n, \tau_{nang15}, \mathrm{mc}_\infty)$ and reports no effective faults, will also report no effective faults in the remaining adversary models. The adversary model $\zeta(n, \tau_{nang15}, \mathrm{mc}_\infty)$ covers all set and reset faults in combinational and memory gates which automatically includes all faults modeled by the adversary models for clock glitches, voltage glitches, and EMPs (again assuming the same $n$). However, this does not hold vice versa so that a design, which for example is evaluated and secure under the EM fault model $\zeta(n, \tau_{sr}, \mathrm{m})$, is not necessarily secure against attackers using optical-based fault injection mechanisms.

## 8.5 Case Study: Integration into VerFI

In this section, we demonstrate the practical application of our new adversary model while integrating it into the state-of-the-art verification tool for fault injections VerFI [AWMN20].

**VerFI.** VerFI is an open-source tool to verify hardware countermeasures against fault injection attacks presented at HOST in 2019 [AWMN20]. The tool works on netlist level and can be configured via a simulation file. This file contains information about the plaintext and key, which should be used for the analysis, different parameters that are cipher related (e.g., duration in clock cycles, port names, end condition), and parameters to define the fault injection. Given that, one can precisely specify the submodules which should be faulted, how many faults should be injected, and which fault injection type should be considered (toggle, stuck-at). Based on this information, VerFI analyzes the given circuit and reports the number of non-detected faults, detected faults, ineffective faults, and the total number of evaluated faults.

**Adjustments to VerFI.** In order to demonstrate the application of our proposed adversary model, we adapted VerFI such that it was able to work with user-defined fault mappings[5]. Therefore, we modified the `library`-file and extended the parameter list for each gate by fault mappings which are specified in form of Boolean expressions. Consequently, we adapted the parsing function which reads in the gates from the `library`-file and stores the corresponding parameters. The required expressions describing the fault mappings are evaluated and stored in LUTs which are used in the fault simulation step to generate the outputs of the faulty gates. Within this fault simulation step each valid combination of fault mappings for a set of target gates (gates in which faults are currently injected) is analyzed before the next set is determined.

**Analyzing a protected LED-64 implementation.** To demonstrate the evaluation of a protected block cipher, we selected an implementation of the lightweight block cipher LED-64 [GPPR11] taken from Impeccable Circuits [AMR+20]. The authors published a list of hardware implementations of common block ciphers where each cipher is implemented with different levels of protection[6]. We decided to use the LED-64 implementation where each state nibble is protected by four bits of redundancy. We constrained the allowed area for fault injections to the xor-gates adding the multiplication results in MixColumns for one resulting nibble and to the following 4-bit state register in the data path as well as in the redundancy. Altogether, the total number of target gates consists of 32 xor-gates and eight registers. The target circuit was analyzed for $n = 4$ while injecting faults in clock cycle 31 only. The plaintext and key were fixed to the values

$$p = \texttt{0x0123456789ABCDEF} \qquad k = \texttt{0xDEADBEEFDEADBEEF}$$

for all following analyses. Hence, compared to the previous examples, we performed a non-exhaustive evaluation with respect to the inputs.

Table 8.4 summarizes the evaluation results provided by the adjusted version of VerFI[7]. The upper three rows report the results for the fault models which were originally provided with VerFI (toggle, stuck-at-1, stuck-at-0) instantiated with our adversary model. The number of fault scenarios is the same for all three cases and is given by $\sum_{k=1}^{4} \binom{32+8}{k}$ since setting $n = 4$ also includes fault injections with $n < 4$.

[5]The adapted version of VerFI can be found at `https://nextcloud.seceng.rub.de/index.php/s/7ZmqynqYNnfPLw8`.

[6]`https://github.com/emsec/ImpeccableCircuits`

[7]Detailed results (for $n < 4$) can be found in the appendix in Section 15.1.

Table 8.4: Fault analysis of LED-64 using VerFI. The top three rows are results produced by using the proposed fault models in VerFI. The lower three rows report results obtained from an adapted version of VerFI reflecting our generic adversary model.

| Fault Model | Detected | Non-detected | Ineffective | Scenarios (sum) |
|---|---|---|---|---|
| $\zeta(4, \tau_{bf}, \mathrm{mc}_\infty)$ | 97 428 | 3 598 | 1 064 | 102 090 |
| $\zeta(4, \tau_{s}, \mathrm{mc}_\infty)$ | 96 660 | 497 | 4 933 | 102 090 |
| $\zeta(4, \tau_{r}, \mathrm{mc}_\infty)$ | 87 372 | 49 | 14 669 | 102 090 |
| $\zeta(4, \tau_{sr}, \mathrm{c}_7)$ | 1 520 | 14 | 162 | 1 696 |
| $\zeta(4, \tau_{sr}, \mathrm{m})$ | 1 520 | 14 | 162 | 1 696 |
| $\zeta(4, \tau_{nang15}, \mathrm{mc}_\infty)$ | 14 383 842 | 72 462 | 1 245 232 | 15 701 536 |

The lower three rows summarize the VerFI report for adversary models instantiated for clock and voltage glitches, electromagnetic pulses, and laser fault injections, respectively. The number of fault scenarios for the clock glitch model results in $\sum_{k=1}^{4} \binom{8}{k} \cdot 2^k$ because we selected for the location parameter $l = \mathrm{c}_7$ which includes all combinational gates with connected outputs to the considered registers. For each combination of $k$ target gates, there are $2^k$ possibilities to combine set and reset faults. Switching to $\zeta(4, \tau_{sr}, \mathrm{m})$ (i.e., modeling faults caused by electromagnetic pulses), results in $\sum_{k=1}^{4} \binom{8}{k} \cdot 2^k$ fault scenarios while only 14 scenarios are not-detected. This number depends on the underlying linear code which in this case has 14 valid codewords with a Hamming weight of four. Evaluating the countermeasure based on the adversary model describing laser fault injections, leads to the largest number of fault scenarios. The number is given by

$$\sum_{k=1}^{4} \sum_{j=0}^{k} \binom{32}{j} \cdot \binom{8}{k-j} \cdot 2^{(k+j)}.$$

The first binomial coefficient describes the number of faults occurring in the combinational gates while the second binomial coefficient determines the number of registers that are faulted. The last term determines the combinations of fault mappings that exist for one combination of $k$ target gates. However, evaluating the target countermeasure, results in the largest number of non-detected faults which mainly is caused by the increased number of fault scenarios.

We showed that our approach provides the possibility of instantiating a more fine-grained fault model. A target design can be analyzed under different adversary models which are tailored to the most common fault injection mechanisms. Additionally, using these results to compare the security to other protection schemes, is much more consistent and straightforward to accomplish.

## 8.6 Discussion

After all, we briefly summarize and discuss the benefits of a unified adversary model to describe fault injection attacks. First, using a unified fault adversary model allows a *distinct evaluation* of developed countermeasures and protection mechanisms under the same assumptions. Second, while our adversary model considers the physical behavior of actual fault injection mechanisms,

it is also designed to work as *generic* and *simple* as possible. This makes an application easy to use and allows a straightforward instantiation in practice. Third, the application of a consolidated fault adversary model enables the possibility to *compare* proposed countermeasures based on the same assumptions and limitations with respect to the attacker. In this sense, our model strives to fulfill these criteria since only a limited number of parameters are necessary to describe and model the adversary in a very *compact* way. In essence, the user directly conceives the properties of the instantiated adversary model and can compare it to evaluations of protection schemes under a similar adversary model which makes it highly *expressiveness*. Fourth, due to the generic form, the adversary model can naturally be adapted to other technologies and hardware primitives (e.g., different memory technologies). With this property the model achieves a high *transferability* being able to customize it to the given circumstances.

**Expansion to More Advanced Logic Gates.** Even given that we limited our fault model in Section 8.3 to unary and binary logic gates, it could be easily and without any restrictions expanded to more advanced logic gates. This includes standard logic gates (e.g., and, or) with more than two inputs and optimized gates like and-or-invert. To expand our adversary model, the user defines additional sets containing all functions with $p$-bit inputs with $p > 2$. The corresponding set would then contain $2^{(2^p)}$ different functions that would transform the $p$-bit input to a 1-bit output. Given the additional sets of functions, the used mappings in $\tau$ need to be adapted in order to accurately describe the occurring faults.

**Limitations of VerFI.** Despite the fact that we were able to integrate our adversary model into the fault verification tool VerFI, we see some limitations with respect to the evaluation results. In VerFI, the user can just evaluate the given design by fixing the plaintext and key to a constant value. This covers not all fault scenarios and could lead to false positives when evaluating a countermeasure against fault injection attacks. Additionally, beyond the practical evaluation using VerFI of this work we see further potential for performance improvements in order to evaluate larger parts of the target design within the same run.

**Comparison to Existing Models.** The most common and already existing fault models are restricted to toggle, stuck-at-1, and stuck-at-0 faults (in our model we call them bit-flip, set, and reset faults, respectively). Compared to these approaches our model can analyze fault injections in digital circuits more precisely and in more detail incorporating the physical behavior of the different fault injection mechanisms. However, an argument against our model could be that a user of a fault verification tool would still cover all worst-case fault scenarios by applying a bit-flip model resulting in a lower number of combinations of fault mappings that need to be tested. The bit-flip model comes with the disadvantage of insufficient precision regarding the description of fault injections. First, it is not possible to distinguish between different fault injection mechanisms. Second, faults in some hardware primitives cannot be accurately modeled like the NOR flash memory mentioned in Section 8.3.2. Third, besides these arguments, our proposed fault adversary model enables the user to precisely reconstruct the cause of failures in a developed countermeasure.

**Limitations of Our Proposed Model.** Despite these clear advantages, our adversary model is not reflecting the parameters of the technology node and the physical layout of integrated

circuits. Since the model considers hardware designs on netlist level, a detailed integration of technology-related parameters is not (yet) possible. Additionally, place-and-route information cannot be used in the evaluation phase resulting in simplified assumptions for e.g., critical paths.

**Practical Application.** Eventually, we discuss the practical application of the proposed adversary model. Particularly, our model is applicable for robustness evaluation, i.e., evaluating the correctness and effectiveness of countermeasures against FIA. Additionally, our approach enables analysis of circuits in a setting with precise adversarial control over the fault injection, i.e., we do not consider random faults and hazards due to environmental effects and conditions (usually considered during safety evaluation in contrast to security analysis). Hence, our presented fault model first and foremost supports the constructive development of robust FIA countermeasures. In particular, the integration into verification tools can assist in correctly, effectively, and efficiently designing protected designs (e.g., [RSS+21]).

The benefits of this approach are manifold. First, analyzing a design requires no setup or expensive equipment (see for example the fault injection setups from Riscure [Ris21] or NewAE [Inc]). Second, no prototyping is required since the circuit under test can be evaluated on a gate-level netlist. Third, the development process is faster, cheaper and less error-prone. Fourth, the designer does not need deep expertise in fault injection setups. Altogether, the application of our proposed adversary model can assist designers in early stages when implementing hardware countermeasures against fault injection attacks.

## 8.7 Conclusion

By reviewing and summarizing existing fault injection mechanisms developed over the last two decades, we created a basic understanding of the physical behavior appearing on the actual hardware when attacking cryptographic implementations. Subsequently, we introduced a generic and abstract (but simple) fault adversary model which can freely be parametrized by selecting three parameters describing the number of faults, the fault types, and the fault locations. Given that, we connected the practical fault injection mechanisms with the theoretically introduced fault adversary model and accurately described how it has to be instantiated to provide a perfect mapping between theory and practice. This connection gave us the opportunity to demonstrate the application of the adversary models – instantiated to model different attack mechanisms – to a practical case study of a protected design of the lightweight cipher LED-64. This case study was accomplished by extending the fault verification tool VerFI by our proposed adversary model. Eventually, we discussed the advantages and benefits of using our consolidated fault adversary model and limitations in existing state-of-the-art verification tools.

# Chapter 9

# Security Notions for Secure Hardware Gadgets

*A well-established countermeasure against SCA is to replace insecure gates in a target digital logic circuit with secure and composable gadgets. These gadgets usually fulfill specific composability notions ensuring to maintain their security properties even in composed circuits. Interestingly, these security guarantees were mainly investigated in the context of side-channel security while ignoring FIA for hardware implementations.*
*In this chapter, we close this gap by transferring existing fault composability notions for software designs to hardware. Afterwards, we introduce a new composability notion for fault-resistant gadgets inspired by the SCA notion PINI.*
*Moreover, we push the definition of composability even further by addressing security notions for combined attacks. Again, existing literature only provides combined security notions for software based on the PNI and PSNI notions. We slightly adapt these notions by transferring them to hardware implementations. Additionally, we present two new PINI influenced combined composability notions and provide corresponding practical gadget instantiations.*
*The contributions in this chapter are extracted from two collaborations with Jakob Feldtkeller, Pascal Sasdrich, and Tim Güneysu [RFSG22, FRSG22]. More precisely, this chapter covers theoretical background and important definitions which are mainly contributions from Jakob Feldtkeller and not part of the contributions of the author of this thesis. To this end, we omit all proofs and refer the interested reader to the original works [RFSG22, FRSG22]. Even though, we still present the definitions and theorems in this chapter since they are important for the remainder of this thesis.*

## Contents of this Chapter

## 9.1 Introduction

Masking is seen as one of the most promising countermeasures against SCA. Over the last 20 years, many countermeasures based on Boolean masking have been proposed, e.g., threshold implementation. However, protecting arbitrary functions against SCA using threshold implementation is not an easy task – especially not for non-linear functions. To this end, the research community focused on designing and developing smaller building blocks – *gadgets* – which can be used to replace gates in insecure circuits generating secure designs.

However, the security guarantees provided by the (glitch-extended) $d$-probing model are not sufficient for secure compositions. Hence, dedicated composability notions have been introduced which restrict leakage propagation to a gadget's boundaries. For this, Barthe et *al.* proposed PNI [BBD+15] and PSNI [BBD+16] while Cassiers and Standaert presented PINI [CS20].

Surprisingly, the research mainly focused on composability notions for designing protected circuits against SCA while mostly ignoring countermeasures against FIA. Nevertheless, Dhooghe and Nikova presented first composability notions for composable gadgets providing fault security in [DN20] targeting software implementations. Inspired by the ideas of PNI and PSNI, they introduced the two notions Non-Accumulation (NA) and Strong Non-Accumulation (SNA) ensuring that faults do not propagate and spread uncontrolled.

In the same work, Dhooghe and Nikova proposed the first composability notions for combined security, i.e., designs that are protected against combined attacks using SCA and FIA. Again, based on the principles of PNI and PSNI, they introduce Non-Interference Non-Accumulative (NINA), Strong Non-Interference Non-Accumulation (SNINA), and Strong Independent Non-Interference Non-Accumulation (SININA). While NINA only considers attacks against SCA or FIA but no use of both attack vectors combined, SNINA incorporates the effects of fault injections on the side-channel security. Eventually, SININA covers the composition of gadgets where the security against fault injections and side-channel attacks is provided independently, i.e., injected faults do not have an influence on the side-channel security.

### 9.1.1 Contribution

In this chapter, we first recap the fault composability notions introduced by Dhooghe and Nikova [DN20] and transfer them to hardware implementations. Please note, for the sake of consistent naming, we rename NA to Fault Non-Interference (FNI) and SNA to Fault Strong Non-Interference (FSNI). Additionally, we present a novel fault composability notion inspired by PINI called Fault-Isolating Non-Interference (FINI).

Moreover, we address composability notions for gadgets against Combined Analysis (CA). Again, we first review the software security notions from [DN20] by embedding them in a hardware context. To this end, we consider gadgets with multiple-output functions and give for the first time formal security arguments for the impact of faulty randomness. Eventually, we propose two new combined composability notions based on the design principles of PINI. Using

these new notions, we also present corresponding hardware gadgets for protection for arbitrary security orders.

## 9.2 Adversary Models

In this section, we briefly recapitulate and state our adversary models in the context of side-channel, fault-injection, and combined security. Additionally, we define probing, fault, and combined security in the context of hardware implementations.

### 9.2.1 Side-Channel Security

In this section, we introduce our side-channel adversary model and corresponding probing security.

#### Adversary Model

An adversary $\mathcal{A}_p$ in the context of stateless probing security [ISW03], is given access to a circuit $\mathbf{C}$ that can be invoked multiple times. Prior to each invocation, $\mathcal{A}_p$ can select up to $d$ wires of $\mathbf{C}$, so called *probes*. The view of the adversary $\mathcal{A}_p$ is defined by the glitch-extended probes [FGP+18], i.e., by the exact values of all registers a probed wire directly depends on[1]. Further, a probe propagates into another wire whenever this wire is required for simulation of the probe [CS20]. For more details, please see Section 2.1.4.

#### Probing Security

In this context, *probing security* [ISW03] is defined as the view of the adversary $\mathcal{A}_p$ always being independent of any secret, i.e., all probes can be simulated without any knowledge apart from the structure of $\mathbf{C}$. Please note, that probing security does not capture horizontal attacks [BCPZ16].

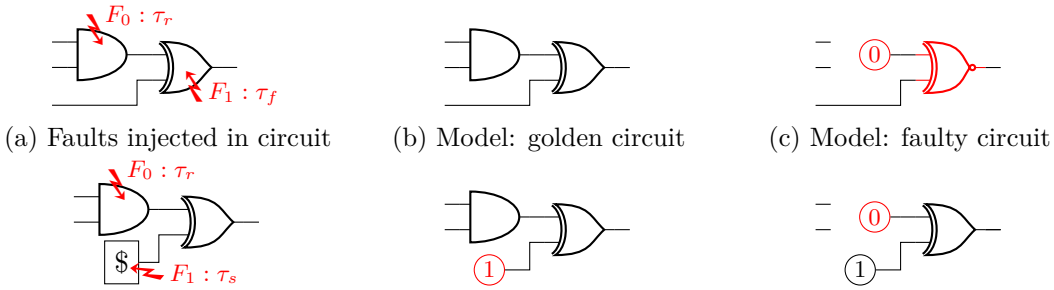
### 9.2.2 Fault-Injection Security

Now, we present our fault-injection adversary model and define fault security in detail.

#### Adversary Model

An adversary $\mathcal{A}_f$ in the context of fault security (more details below) is given access to a circuit $\mathbf{C}$ that can be invoked multiple times. Prior to each invocation, $\mathcal{A}_f$ selects up to $k$ gates in $\mathbf{C}$ and a fault type from the set of allowed fault types $\mathcal{T}$ for each gate. Faults are modeled by transforming the selected gates to a different gate type which is specified by the fault type $t \in \mathcal{T}$ (cf. Chapter 8). Typical fault types are *set*, *reset* (replacing the targeted gate with a constant one or zero, respectively), or *bit flips* (inversion of the gate). A fault propagates into another wire whenever the value of the wire is influenced by the fault. The view of the adversary $\mathcal{A}_f$ is defined by the abort signal (if existent) and the correctness of the outputs. Correctness is defined by equivalence to the *golden circuit*, which is a fault-free version of $\mathbf{C}$.

[1] Glitches are physical defaults that occur when there are timing differences in the propagation path of signals. Hence, providing $\mathcal{A}_p$ with all stable inputs of a probed wire captures all possible leakage via glitches [FGP+18].




(a) Faults injected in circuit (b) Model: golden circuit (c) Model: faulty circuit

(d) Faults injected in circuit (e) Model: golden circuit (f) Model: faulty circuit

Figure 9.1: Fault model and golden circuits for general circuits (a to c) and shared circuits in particular (d to f).

To this end, specific faults are modeled by replacing the faulted gate with a different gate defined by the introduced fault. Afterwards, the fault propagation can be identified and effective and ineffective faults be distinguished by comparing the faulty circuit model to a fault-free circuit model, the so called *golden circuit*. We illustrate the used fault model in Figure 9.1(a-c).

### Ecoding Circuits for Fault Protection

In Section 2.2.4, we introduce countermeasures against FIA. For the definition of composable gadgets, we rely on redundancy in information and define more formally an encoding that transforms values from the unprotected domain to the protected domain.

**Definition 30** (Value Encoding for linear ECCs)**.** *Let $\mathcal{C}$ be a linear code as defined in Definition 9 for $q = 2$. Further, let $\mathsf{E} : \mathbb{F}_2^m \mapsto \mathbb{F}_2^n$ be a deterministic* encode function *following the encoding mechanism of the linear code, and $\mathsf{D} : \mathbb{F}_2^n \mapsto \mathbb{F}_2^m$ be a deterministic* decode function *that receives a codeword $c$ and computes the corresponding message $m$ or aborts.*

We define an *encoded circuit* as a digital circuit that operates on values secured by information redundancy of linear codes. In accordance with Definition 32 and similar to Definition 5 we do not consider the encoding and decoding as part of the circuit, as faulting them inherently violates the definition of fault security.

**Definition 31** (Encoded Circuit)**.** *An* encoded circuit $\mathbf{C}_{\mathsf{F}}^{\mathcal{C}}$ *for a function $\mathsf{F} : \mathbb{F}_2^{m \cdot \ell} \mapsto \mathbb{F}_2^{m \cdot \ell'}$ and a linear code $\mathcal{C}$ is a deterministic circuit realizing a function $\mathsf{F}^{\mathbf{C}} : \mathbb{F}_2^{n \cdot \ell} \mapsto \mathbb{F}_2^{n \cdot \ell'}$, such that $\forall x \in \mathbb{F}_2^{m \cdot \ell}$ it holds that $\mathsf{F}^{\mathbf{C}}(\mathsf{E}(x)) \in \mathcal{C}$ and $\mathsf{F}(x) = \mathsf{D}(\mathsf{F}^{\mathbf{C}}(\mathsf{E}(x)))$ (*functional correctness*).*

### Fault Security

Recently, Dhooghe and Nikova proposed a definition for *active security* [DN20] where an adversary is allowed to inject up to $k$ faults into a circuit $\mathbf{C}$ at arbitrary – but prior to each invocation selected – locations. Then *active security*, in terms of correctness, is given iff for all fault combinations $\mathbf{C}$ either aborts or outputs a correct result, i.e., equivalence with the golden circuit. Here, the view of the adversary contains the (optional) abort signal and the correctness of the result. Please note, that to provide security against some attacks (e.g., SIFA [DEK+18])

no abort signal is allowed. We extend this definition with the adversary model from Chapter 8 by explicitly requiring a protected decoding algorithm for error detection or correction[2].

When analyzing gadgets for secure composability in the presence of fault injections (e.g., gadgets from [DN20]), we further extend the locations $\mathcal{L}$ of valid fault injections to additionally consider faulty inputs. This includes faults in data inputs (due to faulty outputs of other gadgets) as well as randomness gates.

**Definition 32** ($(k,t,l)$-Fault Security)**.** *Let* $\mathsf{G}^{\mathsf{D}}$ *be a gadget realizing a decoding function* $\mathsf{D}$ *such that – given an input with at most* $k$ *faults and an abort signal –* $\mathsf{G}^{\mathsf{D}}$ *either aborts or outputs a correct result. A circuit* $\mathbf{C}$ *together with a decoding* $\mathsf{G}^{\mathsf{D}}$ *is* $(k,t,l)$*-fault secure iff for any set of up to* $k$ *faults of type* $t$ *injected in gates of type* $l$ *in* $\mathbf{C}$*, the concatenation* $\mathsf{G}^{\mathsf{D}}(\mathbf{C}(\cdot))$ *either aborts or outputs a result equal to the golden circuit of* $\mathbf{C}$*.*

The final decoding gadget is necessary for both practicality and security. It is practically impossible to guarantee functional correctness at the output with an unrestricted adversary, as they can always fault the output directly. For security, the output behavior must be independent of any sensitive information. In the following, we write $k$-fault security when we do not restrict the fault type and fault location.

### 9.2.3 Combined Security

Eventually, we introduce our adversary model in the context of combined attacks and define combined security. Moreover, we provide further details about a definition of a golden circuit.

#### Adversary Model

An adversary $\mathcal{A}_c$ in the context of combined security is the combination of the adversaries $\mathcal{A}_p$ and $\mathcal{A}_f$. Hence, $\mathcal{A}_c$ is given access to a circuit $\mathbf{C}$ that can be invoked multiple times and prior to each invocation $\mathcal{A}_c$ can select up to $d$ wires of $\mathbf{C}$ that are probed and up to $k$ gates of $\mathbf{C}$ that are faulted according to a fault type $t \in \mathcal{T}$. The view of the adversary $\mathcal{A}_c$ is defined by the glitch-extended probes, the abort signal (if existent), and the correctness of the outputs of the concatenation $\mathsf{G}^{\mathsf{D}}(\mathbf{C}(\cdot))$, where $\mathsf{G}^{\mathsf{D}}$ is a circuit realizing a fault detection or correction mechanism for up to $k$ faults. Correctness is again defined by equivalence to the golden circuit of $\mathbf{C}$. Here, however, the golden circuit of $\mathbf{C}$ incorporates the faults targeting randomness gates $g \in \mathcal{G}_{\mathsf{rand}}$, i.e., $g \in \mathcal{G}_{\mathsf{rand}}$ are replaced by some $g'$ according to the fault model (see below for more details). Please note, that faulting some gates gives $\mathcal{A}_c$ knowledge about the changed distribution of the dependent values, however, not necessarily the concrete values. This knowledge of changed distributions can cause additional probes that need to be simulated when the independence property of an intermediate Boolean sharing is violated. In general, value distributions are sufficient for simulation and the changed distribution is given to $\mathcal{A}_c$ for faulty gadget inputs.

#### Combined Security

Security in the combined adversary and security model has to amalgamate the definitions of probing security and fault security, while additionally considering any reciprocal effects. For

[2] [DN20] implicitly requires a protected decode gadget as well, however, we explicitly include it into the definition to make this requirement more transparent

this, we adopt and refine the formal notions in [DN20], considering the extended fault model from Chapter 8 and explicit decoding functions. Again, all probes and faults are selected prior to each invocation of the circuit and the view of the adversary is defined by the probed values, the (optional) abort signal, and the correctness of the result of the concatenation $\mathsf{G}^{\mathsf{D}}(\mathbf{C}(\cdot))$, where $\mathsf{G}^{\mathsf{D}}$ is a circuit realizing an error detection or error correction mechanism for up to $k$ faults.

**Definition 33** ($(d, k, t, l)$-Combined Security)**.** *Let $\mathsf{G}^{\mathsf{D}}$ be a gadget realizing a decoding function $\mathsf{D}$, such that, given an input with at most $k$ faults and an abort signal, $\mathsf{G}^{\mathsf{D}}$ either aborts or outputs a corrected result. A circuit $\mathbf{C}$ together with a decoding $\mathsf{G}^{\mathsf{D}}$ is* $(d, k, t, l)$-combined secure *iff for any set of up to $k$ faults of type $t$ injected on gates of type $l$ in $\mathbf{C}$, and any set of up to $d$ probes placed on wires in $\mathbf{C}$ the following holds:*

*Privacy:* *The abort signal and the probes can be simulated without access to any wire of $\mathbf{C}$.*

*Correctness:* *The concatenation $\mathsf{G}^{\mathsf{D}}(\mathbf{C}(\cdot))$ either aborts or outputs a result equal to the golden circuit of $\mathbf{C}$.*

In the following, we write $(d, k)$-combined security when we do not restrict the fault type and location. Further, we emphasize the adversarial knowledge on faults, since injecting faults is strongly linked to placing probes [Cla07, SMC21].

**Remark 1** (Adversarial Knowledge on Faults)**.** *A fault is known iff the adversary exactly learns the targeted gate, and the fault type $t$, i.e., the misbehavior caused by the fault.*

With this, an adversary additionally learns the following details: (i) the location of propagated faults, (ii) the distribution of the faults under a chosen input distribution, and (iii) the probability of fault occurrences under a chosen input distribution. Still, the adversary does not necessarily learn the exact values (but only distributions). This is also the case when the exact position of the fault injection is not modeled, e.g., in the case of faulty inputs (see below).

#### Golden Circuit of Probabilistic Circuits

Accurately modeling CA requires a precise notion of fault propagation and misbehavior in a *shared circuit*. Unfortunately, this is a non-trivial task as, on the one hand, there are multiple valid Boolean sharings for one value, such that faults can be effective in the traditional meaning (i.e., have a different value than the fault-free circuit) but still be functionally correct, which is especially true for faults injected in generated randomness. On the other hand, if we have a faulty sharing, then it is hard to precisely determine the faulty shares, requiring a precise propagation of faults. As a consequence, we introduce an adjusted definition for the golden (fault-free) circuit of a shared circuit, where, if faulted, the randomness-generation gates are already replaced, as shown in Figure 9.1(d-f). Intuitively, this is correct as the actual value of randomness cannot have an impact on the functional correctness.

**Definition 34** (Golden Circuit)**.** *The* golden circuit $\mathbf{C}_{golden}$ *of a shared circuit $\mathbf{C}$ under a set of faults $\mathcal{F}$ is the circuit $\mathbf{C}$ transformed by the faults on generated randomness, i.e., $\{f \in \mathcal{F} \mid f$ targets $g \in \mathcal{G}_{\mathsf{rand}}\}$.*

In the following, we show that Definition 34 is meaningful and sound, since faults in randomness gates do not affect functional correctness and are at most equivalent to a probe for probing security of a circuit $\mathbf{C}$.

**Unaltered Randomness Distribution.** We first start by showing that all faults in randomness gates that do not change the randomness distributions, e.g., bit-flips on uniformly drawn randomness, can be ignored without further impacting functional correctness, probing security, or simulation-based composability notions.

**Theorem 3.** *Faulting a gate $g \in \mathcal{G}_{\mathsf{rand}}$, generating some randomness $r_g$, has no effect on functional correctness, probing security, or simulation-based composability of $\mathbf{C}$ if the randomness distribution of $r_g$ is not changed.*

**Altered Randomness Distribution.** Next, we show that faults injected in a randomness gate which alter the randomness distribution, e.g., set or reset faults, can be replaced by a probe without impacting functional correctness, probing security, or simulation-based composability notions. This further implies that propagation of faulty randomness does not affect probing security or composability (besides providing additional probes). For this, we first provide a lemma claiming that a value with a biased distribution cannot mask some other value before we show the actual claim in Theorem 4.

**Lemma 3.** *Let $x, y \in \mathbb{F}_2$ where $x$ has a biased distribution and the distributions of $x$ and $y$ are independent of each other. Further, let $\circ$ be a binary operator over $\mathbb{F}_2$. Then the distribution of $x \circ y$ and $y \circ x$ is dependent on the distribution of $y$.*

**Theorem 4.** *Faulting a gate $g \in \mathcal{G}_{\mathsf{rand}}$, generating some randomness $r_g$, is equivalent to placing a probe on $r_g$ with respect to functional correctness, probing security, or simulation-based composability of $\mathbf{C}$ if the randomness distribution of $r_g$ is changed.*

Together, Theorems 3 and 4 show that faulting some randomness has no impact on correctness and affects security at most in equivalence to a probe. As probes (backward) and fault (forwards) propagation are converse, fault and probing security is not impacted when defining the golden circuit in replacing/modifying randomness generating gates. Hence, our definition of the golden circuit is sound.

## 9.3 Fault Non-Interference

Dhooghe and Nikova present security notions for composable injection-secure gadgets [DN20]. Inspired by the PNI property, the FNI property ensures that each intermediate fault only propagates to at most a single output. However, this is relaxed by also allowing abortion of computations upon fault detection. Please note, this definition is equal to the definition of Non-Accumulation [DN20], however, with an explicit requirement for the existence of an appropriate decoding gadget.

**Definition 35** (Fault Non-Interference [DN20])**.** *A gadget $\mathsf{G}$ is $k$-FNI iff for any set of $k' \leq k$ faults at inputs and injected on gates in $\mathsf{G}$, the gadget either aborts or gives an output with at most $k'$ bit faults and there exists a decoding gadget $\mathsf{G}^{\mathsf{D}}$, such that given an input with at most $k$ faulty inputs and an abort signal, $\mathsf{G}^{\mathsf{D}}$ either aborts or outputs a correct result.*

## 9.4 Fault Strong Non-Interference

Analogous to PSNI tightening PNI, the notion of FSNI extends FNI and ensures full composability in distinguishing input and intermediate faults. Again, this definition is equal to the definition of Strong Non-Accumulation [DN20], however, with an explicit requirement for the existence of an appropriate decoding gadget.

**Definition 36** (Fault Strong Non-Interference [DN20])**.** *A gadget* $\mathsf{G}$ *is* $k$*-FSNI iff for any set of* $k_1$ *faulty inputs and every set of* $k_2$ *faults injected in gates of* $\mathsf{G}$*, with* $k_1 + k_2 \leq k$*, the gadget either aborts or gives an output with at most* $k_2$ *bit faults and there exists a decoding gadget* $\mathsf{G}^{\mathsf{D}}$*, such that given an input with at most* $k$ *faulty inputs and an abort signal,* $\mathsf{G}^{\mathsf{D}}$ *either aborts or outputs a correct result.*

Both $k$-FNI and $k$-FSNI imply $k$-fault security given a decoding gadget $\mathsf{G}^{\mathsf{D}}$ that can detect or correct $k$ faults. In addition, it holds that the composition of two FSNI gadgets is FSNI again [DN20].

## 9.5 Fault-Isolating Non-Interference

When considering circuits, hardened against fault attacks by replication, we can observe that each injected fault can only propagate within the affected redundant part. Then, having more replications than allowed adversarial faults (maximum fault cardinality) can ensure the desired level of fault security. In addition, we observe that this property is inherently composable as long as the isolation of different redundant parts remains intact. Breaking this down to the core principles, namely the isolation of domains with regard to faults, replication reveals a strong similarity with PINI [CS20]. While PINI isolates probe propagation within share domains, replication isolates fault propagation within *redundancy domains*.

**Definition 37** (Redundancy Domain)**.** *The* redundancy domain $\ell$ *of a redundant circuit is defined by all gates and wires with replication index* $\ell$*.*

In order to generalize the core principle of replication and allow a formal treatment, we now introduce the notion of Fault-Isolating Non-Interference. FINI is the dual to PINI in that it introduces redundancy domains and requires them to be isolated in terms of fault propagation (as illustrated in Figure 9.2). Then, assuming sufficient redundancy domains, detection or correction is always possible by comparing the values in different redundancy domains, regardless of the fault propagation within a single redundancy domain.

In this work, we focus on a realization of FINI via replication since it is an obvious match and transports the ideas and principles more easily. However, we intentionally construct FINI as a general notion that can be applied to other redundancy-based countermeasures with an appropriate definition of redundancy domains. When considering replication, the redundancy domain is defined by the replication index, i.e., all values with replication index $\ell$ belong to the redundancy domain $\ell$.

As FINI, similar to PINI, introduces an isolation between different redundancy domains instead of an isolation between gadgets, faults at inputs are allowed to propagate to outputs of the same redundancy domain. Further, faults injected inside the gadget are restricted to only

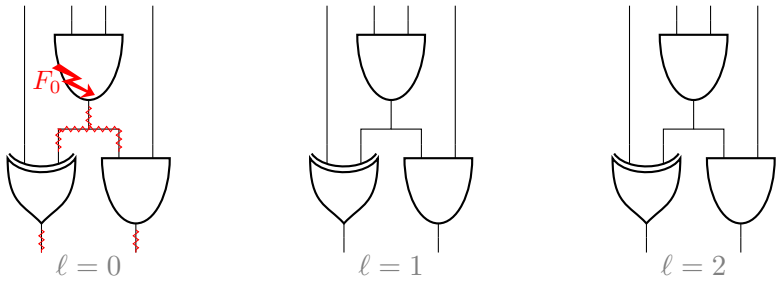



Figure 9.2: Isolation of fault propagation within redundancy domains.

propagate to outputs belonging to a single redundancy domain. We now give a more formal definition in Definition 38.

**Definition 38** (Fault-Isolating Non-Interference)**.** *A gadget* $\mathsf{G}$ *is* $k$*-FINI iff the following holds:*

*(i) For any set* $\mathcal{F}_1$ *of* $k_1$ *faulty redundancy domains and every set of* $k_2$ *faults injected in gates of* $\mathsf{G}$*, with* $k_1 + k_2 \leq k$*, there exists a set of at most* $k_2$ *redundancy domains* $\mathcal{F}_2$*, such that the gadget either aborts or gives an output where all values, except those belonging to the redundancy domains* $\mathcal{F}_1 \cup \mathcal{F}_2$*, are equal to the values of the golden circuit.*

*(ii) There exists a decoding gadget* $\mathsf{G}^{\mathsf{D}}$*, such that given an input with at most* $k$ *faulty redundancy domains and an abort signal,* $\mathsf{G}^{\mathsf{D}}$ *either aborts or outputs a correct result.*

### 9.5.1 FINI Security and Composition

FINI formalizes the intuitive security and compositional properties of replication codes. Here, the main argument for fault security comes from the fact that at most $k$ redundancy domains can be manipulated while up to $k$ manipulated redundancy domains can be detected or corrected by the corresponding decoding function.

**Theorem 5.** *A* $k$*-FINI gadget is* $k$*-fault secure.*

Now we argue that an arbitrary composition of FINI gadgets results in a (larger) FINI gadget. Again, this follows from the properties of replication codes, which introduce a natural isolation of different replications, i.e., a fault injected to one redundancy domain cannot propagate to another redundancy domain. This ensures that the upper bound of faulty redundancy domains remains unchanged after composition.

**Theorem 6.** *The composition of two* $k$*-FINI gadgets is* $k$*-FINI.*

**Remark 2.** *The connection of gadgets has to be consistent, i.e., redundancy domain* $\ell$ *of a gadget is connected only to redundancy domain* $\ell$ *of subsequent gadgets. However, we can permute the index of domains when necessary.*

### 9.5.2 FINI Gadgets

The construction of FINI gadgets for combinational gates follows a simple method: replication of the gate. By applying the compositional property of Theorem 6, the construction method

**Algorithm 6** FINI-secure correction gadget.

```
1: procedure Correct_FINI(a^0, ..., a^{n-1})
2:     for ℓ = 0 to n − 1 do
3:         b^ℓ ← maj(a^0, ..., a^{n-1})
4:     end for
5:     return b^0, ..., b^{n-1}
6: end procedure
```

can be generalized to all combinational or sequential circuits (which also includes the simple case of a single gate).

**Theorem 7.** *The $(k+1)$-times replication of any circuit is $k$-FINI.*

In Theorem 7 we instantiate the number of replications with $k+1$, which is the minimum number required for fault detection via comparison. It is trivial to see that the same claim is true for implementations with more than $k+1$ replications, as the comparison still detects faulty values. Similarly, when using at least $2k+1$ replications, we can use a majority function as decoding gadget additionally resulting in error correction.

The only gadgets that cannot be trivially constructed according to Theorem 7 are gadgets that combine different redundancy domains. The most prominent examples of this category are detection and correction modules. Here, the logic for detection/correction needs to be replicated for each redundancy domain, such that faults injected into this logic only affect one redundancy domain at the output [AMR+20, SRM20]. We show an example for this in Algorithm 6 considering a correction module[3].

## 9.6 Combined Non-Interference

In the context of CA, it is again useful to define notions of secure composition to reduce the analysis complexity of gadgets. In this, we build upon the work of [DN20] by transferring their definitions to the glitch-extended probing model along with necessary refinements. To this end, we use Remark 1 to explicitly specify the information access of a simulator to mimic the view of the adversary. Similarly, we use Definition 34, to overcome the fact that faulting randomness can lead to more faults at the gadget output than allowed.

**Definition 39** ($(d,k)$-Combined Non-Interference)**.** *A gadget $\mathsf{G}$ is $(d,k)$-Combined Non-Interference (CNI) if for any set of $k_1$ faulty inputs, $k_2$ faults injected on gates in $\mathsf{G}$, and $d'$ probes, such that $d' + k_1 + k_2 \leq d$ and $k_1 + k_2 \leq k$ the following holds:*

- *Privacy:* *The probes and the abort signal can be simulated with $d' + k_2$ shares of each input and knowledge of the faults both injected and on inputs.*
- *Correctness:* *The gadget either aborts or outputs a result with at most $k_1 + k_2$ bit faults compared to the golden circuit and there exists a decoding gadget $\mathsf{G}^{\mathsf{D}}$, such that given an input with at most $k$ faulty inputs and an abort signal, $\mathsf{G}^{\mathsf{D}}$ either aborts or outputs a correct result.*

[3] The actual implementation of the function maj is irrelevant for the FINI property.

Specifically, CNI is a combination of PNI with FNI, however, in contrast to the NINA definition in [DN20], we do not add $k_1$ to the number of allowed input shares for simulation, to clarify that faulty inputs must not reveal additional shares. Hence, we restrict the possible leakage through faulty inputs to the knowledge an adversary can have about those inputs according to Remark 1, i.e., the changed distribution under the faults. This ensures that a fault can only leak input shares locally and not by probe propagation from other gadgets. In addition, we refined the definition of NINA by giving a formal description of what a *correct* output is, specifically comparing the output of the gadget with the output of the golden circuit as defined in Definition 34. Those changes also apply to the other composability notions, given below, in comparison with their counterpart from [DN20]. Please note, that knowledge about the faults according to Remark 1 is enough for simulation as a simulator requires only distributions and not concrete values.

## 9.7 Combined Strong Non-Interference

Similar to the underlying definitions, CNI is not sufficient for arbitrary compositions in the context of CA. Hence, Combined Strong Non-Interference (CSNI) is defined as the combination of PSNI and FSNI (similar to SNINA in [DN20]).

**Definition 40** ($(d,k)$-Combined Strong Non-Interference)**.** *A gadget* $\mathsf{G}$ *is* $(d,k)$*-CSNI iff for any set of* $k_1$ *faulty inputs,* $k_2$ *faults injected on gates in* $\mathsf{G}$*,* $d_1$ *probes placed on intermediate values, and up to* $d_2$ *probes placed on shares of each output, such that* $d_1 + d_2 + k_1 + k_2 \leq d$ *and* $k_1 + k_2 \leq k$*, the following holds:*

*Privacy:* *The probes and the abort signal can be simulated with* $d_1 + k_2$ *shares of each input and knowledge of the faults both injected and on inputs.*

*Correctness:* *The gadget either aborts or outputs a result with at most* $k_2$ *bit faults compared to the golden circuit and there exists a decoding gadget* $\mathsf{G}^{\mathsf{D}}$*, such that given an input with at most* $k$ *faulty inputs and an abort signal,* $\mathsf{G}^{\mathsf{D}}$ *either aborts or outputs a correct result.*

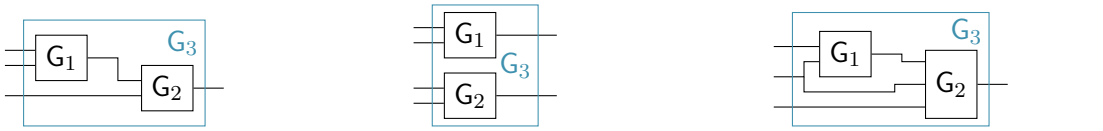
We now show that CSNI gadgets, according to our refined definitions, still can be composed to larger circuits without degrading combined security. We give illustrative examples of valid and invalid compositions in Figure 9.3.

**Theorem 8.** *The composition of two* $(d,k)$*-CSNI gadgets, that is loop-free and where no output of one gadget is used multiple times as input to the other gadget, is* $(d,k)$*-CSNI.*

## 9.8 Indpendet Combined Strong Non-Interference

While CSNI supports arbitrary composition, as shown by Theorem 8, the maximum number of allowed probes and faults is not independent. Indpendet Combined Strong Non-Interference (ICSNI), however, is designed to allow independent security levels for probing and injection attacks, without losing the notion of composition (similar to SININA in [DN20]).

**Definition 41** ($(d,k)$-Indpendet Combined Strong Non-Interference)**.** *A gadget* $\mathsf{G}$ *is* $(d,k)$*-ICSNI if for any set of* $k_1$ *faulty inputs,* $k_2$ *faults injected on gates in* $\mathsf{G}$*,* $d_1$ *probes placed on*




(a) Sequential composition (b) Parallel composition (c) Composition with common input


(d) Composition with loop (e) Output of $\mathsf{G}_1$ is used multiple times

Figure 9.3: Valid (a to c) and invalid (d and e) examples of gadget compositions according to Theorem 8 and Theorem 9. Example (d) is invalid as it contains a loop and (e) is invalid as the output of $\mathsf{G}_1$ is used two times as input to $\mathsf{G}_2$.

*intermediate values, and up to $d_2$ probes placed on shares of each output, such that $d_1 + d_2 \leq d$ and $k_1 + k_2 \leq k$, the following holds:*

*Privacy:* *The probes can be simulated with $d_1$ shares of each input and knowledge of the faults both injected and on inputs.*

*Correctness:* *The gadget outputs a result with at most $k_2$ bit faults compared to the golden circuit.*

**Theorem 9.** *The composition of two $(d, k)$-ICSNI gadgets, that is loop-free and where no output of one gadget is used multiple times as input to the other gadget, is $(d, k)$-ICSNI.*

## 9.9 Combined-Isolating Non-Interference

Combining both PINI and FINI is an obvious step to construct a composability notation for CA that is based on the isolation of domains. However, due to reciprocal effects and the dual nature of faults, which can both manipulate internal values and serve as a probe [Cla07, SMC21], it is insufficient to isolate faults in redundancy domains (FINI) and probes in share domains (PINI) and a more complex notion is required. Those reciprocal effects can be easily seen when considering any HPC gadget where some randomness is faulted to a known value which nullifies the provided security guarantees. Similarly, with known faults, identifying correct *guesses* of shares becomes possible. Consider the example illustrated in Figure 9.4, where a gadget $\mathsf{G}$ with three shares and three redundancy domains is connected to a detection module. Further, assume that all shares in the first redundancy domain are faulty with a known and biased distribution (e.g., set/reset), representing an implicit guess on those shares. Then the error flag leaks information about all faulted shares, since it indicates the correctness of the guess, regardless of how the detection module is realized.

As a result, the domain definition for fault propagation in the CA setting has to be more restrictive than given pure fault attacks. In particular, faults are restricted to propagate only in the combination of both share and redundancy domain. This ensures that a fault can leak at most one share domain even in circumstances where it can be used to learn values (similar to placing a probe).

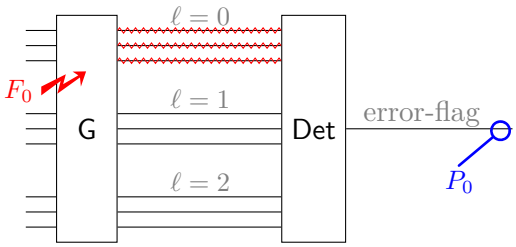



Figure 9.4: Insecure fault propagation in combined attack model. If all shares within one redundancy domain are faulty with a known biased distribution then the error flag indicates whether the faults represent a valid guess of the shares.

**Definition 42** (Shared Redundancy Domain)**.** *The* Shared Redundancy Domain (SRD) $(i, \ell)$ *of a replicated and shared circuit is defined by all gates and wires with share index $i$ and replication index $\ell$.*

Here, in contrast to faults, it is not necessary to further restrict the propagation of probes. On the contrary, as all replicated wires carry the same value, it is sufficient to probe one of those wires to learn all those values. Hence, each share domain consists of multiple SRDs.

We now define Combined-Isolating Non-Interference (CINI) as a composability property for combined security, where faults are isolated within SRDs and probes within share domains. That is, for every set of probes and faults the number of input share domains $\mathcal{A}_c$ can learn is bounded by the sum of the cardinality and position of the probes and faults, and the number of faulty output SRDs is bounded by the cardinality and position of the faults, as visualized in Figure 9.5.

**Definition 43** (Combined-Isolating Non-Interference)**.** *A gadget* $\mathsf{G}$ *is $(d, k)$-CINI iff for any set $\mathcal{F}_1$ of $k_1$ faulty SRDs, every set of $k_2$ faults injected in gates of* $\mathsf{G}$*, any set of $d_1$ probes placed on intermediate values, and any set $\mathcal{S}_2$ of $d_2$ share domains, such that $k_1 + k_2 \leq k$ and $d_1 + d_2 + k_1 + k_2 \leq d$, there exists a set $\mathcal{F}_2$ of at most $k_2$ SRDs and a set $\mathcal{S}_1$ of at most $d_1 + k_2$ share domains such that the following holds:*

*Correctness:* *The gadget either aborts or gives an output where all values, except those belonging to the SRDs $\mathcal{F}_1 \cup \mathcal{F}_2$, are equal to the golden circuit, and there exists a decoding gadget* $\mathsf{G}^{\mathsf{D}}$*, such that given an input with at most $k$ faulty SRDs and an abort signal,* $\mathsf{G}^{\mathsf{D}}$ *either aborts or outputs a correct result.*

*Privacy:* *The abort signal, the outputs of the share domains in $\mathcal{S}_2$, the outputs violating the independence property of Boolean sharing, and the probes can be simulated with the inputs of the share domains in $\mathcal{S}_1 \cup \mathcal{S}_2$ and knowledge of the faults both injected and on inputs in $\mathcal{F}_1$.*

Please note, that CINI restricts the number of probes and faults together to be smaller than or equal to the order of probing security $d$. Hence, the order of fault security is always dependent on the order of probing security. In Section 9.10 we show how to achieve independence between fault and probing security.

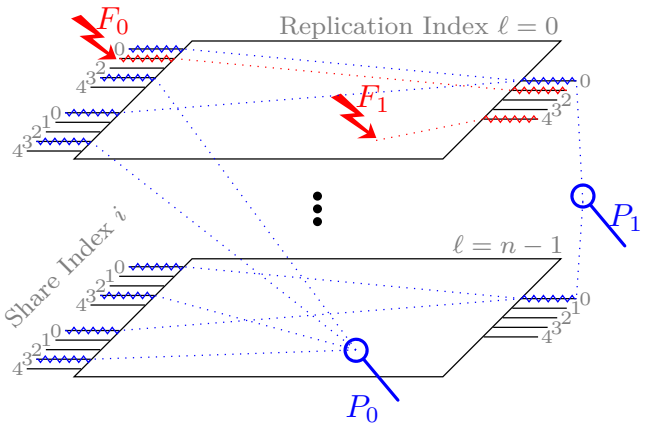



Figure 9.5: Propagation of probes and faults in the CINI context. While probes leak entire share domains (all inputs with same share index), faults are restricted to influence at most one output SRD. Further, probes at outputs restrict the leaked share domain of the inputs to be the same. Similarly, input faults are restricted to affect the same SRD at the output.

### 9.9.1 CINI Security and Composition

Intuitively, the security of CINI comes from the fact, that there are always more share domains than the number of probes and faults an adversary is allowed to place or inject. This is sufficient for security, as the isolation of probe and fault propagation within share domains and SRDs, respectively, restricts the possible leakage. Theorem 10 states that $(d, k)$-CINI gadgers are always $(d, k)$-combine secure.

**Theorem 10.** *A $(d, k)$-CINI gadget is $(d, k)$-combined secure.*

Moreover, Theorem 11 states that circuits constructed from CINI gadgets are again CINI secure.

**Theorem 11.** *The loop-free composition of two $(d, k)$-CINI gadgets is $(d, k)$-CINI.*

### 9.9.2 CINI Gadgets

One explicit goal of CINI is the trivial implementation of linear functions, e.g., addition, by simply sharing and replicating them. This is possible, as replication and Boolean sharing are linear themselves.

**Theorem 12.** *An implementation with $d + 1$ shares and $(k + 1)$-times replication of a linear function is $(d, k)$-CINI in the glitch-robust probing model.*

Implementing non-linear functions, e.g., multiplication, is more complex since values need to be combined across share-domain boundaries. Hence, similar to PINI, we have to ensure that the masks of such terms are refreshed beforehand. In addition, a faulty value is not allowed to cross SRD boundaries, which requires fault detection or correction for terms combining different

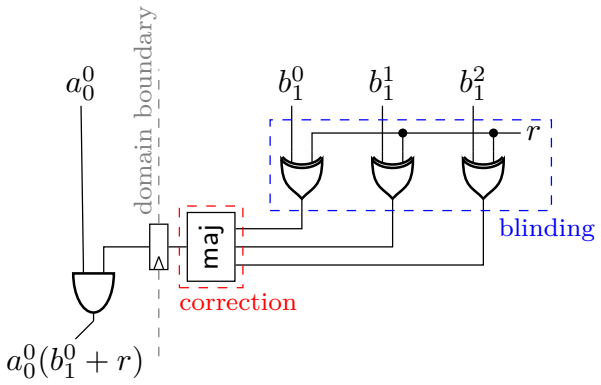



Figure 9.6: CINI requires all values crossing a domain boundary to be both correct and blinded.

share domains. Both principles are illustrated in Figure 9.6. As no fault is allowed to affect more than one SRD, each intermediate value is only allowed to be used in one SRD (except for generated randomness), requiring detection/correction within gadgets for values that are used across SRD boundaries.

Based on those design principles, we can extend the refresh-then-multiply technique, used in the $\mathsf{HPC}_1$ gadget [CGLS21] (originally PINI), to a refresh-and-correct-then-multiply technique to construct a CINI gadget (given in Algorithm 7). The major difference to the original $\mathsf{HPC}_1$ gadget is the replication of the logic and the majority function for correction in Line 18. Also, and in contrast to $\mathsf{HPC}_1$, all intermediate values are only allowed to be used once (see Remark 4). In the following, we state that the derived gadgets are CINI secure. Please note, the proofs can be found in the original paper [FRSG22].

**Theorem 13.** *The gadget* $\mathsf{HPC}_1^{\mathsf{C}}$ *as defined in Algorithm 7 with a register-free majority function is* $(d, k)$*-CINI in the glitch-robust probing model.*

**Remark 3.** *We emphasize that faults targeted at randomness count as* internal *faults, i.e., contribute to* $k_2$*.*

**Remark 4.** *For security, it is essential that the majority function (Line 18) is computed for each SRD individually.*

**Remark 5.** *For security it is also essential that the mask refreshing in Line 12 is done one by one, i.e.,* $\tilde{v}_j^\ell \leftarrow (((b_j^\ell + r_{j,0}) + \ldots) + r_{j,d})$*.*

**Remark 6.** *The assumption that the majority function is implemented register free is done for simplicity and readability of the proof. In fact, when there are registers in* maj *then probes on* $v_{i,j}^\ell$ *and potential probes within* maj *are strictly less powerful as a probe on* $v_{i,j}^\ell$ *without registers in* maj*.*

As mentioned, the security of $\mathsf{HPC}_2$ [CGLS21] relies on the fact that cross-domain leakage is prevented by some $r_{i,j}$ that is observable in only one of $(a_i + 1) \cdot r_{i,j}$ or $a_i \cdot (b_j + r_{i,j})$ and, hence, ensures a proper masking of $b_j$ (if $b_j$ is not an input to the simulator). However, this only holds because both terms get the same $a_i$ as input and one of them is using the negated form. Now, when replicating the gadget there exist some terms $(a_i^0 + 1) \cdot r_{i,j}$ and $a_i^1 \cdot (b_j^1 + r_{i,j})$ such that $a_i^\ell$

**Algorithm 7** $\mathsf{HPC}_1^{\mathsf{C}}$: CINI multiplication
(with difference to $\mathsf{HPC}_1$ highlighted).

**Require:** $n = 2k + 1$
**Require:** $a_i^\ell = a_i^{\ell'}$ and $b_i^\ell = b_i^{\ell'}$ for $0 \le \ell, \ell' \le n, 0 \le i \le d$
**Require:** $\sum_{j=0}^{d} a_j^\ell = a$ and $\sum_{j=0}^{d} b_j^\ell = b$ for $0 \le \ell \le n$
**Ensure:** $c_i^\ell = c_i^{\ell'}$ for $0 \le \ell, \ell' \le n, 0 \le i \le d$
**Ensure:** $\sum_{i=0}^{d} c_i^\ell = a \cdot b$ for $0 \le \ell \le n$

1: **procedure** $\mathsf{HPC}_1^{\mathsf{C}}(a_0^0, \ldots, a_d^n, b_0^0, \ldots, b_d^n)$
2: **for** $i = 0$ **to** $d$ **do** ▷ Initialize randomness
3: **for** $j = i + 1$ **to** $d$ **do**
4: $\tilde{r}_{i,j} \xleftarrow{\$} \mathbb{F}_2$
5: $\tilde{r}_{j,i} \leftarrow \tilde{r}_{i,j}$
6: $r_{i,j} \xleftarrow{\$} \mathbb{F}_2$
7: $r_{j,i} \leftarrow r_{i,j}$
8: **end for**
9: **end for**

10: **for** $\ell = 0$ **to** $n - 1$ **do** ▷ Refreshing
11: **for** $j = 0$ **to** $d$ **do**
12: $\tilde{v}_j^\ell \leftarrow b_j^\ell + \sum_{i=0, i \neq j}^{d} \tilde{r}_{i,j}$
13: **end for**
14: **end for**

15: **for** $\ell = 0$ **to** $n - 1$ **do** ▷ Correction
16: **for** $i = 0$ **to** $d$ **do**
17: **for** $j = 0$ **to** $d$ **do**
18: $v_{i,j}^\ell \leftarrow \mathsf{maj}(\tilde{v}_i^0 \ldots \tilde{v}_i^{n-1})$
19: **end for**
20: **end for**
21: **end for**

22: **for** $\ell = 0$ **to** $n - 1$ **do** ▷ Multiplication
23: **for** $i = 0$ **to** $d$ **do**
24: $w_i^\ell \leftarrow a_i^\ell \cdot \mathsf{Reg}[v_{i,i}^\ell]$
25: **for** $j = 0$ **to** $d, j \neq i$ **do**
26: $z_{i,j}^\ell \leftarrow a_i^\ell \cdot \mathsf{Reg}[v_{j,i}^\ell] + r_{i,j}$
27: **end for**
28: $c_i^\ell \leftarrow \mathsf{Reg}[w_i^\ell] + \sum_{j=0; j \neq i}^{d} \mathsf{Reg}[z_{i,j}^\ell]$
29: **end for**
30: **end for**
31: **return** $c_0^0, \ldots, c_d^n$
32: **end procedure**

**Algorithm 8** $\mathsf{HPC}_2^{\mathsf{C}}$: CINI multiplication for $d \leq 2$, $k \leq 2$
(with difference to $\mathsf{HPC}_2$ highlighted).

**Require:** $d \leq 2, k \leq 2, n = 2k+1$
**Require:** $a_i^\ell = a_i^{\ell'}$ and $b_i^\ell = b_i^{\ell'}$ for $0 \leq \ell, \ell' \leq n, 0 \leq i \leq d$
**Require:** $\sum_{j=0}^{d} a_j^\ell = a$ and $\sum_{j=0}^{d} b_j^\ell = b$ for $0 \leq \ell \leq n$
**Ensure:** $c_i^\ell = c_i^{\ell'}$ for $0 \leq \ell, \ell' \leq n, 0 \leq i \leq d$
**Ensure:** $\sum_{i=0}^{d} c_i^\ell = a \cdot b$ for $0 \leq \ell \leq n$

**procedure** $\mathsf{HPC}_2^{\mathsf{C}}(a_0^0, \ldots, a_d^n, b_0^0, \ldots, b_d^n)$
  **for** $i = 0$ **to** $d$ **do** ▷ Initialize randomness
    **for** $j = i+1$ **to** $d$ **do**
      $r_{i,j} \xleftarrow{\$} \mathbb{F}_2$
      $r_{j,i} \leftarrow r_{i,j}$
    **end for**
  **end for**

  **for** $\ell = 0$ **to** $n-1$ **do** ▷ Masking
    **for** $i = 0$ **to** $d$ **do**
      **for** $j = 0$ **to** $d, j \neq i$ **do**
        $\tilde{v}_{i,j}^\ell \leftarrow b_j^\ell + r_{i,j}$
      **end for**
    **end for**
  **end for**

  **for** $\ell = 0$ to $n-1$ **do**
    **for** $i = 0$ **to** $d$ **do** ▷ Correction and partial products
      $w_i^\ell \leftarrow a_i^\ell \cdot \mathtt{Reg}[b_i^\ell]$

      **for** $j = 0$ **to** $d, j \neq i$ **do**
        $u_{i,j}^\ell \leftarrow (a_i^\ell + 1) \cdot \mathtt{Reg}[r_{i,j}]$
        $v_{i,j}^\ell \leftarrow \mathsf{maj}(\tilde{v}_{i,j}^0 \ldots \tilde{v}_{i,j}^{n-1})$
        $z_{i,j}^\ell \leftarrow a_i^\ell \cdot \mathtt{Reg}[v_{i,j}^\ell]$
      **end for**
    **end for**
    **for** $i = 0$ **to** $d$ **do** ▷ Reduction
      $c_i^\ell \leftarrow \mathtt{Reg}[w_i^\ell] + \sum_{j=0, j\neq i}^{d} (\mathtt{Reg}[u_{i,j}^\ell] + \mathtt{Reg}[z_{i,j}^\ell])$
    **end for**
  **end for**
  **return** $c_0^0, \ldots, c_d^n$
**end procedure**

comes from different replications. Hence, by faulting $a_i^0$ or $a_i^1$ it is possible to force both $(a_i^0 + 1)$ and $a_i^1$ to be true and, hence, $r_{i,j}$ is observable in both corresponding terms. However, in a combined attack setting this flaw cannot be exploited for $d \leq 3$ and $k \leq 3$, as the corresponding

attack requires one fault (at $a_i^0$) and two probes (at $c_i^0$ and $c_i^1$). Hence, for the remaining cases, we describe an $\mathsf{HPC}_2$-based CINI gadget in Algorithm 8 and prove the security in Chapter 11 via a tool-based analysis. As $\mathsf{HPC}_3$ [KM22] also depends on the masked shares multiplication trick it suffers from the same limitations. Interestingly, $\mathsf{HPC}_2^{\mathsf{C}}$ achieves a tight realization of CINI, meaning that a well-placed fault indeed reduces the order of probing security by one (cf. Chapter 11). In contrast, $\mathsf{HPC}_1^{\mathsf{C}}$ actually achieves a higher security level than strictly required for some orders, e.g., the $(2,2)$-$\mathsf{HPC}_1^{\mathsf{C}}$ gadget is also $(2,1)$-ICINI and $(1,2)$-ICINI (see Section 9.10). This indicates some overhead in terms of randomness for $\mathsf{HPC}_1^{\mathsf{C}}$.

While potentially more efficient, gadgets based on detection have to avoid SCA leakage caused by fault propagation and construct a SCA-secure tree of detection flags. Both are non-trivial problems that we leave for future work.

## 9.10 Independent Combined-Isolating Non-Interference

The presented CINI definition requires that the number of probes and faults together is smaller than or equal to the order of probing security $d$. This means that it is of course possible to build a gadget that can resist $d'$ probes together with $k'$ faults for arbitrary $d'$, and $k'$, however, it requires an implementation with at least $d'+k'+1$ shares and is, hence, effectively a $(d'+k', k')$ gadget. This results in significant overhead in the number of shares and in consequence also in other metrics.

To avoid this overhead, we define Independent Combined-Isolating Non-Interference (ICINI), a composability notion for CA security where the order of probing and fault security can be selected independently. This independence requires only a small change compared to the CINI definition, namely that we now have $d_1 + d_2 \leq d$ instead of $d_1 + d_2 + k_1 + k_2 \leq d$, separating the number of injected faults and the order of probing security.

**Definition 44** (Independent Combined-Isolating Non-Interference)**.** *A gadget* $\mathsf{G}$ *is* $(d,k)$*-ICINI iff for any set* $\mathcal{F}_1$ *of* $k_1$ *faulty SRDs, every set of* $k_2$ *faults injected in gates of* $\mathsf{G}$*, any set of* $d_1$ *probes placed on intermediate values, and any set* $\mathcal{S}_2$ *of* $d_2$ *share domains, such that* $k_1+k_2 \leq k$ *and* $d_1 + d_2 \leq d$*, there exists a set* $\mathcal{F}_2$ *of at most* $k_2$ *SRDs and a set* $\mathcal{S}_1$ *of at most* $d_1$ *share domains such that the following holds:*

*Correctness:* *The gadget either aborts or gives an output where all values, except those belonging to the SRDs* $\mathcal{F}_1 \cup \mathcal{F}_2$*, are equal to the golden circuit, and there exists a decoding gadget* $\mathsf{G}^{\mathsf{D}}$*, such that given an input with at most* $k$ *faulty SRDs,* $\mathsf{G}^{\mathsf{D}}$ *outputs a correct result.*

*Privacy:* *The outputs of the share domains in* $\mathcal{S}_2$ *and the probes can be simulated with the inputs of the share domains in* $\mathcal{S}_1 \cup \mathcal{S}_2$ *and knowledge of the faults both injected and on inputs in* $\mathcal{F}_1$*.*

Please note, that in contrast to CINI the parameters for probing and fault security are now clearly separated. This also requires the use of correction countermeasures, as detection violates this separation [DN20]. To see this, assume an intermediate value that gets randomized by some randomness, i.e., $x_i^\ell + r$, where a reset fault is injected on $x_i^\ell$. The detection then removes the $r$ in some other fault domain $\ell'$ and an appropriate probe leaks $x_i$ (cf. Figure 11.3 in Chapter 11).

### 9.10.1 ICINI Security and Composition

The security and composition of ICINI really on the same fundamental properties as for CINI, i.e., more domains than allowed attack points and isolation of probe and fault propagation within the respective domains.

**Theorem 14.** *A $(d, k)$-ICINI gadget is $(d, k)$-combined secure.*

**Theorem 15.** *The loop-free composition of two $(d, k)$-ICINI gadgets is $(d, k)$-ICINI.*

### 9.10.2 ICINI Gadgets

One can observe that the provided CINI gadget for linear functions already adheres to the more restrictive ICINI property. The reason is the natural isolation of each SRD when implementing the function for each share and replication individually.

**Theorem 16.** *An implementation with $d + 1$ shares and $(2k + 1)$-times replication of a linear function is $(d, k)$-ICINI in the glitch-robust probing model.*

As with PINI and CINI, implementing a non-linear gadget is more complex and requires careful separation of probe and fault propagation. Of course, the same design principles as for CINI apply also for ICINI, namely (i) the masking of values crossing share-domain boundaries need to be refreshed, and (ii) values crossing SRD boundaries need to be corrected. In addition ICINI requires (iii) remasking is done with more than $k$ random values. This ensures that an attacker cannot remove the randomness from a refreshed value.

Interestingly, increasing the amount of randomness is sufficient to make the CINI multiplication gadget ICINI in the case of $\mathsf{HPC}_1^{\mathsf{C}}$.

**Theorem 17.** *The gadget $\mathsf{HPC}_1^{\mathsf{I}}$ as defined in Algorithm 9 with a register-free majority function is $(d, k)$-ICINI in the glitch-robust probing model.*

Remark 4, Remark 5, and Remark 6 also hold for the ICINI gadget in Algorithm 9 with the majority function in Line 20 and the masking in Line 14 and 30. Please note, the gadget $\mathsf{HPC}_2^{\mathsf{C}}$ cannot be easily transferred to ICINI since the contradiction mentioned in Section 9.9.2 then also occurs for smaller orders ($d \geq 2$ and $k \geq 1$).

## 9.11 Conclusion

In this chapter, we first present composability notions for gadgets protecting hardware implementations against FIA. We start by reviewing existing composability notions designed for software and transfer them to designs targeting hardware implementations. Additionally, we introduce a new security notion based on PINI.

In the second part of this chapter, we push the research of composability notions even further by approaching security notions for combined attacks. Again, we start by transferring software notions to the context of hardware implementations. Furthermore, we present two new composability notions with corresponding gadget instantiations for arbitrary security orders with respect to SCA and FIA.

**Algorithm 9** $\mathsf{HPC}_1^{\mathsf{I}}$: ICINI multiplication.

**Require:** $n = 2k + 1$
**Require:** $a_i^\ell = a_i^{\ell'}$ and $b_i^\ell = b_i^{\ell'}$ for $0 \le \ell, \ell' \le n, 0 \le i \le d$
**Require:** $\sum_{j=0}^{d} a_j^\ell = a$ and $\sum_{j=0}^{d} b_j^\ell = b$ for $0 \le \ell \le n$
**Ensure:** $c_i^\ell = c_i^{\ell'}$ for $0 \le \ell, \ell' \le n, 0 \le i \le d$
**Ensure:** $\sum_{i=0}^{d} c_i^\ell = a \cdot b$ for $0 \le \ell \le n$

**procedure** $\mathsf{HPC}_1^{\mathsf{I}}(a_0^0, \ldots, a_d^n, b_0^0, \ldots, b_d^n)$
  **for** $i = 0$ **to** $d$ **do** ▷ Initialize randomness
    **for** $j = i + 1$ **to** $d$ **do**
      **for** $m = 0$ **to** $k - 1$ **do**
        $\tilde{r}_{i,j,m} \xleftarrow{\$} \mathbb{F}_2$
        $\tilde{r}_{j,i,m} \leftarrow \tilde{r}_{i,j,m}$
        $r_{i,j,m} \xleftarrow{\$} \mathbb{F}_2$
        $r_{j,i,m} \leftarrow r_{i,j,m}$
      **end for**
    **end for**
  **end for**
  **for** $\ell = 0$ **to** $n - 1$ **do** ▷ Refreshing
    **for** $j = 0$ **to** $d$ **do**
      $\tilde{v}_j^\ell \leftarrow b_j^\ell + \sum_{i=0, i\neq j}^{d} \sum_{m=0}^{k-1} \tilde{r}_{i,j,m}$
    **end for**
  **end for**
  **for** $\ell = 0$ **to** $n - 1$ **do** ▷ Correction
    **for** $i = 0$ **to** $d$ **do**
      **for** $j = 0$ **to** $d$ **do**
        $v_{i,j}^\ell \leftarrow \mathsf{maj}(\tilde{v}_i^0 \ldots \tilde{v}_i^{n-1})$
      **end for**
    **end for**
  **end for**
  **for** $\ell = 0$ **to** $n - 1$ **do** ▷ Multiplication
    **for** $i = 0$ **to** $d$ **do**
      $w_i^\ell \leftarrow a_i^\ell \cdot \mathsf{Reg}[v_{i,i}^\ell]$
      **for** $j = 0$ **to** $d, j \neq i$ **do**
        $z_{i,j}^\ell \leftarrow a_i^\ell \cdot \mathsf{Reg}[v_{j,i}^\ell] + \sum_{m=0}^{k-1} r_{i,j,m}$
      **end for**
      $c_i^\ell \leftarrow \mathsf{Reg}[w_i^\ell] + \sum_{j=0; j\neq i}^{d} \mathsf{Reg}[z_{i,j}^\ell]$
    **end for**
  **end for**
  **return** $c_0^0, \ldots, c_d^n$
**end procedure**

# Part IV

# Novel Frameworks for Formal Hardware Verification

# Chapter 10

# Formal Verification of Countermeasures against Fault-Injection Attacks

*Over the last two decades, researchers proposed a plethora of countermeasures to secure cryptographic implementations against FIA. However, the design process and implementation are still error-prone, complex, and manual tasks which require long-standing experience in hardware design and physical security. Moreover, the validation of the claimed security is often only done by empirical testing in a very late stage of the design process. To prevent such empirical testing strategies, approaches based on formal verification are applied instead providing the designer early feedback.*
*In this chapter, we present a fault verification framework to validate the security of countermeasures against fault-injection attacks designed for ICs which has been presented in a joint work with Aein Rezaei Shahmirzadi, Pascal Sasdrich, Amir Moradi, and Tim Güneysu [RSS+21]. The verification framework works on netlist level, parses the given digital circuit into a model based on Binary Decision Diagrams, and performs symbolic fault injections. This verification approach constitutes a novel strategy to evaluate protected hardware designs against fault injections offering new opportunities for performing full analyses under a given fault model.*
*Eventually, we apply the proposed verification framework to real-world implementations of well-established countermeasures against fault-injection attacks. Here, we consider protected designs of the lightweight ciphers CRAFT and LED-64 as well as AES. Due to several optimization strategies, our tool is able to perform more than* 90 *million fault injections in a single-round CRAFT design and evaluate the security in under* 50 min *while the symbolic simulation approach considers all* $2^{128}$ *primary inputs.*

## Contents of this Chapter

## 10.1 Introduction

With the discovery of DFA in 1997 by Biham and Shamir [BS97], researchers from academia and industry proposed a plethora of countermeasures to secure cryptographic operations against FIA. More precisely, modern countermeasures leverage redundancy in time, area, or information and can be classified as detection-based, correction-based, and infection-based techniques. While detection-based countermeasures were intensively investigated in [AMR+20], using linear error codes, originally known from coding theory, to protect symmetric block ciphers, this approach was extended in [SRM20] such that linear error codes also were deployed to correct occurring faults. The last class – infection-based countermeasures – was investigated in [GST12] and infects the state of a cryptographic algorithm with random bits in case a fault occurred intending to generate useless outputs for an attacker.

However, despite extensive theoretical research on efficient and effective countermeasures, in particular the process of designing and implementing such methods in practice is still an error-prone, complex, and manual task, requiring longstanding experience and expertise in hardware design and physical security. Furthermore, the correctness and security of implemented designs and countermeasures are predominantly evaluated through empirical testing of prototypes or final products, making it difficult to correct or adjust design deficiencies and security flaws. To counteract this issue, formal verification can support the designer during the design process and provide an early indication of deficiencies and flaws.

As a consequence, the approach of empirical testing is replaced by security proofs which, however, require an appropriate definition of the adversary model and an abstraction of fault injection methods. In Chapter 8 we propose a new and generic fault model which allows to precisely define and describe an attacker. While the work from Arribas et *al.* [AWMN20] already presents a fault verification tool called VerFI, directly working on a gate-level netlist of a cryptographic circuit, it works with a limited set of fault models (i.e., bit-flip and stuck-at) and exposes some open challenges, particularly with respect to the reliability of the reported results. More precisely, as the tool is simulation-based, the user has to select dedicated test vectors, which can lead to undetected corner cases and false positive results. To this end, we close this gap by proposing a verification approach that inherently prevents misleading evaluation results and reports.

### 10.1.1 Contributions

We propose a formal verification approach and corresponding tool for countermeasures against FIA for cryptographic algorithms implemented on ICs. Hence, similar to VerFI [AWMN20], our approach works on a given gate-level netlist serving as starting point to create a model of the underlying digital logic circuit. However, instead of relying on empirical testing methods, we present a formal verification approach that is based on BDDs. This data structure inherently provides the possibility to observe the output of a Boolean function considering all combinations of the input variables. Hence, we avoid false positives that could be created by selecting inauspicious test vectors. Additionally, we propose a symbolic fault injection approach allowing to cover all possible fault events that can occur in digital logic circuits under a given fault model while avoiding to detect any undiscovered corner cases that may lead to successful fault injection attacks.

Furthermore, instead of fixing the applied fault model to a predefined subset as in VerFI, we incorporate the generic fault model presented in Chapter 8. This permits a precise definition and description of the adversary while analyzing the countermeasure's claims.

In order to achieve reasonable performance with respect to the evaluation time and circuit size, we present different kinds of optimization strategies. On the one hand, these strategies address the reduction of the complexity of the number of fault combinations that can occur in a digital logic circuit. On the other hand, we propose several approaches to increase the performance of our fault verification tool. The tool is publicly available and can be accessed via `https://github.com/Chair-for-Security-Engineering/FIVER`.

## 10.2 Fault-Injection Analysis

In this section, we briefly discuss existing countermeasures against fault injections and review the fault verification tool VerFI.

### 10.2.1 Fault Analysis Techniques and Countermeasures

In the context of cryptographic fault analysis, several techniques have been introduced to recover a secret key after a successful fault injection. Examples include DFA [BS97], SFA [FJLT13], Differential Fault Intensity Analysis (DFIA) [GYTS14], IFA [Cla07], and SIFA [DEK+18]. For more information, please see Section 2.2.3.

Depending on the situation and the scenario, various factors, such as access to faulty results, the precision of the fault injection, or the underlying cryptographic algorithm affect the final choice of the analysis technique. However, due to the efficiency of such powerful attacks, the research community has also dedicated a considerable body of research to propose methodologies for counteracting fault injections. For this, all approaches and countermeasures commonly rely on redundancy in terms of area, time, information, or any combination of them.

For instance, an encryption function can be instantiated twice (or multiple times) to form a basic detection scheme (based on spatial redundancy) allowing to check the consistency of the outputs through a simple comparison [MSY06]. Another trivial way to detect the presence of a fault is *recomputation* (as temporal redundancy), i.e., the construction recomputes the output multiple times using the same dedicated function and compares the results [MSY06]. In [AMR+20], a code-based approach based on CED schemes, i.e., information redundancy, has been proposed, where fault propagation in hardware implementations has been taken into account. More precisely, the authors guarantee the detection of any induced fault in any location of the design, including data path, Finite State Machine (FSM), control signals, and consistency check modules at any clock cycle. However, since the proposed technique is vulnerable against advanced attacks such as IFA and SIFA, the detection facility of [AMR+20] was extended to fault correction in [SRM20] to also protect implementations against IFA and SIFA.

Additionally, the authors proposed important properties and guidelines to design resilient hardware countermeasures against fault injection attacks. The most significant criteria, called *Independence Property*, was introduced in [AMR+20] and demands that a digital circuit is separated into independent parts such that each computes exactly one output bit. Then, a checkpoint is placed at the output of each separate part ensuring to detect or correct any fault within the capabilities of the underlying countermeasure. To this end, introducing a checkpoint

after each non-linear function ensures to stop fault propagation as early as possible and prevents unnecessary complexity when designing large circuits fulfilling the independence property over several non-linear functions. Hence, in the context of designing countermeasures for block ciphers, a checkpoint should be introduced after each substitution layer which ensures to detect occurring faults in each round.

A couple of more techniques have been proposed to protect against SIFA [BKHL20, SJR+20, RAD20]. A combined countermeasure against SCA and SIFA has been introduced in [DDE+20]. In this approach, the non-linear layer should be implemented by Toffoli gates and the whole design should be masked. Then, the authors claimed that a simple duplication can prevent a single-fault SIFA. In [BBB+21], a randomized duplication-based approach is presented, where no masking is needed.

### 10.2.2 State-of-the-Art Fault Verification

Practical evaluation of countermeasures against fault attacks on physical devices and real products is a complex and time-consuming task and needs considerable expertise and experience. Hence, this certainly highlights the necessity of verification tools and automated analysis techniques to accelerate evaluation and assist designers in the analysis of countermeasures. Moreover, it can help to reduce the cost of the fabrication process while maintaining the desired level of security.

In 2017, the authors of [BGE+17] presented a tool for automatic construction of algebraic fault attacks called *AutoFault*. AutoFault works on gate-level netlists and uses a SAT solver to detect possible vulnerabilities in a given design without deeper knowledge of the cipher construction. However, the user has to define a list of fault locations which limits evaluations to the corresponding areas of a target design. Hence, if AutoFault is used to verify countermeasures against fault attacks, the tool could report false positive results since a full coverage of all possible fault events is impossible.

In [KRH17], a framework, called XFC, for fault characterization in block ciphers was presented. It receives the block cipher specification and a fault model in order to determine locations for fault injection during the execution of the encryption. By tracing the fault propagation and its effects on the ciphertext, the tool evaluates the exploitability of a fault in terms of DFA and returns the computational complexity of the recoverable part of the (round) key. However, while XFC is mostly limited to a specific class of DFAs, ExpFault [SMD18] is designed to cover even more fault analysis techniques. Unfortunately, even though both tools can help adversaries to find the optimal location to inject faults and facilitate fault attacks, they are not able to assist designers in assessing the security of implementations equipped with fault attack countermeasures. As a consequence, this issue was addressed in a framework called *SAFARI* [RRHB20]. More precisely, this framework uses XFC to automatically identify locations that can be exploited by fault injection attacks, given a description of the (unprotected) target block cipher in a dedicated block cipher specification language. Then, based on user-defined security levels, SAFARI automatically equips the given block cipher with a parity or redundant-based countermeasure and returns HDL or C code accordingly. Recently, another work that focuses on the exploitability of fault injection attacks on microcontrollers was presented at CHES in 2019 [HBZL19]. The authors propose a tool called *TADA* which automatically detects vulnerabilities

of a block cipher software implementation on assembly level and returns exploitable faults with the help of an SMT solver.

Unfortunately, all aforementioned works aim to detect exploitable fault injection attacks on hardware or software implementations of block ciphers, hence taking an adversarial perspective. Only [RRHB20] additionally applies protection mechanisms to susceptible areas. However, none of these works take the perspective of the designer in targeting the assessment and formal verification of countermeasures against fault injection attacks implemented for hardware devices. More specifically, all those approaches are not able to verify a given design considering all possible fault events that could occur under all valid input combinations.

There are a few works addressing this topic by proposing open-source fault simulators for fault diagnosis [NCP92, LH96, BN08]. However, their application to cryptographic fault analysis is quite limited as they can simulate only a single-bit fault injection. As a result, VerFI [AWMN20] is the first automated open-source cryptographic fault diagnosis tool designed to evaluate fault-protected cryptographic implementations. For this, the tool directly operates on the gate-level netlist of a hardware design and is able to assess detection, infection, and correction-based countermeasures. Moreover, the user is able to define a fault model, a bounded adversary model, the location of the faults, the desired clock cycles for fault injections, and some input test vectors for simulation. Then, the tool simulates the circuit considering the parameterized fault injection and provides the coverage for every test vector and a final overall result including the total number of faults, all the non-detected faults per input test vector, reporting the location and type of faults, and the corresponding faulty outputs, which may assist the designer to identify the design failures.

### 10.2.3 Limitations of VerFI

Although VerFI facilitates the verification of fault-protected implementations, the result of the analysis depends on the selected input test vector(s). Hence, it is possible that the tool indicates the security of a design under a certain set of test vectors, while different test vectors would lead to observable or exploitable faults.

For this, let us consider a simple PRESENT S-box implementation [BKL+07], protected by a single bit of parity, as depicted in Figure 10.1. More precisely, the $S$ function receives a 4-bit input $S_{in} = \langle a, b, c, d \rangle$ and provides the 4-bit S-box output $S_{out} = \langle x, y, z, t \rangle$, where $a$ and $x$ are most significant bits. Simultaneously, the redundant part $S'$ operates on the 4-bit input, independent of $S$, and estimates the parity bit of the S-box output. Eventually, the consistency check module verifies the correctness of the $S$ output given the estimated parity bit and indicates a fault in case of inconsistency. However, such a design is not necessarily secure against single-bit fault injection in case fault propagation occurs in the $S$ function. In other words, for some test vectors, an attacker can inject a single-bit fault in such a way that an even number of faulty bits appear at the output, hence no opportunity to be detected by the consistency check module. One of such cases is shown in Figure 10.1, where a single-bit fault propagates to two different output bits ($x$ and $y$) depending on input test vector, i.e., $S_{in} \in \{$`0x1,0x2,0x3,0x9,0xA,0xB`$\}$. More precisely, the tool reports all single-bit faults are detected when $S_{in} \in \{$`0x0,0x4,0x5,0xD,0xE,0xF`$\}$ and there is at least one non-detected fault in the rest test vectors due to fault propagation. To mitigate this issue *independence property* has been defined in [AMR+20] to guarantee an $n$-bit induced fault affects at most $n$-bit output

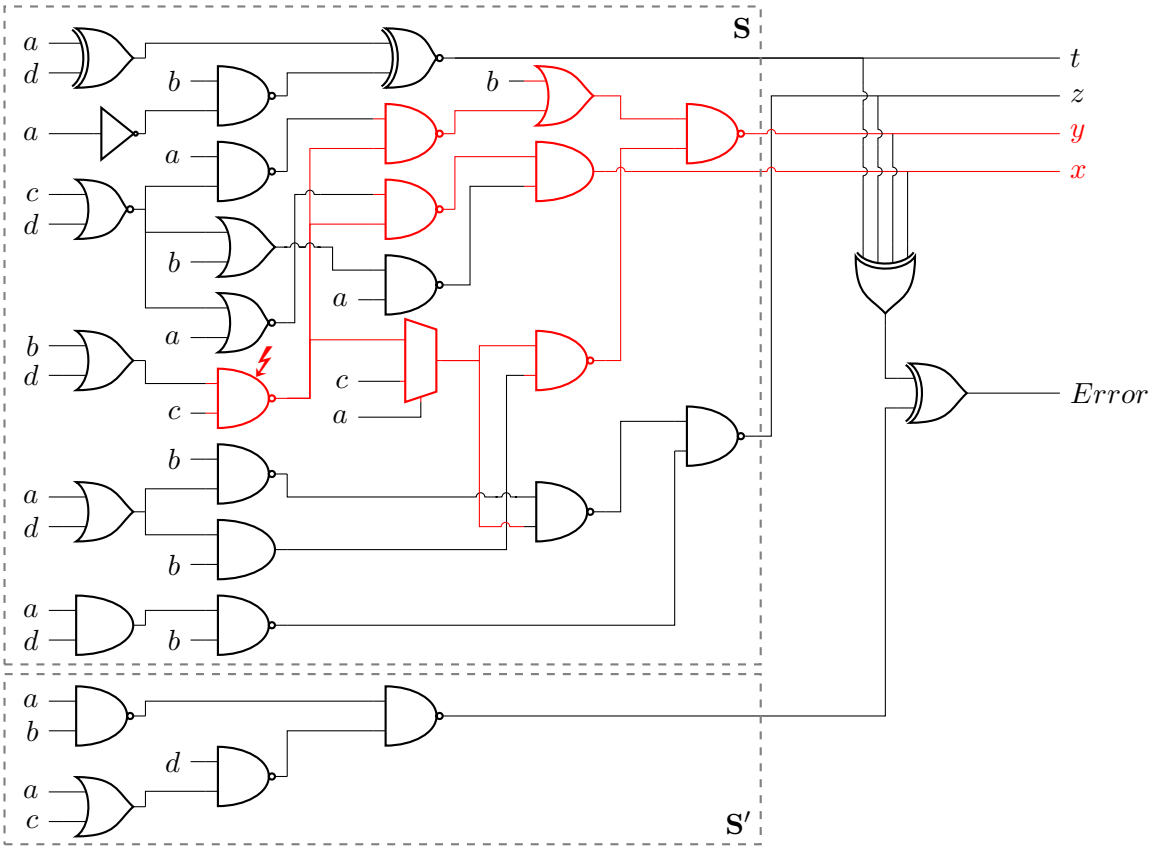



Figure 10.1: A simple PRESENT S-box implementation protected by a single-bit parity. The shown fault propagation happens when $\langle a, b, c, d \rangle \in \{$`0x1,0x2,0x3,0x9,0xA,0xB`$\}$.

bits. To this end, no cell should be involved in the computation of multiple output wires. As one can see, this property is not fulfilled in the given example, leading to insecure implementation for some test vectors in the underlying adversary model.

Hence, VerFI confirms the security of the design if the evaluation is only based on a limited number of test vectors, while additional test vectors could reveal the flaw. This behavior becomes even more challenging with increasing circuit size and number of inputs, as using VerFI it is almost impossible to check all input combinations for such a fault-protected cryptographic design. As a consequence, this highlights the importance of an automated tool that does not rely on the simulation of test vectors but symbolically checks all possible cases under given fault models.

## 10.3 Verification Concept

Before we present our verification approach in more detail, we introduce appropriate models for fault events and describe important terms required for diagnosing fault effects. The proposed verification approach covers the generation of circuit models, symbolic fault injection, and the corresponding fault diagnosis.

### 10.3.1 Fundamental Terminology

Formal verification of security requires formal descriptions and definitions of adversary models and security properties. More precisely, given capabilities and limitations of an adversary model, formal verification can prove security properties of any design under verification in the presence of the given adversary models. To this end, we briefly outline the fundamentals of our basic fault injection models as follows.

**Fault Events.** Any accidental condition that results in a malfunction or misbehavior of a digital logic circuit is considered as a fault event. However, while environmental faults have an erratic and random nature, adversarial faults are often precisely located and purposely injected into the circuit. Depending on their retention time, fault events can be classified as *transient*, *persistent*, or *permanent*. While transient fault events have a dynamic nature and volatilize after certain periods or changes in the circuit, elimination of persistent fault events requires a full reset of the circuit or system, whereas permanent faults are of static nature and will remain permanently. However, when modeling fault events, considering only transient fault events is sufficient, as any persistent or permanent fault event can be modeled as a repetitive transient fault event.

Observing that digital logic circuits are used to implement the computation of arbitrary Boolean functions $\mathsf{F} : \mathbb{F}_2^p \mapsto \mathbb{F}_2^q$, any fault event in such a digital logic circuit can be precisely modeled by another Boolean function $\mathsf{F}' : \mathbb{F}_2^p \mapsto \mathbb{F}_2^q$ as we propose in Chapter 8. More precisely, we model fault events at the structural level of logic circuits, assuming logic gates as atomic components, while the misbehavior of a single logic gate is considered as a fault event. As a consequence, changing the functionality of the misbehaving logic gate in the context of the entire circuit results in a clear specification of the *faulty* function $\mathsf{F}'$ that can be compared to the *golden*, i.e., fault-free, function $\mathsf{F}$.

**Fault Positions.** Given our adversary model, as introduced in Chapter 8, the adversarial capabilities are mostly determined and limited by the number of fault events that can be purposely injected into a single evaluation of a digital logic circuit[1]. More specifically, any fault injection might be limited and constrained in *spatial* or *temporal* dimension. For the spatial dimension, we mostly distinguish between *combinational* and *sequential* logic gates that are considered as sources for transient fault events. Further, in addition to the spatial locations, each adversary is also limited in the number of fault injections, i.e., the number of fault events that can be caused simultaneously within the same clock cycle. In fact, for the *temporal* dimension, we distinguish between *univariate* and *multivariate* fault injections. In that sense, univariate fault injections only consider fault events occurring in the same clock cycle, while for multivariate fault injections, fault events can occur in different clock cycles.

As a result, the total number of possible fault events, ultimately describing and limiting the adversarial capabilities, is derived as the product of the spatial and temporal limitations. This means, given $n$ fault events in spatial dimension, and $v$ fault events in temporal dimension, the total number of fault events that is injected into a single circuit evaluation is yielded by $n \times v$. Further, depending on the adversary model and if necessary, the distribution of fault

[1] In the context of this work, we focus on analysis and verification of clock-synchronous digital logic circuits only.

events can be adjusted, e.g., according to a *uniform* or *biased* distribution. However, whenever possible, we opt for an exhaustive fault verification, hence, allowing to cover any possible fault distribution.

**Classifying Fault Effects.** As indicated before, diagnosis of fault events and effects requires knowledge of the expected behavior. As a consequence, comparing the faulty behavior to the expected behavior of a golden circuit allows to evaluate and examine the *fault effectivity*.

In the context of pure *fault-detection* countermeasures, fault handling is delegated and escalated to the system. More precisely, in such a context, the circuit under diagnosis might expose a misbehavior that is detected and clearly communicated as such to the system level. As a consequence, for fault-detection countermeasures, we usually distinguish between *ineffective*, *detected*, and *effective* faults. For this, each fault event that does not lead to observable misbehavior is classified as *ineffective*, while all fault events that clearly lead to misbehavior that is not detected by the circuit are marked as *effective* faults, leaving the remaining events in the class of *detected* fault events. In contrast to this, *fault-correction* countermeasures attempt to correct any detected misbehavior immediately such that only *ineffective* or *effective* faults can be observed.

### 10.3.2 Verification Approach

In the following, we present our verification approach for fault injection countermeasures on hardware devices. More precisely, we explain how we use a Verilog gate-level netlist of a digital logic circuit to create an appropriate model. This model is used as a foundation to introduce techniques using BDDs to perform efficient evaluations of fault events. Eventually, we provide more insights into optimization strategies that allow supporting larger circuits.

**Requirements for Cryptographic Fault Verification.** There are some simulation tools [NCP92, LH96, BN08] in the field of integrated circuits testing, also known as reliability analysis, that examine the working environment stress, e.g., thermal cycling and vibration, or the potential manufacturing failures in a chip. However, they are not suitable for cryptographic fault analysis and verification of fault-protected implementations as they are commonly limited to single-bit faults. Moreover, often the user cannot set different fault models or specify desired locations for fault injection. It becomes even more challenging if the evaluated design incorporates dedicated countermeasures against fault attacks. In particular for detection- and infection-based countermeasures, the design returns a fixed value or a random value completely unrelated to the secret key if a fault event was recognized. Hence, the evaluation tool must be able to anticipate the behavior of the design and the integrated countermeasure for correct evaluation. While all these facts are supported by the recently-introduced fault-diagnostics tool VerFI [AWMN20], the result of such an evaluation is based on the given test vector(s). This may lead to a false-positive result. Namely, some faults may appear at the output only for certain input values and might be not detected by every test vector, like the example provided in Section 10.2.3. In this work, we mainly focus on cryptographic fault analysis that naturally covers every possible test vector, avoiding false positives.

**Abstraction Levels.** The circuit definitions introduced in Section 5.1 allow us to introduce two abstraction levels *structural* and *functional*. The structural level incorporates the edges and vertices of the DAG, i.e., the wires in the circuit $\mathbf{C}$ connecting the Boolean gates from $\mathcal{G}$. In a verification environment, the structural level is used to define and distinguish different areas of the original circuit, e.g., the register stages or dedicated modules that should be considered in an analysis. Furthermore, the structural level gives us the opportunity to develop special optimization strategies as we describe later in this section. However, the actual faults are injected at the functional level – directly in the combinational or sequential gates. At this level, we can precisely and generically cover several known fault models summarized in Chapter 8.

**From Netlist to Direct Acyclic Graph.** Figure 10.2 depicts the verification approach which we follow in this chapter. As already mentioned above, we analyze hardware circuits based on their (Verilog) gate-level netlist. In the first step, the netlist is transformed into the circuit model introduced in Section 5.1 and therefore converted into a DAG $\mathbf{D}$. The underlying data structure allows us to perform several preprocessing steps at the structural level, as follows.

- First, each node $d \in \mathbf{D}$ is attached with an information holding the gate type. It is accessed by the function $\texttt{type}\,(d)$ and returns one of the following values from $\mathcal{G}_t$.

$$\mathcal{G}_t = \{\mathsf{in}, \mathsf{out}, \mathsf{not}, \mathsf{buf}, \mathsf{reg}, \mathsf{and}, \mathsf{nand}, \mathsf{or}, \mathsf{nor}, \mathsf{xor}, \mathsf{xnor}\}$$

- Second, dependencies between the existing nodes in $\mathbf{D}$ are identified. To be more precise, each node $d \in \mathbf{D}$ is equipped with its *propagation path*, i.e., with a list of nodes that are influenced by the output of $d$.

- Third, all nodes in $\mathbf{D}$ are separated into two classes depending on whether the corresponding logic gate $g$ is from $\mathcal{G}_\mathrm{m}$ or from $\mathcal{G}_\mathrm{c}$, i.e., $g$ is whether a sequential or combinatorial gate, respectively. We access this information for a given node $d$ by the function $\texttt{location}\,(d)$.

- Forth, the structural level is perfectly suited to extract topological characteristics of the underlying circuit. This includes the assignment of each node $d \in \mathbf{D}$ to its logic stage.

A single logic stage consists of all combinatorial gates between two successive register stages. Special cases are 1) the first logic stage where the combinatorial gates are between the primary inputs and the first register stages, and 2) the last logic stage where the combinatorial gates are between the last register stages and the circuit's primary outputs. We use the function $\texttt{stage}\,(d)$ to refer to the logic stage of node $d$.

**Symbolic Simulation using BDDs.** The next step in our verification approach consists of mapping the Boolean function associated with each node $d \in \mathbf{D}$ to a BDD which includes the entire subgraph spanned by the node $d$. Therefore, the DAG is topologically sorted and each node $d$ is evaluated starting from the primary inputs. For each primary input, i.e., $\texttt{type}\,(d) = \mathsf{in}$, a new BDD variable is introduced. For all remaining nodes $d \in \mathbf{D}$ a BDD is constructed from the fan-in BDDs, based on the Boolean function associated with the node $d$. As an example, let us assume that a node $d$ is associated with an $\mathsf{and}$-gate. Therefore, $d$ has two input edges connected to two previous nodes which have already been evaluated (due to the topological sorting) and the

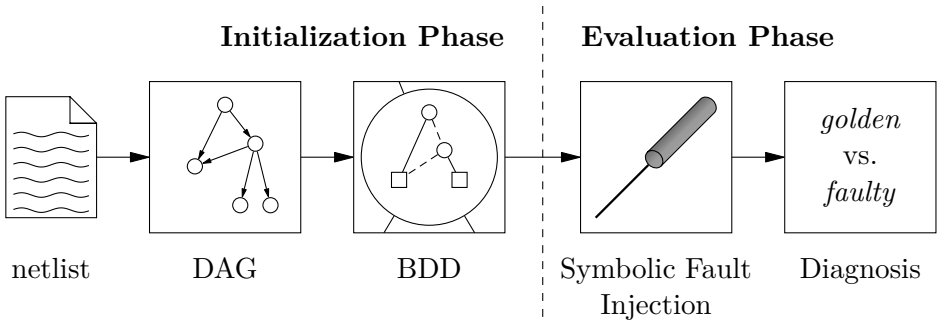



Figure 10.2: Flow of the proposed verification approach.

corresponding BDDs have been constructed. Then, the BDD for $d$ is constructed by computing the logical *and* of both fan-in BDDs based on the concept given in Section 5.2.

We decided to select BDDs as the underlying data structure to model a digital logic circuit since they offer many advantages. First, BDDs were originally proposed for defining, analyzing, testing, and implementing digital Very Large Scale Integration (VLSI) circuits [Jr.78]. Therefore, they seem to be a natural choice for our application, i.e., verifying hardware countermeasures against fault injection attacks. Second, one core idea of BDDs is to work with symbolic simulations which inherently consider all possible states of the BDD variables. Hence, the verification of a digital logic circuit is not limited to a predefined set of test vectors (inputs) but rather all valid inputs are tested and evaluated. This procedure avoids false positives as discussed in Section 10.2.2. Third, since executions of Boolean functions over BDDs are elementary operations (cf. Section 5.2), injecting faults by exchanging the associated Boolean function of the target node in the DAG $\mathbf{D}$ with a faulty one is a straightforward task that results in simple re-computation of the corresponding BDD.

**Symbolic Fault Injection.** Our concept of symbolic fault injection is based on the very generic approach presented in Chapter 8, i.e., modeling faults by replacing the original Boolean operation $\mathsf{F}$ of a target gate $g \in \mathbf{C}$ with another Boolean operation $\mathsf{F}'$ chosen from the same set of functions according to a predefined mapping $\tau$.

In the context of our verification approach, when analyzing the effect of a fault injection in a target logic gate, the corresponding graph node $d \in \mathbf{D}$ is replaced with another graph node $d'$, according to a fault mapping $\tau$. However, since each graph node is explicitly associated with a Boolean operation, the replacement of the graph node not only changes the structural description of the circuit node but also affects the functional behavior. More precisely, while $d$ is associated with a Boolean operation $\mathsf{F}$, the replaced graph node $d'$ is associated with a different Boolean operation $\mathsf{F}'$, necessitating a re-generation of the BDD for all subsequent graph nodes (affected by the fault injection). Note that for the remainder of this work, we denote the structurally modified DAG $\mathbf{D}'$ as the *faulty model*, while we refer to the original, fault-free DAG $\mathbf{D}$ as the *golden model*.

In fact, our verification approach is designed to analyze the golden model $\mathbf{D}$ under all possible fault events that can occur for a given fault model $\zeta(n, t, l)$. More specifically, the fault model $\zeta$ determines the fault verification process in terms of the number of graph nodes $n$ that should

be faulted, the fault mapping $\tau$ that is considered to replace the target nodes, and which circuit location $l$ should be considered (i.e., combinational logic, sequential logic, or both).

Therefore, in the first step, our verification approach creates a set of nodes $\Lambda$. Particularly, the nodes in $\Lambda$ are extracted from the golden model $\mathbf{D}$ according to the considered location parameter $l$, such that

$$\Lambda = \{\, d \in \mathbf{D} \mid \texttt{location}\,(d) = l \,\}\,.$$

In a next step, the nodes in $\Lambda$ are separated into $s$ subsets $\theta_i$ ($s$ denotes the total number of logic stages) holding all nodes belonging to the same logic stage, i.e., each subset $\theta_i$ is defined as $\theta_i = \{\, \lambda \in \Lambda \mid \texttt{stage}\,(\lambda) = i \,\}$ for $0 \leq i < s$. Therefore, the set of all valid gates $\lambda \in \Lambda$ categorized based on logic stages is noted by $\Theta = \{\theta_0, \theta_1, ..., \theta_{s-1}\}$. In particular, such a categorization allows to distinguish between univariate and multivariate fault injections, i.e., different fault injections with respect to the temporal dimension.

Besides considering the location parameter $l$ and the separation in the temporal dimension to create valid sets of nodes that should be faulted, we further consider the number of fault events $n$ that should be injected simultaneously in a single subset $\theta_i$. Therefore, we introduce the sets $\Gamma_i$ for $0 \leq i < s$ which hold all combinations of $n$ nodes that are available in a single subset $\theta_i$, formally defined as

$$\Gamma_i = \{\, \gamma \mid \gamma = \{d \in \theta_i\}, |\gamma| = n \,\}\,.$$

Note, however, that the cardinality of each $\Gamma_i$, i.e., the number of valid combinations of nodes that need to be evaluated in each logic stage, drastically increases in $|\theta_i|$ and $n$ since $|\Gamma_i| = \binom{|\theta_i|}{n}$.

Next, given a valid set of target nodes $\gamma \in \Gamma_i$, each node in $\gamma = \{d_0, \ldots, d_{n-1}\}$ is associated with a Boolean operation $\mathsf{F}$ which is replaced by faulty operations according to the fault type $t$ defined in $\zeta(n, t, l)$. In particular, the fault type $t$ (e.g., bit-flip, set, or reset) is defined and described by a fault mapping $\tau$ (cf. Chapter 8). For example, a fault mapping for the gate type `and` could be defined by $\tau : \{\texttt{and}\} \mapsto \{\texttt{set}, \texttt{reset}, \texttt{nand}\}$. Hence, the number of different fault mappings that can occur for one $\gamma$ depends on the cardinality of the corresponding fault mapping and is determined by

$$\prod_{j=0}^{n-1} \Big|\tau(\texttt{type}\,(d_j)\,)\Big|,\ d_j \in \gamma.$$

Eventually, each of these valid combinations leads to a new faulty model $\mathbf{D}'$ which should be compared to the golden model $\mathbf{D}$ to determine the effect of the injected fault (more details are provided in the subsequent paragraph).

As already mentioned above, our verification approach considers univariate and multivariate fault injections. For univariate fault injections, faults are injected in only a single set $\Gamma_i$. In contrast, for multivariate fault injections, $v$ different sets $\Gamma_i$ are selected, where $v$ denotes the number of different logic stages that can be faulty at the same time (e.g., setting $v = 2$ would describe a bivariate fault injection). Note, that in each logic stage (temporal dimension), $n$ nodes can be faulted such that $v \times n$ nodes of the golden model $\mathbf{D}$ are affected. Therefore, analyzing multivariate fault injections drastically increases the combinations of nodes that need to be evaluated. More precisely, each selection of $v$ different sets creates

$$\prod_{v} |\Gamma_i|$$

valid combinations.

**Fault Diagnosis.** The ultimate goal of our fault verification approach is the diagnosis of fault effectiveness and severity (cf. Figure 10.2). For this, given a golden model $\mathbf{D}$ and a faulty model $\mathbf{D}'$, the effects of fault injection in $\mathbf{D}$ resulting in $\mathbf{D}'$, are evaluated by analyzing and comparing both models. Particularly, the output nodes of both models are combined to new BDDs (commonly by an exclusive-or which we highlight in more detail in Section 10.4.2) in order to detect any differences in the outputs considering all valid assignments of the primary input variables. This strategy is especially beneficial for the data structure of BDDs since counting the number of satisfying variable assignments, i.e., leading to a logical 1, can be accomplished efficiently. Hence, determining and counting the satisfiability of the BDDs combining the outputs of the golden and faulty models, directly yields the number of input combinations leading to a difference in both models.

More precisely, based on the analyzed fault-injection countermeasure, incorporated in the design under test, the combined BDDs of the golden and faulty models are used to determine the number of *effective*, *ineffective*, and *detected* faults, as well as the total number of fault events as introduced in Section 10.3.1. Note, however, that the exact fault diagnosis procedure depends on the underlying countermeasure which we discuss in more detail in Section 10.4.2.

**Optimizations.** As already indicated before, this verification approach poses some challenges with respect to the complexity when analyzing large circuits or when the number of fault injections $n$ increases. Therefore, we further propose two optimization strategies which both rely on the structural analysis of the circuit model while being independent of the functional behavior of the circuit, i.e., the realized logical function.

The first strategy benefits from the identification of fault propagation paths, which are determined in the initialization phase. More precisely, the propagation paths are determined by a backwards-iterating algorithm given a topological sorting of the DAG $\mathbf{D}$. Hence, in a breadth-first search, the algorithm considers each node $d \in \mathbf{D}$ and adds the propagation paths of all nodes $d_i$ connected to the output edges of $d$ along with the node $d_i$ itself. This procedure generates topologically sorted lists of propagation paths since the nodes from the deepest logic levels are added first. Then, assuming a target node $\lambda \in \Lambda$ is faulted, i.e., the associated Boolean function is replaced, we can observe that not all BDDs associated with nodes $d \in \mathbf{D}$ need to be re-evaluated. In particular, evaluating only the nodes that are located on the fault propagation path is sufficient and reduces computational overhead especially when injecting faults on gates in deeper logic levels.

The second optimization strategy reduces the number of nodes that are tested in the evaluation phase (cf. Figure 10.2). This is achieved by creating a subset $\Lambda_{\mathrm{red}} \subset \Lambda$ which is used instead to generate the valid combinations of target nodes in $\Theta$, using the following ideas and observations. First, registers form synchronization points in a digital logic circuit $\mathbf{C}$ (cf. Definition 20) and occurring fault events will eventually manifest in such register stages. Hence, it is straightforward that all nodes in $\Lambda$ associated with a register in $\mathbf{C}$ also need to be included in $\Lambda_{\mathrm{red}}$. Second, all nodes $d \in \mathbf{D}$ associated with gates that are directly connected to registers (i.e., whose output edges are connected to a register input) are added to $\Lambda_{\mathrm{red}}$ as well since they influence the behavior of the synchronization points immediately. Third, we observed that most digital logic circuits have *sensitive gates* directing faults from several locations through the circuit, eventually manifesting in registers. More precisely, we cluster gates to Boolean functions $\tilde{\mathsf{F}} : \mathbb{F}_2^p \mapsto \mathbb{F}_2^1$ with $p > 1$, i.e., to Boolean functions that providing only single-bit outputs. The

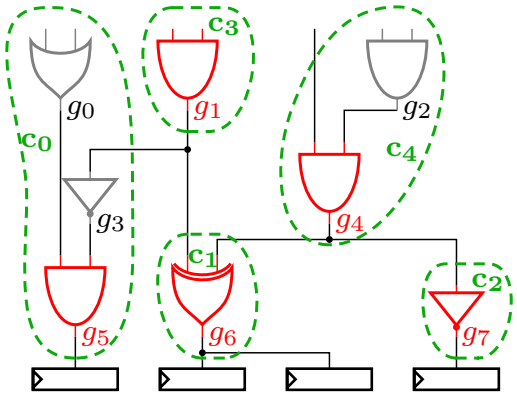



Figure 10.3: Clustering gates of a given digital logic circuit to reduce the verification complexity.

output gates of such clusters symbolize sensitive gates and fault propagations within such clusters are local and always pass them. Hence, fault injections in these clusters can be modeled by considering the sensitive gates only. Therefore, we add all nodes associated with sensitive gates to the reduced set of nodes $\Lambda_{\text{red}}$.

Note, this approach conservatively selects cluster of gates, i.e., each gate with fan-out greater than one is treated as a sensitive gate regardless of the fact that output signals are may recombined by another gate such that they only influence a single wire. Additionally, analyses using the reduced set of nodes $\Lambda_{\text{red}}$ should only be performed in the bit-flip model, i.e., $\zeta(n, \tau_{bf}, l)$. Again, this conservative approach models a worst-case scenario and ensures that a fault event is definitely effective. Hence, both restrictions guarantee full coverage of all possible fault events that otherwise may occur in a non-reduced set $\Lambda$.

However, switching to the introduced circuit model $\mathbf{D}$, nodes associated with registers and nodes directly connected to registers can be extracted from $\mathbf{D}$ in a straightforward way. All nodes $d \in \mathbf{D}$ associated with sensitive gates are identified by iterating over all nodes and extracting each node $d$ with more than one output edge. All three steps are formally defined in Algorithm 10 describing the complete process of generating the reduced set of nodes $\Lambda_{\text{red}}$. Here, Line 6 adds all nodes associated with registers to the reduced subset $\Lambda_{\text{red}}$ while Line 11 considers the input nodes directly connected to the registers. Eventually, Line 21 adds all nodes associated with sensitive gates to $\Lambda_{\text{red}}$.

Finally, we visualize the determination of sensitive gates and corresponding clusters of gates by an exemplary circuit depicted in Figure 10.3. Altogether, the exemplary circuit consists of five clusters denoted by $c_0, ..., c_4$. While gates $g_5$, $g_6$, and $g_7$ influence the existing registers directly, they also represent sensitive gates since faults always propagate through them. However, cluster $c_4$ consists of gate $g_4$ and $g_2$ where only $g_4$ is considered in a verification approach where Algorithm 10 is applied. To summarize, with the presented reduction approach it is enough to cover $\Lambda_{\text{red}} = \{\mathsf{regs}, g_1, g_4, g_5, g_6, g_7\}$ instead of $\Lambda = \{\mathsf{regs}, g_0, \ldots, g_7\}$.

## 10.4 The Tool

In this section, we present our fault verification tool FIVER (Fault Injection VERification) which realizes the approaches and concepts from Section 10.3. For this, we briefly introduce the

**Algorithm 10** Complexity Reduction.

**Require:** Golden circuit model **D**, set of valid fault location (nodes) $\Lambda$
**Ensure:** Set of reduced fault locations $\Lambda_{\text{red}}$

1: $\Sigma \leftarrow \emptyset$, $\Lambda_{\text{red}} \leftarrow \emptyset$
2: **for** $\forall d \in \mathbf{D}$ **do**
3: **if** type $(d) =$ reg **or** type $(d) =$ out **then**
4: $\Sigma \leftarrow \Sigma \cup d$
5: **if** $d \in \Lambda$ **then**
6: $\Lambda_{\text{red}} \leftarrow \Lambda_{\text{red}} \cup d$
7: **end if**
8: **end if**
9: **end for**
10: **for** $\sigma \in \Sigma$ **do**
11: $\Lambda_{\text{red}} \leftarrow \Lambda_{\text{red}} \cup$ node_in$(\sigma)$
12: $\Phi \leftarrow \sigma$
13:
14: **while** $\Phi \neq \emptyset$ **do**
15: $\alpha \leftarrow \Phi[0]$, delete$(\Phi[0])$
16: **for** $\forall n \in$ node_in$(\sigma)$ **do**
17: **if** type $(n) \neq$ reg **and** type $(n) \neq$ in **then**
18: $\Phi \leftarrow \Phi \cup n$
19: **end if**
20: **if** out_degree$(\alpha) > 1$ **and** $\alpha \in \Lambda$ **and** $\alpha \notin \Lambda_{\text{red}}$ **then**
21: $\Lambda_{\text{red}} \leftarrow \Lambda_{\text{red}} \cup \alpha$
22: **end if**
23: **end for**
24: **end while**
25: **end for**

applied BDD library and explain the general tool flow. Finally, we present some optimization strategies to improve the overall performance of our tool.

### 10.4.1 Colorado University Decision Diagram Package

The Colorado University Decision Diagram (CUDD) package is a BDD library developed by Fabio Somenzi at the University of Colorado [Som18]. The library is written in C but provides an interface to C++, used by our tool. Besides a large set of BDD operations offered by CUDD, it provides a large assortment of variable reordering methods. These methods allow reordering BDD variables such that the size of the underlying BDD is optimized. This is especially beneficial when the size of the evaluated circuit increases.

### 10.4.2 Tool Flow

In this section, we introduce our fault verification tool in more detail. To start the analysis of a target design, a configuration file needs to be provided first. Afterwards, the internal toolchain

is evoked and executed. At the end of the toolchain, an evaluation function is called in order to determine the number of effective, ineffective, and detected faults and generate the final evaluation report.

**Configuration.** Our fault verification tool uses a configuration file to specify and execute the desired analysis. This configuration file includes and sets parameters controlling the execution environment and host resources (e.g., CPU cores or memory) as well as the fault model parameters, including the number of fault injections $n$, the number of simultaneous fault injections $v$ in temporal dimension (e.g., univariate, bivariate, etc.), the location parameter $l$, and whether the complexity reduction approach should be applied or not. Furthermore, the definition of the fault mapping $\tau$ needs to be provided allowing the software to consider custom fault mappings for evaluation and diagnosis. However, along with the software on GitHub[2], we provide template definitions for common fault mappings (e.g., bit-flip, set, reset). Finally, a reference to a blacklist of entity names can be provided, excluding all matching modules from the fault injection process during the evaluation phase.

**Toolchain.** The toolchain is guided by the verification approach introduced in Figure 10.2. Hence, the Verilog netlist of the target design is parsed first. The outcome is an intermediate representation of the circuit containing the gate type, the list of input nodes, and additional annotations. Based on this intermediate representation, the DAG $\mathbf{D}$ of the golden model is generated. Besides, as this function already processes the intermediate representation, the annotations are used to identify the blacklisted entities. Afterwards, the CUDD library is used to process a topologically sorted representation of the DAG $\mathbf{D}$ and creates a BDD for each node $d \in \mathbf{D}$ based on the associated type. Further, if the configuration file enables the complexity reduction, Algorithm 10 is evoked, while the initialization phase is concluded by extracting all related graph properties including the number of logic stages, propagation paths, and nodes that need to be considered during the analysis.

During the subsequent evaluation phase, a fault verification function is called by passing the fault model parameters, the list of valid nodes, the golden model, and a BDD manager required for the CUDD library. Altogether, this function handles the most workload by iterating over four nested loops considering the number of fault injections $n$, the distinction in the temporal dimension, over all valid nodes, and finally on the lowest level over the defined fault mappings in $\tau$. Note that $n$ only determines the upper bound for the number of simultaneous fault injections, i.e., fault injections smaller than $n$ are considered in the analysis as well. This procedure is very common in the evaluation of countermeasures against fault injections and is done in the same way in Chapter 6 and in related work [SMG16]. However, on the lowest level, the tool performs the actual fault injection by replacing the types of the target nodes resulting in the faulty model $\mathbf{D}'$.

**Evaluation.** During evaluation and fault diagnosis, the faulty model $\mathbf{D}'$ is compared to the golden model $\mathbf{D}$. More precisely, the verification tool creates a new BDD for each output node in $\mathbf{D}$ and $\mathbf{D}'$. Specifically, let us denote the associated BDDs by $\mathsf{B}_i^{\mathbf{D}}$ and $\mathsf{B}_i^{\mathbf{D}'}$ for $0 \leq i < o$ and for the golden and faulty model, respectively.

[2] `https://github.com/Chair-for-Security-Engineering/FIVER`

Then, the evaluation BDDs $\mathsf{B}_i$ are generated such that $\mathsf{B}_i = \mathsf{B}_i^{\mathbf{D}} \oplus \mathsf{B}_i^{\mathbf{D}'}$, i.e., all output BDD pairs of the golden and faulty model are combined by an exclusive-or. This procedure allows to identify all input assignments under which the BDDs of the output nodes differ, i.e., a fault occurred and is visible at the output. In order to track occurring faults over the entire model, all $\mathsf{B}_i$ are further combined by an OR-operation leading to a single BDD $\mathsf{B}$. However, as in most detection-based countermeasures, an additional error flag indicates if a fault was detected, this information can be used to distinguish effective and detected faults. Particularly, if the BDD $\mathsf{E}$ of the error flag produces a zero while $\mathsf{B}$ generates a one, a fault injection leads to an effective (undetected) fault. Consequently, in case $\mathsf{E}$ and $\mathsf{B}$ leading both to a one, a fault was successfully detected by the design. As already introduced in Section 10.3, the data-structure of BDDs naturally covers all combinations for the given BDD variables. The number of combinations leading to a true assignment in a given BDD can efficiently be determined by a function counting the minterms which is also provided as part of the CUDD library. Hence, the number of effective faults is determined as `countMinterms`$(\mathsf{B} \cdot \overline{\mathsf{E}})$ while the number of detected fault is obtained by `countMinterms`$(\mathsf{B} \cdot \mathsf{E})$. Knowing the total number of fault events, the number of ineffective faults can be easily calculated by subtracting the number of effective and detected faults.

Note, that if countermeasures without an error flag should be analyzed, the evaluation function, i.e., the function that combines the output BDDs, needs to be adapted. However, this is easily possible without any deeper knowledge of the applied BDD library.

**Report.** As a final step, the tool reports all verification results in a text file. This includes a summary of the number of effective, ineffective, and detected faults, as well as the total number of fault scenarios that were tested[3].

Besides, for each detected effective fault, a clear description of the fault is added to the report. More precisely, all faulted gates leading to effective faults are listed as well as the function used to model the fault injection. This allows the designer to accurately determine the cause of the effective fault event in order to fix the flaw in the evaluated countermeasure.

### 10.4.3 Optimizations

In Section 10.3.2, we already introduced two optimization strategies based on determining the propagation paths of all nodes in $\mathbf{D}$ and on the reduction of the number of target nodes that need to be faulted, which we called *complexity reduction*. Besides those two approaches, we applied further optimizations which are directly related to the tool.

**Incremental Faulting.** The first approach optimizes the application of replacing the Boolean functions defined in a given fault mapping $\tau$. It is only effective for analysis with $n > 1$ and for nodes with $|\tau(d)| > 1$, i.e., for nodes that are changed to more than one function modeling a fault injection. Therefore, let us consider a current state of the faulty model $\mathbf{D}'$ where $n$ nodes are faulted. The tool would step on to the next valid set of fault mappings. But instead of just starting from a new golden model, the tool computes the difference between the previously applied fault mappings and the new fault mappings. Hence, if for example only the fault

[3] All numbers are reported on a logarithmic scale to avoid overflows in counting the statistics.

mapping for one single node changes, only the type of this node is adapted (triggering a re-evaluation of related BDDs) and not all BDDs associated with the remaining $n-1$ need to be re-evaluated. This *incremental faulting* approach can notably reduce computational time since the number of evaluations can be reduced and the same fault events are not performed multiple times.

**Resetting Faulty Model.** The next optimization approach addresses the *resetting* of the faulty circuit. More precisely, after a set of valid nodes $\gamma' \in \Gamma_i$ was faulted and analyzed, the tool proceeds with the next valid set $\gamma'' \in \Gamma_i$. Therefore, the functions from the nodes $\gamma'$ in $\mathbf{D}'$ need to be restored to the original functions defined in the golden model $\mathbf{D}$. In a straightforward approach, the golden model $\mathbf{D}$ could just be copied to the faulty model $\mathbf{D}'$ such that $\mathbf{D}'$ is fault free and the fault injections into the nodes defined in $\gamma''$ could be performed. However, this process can be very time-intensive especially for larger models $\mathbf{D}$ and therefore for larger circuits $\mathbf{C}$. Instead, we only change the types of the nodes defined in $\gamma'$ to the original types from the golden model $\mathbf{D}$. Even though this procedure triggers a re-evaluation of all BDDs placed in the propagation paths of the nodes in $\gamma'$, it turns out that this process increases the performance notably.

**Multithreading.** Finally, we parallelized the execution of our tool by using OpenMP[4]. The given problem is perfectly suited for parallelization since each set of valid nodes in $\Gamma_i$ can be evaluated independently. Therefore, the loop that iterates over the sets defined in $\Gamma_i$ is parallelized into the number of threads set up in the configuration file.

## 10.5 Case Studies

In this section, we apply the tool proposed in Section 10.4 to various cryptographic hardware implementations. More precisely, we evaluated detection-based and correction-based countermeasures against fault injection attacks attached to the lightweight ciphers CRAFT and LED-64 as well as the full block cipher AES. All designs were taken from [AMR+20, SRM20][5] while we unrolled the designs and only evaluated one or two rounds of the given circuit (for sake of complexity). Further, to obtain the Verilog gate-level netlists, we used the Synopsys design compiler with version E-2010.12-SP2.

In the next two subsections, we first present the evaluation results of the considered case studies, before discussing the limitations of our tool with respect to the size of a given circuit and the applied fault models.

### 10.5.1 Evaluation Results

We start our experiments by evaluating the counterexample from Section 10.2.2. Due to the symbolic fault injection approach, our tool is able to detect the existing flaws in the design and reports the corresponding gates leading to effective fault injections. However, we proceed our analyses with the lightweight cipher CRAFT [BLMR19] since it is built upon a simple

[4] `https://www.openmp.org/`

[5] The HDL code can be accessed at `https://github.com/emsec/ImpeccableCircuits`

structure leading to a small hardware footprint (i.e., a reasonable number of logic gates). As a next step, we decided to analyze LED-64 which is also a lightweight cipher but has a more complex structure [GPPR11]. Eventually, we challenge our tool by evaluating an AES-128 as it is roughly 14 times larger than the LED-64 design. All results are summarized in Table 10.1 obtained from a system running Ubuntu 18.04.2 with an Intel Xeon E5-1660 CPU with 3.2 GHz and 128 GB RAM. For all upcoming results, we fixed the number of threads used by our tool to eight while each thread could use up to 8 GB RAM (this is sufficient for most analyses considered in this chapter). More details about the performance with respect to the number of used cores and amount of memory can be found in Appendix 15.2 in Figure 15.1 and Figure 15.2.

**CRAFT.** For CRAFT, we consider detection-based countermeasures for a single-round design (protected against 1-bit, 2-bit, and 3-bit fault injections), a two-rounds design (with the same protection levels), and a two-rounds design which is protected against multivariate 1-bit and 2-bit attacks. Additionally, we provide evaluation results for correction-based countermeasures for single-round designs protected against 1-bit and 2-bit fault injections. For the analysis of single-round designs protected by a detection-based countermeasure, we instantiate the fault model as $\zeta(n, \tau_{bf}, cs)$, i.e., we consider bit-flip faults in combinational and sequential gates. The number of injected faults $n$ is adjusted to the countermeasure meaning that it is set to the maximum protection level of the considered design. The evaluations for the 1-bit and 2-bit designs are executed within 0.021 s and 1.496 s, respectively. However, the evaluation of the 3-bit design is more challenging because more than 90 million combinations need to be tested. Without any complexity reduction, this evaluation takes roughly 50 min while the application of Algorithm 10 decreases the evaluation time to only six minutes. Note, that all these analyses are performed under all input combinations for plaintext and key, i.e., $2^{128}$ valid inputs.

To demonstrate the functionality of FIVER, we also analyzed a subset of the provided countermeasures with fault models instantiated such that they describe fault injections exceeding the capabilities of our tool. The corresponding experiments are marked by red crosses in Table 10.1. As expected, the reports contain detailed lists of gates leading to effective fault injections.

The analysis of the two-round design gets more complex because each output depends on more primary inputs. While the BDD generations for the 1-bit and 2-bit design could be accomplished by the CUDD library and an evaluation could be executed without any complications, the structure of the 3-bit protected design is too complex such that the parsing and BDD generation process fails.

Next, we analyze the two-rounds design protected against multivariate attacks which consists of two register stages and therefore three logic stages. For the 1-bit protected design, we perform a bivariate ($v = 2$) and a multivariate ($v = 3$) evaluation, where the multivariate analysis takes roughly 7.5 h due to the increased number of combinations (even though we enabled the complexity reduction) as explained in Section 10.3.2. However, a bivariate analysis for the 2-bit protected design with $\zeta(2, \tau_{bf}, cs)$ is out of scope since the number of combinations is too large (over 200 billion). Nevertheless, switching to the fault model $\zeta(2, \tau_{sr}, s)$ which can be used to model fault injections caused by electromagnetic pulses (cf. Chapter 8), could reduce the number of combinations such that the analysis finishes within 23 h. Note that applying Algorithm 10 would not reduce the complexity since $\zeta(2, \tau_{sr}, s)$ only considers nodes associated with register anyways and they would also be added to $\Lambda_{\text{red}}$.

Table 10.1: Evaluation results for various ciphers protected against different levels of fault injections. A red cross in the last column indicates that the tool found effective faults which, however, is expected since the capabilities of the countermeasures were exceeded for these experiments. Experiments with simulation times marked by $\infty$ were not finished in a reasonable time such that we only report the number of combinations.

| **Redundancy (Capability*)** [bits] | **Verification Parameter** $\zeta(n,t,l)$ | Variate | Complexity Reduction | **Design Properties** Comb. Gates | Seq. Gates | Logic Stages | **Analysis Results** Combinations | Time [s] | Security |
|---|---|---|---|---|---|---|---|---|---|
| **CRAFT – 1 round (detection)** | | | | | | | | | |
| 1 (1) | $\zeta(1,\tau_{bf},cs)$ | univariate | no | 845 | 80 | 2 | 766 | 0.021 | ✓ |
| 1 (1) | $\zeta(2,\tau_{bf},cs)$ | univariate | no | 845 | 80 | 2 | 151 561 | 0.769 | ✗ |
| 3 (2) | $\zeta(2,\tau_{bf},cs)$ | univariate | no | 1 410 | 112 | 2 | 329 730 | 1.496 | ✓ |
| 3 (2) | $\zeta(3,\tau_{bf},cs)$ | univariate | no | 1 410 | 112 | 2 | 64 320 469 | 441 | ✗ |
| 4 (3) | $\zeta(3,\tau_{bf},cs)$ | univariate | no | 1 679 | 128 | 2 | 91 737 144 | 2 937 | ✓ |
| | | | yes | 1 679 | 128 | 2 | 4 665 200 | 360 | ✓ |
| **CRAFT – 2 rounds (detection)** | | | | | | | | | |
| 1 (1) | $\zeta(1,\tau_{bf},cs)$ | univariate | no | 1 571 | 160 | 3 | 1 491 | 0.378 | ✓ |
| 1 (1) | $\zeta(2,\tau_{bf},cs)$ | univariate | no | 1 571 | 160 | 3 | 417 882 | 62 | ✗ |
| 3 (2) | $\zeta(2,\tau_{bf},cs)$ | univariate | no | 2 526 | 224 | 3 | 868 500 | 157 | ✓ |
| 3 (2) | $\zeta(3,\tau_{bf},cs)$ | univariate | no | 2 526 | 224 | 3 | 250 984 950 | $\infty$ | – |
| | | | yes | 2 526 | 224 | 3 | 7 364 279 | 408 | ✗ |
| **CRAFT – 2 rounds – multivariate (detection)** | | | | | | | | | |
| 1 (1) | $\zeta(1,\tau_{bf},cs)$ | bivariate | no | 1 720 | 160 | 3 | 682 832 | 140 | ✓ |
| 1 (1) | $\zeta(1,\tau_{bf},cs)$ | trivariate | yes | 1 720 | 160 | 3 | 99 542 528 | 26 955 | ✓ |
| 3 (2) | $\zeta(2,\tau_{sr},s)$ | bivariate | no | 2 915 | 224 | 3 | 38 651 200 | 81 897 | ✓ |
| **CRAFT – 1 round (correction)** | | | | | | | | | |
| 3 (1) | $\zeta(1,\tau_{bf},cs)$ | univariate | no | 2 868 | 112 | 2 | 2 788 | 0.081 | ✓ |
| 3 (1) | $\zeta(2,\tau_{bf},cs)$ | univariate | no | 2 868 | 112 | 2 | 3 201 690 | 22 | ✗ |
| 7 (2) | $\zeta(2,\tau_{bf},cs)$ | univariate | no | 17 460 | 176 | 2 | 129 651 034 | 3 543 | ✓ |
| | | | yes | 17 460 | 176 | 2 | 10 923 888 | 130 | ✓ |
| **LED-64 – 1 round (detection)** | | | | | | | | | |
| 1 (1) | $\zeta(1,\tau_{bf},cs)$ | univariate | no | 1 541 | 0 | 1 | 1 301 | 0.064 | ✓ |
| 1 (1) | $\zeta(2,\tau_{bf},cs)$ | univariate | no | 1 541 | 0 | 1 | 846 951 | 9.558 | ✗ |
| 3 (2) | $\zeta(2,\tau_{bf},cs)$ | univariate | no | 2 435 | 0 | 1 | 1 730 730 | 27 | ✓ |
| 3 (2) | $\zeta(3,\tau_{bf},cs)$ | univariate | no | 2 435 | 0 | 1 | 1 072 477 550 | 12 722 | ✗ |
| 4 (3) | $\zeta(3,\tau_{bf},cs)$ | univariate | no | 2 916 | 0 | 1 | 1 654 087 449 | 17 348 | ✓ |
| | | | yes | 2 916 | 0 | 1 | 3 983 413 | 94 | ✓ |
| **AES-128 – 1 round (detection)** | | | | | | | | | |
| 1 (1) | $\zeta(1,\tau_{bf},cs)$ | univariate | no | 24 864 | 0 | 1 | 24 432 | 22 | ✓ |
| 4 (2) | $\zeta(2,\tau_{bf},cs)$ | univariate | no | 34 159 | 0 | 1 | 298 473 528 | $\infty$ | – |
| | | | yes | 34 159 | 0 | 1 | 56 632 584 | 471 281 | ✓ |

* The capability determines the maximum number of faults that can be detected or corrected by the corresponding countermeasure.

Eventually, we analyze the single-round CRAFT designs protected by correction-based countermeasures against 1-bit and 2-bit fault injections. To do so, we first slightly adapted the

fault diagnosis function such that we count the effective faults by `countMinterms` ($\mathcal{B}$) (cf. Section 10.4.2) as correction-based countermeasures do not provide an error flag. Therefore, we only count effective and ineffective faults but we cannot distinguish the number of corrected faults. However, the analysis of the 1-bit protected design is performed without reducing the set of target nodes and within 0.081 s. Switching to a 2-bit protected design requires seven bits of redundancy resulting in an increased number of target gates. Nevertheless, our tool can analyze the 130 million fault combinations in under 1 h and validates the security of the design. Again, applying the proposed approach to reduce the complexity (i.e., Algorithm 10), reduces the number of fault combinations to roughly 10 million while the simulation time is decreased to only 130 s.

**LED.** In our next case study, we analyze a single-round design of LED-64 protected by detection-based countermeasures against 1-bit, 2-bit, and 3-bit fault injections. For all three designs, we selected $\zeta(n, \tau_{bf}, cs)$ as fault model to allow a fair comparison to the CRAFT case study. As for CRAFT, the 1-bit and 2-bit countermeasures can be analyzed in a few seconds although the evaluation time increases compared to the analyses for the CRAFT design. Switching to the 3-bit protected design results in over 1.6 billion combinations that need to be tested. Nevertheless, our tool is able to perform this evaluation in under 5 h without any complexity reduction applied. Enabling the complexity reduction reduces the evaluation time roughly by a factor of 185. We also tried to analyze a two-round design of LED-64 but due to the increased dependencies of the outputs on the primary inputs, we are not able to parse the circuit into BDDs.

**AES.** In our last case study, we analyzed AES-128 protected by detection-based countermeasures against 1-bit and 2-bit fault injections. While the analysis for the 1-bit protected design can easily be managed by our tool (in only 22.5 s), the 2-bit protected design is more challenging. Hence, due to the enormous amount of gates (over 34 000), the number of combinations drastically increases. Therefore, we are only able to analyze the design by applying Algorithm 10 to reduce the complexity. Even then, the evaluation takes roughly 5.5 d but, nevertheless, it is manageable by our tool.

### 10.5.2 Limitations

Given the results of the three case studies, we now identify limitations of our tool with respect to circuit sizes and fault models.

**Circuit Size.** In two cases (CRAFT two rounds, LED-64 two rounds) our tool is not able to parse the circuit into the proposed data structures, i.e., the construction of the BDDs does not terminate. These problems occur because the outputs of the given circuit and therefore the BDDs of the output nodes in $\mathbf{D}$ depend on too many inputs, i.e., the depth of the BDDs increases. More precisely, the two-round CRAFT design (equipped with 3-bit protection) only consists of 3 739 gates which is clearly not the limiting factor since larger designs (e.g., CRAFT correction, AES) can be processed by our tool. This leads to the conclusion, that the circuit structure (i.e., the realized Boolean function) instead of circuit size prevents the

parsing into BDDs. For the two round LED-64 design, each output BDD depends on all 64-bit plaintext variables and on all 64-bit key variables. Hence, circuits with similar structures and dependencies are out of scope for our tool which, however, is expectable since otherwise common block ciphers could probably be broken. More precisely, if our tool could successfully parse a two-round LED-64 design, parsing an entire unrolled implementation of LED-64 would probably also be possible since the dependencies in the cipher would not increase. Therefore, the whole cipher could be analyzed over all possible combinations of input variables, i.e., considering all valid plaintexts and keys.

**Fault Model.** Limitations with respect to the fault model naturally occur when the number of simultaneous fault injections $n$ increases or multivariate fault injections should be analyzed. One of these limitations appears for the multivariate 2-bit protected CRAFT design under the model $\zeta(2, \tau_{bf}, cs)$ for $v = 2$. Evaluating this design without any complexity reduction would require to test more than 200 billion different fault combinations. For the given circuit size (i.e., 3 396 gates), this exceeds the capabilities of our tool. However, as already pointed out in Section 10.3, such a limitation naturally occurs due to the growth of the binomial coefficient. With the introduction of Algorithm 10, we can counteract this growth, but further improvement still remains an open research challenge.

**Circuit Structure.** As already indicated in Section 5.1, our tool is limited to unrolled digital logic circuits. This is mainly due to the underlying data structure of DAGs that does not allow any loops in our model. Therefore, a designer of a countermeasure has to unroll a target design before it can be processed by our verification tool.

**Iterative Block Ciphers.** Although our tool is mostly limited to the analyses of single-round or two-round implementations, we do not see major obstacles with respect to the verification of common countermeasures and the corresponding assertions. Particularly, when considering countermeasures based on linear error codes [AMR+20, SRM20], the underlying scheme usually protects each round with the same mechanism. Hence, an evaluation of a single round (univariate) or two rounds (multivariate) would be sufficient to verify the correctness of a protection mechanism. Similarly, countermeasures that are based on duplication are often equipped with detection or majority voting modules positioned at the end of a cipher execution. Again, these schemes could be seamlessly verified by our framework, focusing the analysis on the last round of the target scheme.

## 10.6 Conclusion

In this chapter, we present a framework to verify the security of countermeasures against fault-injection attacks designed for ICs. Given a Verilog gate-level netlist, our tool relies on BDDs to model the underlying Boolean function of the digital logic circuit and uses symbolic simulation to avoid false positive results while covering all possible input combinations. Further, assuming different fault models under consideration, our framework automatically identifies potential fault locations and performs a full analysis under all given models. Since the complexity of evaluations of digital logic circuits increases with circuit size, we propose various performance optimization strategies, ranging from algorithmic to programming-specific techniques.

Eventually, we conduct several case studies to demonstrate the application on real-world digital logic circuits implementing well-established countermeasures against fault-injection attacks. More precisely, we successfully analyze implementations of the lightweight ciphers CRAFT and LED-64 as well as the widespread AES. In fact, our tool is able to analyze more than 90 million fault injections for a single round of CRAFT in under 50 min while still testing all $2^{128}$ assignments of the primary inputs.

# Chapter 11

# Verification of Combined Attacks

*Side-Channel Analysis and FIA are well known and research provides many specialized countermeasures to protect cryptographic implementations against them. Still, only a limited number of combined countermeasures, i.e., countermeasures that protect implementations against multiple attacks simultaneously, were proposed in the past. Due to increasing complexity and reciprocal effects, the design of efficient and reliable combined countermeasures requires longstanding expertise in hardware design and security. With the help of formal security specifications and adversary models, automated verification can streamline development cycles, increase quality, and facilitate the development of robust cryptographic implementations.*
*In this chapter, we present the first automated verification framework that can verify physical security properties of hardware circuits with respect to combined physical attacks. To this end, we conduct several case studies to demonstrate the capabilities and advantages of our framework, analyzing secure building blocks (gadgets), S-boxes build from Toffoli gates, and the ParTI scheme. For the first time, we reveal security flaws in analyzed structures due to reciprocal effects, highlighting the importance of continuously integrating security verification into modern design and development cycles. Moreover, we demonstrate by practical measurements that precisely injected faults can lead to decreased side-channel protection in cryptographic algorithms protected by CINI gadgets. The contributions and results of this chapter are extracted from collaborations with Jakob Feldtkeller, Pascal Sasdrich, and Tim Güneysu [RFSG22, FRSG22]. Please note, any proofs that were presented in the original works are omitted in this chapter since they are not part of the contribution of the author of this thesis.*

## Contents of this Chapter

## 11.1 Introduction

Although SCA and FIA are well-known and well-studied threats to secure implementations of cryptographic algorithms, both threats and potential security mechanisms are mostly addressed and evaluated in isolation. More precisely, while *masking* and *redundancy* provide strong security guarantees against SCA and FIA, respectively, the impact and threat of combining FIA and SCA has long been neglected and underestimated, hence, only a limited number of combined countermeasures can be found in the literature.

As one of the first attempts, ParTI [SMG16], a first-order secure TI of LED-64, was protected against FIA based on a detection scheme using linear error codes for information redundancy. Based on the concepts presented in [SRM20], this TI can be extended to also provide correction-based protection instead of only detecting faults. Subsequently, different approaches have been investigated, e.g., using concepts of MPC [RMB+18], MACs [MAN+19], orthogonal error correction [RSBG20], or transformation and encoding [SJR+20], to build robust countermeasures against combined attacks. However, as it is well known, secure design and implementation of cryptographic algorithms require long-standing expertise and experience in the fields of hardware design and security, as the complexity, effort, and cost of these tasks increase dramatically with design complexity. For this, most recent approaches focus on the design and implementation of smaller building blocks, e.g., based on Toffoli gates [DDE+20] or masked multiplication *gadgets* [DN20], to compose larger designs from provably secure components.

Due to the ever-increasing and growing design complexity, formal verification nowadays is a fundamental part of VLSI design cycles in industry [BK18]. Regardless of this, existing formal verification mostly focuses on functional correctness (e.g., in form of model checking [CCGR99]) and is applied for example to entire CPUs as shown in [SSR+18] for RISC-V processors. Additionally, a widespread application of formal verification can be found in safety-critical environments, e.g., in the automotive industry [GLH18]. However, verification of secure implementation is often not considered. In light of this, recent hardware security research also stimulates progress and innovation for formal models of *active* and *passive* adversaries and the physical execution environments of modern electronic devices. Ideally, sound and accurate formal models can simplify and assist in the verification of security and functional correctness of cryptographic implementations to shorten and streamline hardware development cycles. For this, in the context of *masking*, formal security verification is commonly performed in the abstract and elegant Ishai-Sahai-Wagner (ISW) threshold $d$-probing model [ISW03], modeling side-channel leakage in terms of $d$ adversarial probes, providing access to intermediate values of a digital logic circuit during operation. Inspired by the sophistication of this formal security model, we present a consolidated fault adversary model in Chapter 8. However, as mentioned before, the complexity of verification increases dramatically with increasing design size and model complexity, rendering manual security verification nearly infeasible.

These challenges inevitably led to the research and development of automated security reasoning and computer-aided security verification tools. In connection with SCA and the $d$-probing model, existing verification tools either focus on specific countermeasures [ANR18], direct security reasoning [BGI+18, BBC+19b, KSM20, GHP+21, HB21, BMRT21], or assume securely masked gadgets and verify the correct composition under secure composability notions [BGR18, BDM+20, CGLS21]. In the context of FIA and active information tampering, state-of-the-art verification tools evaluate specific countermeasures [HPB21], algebraic vulner-

abilities [KRH17], or use simulation to ensure the robustness of redundancy-based hardware countermeasures [AWMN20]. Additionally, we presented FIVER, which is based on formal verification, in the previous chapter. As mentioned before, security threats from SCA and FIA are mainly considered in isolation and none of the existing security verification tools allow security reasoning under CA.

### 11.1.1 Contributions

Based on the formal definitions of combined security and composability notions presented in Chapter 9, we present the first automated framework for formal security verification of hardware circuits under CA, reconciling and uniting previous concepts that only addressed each physical attack in isolation. More specifically, this framework automatically verifies security of digital logic circuits and secure composability of fundamental building blocks (gadgets) in the presence of combined attacks. In light of this, we show that, due to unnoticed reciprocal effects, combined verification requires concepts and techniques beyond simple combination of SCA and FIA verification. Using these features, we conduct several case studies, i.e., analyzing the gadgets proposed in [DN20], the gadgets we proposed in Section 9.5, Section 9.9, and Section 9.10, dedicated SIFA countermeasure recently proposed in [DDE+20], and the ParTI protection scheme [SMG16]. In particular, these case studies helped to reveal unknown security flaws in [DN20] which impact software and hardware implementations, likewise. Consequently, the automated verification framework is able to perform stand-alone SCA or FIA verification as well as CA and is publicly available on GitHub[1]. We believe that our open-source tool VERICA sparks and supports new research on combined attacks and countermeasures.

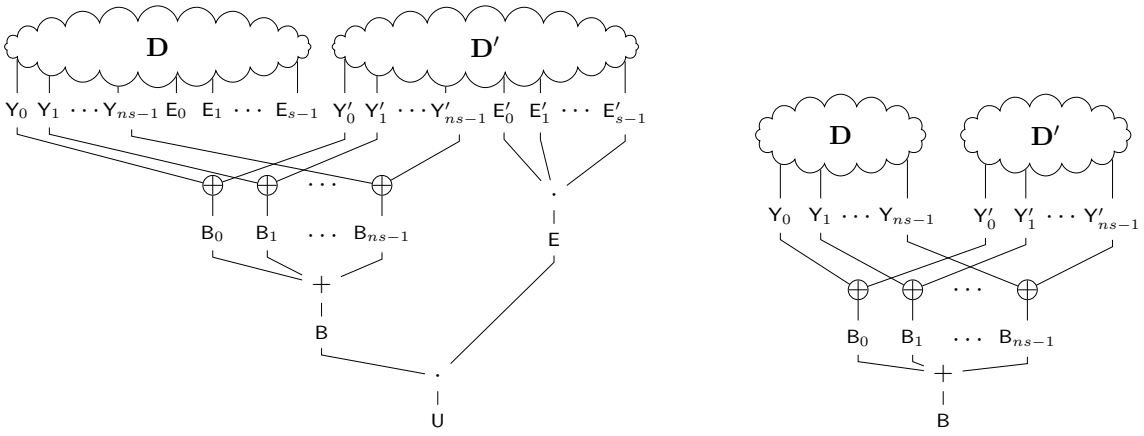
Additional to the verification results, we demonstrate that precisely injected faults in a CINI gadget can reduce the side-channel security by performing practical experiments. Hence, we confirm the verification results reported by VERICA with side-channel measurements on a FPGA evaluation platform.

## 11.2 Side-Channel Analysis Verification

The analysis and verification of resistance against SCA of a target circuit $\mathbf{C}$ closely follow the realization of SILVER originally presented in [KSM20]. SILVER is a formal verification framework that utilizes BDDs to verify the probing security and composability of digital logic circuits given as (Verilog) gate-level netlist. Due to the data structure of BDDs, the tool can efficiently check the statistical independence of two Boolean functions.

Hence, as introduced in Chapter 9, this perfectly enables verification in the glitch-extended $d$-probing model. Additionally, for gadgets solely designed against a SCA attacker, we consider the PNI and PSNI composability notions as recapitulated in Definition 6 and Definition 7, respectively. Besides, our composability verification approach also covers the analysis of the recently introduced PINI notation [CS20], closely following the approach of [KSM20].

[1] `https://github.com/Chair-for-Security-Engineering/VERICA`




(a) Detection-based. (b) Correction-based.

Figure 11.1: Evaluation strategies for detection- and correction-based countermeasures.

## 11.3 Fault-Injection Analysis Verification

Our verification for resistance against (stand-alone) FIA primarily adapts the concepts of our consolidated fault model introduced in Chapter 8 and our fault verification tool FIVER (see Chapter 10). We adopt the features from FIVER and further introduce extended verification methods in the following section. More specifically, we discuss fault diagnosis strategies to analyze detection or correction countermeasures (slightly adapted compared to the strategies from FIVER in order to cover shared circuits as well), approaches to evaluate resistance against statistical (ineffective) fault attacks, as well as composability of gadgets according to FNI and FSNI, as defined in Definition 35 and Definition 36, respectively.

**(Share-wise) Detection.** Even though the primary goal of this work is the verification of combined countermeasures, our framework still supports stand-alone FIA verification for detection-based countermeasures.

Given a faulty circuit model $\mathbf{D}'$, fault-free operation is indicated by $\mathsf{E} = \prod_{i=0}^{s-1} \mathsf{E}'_i$, assuming that each (share-wise) error flag $\mathsf{E}'_i = 1$ indicates a correct operation (see Figure 11.1a). Additionally, given the same input assignments, mismatching circuit outputs are derived as $\mathsf{B} = \sum_{i=0}^{ns-1} (\mathsf{Y}_i \oplus \mathsf{Y}'_i)$. Eventually, the number of effective faults, i.e., undetected faults visible at the circuit's output, is derived by counting satisfying assignments of $\mathsf{U} = \mathsf{E} \cdot \mathsf{B}$

**(Share-wise) Correction.** For pure correction-based countermeasures, usually coming without dedicated error detection flags, this procedure is simplified to only counting satisfying assignments of $\mathsf{B} = \sum_{i=0}^{ns-1} (\mathsf{Y}_i \oplus \mathsf{Y}'_i)$ (see Figure 11.1b).

**Statistical Ineffective Fault Analysis.** SIFA exploits statistical dependencies between fault injections and input values that alter the output distribution of the correct unshared data [DEK+18, DEG+18, DDE+20]. According to Hadžić et *al.* [HPB21], absence of vulnerabilities against SIFA can be proven through statistical independence of detection values and

processed secrets. In [HPB21], this is verified indirectly via *Boolean dependency analysis*, *factorization*, and *properties of masked computations*. However, as our framework naturally supports verification of statistical independence (due to the integration of analysis techniques provided by SILVER [KSM20]) and injections of faults in all available gates (features from FIVER), we opted to implement direct verification of SIFA.

Consequently, the fundamental step of the verification procedure is the derivation of the fault detection values. While for detection-based countermeasures this is implicitly given as $\mathsf{E} = \prod_{i=0}^{s-1} \mathsf{E}'_i$, for correction-based countermeasures this has to be explicitly constructed according to $\mathsf{E} = \prod_{i=0}^{ns-1} \overline{(\mathsf{Y}_i \oplus \mathsf{Y}'_i)}$. Eventually, in both cases, we verify statistical independence of $\mathsf{E}$ and the processed secrets to reason about resistance against SIFA[2].

**Composability Notions.** For secure composition of detection-based or correction-based gadgets, we specifically verify the notions of FNI (see Definition 35), FSNI (see Definition 36), and FINI (see Definition 38). For FNI and FSNI, we ensure that at most $k$ or $k_2 = k - k_1$ circuit outputs differ, i.e., $\mathsf{Y}_i \oplus \mathsf{Y}'_i = 1$, for all input assignments in case of FNI or FSNI, respectively.

For each verification that analyzes the FNI and FSNI notions, we additionally consider faults in primary gadget inputs and randomness gates $g \in \mathcal{G}_{\mathsf{rand}}$. This is independent of the location parameter $l$ of the fault model $\zeta(f, t, l)$. Further, considering the effects of faulty randomness, as discussed in Theorem 3 and Theorem 4, we adjust the golden circuit model according to Definition 34 and proceed with the fault diagnosis as before. More precisely, a fault injection in a randomness gate $g \in \mathcal{G}_{\mathsf{rand}}$ could lead to more than $k$ errors at the output of the target gadget $\mathsf{G}$ if the outputs are plainly compared to the fault-free gadget. We address these special cases in our verification process by tracking if a current fault injection includes faults in randomness gates. In case there is at least one randomness gate faulted, we apply the corresponding modification to the golden circuit model as well (i.e., altering the randomness gates in the golden circuit model according to the faulty circuit model), and compare the adapted golden circuit model with the faulty circuit model (cf. Figure 9.1d to 9.1f). This allows us to apply the same strategy of counting the satisfied BDDs $\mathsf{B}_i$ as explained above and compare the result to the threshold $k$ or $k_2$ for FNI or FSNI, respectively. The influence of this procedure on the side-channel verification is discussed in the next section.

In order to support the verification under the FINI notion, the inputs and outputs require an additional annotation to identify the different redundancy domains (cf.Definition 37). This information is used to track the redundancy domains of erroneous outputs and faulted inputs of a design under test. Based on Definition 38, we remove all input redundancy domains from the set of output redundancy domains and check if the cardinality of the resulting set exceeds the number of internal faults (if not, the design is FINI).

## 11.4 Combined Verification

After the introduction of stand-alone side-channel verification and fault injection verification, we now introduce our approach to perform combined verification. For this, we again rely on the glitch-extended $d$-probing model, now combined with the $\zeta(n, t, l)$ fault model. More specifically, our combined framework enables to verify $(d, k, t, l)$-combined security as well as

[2]For verification of SFA, the same concepts can be applied.

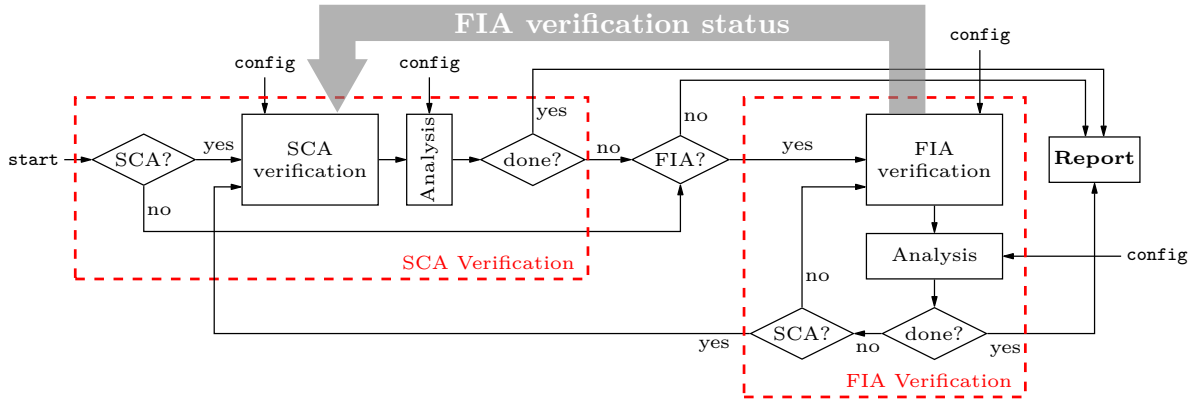



Figure 11.2: Concept of our combined verification approach.

secure composition under the CNI, CSNI, and ICSNI notions. Figure 11.2 visualizes the general verification concept, allowing to perform (stand-alone) SCA and FIA, as well as CA verification.

**$(d, k)$-Combined Security Verification.** Combined security verification always starts with verification of glitch-extended $d$-probing security, ensuring the secure implementation of the countermeasure in the absence of any faults. Afterwards, symbolic fault injection, fault diagnosis, and passive security verification are performed in an incremental *active-then-passive* approach, i.e., continuously increasing the number of faults while in turn performing fault diagnosis and SCA verification on the faulty circuit model. Hence, combined verification requires that the statistical independence checks for the SCA verification can also be checked on the faulty circuit model and not just on the golden circuit model. However, in a purely combined security verification scenario, the current state of the fault injection verification does not influence the checks for the probing security. More precisely, both verification techniques are applied independently and the number of faults does not reduce the number of probes.

**Composability Verification.** Verification of combined composability notions, i.e., CNI, CSNI, ICSNI, CINI, and ICINI again starts with verification in the absence of faults, hence is reduced to verification of PNI or PSNI.

In the presence of faults, verification of $(d, k)$-CNI and $(d, k)$-CSNI requires the reduction of available adversarial probes to $d' = d - k_1 - k_2$ as introduced in Section 9.6 and Section 9.7, respectively. Additionally, the attacker is allowed to learn $d - k + k_1 = d - k_2$ shares, given $k_1$ input faults, assuming that input faults provide additional probes. This interaction between the FIA verification and the SCA verification is highlighted in Figure 11.2 by the gray arrow and is very important for the *combined verification* of *combined composability notions*. Next, our framework checks PNI and FNI or PSNI and FSNI (under these modifications) for CNI or CSNI verification, respectively. As discussed in Chapter 9, we apply Definition 34 (i.e., adapt the golden circuit model in case randomness gates are faulted) in order to verify the FNI and FSNI notions. Note, this modification ensures that the detection or correction capabilities of the gadget under test are not exceeded. However, we do not automatically assume that such a fault does not influence the side-channel security. This is separately verified by the

SCA verification and therefore ensures that faults in randomness gates do not violate the SCA security assumptions.

For the verification of ICSNI, the FSNI and PSNI properties are verified independently without any modifications on probes and shares.

In order to check the correctness of the CINI notion, we implemented a similar strategy as for FINI but instead of relying on redundancy domains, we use SRDs as defined in Definition 42. Note that all faults on randomness are treated as internal faults, as discussed before. To verify the privacy, we modify the PINI strategy such that the number of allowed probes depends on the number of injected faults (cf. Definition 43).

For the integration of the ICINI notion, we independently inject faults and subsequently check the composability in the PINI model. The composability with respect to fault injections is accomplished by applying the same checks as for CINI (i.e., determining the number of faults in different SRDs).

**Optimizations.** Adopting reduction strategies from FIVER, we reduce the combined verification complexity through *incremental probing-security verification*. Particularly, we only consider altered probe combinations, i.e., probing the fault propagation path, during combined verification. In more detail, we compute all probe combinations that include at least one gate that was either altered by the fault injection or lies in one of the propagation paths of the altered gates. Hence, all probe combinations that can solely be created by gates that were not affected by the preceding fault injection can be neglected in the probing verification process leading to an increased verification performance.

**Soundness of our Verification Approach.** In this paragraph, we briefly discuss our verification approach and its soundness. All verification techniques utilized for our strategies are based on the $d$-probing model [ISW03] and the fault model presented in Chapter 8. Hence, our verification can at most be as precise as the abstractions made by these two models.

Nevertheless, we verify a circuit under test against these models in an *active-then-passive* approach as explained above. Since we always verify stand-alone SCA security on a fault-free model first, following an active-then-passive approach can be justified by Theorem 18.

**Theorem 18.** *Let* $\mathbf{C}$ *be an arbitrary circuit and* $\mathsf{G}^{\mathsf{D}}$ *a gadget realizing a decoding function* $D$*, such that, given an input with at most* $k$ *faults and an abort signal,* $\mathsf{G}^{\mathsf{D}}$ *either aborts or outputs a corrected result. Then* $\mathbf{C}$ *is combined secure iff the* active-then-passive *verification approach with preceding* passive *verification of* $\mathsf{G}^{\mathsf{D}}(\mathbf{C}(\cdot))$*, such that no fault or probe targets* $\mathsf{G}^{\mathsf{D}}$*, does not find an attack.*

## 11.5 Verification of Combined Gadgets

After the introduction of our verification concept, we now use our proposed framework VERICA to analyze the security of combined gadgets. Therefore, we first evaluate gadgets presented for software implementations from [DN20]. Afterwards, we verify the security of our FINI, CINI and ICINI gadgets presented in Chapter 9.

### 11.5.1 Verification of Gadgets from [DN20]

We start our case study by evaluating designs extracted from the gadget descriptions provided in [DN20]. The algorithms are originally designed for software implementations such that we added register stages at critical locations to stop glitches. Gruber et *al.* [GPK+21] presented a similar approach where they combined DOM gadgets with repetition codes. However, their work does not target the protection of gadgets but rather the protection of an entire cipher.

**Designs.** Before we present and discuss evaluation results, we briefly summarize and explain the different gadget variants and their design and security properties. Here, we adapt the naming NINA, SNINA, and SININA from [DN20] to describe the different gadget types.

**$(d, k)$-NINA:** NINA corresponds to our CNI security definition from Definition 39. We can construct a $(d, k)$-NINA gadget by implementing a shared xor gate which is $d$-order secure with respect to the threshold $d$-probing model. In order to fulfill the FNI security notion, the shared xor gate is replicated $k$ times such that no additional detection or correction mechanisms are required.

**$(d, k)$-SNINA.** [DN20, Algorithm 2] presents a detection-based design for the protected multiplication of duplicated shared values. As mentioned above, we opted to implement the gadgets in hardware while adding necessary registers on intermediate multiplication results (i.e., $u_{i,j,l}$) to ensure security in the glitch-extended $d$-probing model.

**$(d, k)$-SININA.** [DN20, Algorithm 5] constructs a protected multiplication gadget for duplicated shared values, relying on error correction instead of error detection. Again, we opted to implement Algorithm 5 in hardware and inserted additional registers where necessary to stop propagation of glitches.

All the different gadget variants, implemented in Verilog, are provided at GitHub and have been synthesized using Synopsys Design Compiler using a subset of cells in the NanGate 45 nm Open Cell Library (OCL). In addition, each detection-based gadget has been modified to provide a separate error detection flag per output share instead of returning a null output, as suggested in [DN20].

**Verification.** Combined verification of the gadgets is performed for an adversary who is able to precisely inject (at most) two independent set/reset faults into arbitrary gates of the circuits. As noted in [IPSW06], this fault model is more powerful for CA than the commonly employed bit-flip fault model. More specifically, each fault in the set/reset model can be modeled as an additional probe since an adversary is able to precisely inject values into the circuit, hence learning information on the processed data. In contrast to this, the commonly-used bit-flip model certainly maximizes the number of effective faults, however, an injected fault does not reveal information on processed data of the circuit and, hence, cannot be modeled as additional probe. Further, for CA, each gadget instance has been analyzed with respect to composability under the stand-alone PNI, PSNI, FNI, and FSNI security notions, first. Afterwards, we verify combined composability notions, i.e., CNI, CSNI, or ICSNI.

Table 11.1: Combined verification results for different gadget variants according to [DN20].

| Gadget | Design | | | | | SCA | | | FIA | | | Combined | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $d$ | $k$ | rand. | comb. | memory | PNI | PSNI | Time | FNI | FSNI | Time | | $(d,k)$ | Time |
| NINA | 1 | 1 | 0 | 4 | 0 | 1✓ | – | 0.460 s | 1✓ | – | 0.429 s | CNI | $(1,1)^{✓}$ | 0.430 s |
| NINA | 1 | 2 | 0 | 6 | 0 | 1✓ | – | 0.455 s | 2✓ | – | 0.445 s | CNI | $(1,2)^{✓}$ | 0.492 s |
| NINA | 2 | 1 | 0 | 6 | 0 | 2✓ | – | 0.471 s | 1✓ | – | 0.451 s | CNI | $(2,1)^{✓}$ | 0.436 s |
| NINA | 2 | 2 | 0 | 9 | 0 | 2✓ | – | 0.442 s | 2✓ | – | 0.444 s | CNI | $(2,2)^{✓}$ | 0.442 s |
| SNINA | 1 | 1 | 1 | 22 | 16 | – | 1✓ | 0.476 s | – | 1✓ | 0.449 s | CSNI | $(1,1)^{✓}$ | 0.473 s |
| SNINA | 1 | 2 | 1 | 38 | 26 | – | 1✓ | 0.451 s | – | 2✓ | 0.500 s | CSNI | $(1,2)^{✓}$ | 0.519 s |
| SNINA | 2 | 1 | 3 | 57 | 33 | – | 2✓ | 0.566 s | – | 1✓ | 0.456 s | CSNI | $(2,1)^{✗}/(1,1)^{✓}$ | 0.592 s |
| SNINA | 2 | 2 | 3 | 96 | 54 | – | 2✓ | 0.821 s | – | 2✓ | 0.673 s | CSNI | $(2,2)^{✗}/(1,1)^{✓}$ | 1.062 s |
| SININA | 1 | 1 | 2 | 90 | 30 | – | 1✓ | 0.450 s | – | 1✓ | 0.461 s | ICSNI | $(1,1)^{✗}/(0,0)^{✓}$ | 0.456 s |
| SININA | 1 | 2 | 3 | 360 | 50 | – | 1✓ | 0.555 s | – | 2✓ | 1.395 s | ICSNI | $(1,2)^{✗}/(0,0)^{✓}$ | 17.985 s |
| SININA | 2 | 1 | 6 | 207 | 63 | – | 2✓ | 1.334 s | – | 1✓ | 0.511 s | ICSNI | $(2,1)^{✗}/(0,0)^{✓}$ | 73.574 s |
| SININA* | 2 | 2 | 9 | 825 | 105 | – | 2✓ | 76.030 s | – | 2✓ | 5.300 s | ICSNI | $(2,2)^{✗}/(0,0)^{✓}$ | >2.7 h |

* Due to the high verification complexity, we interrupted the combined analysis after testing $(2,1)$-SININA where VERICA already reported a failure.

**Results.** All verification results provided in Table 11.3 were generated under a 64-bit Linux Operating System (OS) environment on an Intel Xeon E5-1660v4 CPU with 16 cores, a clock frequency of 3.20 GHz, and 128 GB of RAM. More precisely, each gadget variant has been instantiated for all combinations of $d \in \{1,2\}$ and $k \in \{1,2\}$.

Starting with the combined analysis of the proposed NINA gadgets, Table 11.3 reports the expected security under the CNI notion for all four gadgets. Note, however, according to Definition 39, the number of fault injections is limited by the side-channel security order $d$. Nevertheless, we can construct gadgets achieving higher protection against faults (by introducing more duplications) than against probes as shown for the $(1,2)$-NINA gadget in Table 11.3. Hence, once the number of fault injections is equal to or greater $d$, the gadget does not provide any protection against SCA but is still protected against fault injections. VERICA can handle these cases and therefore verifies $(1,2)$-CNI security.

The analysis of the SNINA gadgets shows that the $(1,1)$ and $(1,2)$ gadgets are secure under the CSNI notion. However, the remaining two gadgets are vulnerable to combined attacks. For example, the definition of the $(2,1)$-SNINA gadget allows to inject one single-bit fault while providing probing security up to the first order. However, injecting one precise fault at the input of this gadget leads to information leakage at the corresponding error flag. This is visualized in Figure 11.3 for the detection path of output $c_2$. It is assumed that the attacker injects a set/reset fault at input $a_2^1$ which is the third share of $a$ belonging to the second duplicate of the and gate. Without any loss of generality, we assume the attacker injects a set fault which leads to a propagation of the random input $r_1$ to gate $g_2$. The second input of $g_2$ provides the same multiplication result but from the second instantiation. Hence, the randomness $r_1$ is canceled out in $g_2$ and only $a_2 b_0$ is sampled in the subsequent register. Since all other data paths are fault-free, the remaining registers only store a logical 0 which eventually leads to leakage of the shares $a_2$ and $b_0$ at the output of the gadget which violates the PSNI property. Note, that

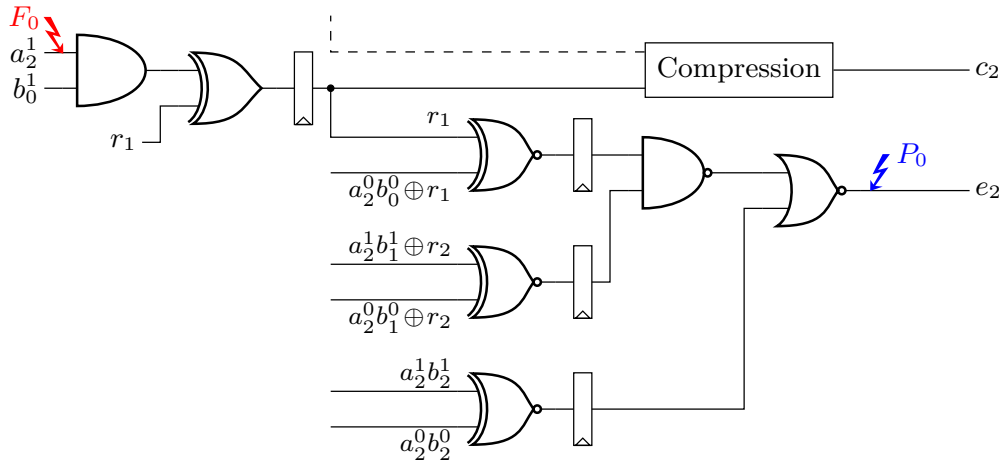



Figure 11.3: Combined attack for the $(2,1)$-SNINA gadget proposed in [DN20] considering a set/reset fault $F_0$.

this phenomenon is due to the difficulties to implement appropriate error detection signals in hardware (it is not caused by flaws in the definitions of [DN20, Algorithm 2] and does not occur in software implementations). Since the same data is processed in the different duplications and this data is merged in the detection paths, a violation of the CSNI property is expected and hard to prevent.

Eventually, all SININA gadgets are insecure under the ICSNI security notions. Figure 11.4 shows a schematic of a $(1,1)$-SININA gadget as suggested in [DN20] with the required modifications to secure it against hardware glitches. By Definition 41, a $(d,k)$ gadget should be secure even if an attacker injects up to $k$ faults and uses $d$ probes. However, in case an attacker injects a fault $F_0$ in one of the registers containing the partial products that were refreshed in the multiplication module (the most left modules in Figure 11.4), the probe $P_0$ can be used to observe the corresponding output after the compression step (xor gates in Figure 11.4). It can clearly be seen that the attacker is able to learn one share of $a$ and one share of $b$ which violates the PSNI property (cf. Definition 7). Hence, simultaneous protection against side-channel attacks and fault injection attacks is not guaranteed. Even though we slightly adapted the ICSNI definition and the gadget implementation (i.e., we explicitly consider faults in randomness gates in the security notion and add registers to the hardware implementation) compared to the original proposals from [DN20], the same flaws occur and can be transferred to the original work (i.e., the flaws also occur for software implementations). Hence, using VERICA, we were able to detect flaws in [DN20, Algorithm 5] which does not provide security against combined attacks.

Note, VERICA is not able to finish the verification of the $(2,2)$-SININA gadget since the design is complex to be analyzed by the proposed algorithms. However, within 2.7 h VERICA already failed checking the properties of $(2,1)$-SININA, such that $(2,2)$-ICSNI cannot be fulfilled.

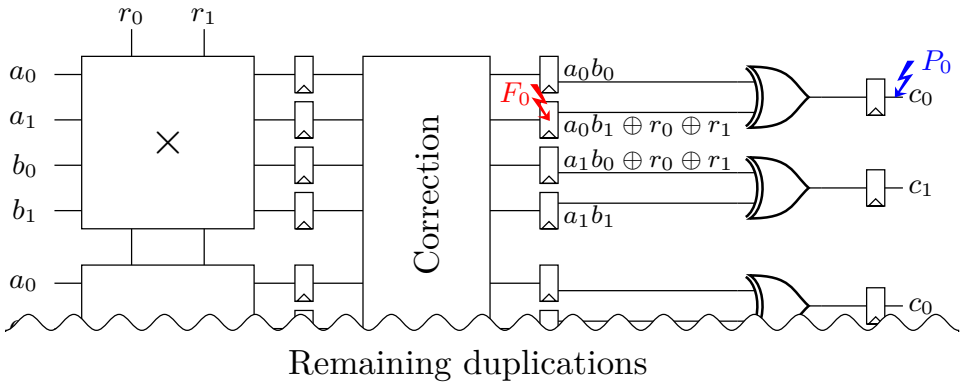



Figure 11.4: Combined attack for the $(1,1)$-SININA gadget proposed in [DN20] considering a set/reset fault $F_0$.

### 11.5.2 Verification of FINI, CINI, and ICINI Gadgets

In our second case study, we first describe the implementations of our FINI, CINI, and ICINI introduced in Chapter 9 in more detail. Afterwards, we conduct a formal verification to prove the claimed security using VERICA.

**Implementations.** We construct FINI gadgets according to Section 9.5, i.e., the target design is instantiated $k+1$ times for detection-based and $2k+1$ times for correction-based implementations. In addition, detection and correction gadgets follow the *independence property* with respect to redundancy domains by replicating the respective logic as well (cf. Algorithm 6).

For CINI and ICINI linear functions are constructed by instantiating the function $(d+1)(2k+1)$ times, i.e., for each SRD one instance, while multiplications are implemented according to Algorithm 7, Algorithm 8, and Algorithm 9. Please note, that an implementation has to adhere to Remark 4 and Remark 5. In Table 11.2 we provide the number of logic elements, dependent on the order of fault and probing security, required to instantiate the multiplication gadgets $\mathsf{HPC}_1^{\mathsf{C}}$ and $\mathsf{HPC}_1^{\mathsf{I}}$. Those numbers are without considering the implementation of the majority function, as there is no closed formula for the related implementation cost. Please note that the gadgets have the same implementation cost except for required randomness and the number of xor gates.

**Verification Results.** The implementations and verification results are shown in Table 11.3. We instantiated and analyzed a FINI and gadget equipped with a detection and a correction gadget for $k \in \{1,2,3,4\}$, respectively. All designs fulfill the FINI security notion while the evaluation of the correction gadget for $k=4$ requires a notable verification time of 6.24 h.

Next, we analyze the introduced CINI gadget from Algorithm 7 for $d \in \{1,2,3\}$ and $k \in \{1,2,3\}$ and report the required resources. We could verify the security for all gadgets except for the $(3,3)$-CINI gadget, as VERICA was not able to finish the analysis in a reason-

Table 11.2: Number of elements (without implementation of maj).

| | $\mathsf{HPC}_1^{\mathsf{C}}$ | $\mathsf{HPC}_1^{\mathsf{I}}$ |
|---|---|---|
| and | $(2k+1)(d+1)^2$ | $(2k+1)(d+1)^2$ |
| xor | $3(2k+1)d(d+1)$ | $(2k+1)^2d(d+1)$ |
| reg | $(2k+1)(d+1)^2$ | $(2k+1)(d+1)^2$ |
| maj | $(2k+1)(d+1)^2$ *[input size: $2k+1$]* | $(2k+1)(d+1)^2$ *[input size: $2k+1$]* |
| rand | $d(d+1)$ | $d(d+1)k$ |

able time. As already discussed in Section 9.9.2, we can construct for some parameters $d$ and $k$ a more tight CINI gadget applying Algorithm 8. Table 11.3 verifies the security for $d \in \{1,2\}$ and $k \in \{1,2\}$ while it reveals security flaws for a $(3,1)$ configuration. We confirm in Section 11.7 by a practical evaluation that well-placed faults can lead to a reduced side-channel security order.

Eventually, we instantiated and verified the ICINI gadgets for $d \in \{1,2,3\}$ and $k \in \{1,2,3\}$ (cf. Algorithm 9). The verification of these gadgets is more challenging since the number of faults does not reduce the number of probes. Therefore, VERICA is not able to verify several designs in a reasonable time (marked by $\infty$).

## 11.6 Verifcation of Cryptographic Primitives

In this section, we analyze entire cryptographic primitives. We start by demonstrating that our framework is able to validate protection against SIFA checking directly the independence of the secrets and the error detection flag as introduced in Section 11.3. This strategy is not supported by FIVER and is an additional feature provided by VERICA.

Afterwards, VERICA verifies a 4-bit S-box protected by the ParTI scheme which is one of the first countermeasures providing protection against SCA and FIA independently. Even though ParTI does not claim protection against *combined attacks*, we use the scheme to demonstrate that VERICA is able to check $(d,k)$-combined security (cf. Section 11.4) and that the mere combination of SCA and FIA countermeasures does not automatically result in combined security.

### 11.6.1 SIFA Constructions

While most countermeasures against SIFA are based on correction mechanisms [SJR+20, GPK+21], Daemen et *al.* proposed to use incomplete sub-circuits to avoid that effective faults become ineffective [DDE+20]. More precisely, the protected circuits are constructed from three basic circuits, i.e., a Toffoli gate $p_T(a,b,c) \mapsto \{a \oplus b \odot c, b, c\}$ [Tof80], a modified Toffoli gate $p_\chi(a,b,c) \mapsto \{\bar{a} \oplus b \odot c, b, c\}$, and a simple xor-gate.

To achieve the desired security, the basic circuits are masked and used as building blocks to construct cryptographic primitives. The masked circuits are given by

$$p_{TS}(a_0,a_1,b_0,b_1,c_0,c_1) \mapsto \{a_0 \oplus (b_0 \cdot c_1) \oplus (b_0 \cdot c_0), a_1 \oplus (b_1 \cdot c_1) \oplus (b_1 \cdot c_0), b_0, b_1, c_0, c_1\}$$
$$p_{\chi S}(a_0,a_1,b_0,b_1,c_0,c_1) \mapsto \{a_0 \oplus (\overline{b_0} \cdot c_1) \oplus (\overline{b_0} \cdot c_0), a_1 \oplus (b_1 \cdot c_1) \oplus (b_1 \cdot c_0), b_0, b_1, c_0, c_1\}$$

Table 11.3: Implementation and verification results for FINI, CINI, and ICINI gadgets synthesized with the 45 nm Open Cell Library.

| Gadget | Design | | | | | | Verification | | |
|---|---|---|---|---|---|---|---|---|---|
| | $d$ | $k$ | rand. | comb. | reg. | area [GE] | Def. | $(d,k)$ | Time |
| Detect | – | 1 | 0 | 3 | 0 | 4.7 | FINI | $(0,1)^{\checkmark}$ | 0.387 s |
| | – | 2 | 0 | 6 | 0 | 9 | | $(0,2)^{\checkmark}$ | 0.397 s |
| | – | 3 | 0 | 13 | 0 | 15 | | $(0,3)^{\checkmark}$ | 0.429 s |
| | – | 4 | 0 | 18 | 0 | 19.7 | | $(0,4)^{\checkmark}$ | 1.280 s |
| Correct | – | 1 | 0 | 15 | 0 | 17 | FINI | $(0,1)^{\checkmark}$ | 0.383 s |
| | – | 2 | 0 | 75 | 0 | 98.3 | | $(0,2)^{\checkmark}$ | 0.445 s |
| | – | 3 | 0 | 147 | 0 | 194.3 | | $(0,3)^{\checkmark}$ | 16.501 s |
| | – | 4 | 0 | 297 | 0 | 390 | | $(0,4)^{\checkmark}$ | 6.24 h |
| $\mathsf{HPC}_1^{\mathsf{C}}$ | 1 | 1 | 2 | 78 | 24 | 238 | CINI | $(1,1)^{\checkmark}$ | 0.409 s |
| | 2 | 1 | 6 | 189 | 54 | 567 | | $(2,1)^{\checkmark}$ | 0.485 s |
| | 3 | 1 | 12 | 356 | 96 | 1 032 | | $(3,1)^{\checkmark}$ | 39.544 s |
| | 1 | 2 | 2 | 340 | 40 | 685 | | $(1,2)^{\checkmark}$ | 1.490 s |
| | 2 | 2 | 6 | 795 | 90 | 1 595 | | $(2,2)^{\checkmark}$ | 6.321 s |
| | 3 | 2 | 12 | 1420 | 160 | 2 860 | | $(3,2)^{\checkmark}$ | 4.662 min |
| | 1 | 3 | 2 | 590 | 56 | 1 087 | | $(1,3)^{\checkmark}$ | 16.817 min |
| | 2 | 3 | 6 | 1362 | 126 | 2 502 | | $(2,3)^{\checkmark}$ | 3.897 h |
| | 3 | 3 | 12 | 2456 | 224 | 4 509 | | * | $\infty$ |
| $\mathsf{HPC}_2^{\mathsf{C}}$ | 1 | 1 | 1 | 66 | 36 | 294 | CINI | $(1,1)^{\checkmark}$ | 0.389 s |
| | 2 | 1 | 3 | 189 | 90 | 768 | | $(2,1)^{\checkmark}$ | 0.775 s |
| | 1 | 2 | 1 | 210 | 60 | 640 | | $(1,2)^{\checkmark}$ | 0.804 s |
| | 2 | 2 | 3 | 615 | 150 | 1 730 | | $(2,2)^{\checkmark}$ | 5.643 s |
| | 3 | 1 | 6 | 372 | 168 | 1 460 | | $(3,1)^{\times}/(2,1)^{\checkmark}$ | 18.386 h |
| $\mathsf{HPC}_1^{\mathsf{I}}$ | 1 | 1 | 2 | 78 | 24 | 240 | ICINI | $(1,1)^{\checkmark}$ | 0.397 s |
| | 2 | 1 | 6 | 189 | 54 | 573 | | $(2,1)^{\checkmark}$ | 4.329 s |
| | 3 | 1 | 12 | 356 | 96 | 1 044 | | * | $\infty$ |
| | 1 | 2 | 4 | 360 | 40 | 728 | | $(1,2)^{\checkmark}$ | 7.153 s |
| | 2 | 2 | 12 | 855 | 90 | 1 725 | | * | $\infty$ |
| | 3 | 2 | 24 | 1540 | 160 | 3 120 | | * | $\infty$ |
| | 1 | 3 | 6 | 646 | 56 | 1 203 | | $(1,3)^{\checkmark}$ | 4.743 h |
| | 2 | 3 | 18 | 1530 | 126 | 2 852 | | * | $\infty$ |
| | 3 | 3 | 36 | 2792 | 224 | 5 209 | | * | $\infty$ |

* Due to the extensive amount of combinations, these gadgets could not be verified with VERICA.

while necessary registers are added to achieve glitch-extended probing security. In the following, we first analyze these building blocks, before proceeding with the 3-bit permutation in Xoodoo [DHAK18], the 5-bit S-box in Keccak [BDPA13], and the AES S-box presented in [HPB21].

Table 11.4: Verification results for designs based on Toffoli gates [DDE+20, HPB21].

| Implementation | Design | | $\zeta(0, \tau_{sr}, \text{mc}_\infty)$ | | $\zeta(1, \tau_{sr}, \text{mc}_\infty)$ | | $\zeta(2, \tau_{sr}, \text{mc}_\infty)$ | |
|---|---|---|---|---|---|---|---|---|
| | comb. | mem. | SIFA | Prob. | SIFA | Prob. | SIFA | Prob. |
| $p_{TS}$ | 8 | 6 | – | $1^{\checkmark}$[0.47 s] | $1^{\checkmark}$[0.45 s] | $1^{\checkmark}$[0.45 s] | $1^{\times}$[0.46 s] | $1^{\checkmark}$[0.44 s] |
| $p_{\chi S}$ | 10 | 6 | – | $1^{\checkmark}$[0.45 s] | $1^{\checkmark}$[0.44 s] | $1^{\checkmark}$[0.45 s] | $1^{\times}$[0.46 s] | $1^{\checkmark}$[0.45 s] |
| $\chi_3$ | 30 | 30 | – | $1^{\checkmark}$[0.43 s] | $1^{\checkmark}$[0.46 s] | $0^{\times}$[0.46 s] | $1^{\times}$[0.46 s] | $0^{\times}$[0.49 s] |
| $\chi_5$ | 52 | 42 | – | $1^{\checkmark}$[0.44 s] | $1^{\checkmark}$[0.48 s] | $0^{\times}$[0.44 s] | $1^{\times}$[0.48 s] | $0^{\times}$[0.54 s] |
| AES S-box, $g_{104}$ [HPB21] | 631 | 0 | – | $0^{\times}$[13.80 s] | $0^{\times}$[194.89 s] | $0^{\times}$[191.93 s] | [$\infty$] | [$\infty$] |
| AES S-box, full [HPB21] | 634 | 0 | – | $0^{\times}$[13.90 s] | $1^{\checkmark}$[194.58 s] | $0^{\times}$[194.70 s] | [$\infty$] | [$\infty$] |

This case study should demonstrate the functionality of the SIFA extension introduced in Section 11.3. Due to the combination of statistical independence checking (as presented in SILVER) and fault injection (as presented in FIVER), the verification support of SIFA-based countermeasures can be realized by checking the statistical independence of the secrets and the error detection flag.

**Masked Toffoli Gates.** The masked Toffoli gate $p_{TS}$ consists of four simple Toffoli gates that process the two shares of the masked input data. We first verify the security against side-channel attacks in the glitch-extended $d$-probing model and no fault injections, i.e., $\zeta(0, \tau_{sr}, \text{mc}_\infty)$. As expected and shown in Table 11.4, the design is secure against first-order side-channel attacks. Next, we perform an analysis with enabled fault injections using $\zeta(1, \tau_{sr}, \text{mc}_\infty)$ as fault model and the SIFA strategy presented in Section 11.3. As claimed by the authors, the fault detection value is independent of the secret input values, i.e., the unshared input data $a = a_0 \oplus a_1$, $b = b_0 \oplus b_1$, and $c = c_0 \oplus c_1$. Hence, the design is secure against single-bit SIFA attacks. Interestingly, the design is still first-order secure in the glitch-extended $d$-probing model even in the presence of single-bit faults. Eventually, we also evaluate the masked Toffoli gate under the fault model $\zeta(2, \tau_{sr}, \text{mc}_\infty)$. The design is still first-order secure (with the same argument as above), however, the tool reports only SIFA security under single-bit faults (which is expected).

We performed the same experiment for the adapted Toffoli gate $p_{\chi S}$. The verification results are similar to the results for $p_{TS}$.

**Xoodoo 3-bit S-box.** VERICA confirms the first-order probing security of the 3-bit S-box used on Xoodoo, implemented with the masked Toffoli gates $p_{TS}$ and $p_{\chi S}$. However, as the countermeasure was not designed to provide $(1, 1)$-combined security, single-bit faults lead to successful first-order probing attacks.

**Keccak 5-bit S-box.** Similarly, the 5-bit S-box used in Keccak, again implemented with Toffoli gates, is probing secure in the glitch-extended $d$-probing model but does not provide $(d, k)$-combined security.

**AES S-box.** Eventually, we examine two different implementations of the AES S-box presented in [HPB21][3] which is also built from Toffoli gates. The first implementation is automatically optimized to reuse the intermediate results $g_{104}$ in order to demonstrate a flaw in the SIFA protection [HPB21], which is confirmed by VERICA. In contrast to this, the (non-optimized) full AES S-box design in [HPB21] is secure against single-bit SIFA attacks. Interestingly, both designs are not secure in the glitch-extended $d$-probing model, since no register stages have been added to stop potential glitches. Unfortunately, due to the increasing complexity and number of fault combinations, VERICA is not able to analyze the S-boxes for two simultaneous injected faults.

### 11.6.2 ParTI Verification

In 2016, ParTI presented a first-order secure TI with linear ECCs to provide protection against SCA and FIA [SMG16]. As a case study, the authors applied their approach to the lightweight cipher LED-64 [GPPR11]. Even though the design is not secure against combined attacks, we implemented the scheme to demonstrate that VERICA is able to check $(d, k)$-combined security (cf. Section 11.4).

**Designs.** In this work, we implement a first-order secure LED-64 S-box based on the TI scheme. We create two designs that are additionally protected against FIA by applying ECCs. The first design uses the error detection capabilities of the $[8, 4, 4]$-code as was done for ParTI. Hence, this design is expected to be secure against first-order SCA and up to 3-bit faults should be detectable (Hamming distance of the code is four). In the second design, we utilized the error-correcting capabilities of the linear code such that the implementation is able to correct single-bit faults (and, again, secure against first-order SCA without fault injections). For both implementations, we satisfy the *independent property* introduced in [AMR+20, SRM20]. In the following, we present the verification results provided by VERICA.

**Verification.** Table 11.5 shows the verification results for both S-boxes. As expected, the first design provides the claimed probing security (during fault-free operation). Similarly, all faults in the $\zeta(1, \tau_{sr}, \mathrm{mc}_\infty)$ model are detected by the linear ECC. As expected, the design is not $(1, 1)$-combined secure which was also not claimed in [SMG16] For verification of correction capabilities, we excluded the final correction stage from the analysis since injected faults cannot be detected or corrected there. Nevertheless, the verification results are similar to the design that only uses detection. However, due to the increased number of gates, the verification complexity increases significantly.

## 11.7 Practical Evaluation

Besides verifying our CINI-designs with VERICA, we additionally perform practical measurements. More precisely, we synthesized a pipelined $(2, 2)$-CINI PRESENT S-box for the Sakura-G side-channel evaluation board which is equipped with a Xilinx Spartan 6 FPGA. We focus our practical evaluation on $\mathsf{HPC}_2^{\mathsf{C}}$ gadgets since they tightly fulfill Definition 43. The design requires

[3]The source files are publicly available at `https://extgit.iaik.tugraz.at/scos/danira`

Table 11.5: Verification results of a LED-64 S-box protected by a combination of TI and error detection or correction+[SMG16].

| Implementation | Design | | $\zeta(0, \tau_{sr}, \mathrm{mc}_\infty)$ | | $\zeta(1, \tau_{sr}, \mathrm{mc}_\infty)$ | |
|---|---|---|---|---|---|---|
| | comb. | memory | Det./Corr. | Prob. | Det./Corr. | Prob. |
| ParTI S-box (Detection) | 678 | 78 | – | $1^{\checkmark}$[0.866 s] | $1^{\checkmark}$[1.010 s] | $0^{\times}$[1.950 s] |
| ParTI S-box (Correction) | 2063 | 72 | – | $1^{\checkmark}$[4.103 s] | $1^{\checkmark}$[3.677 s] | $0^{\times}$[336.239 s] |

twelve bits of fresh randomness which is provided by a KECCAK core instantiated as PRNG. The FPGA is supplied with a 4 MHz clock and the current is measured indirectly via the voltage drop over a shunt in the supply path. To acquire suitable power traces, we use a ZFL-2000GH+ LNA configured with a 25.5 dB gain and a Spectrum M4 oscilloscope (8 bit resolution) with a sample rate of 2.5 GS/s.

For the first experiment, we instantiate a fault-free design on the FPGA and evaluate the first three statistical moments using a univariate Welch's $t$-test as described in Section 2.1.3.

Figure 11.5 depicts the corresponding results while Figure 11.5a shows a sample trace of the S-box evaluation. As expected, the $t$-test only indicates leakage in the third statistical moment.

For the second experiment, we inject a persistent stuck-at-zero fault into one of the randomness gates. The corresponding measurement is shown in Figure 11.6. As formally noted in Definition 43, and confirmed by our measurements shown in Figure 11.6c, the one-bit fault reduces the side-channel security by one order.

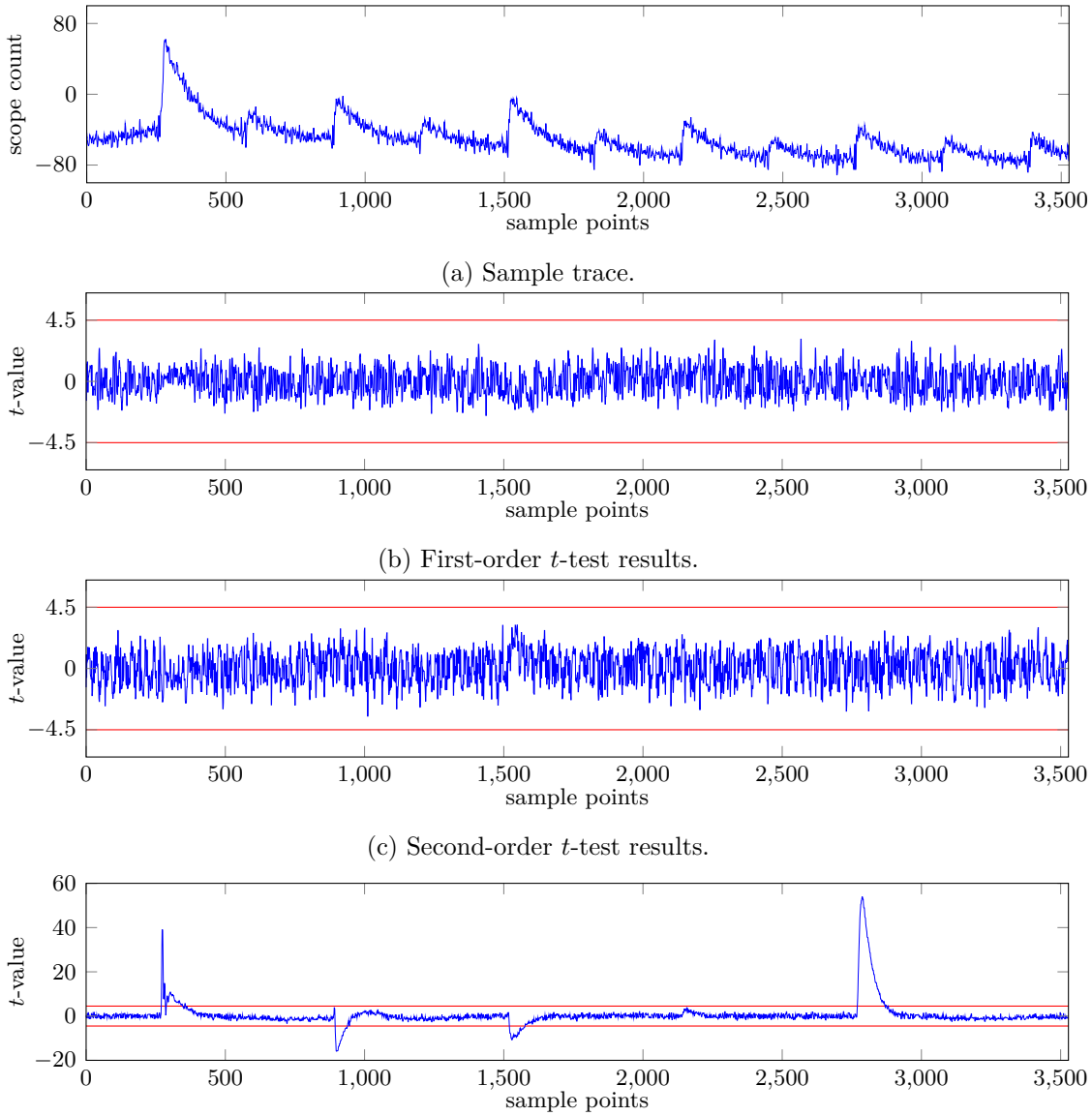
Eventually, for our last experiment, we inject two persistent faults in the randomness gates. We expect that the two injected faults reduce the side-channel security by two orders which is confirmed in Figure 11.7.

## 11.8 Conclusion

In this chapter, we introduce and present the first formal verification framework that can validate side-channel security and fault injection resistance as well as the protection against combined attacks. We demonstrate the functionality and advantages of VERICA in three extensive case studies where we analyze combined gadgets, protection mechanisms against SIFA based on Toffoli gates, and entire S-box implementations according to the ParTI protection scheme.

Additionally, we confirm by practical side-channel measurements that precisely injected faults can decrease the order of the side-channel security. For this, we generate a PRESENT S-box constructed from $(2, 2)$-CINI gadgets and inject persistent faults in random inputs. To determine the security order with respect to the side-channel protection, we perform an evaluation based on the well-established TVLA.

**Limitations.** However, even though VERICA can assist the designer in creating secure hardware implementations of cryptographic primitives, it has some limitations. As shown in our case studies, CA becomes more difficult with an increasing number of gates in the design under test. This is naturally expected since the number of valid fault injections drastically increases (especially for multi-bit fault injections) while for each valid fault injection a separate verification of the side-channel security is conducted (for which the complexity also increases with the




(a) Sample trace.

(b) First-order $t$-test results.

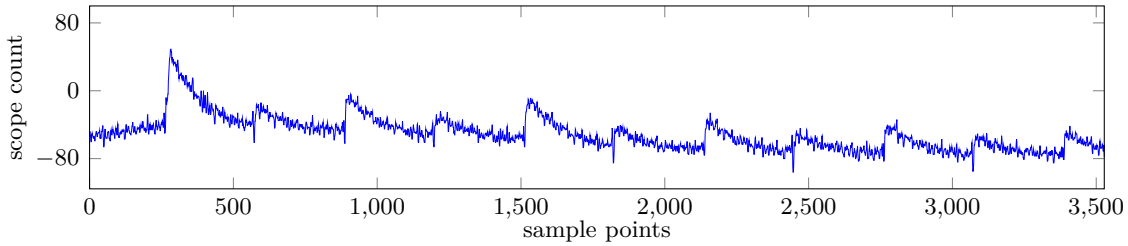
(c) Second-order $t$-test results.

(d) Third-order $t$-test results.

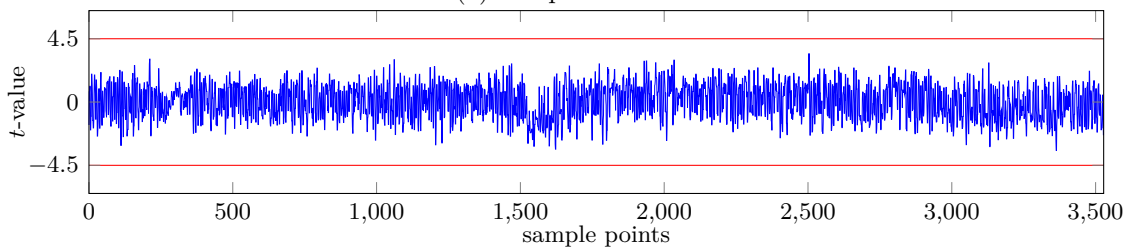
Figure 11.5: Measurement results of a fault free PRESENT S-box generated from $(2, 2)$-CINI gadgets (100 million traces).

number of gates and security order). Note, even more powerful computers (i.e., using more cores and more memory) could not analyze these large circuits since the problem gets too complex. More precisely, the number of valid fault combinations and valid probe combinations increases exponentially with the number of gates and the corresponding order (i.e., with the number of simultaneously injected faults and the probing threshold, respectively).

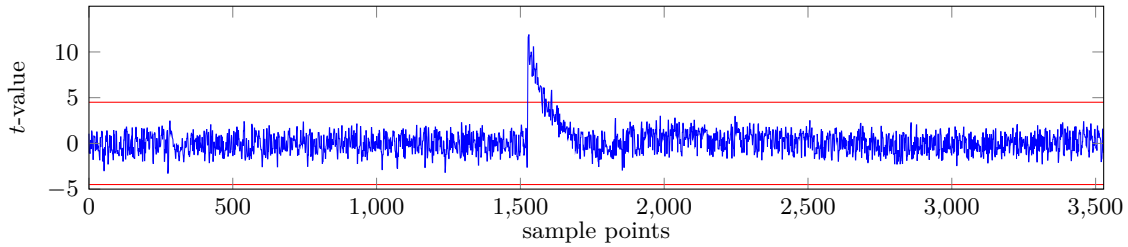
**Correctness of VERICA.** VERICA relies on the theoretical foundation of the security notions and their corresponding proofs presented in Chapter 9. We transferred these notions to software



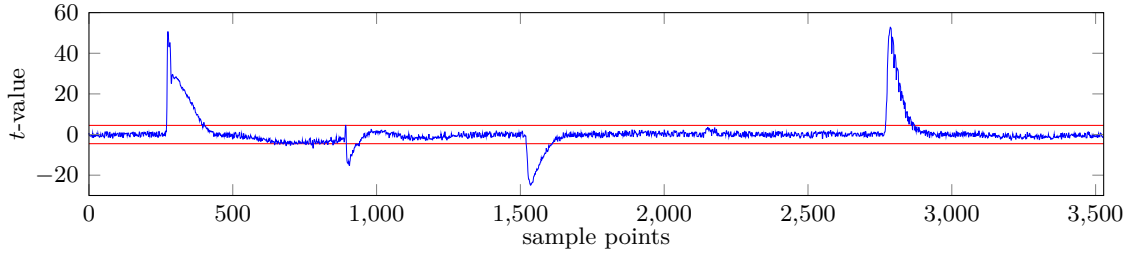

(a) Sample trace.




(b) First-order $t$-test results.



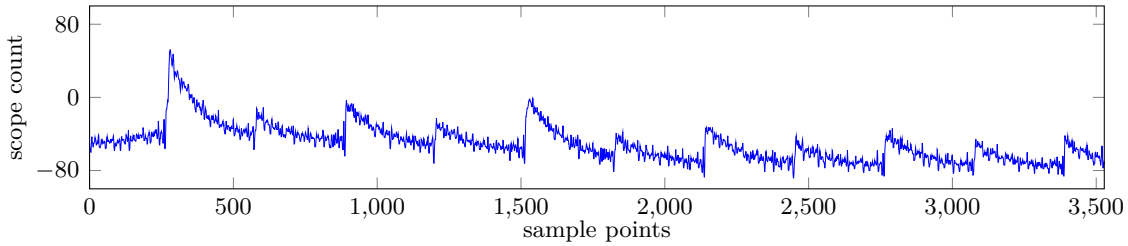

(c) Second-order $t$-test results.



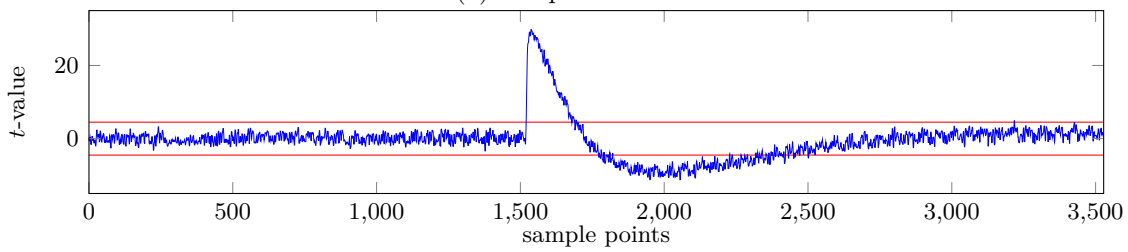

(d) Third-order $t$-test results.

Figure 11.6: Measurement results of a PRESENT S-box generated from $(2, 2)$-CINI gadgets with 1-bit fault injection (100 million traces).

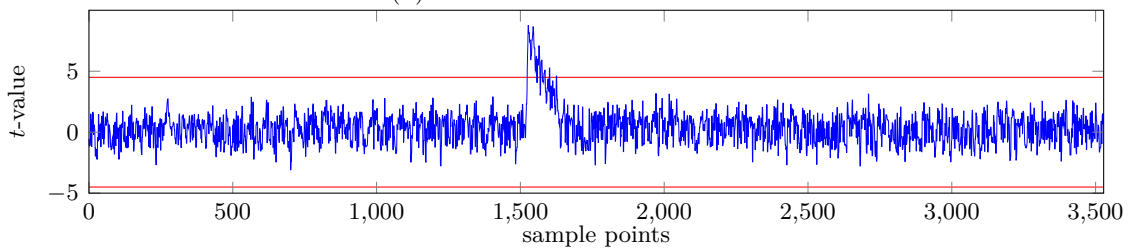
and verified them by (smaller) hand-verified examples. These hand-verified examples are further used in test strategies to ensure the correct functionality of different methods used in VERICA. However, we cannot fully guarantee the correctness of our source code due to the huge size of the project. Therefore, we additionally rely on the scrutiny of the community by releasing the source code and results of the case studies to ensure correct functionality and implementations.



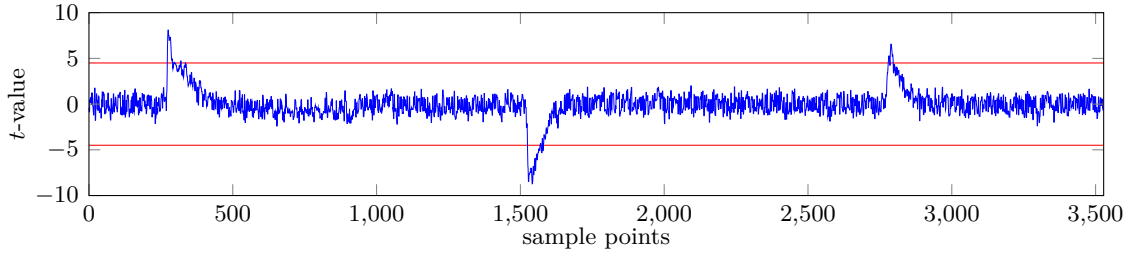

(a) Sample trace.




(b) First-order $t$-test results.




(c) Second-order $t$-test results.




(d) Third-order $t$-test results.

Figure 11.7: Measurement results of a PRESENT S-box generated from $(2, 2)$-CINI gadgets with 2-bit fault injection (1 million traces).

**Future Work.** In our case studies, we reveal some security flaws in the hardware implementations of combined gadgets. Hence, an interesting question is how these gadgets need to be adapted and implemented on hardware such that they resist combined attacks.

Eventually, VERICA reports that the shared implementations of the Toffoli gates from Section 11.6.1 are even first-order secure against SCA in the presence of single-bit and two-bit faults. However, using the shared Toffoli gates to construct larger circuits (e.g., S-boxes), leads to implementations that are not protected against SCA in the presence of faults. This ob-

servation could be used in the future to formulate composability notions for gadgets that are protected against SIFA.

# Part V

# Efficient Hardware Implementations of BIKE

# Chapter 12

# Folding BIKE – Scalable Hardware Implementation for FPGAs

*In this chapter, we investigate different strategies to efficiently implement the BIKE algorithm on FPGAs. To this extent, we improve already existing polynomial multipliers, propose efficient strategies to realize polynomial inversions, and implement the Black-Gray-Flip decoder for the first time. Additionally, our implementation is designed to be scalable and generic with the BIKE specific parameters. Altogether, the fastest designs achieve latencies of* 2.69 ms *for the key generation,* 0.1 ms *for the encapsulation, and* 1.89 ms *for the decapsulation considering the lowest security level. This chapter is based on a joint work with Johannes Mono and Tim Güneysu originally presented in [RMG22].*

**Contents of this Chapter**



## 12.1 Introduction

Since the introduction of Shor's algorithms [Sho99], there is extensive research to find new schemes which are secure even in the presence of quantum adversaries. One such promising research area is code-based cryptography where hard problems from coding theory are used to create cryptographic schemes. The first scheme based on linear error codes was proposed by McEliece in 1978 [McE78]. Even though the McEliece cryptosystem is assumed to be secure against classical and quantum-based attacks, one disadvantage is its large public key. In order to decrease the key size (and the corresponding memory requirements and transmission bandwidth), a new class of linear codes was designed, so-called QC-MDPC codes. They were first presented in [MTSB13] and gained more and more attention in recent years due to performance and security features.

In 2017, the NIST announced the Post-Quantum Cryptography Standardization Project aiming to find and standardize suitable PQC schemes. One of the submissions is the BIKE scheme built upon QC-MDPC codes. With the advancement of the submission to the third round, the

BIKE team reduced the number of algorithms proposed in earlier specifications [ABB+19] to one single algorithm, now just called BIKE. The remaining algorithm (called BIKE-2 in earlier submissions) is based on the Niederreiter framework [Nie86] including some tweaks [ABB+20b]. Recently, NIST announced that BIKE proceeds to the fourth round which means that the algorithm is still considered for standardization [oST22].

After the announcement of the PQC Standardization Project, NIST published a list of selection criteria including security, cost and performance as well as algorithm and implementation characteristics on various platforms [AASA+19]. Currently, there is a software reference implementation of BIKE, an optimized software implementation for Intel CPUs [DGK20a], and an efficient microcontroller implementation [BOG19].

### 12.1.1 Related Work

After the introduction of QC-MDPC codes by Misoczki et *al.*, the authors of [HvMG13] were the first researchers who implemented the McEliece cryptosystem with QC-MDPC codes on FPGAs. Besides an exploration of different decoders suited for efficient hardware implementations, they decided to follow a design strategy targeting a high-speed implementation. To this end, they stored all keys and intermediate results directly in the FPGA logic and did not use any external or internal memories.

One year later, von Maurich and Güneysu presented a lightweight implementation of McEliece using QC-MDPC codes [vMG14]. They divided each vector into chunks of 32 bit and processed them separately. This approach incorporated the internal memory of the FPGA to keep the amount of registers as low as possible.

The authors of [HC17] proposed an area time efficient hardware implementation for QC-MDPC codes outperforming the results from [HvMG13]. The improvements were mainly gained by a custom-designed decoder equipped with a hardware module estimating the Hamming weight of larger vectors.

With the submission to the second round, the BIKE team presented an FPGA implementation of one of the discarded algorithms called BIKE-1 including the key generation and encapsulation [ABB+19]. Their design strategy was very similar to the one presented in [vMG14] but included two optimization levels which parallelized the encoding process.

Recently, Reinders et *al.* proposed an efficient hardware design with a constant-time decoder, also designed for the older BIKE-1 algorithm [RMGS20]. However, the proposed decoder differs from the introduced decoder of the current BIKE specification. Additionally, as they opted for BIKE-1, they did not implement any polynomial inversion.

An efficient algorithm to accomplish polynomial inversions was presented in [HGWC15] and is based on the classic Itoh-Tsujii Algorithm (ITA) [IT88]. Here and in many other parts of BIKE, polynomial multiplications are an essential building block that can be realized by different design strategies. Two of them – i.e., a row-by-row strategy and a strategy dividing the vectors into chunks – were described in the above-mentioned works [HvMG13] and [vMG14], respectively. Another strategy was recently introduced by Hu et *al.* in [HWCW19] where the authors decomposed the quasi-cyclic matrix (constructed from one of the polynomials) into sub-matrices achieving an enhanced area-time product.

### 12.1.2 Contribution

We present the first hardware implementation of the entire BIKE algorithm proceeded to the fourth round in the NIST PQC standardization process. The first challenging part is the implementation of the polynomial inversion required for the key generation. We investigate different optimization strategies for hardware platforms which eventually leads to a highly optimized design. The inversion module as well as other parts of BIKE require a polynomial multiplier. We slightly improve the multiplier proposed in [HWCW19] and reduce the overall latency. Additionally, we provide the first hardware implementation of the BGF decoder originally proposed in [DGK20c]. The implementation is constant-time with respect to the processing of secret values (i.e., the operation times of all modules are independent of any secret values) and is thus secure against timing attacks.

By implementing a parameterized design, we can scale our approach down to small devices (resulting in higher latency) or scale it up for low-latency applications (resulting in a bigger implementation). Additionally, we wrote SageMath scripts to achieve a design which is completely generic with respect to all parameters used in BIKE. All HDL files are available at `https://github.com/Chair-for-Security-Engineering/BIKE`.

## 12.2 Efficient Hardware Implementation

In this section, we first state and discuss our design considerations. Afterwards, we present our design strategies for each required submodule to assemble BIKE and discuss our approaches in more detail.

### 12.2.1 Design Considerations

In general, our implementation tries to keep the footprint as small as possible while providing a reasonable throughput. This goal is achieved by storing all polynomials in BRAMs instead of using registers even if that means forgoing the possibility to access all bits of a polynomial at the same time. This strategy drastically reduces the amount of required registers (and consequently slices) because otherwise each polynomial (e.g., the error vectors, the public key, private key, ciphertext) would consume $r$ registers resulting in exploding implementation costs. Nevertheless, we decided to use registers whenever values of $\ell$ bits (e.g., $m$ or $c_1$) need to be stored as spending an entire BRAM would waste hardware resources.

Besides these trade-offs, our implementation is developed to be generic with the BIKE specific parameters in case they need to be adapted (e.g., for security reasons). Additionally, we introduce a scaling parameter $b$ to define the internally applied data bus width affecting the bus width of all BRAMs and the level of parallelization of several submodules. Therefore, all polynomials are divided into chunks of $b$ bits which will be further processed by the required submodules (e. g., multiplier or inversion). By writing $a[i]$, we denote $b$ bits of the polynomial $a$ which are stored at address $i$ where the Least Significant Bit (LSB) $a_0$ of $a$ is stored in the LSB of $a[0]$. In our evaluation, we consider $b \in \mathcal{B} = \{32, 64, 128\}$ as these values are common bus widths and larger values would exceed the available hardware resources on Xilinx's Artix-7 FPGAs[1].

[1]Note that the NIST recommended to use Artix-7 FPGAs.

The generations of $(h_0, h_1)$, $\sigma$, and $m$ require a source of randomness. In our design, we assume that the target device is equipped with an appropriate Random Number Generator (RNG) since the implementation of a secure RNG is out of scope of this work. All modules requiring such randomness have implemented ports which could be connected to an available source of randomness.

Our goal is to comply with the BIKE specification. Thus, we can generate and extract testvectors from the reference implementation and can validate the output of our design.

### 12.2.2 Sampler

**With Predefined Hamming Weight.** The first step in the key generation (cf. Algorithm 2) is to sample the polynomials $(h_0, h_1)$ representing the first part of the secret key. Since both polynomials are defined to have a Hamming weight of $w/2$, they can be sampled in parallel.

The samplers are realized by rejection sampling [DG19] and both expect a $\lceil \log_2(r) \rceil$-bit input $x_{\text{rand},i}$ of fresh randomness every two clock cycles with $i \in \{0, 1\}$. The input $x_{\text{rand},i}$ determines the non-zero positions in the polynomial $h_i$. For the sampler, we decided to fix $b$ to 32 bits. Increasing $b$ would not improve the throughput because for each random input $x_{\text{rand},i}$ only one bit in the target polynomial needs to be adjusted. Hence, working on larger values of $b$ would increase the required hardware resources in terms of reading larger chunks from the memory which need to be processed by the sampler (more details below).

The sampler divides the random input $x_{\text{rand},i}$ into two parts consisting of the lower five bits $x_{\text{pos},i}$ and the remaining upper bits $x_{\text{addr},i}$. Within the first clock cycle of sampling one single bit, the sampler reads the 32-bit chunk of the polynomial $h_i[x_{\text{addr},i}]$. The lower five random bits $x_{\text{pos},i}$ are buffered in registers. In the next clock cycle, these bits are used to create a bit vector determined by $2^{x_{\text{pos},i}}$ (target bit position is set to one). The vector is added (xored) to $h_i[x_{\text{addr},i}]$ and the result is written back to the memory. If a bit is set and $x_{\text{rand},i} < r$, a counter, which monitors the Hamming weight of the sampled polynomial, is enabled. Given that, increasing $b$ would not improve the throughput but instead, more hardware resources would be necessary (more xor-gates) to adjust a single bit in $h_i$.

Although rejection sampling avoids biased values obtained by e.g., reducing $x_{\text{rand},i}$ modulo $r$, it does not finish in constant time. Therefore, we will briefly discuss its (timing) side-channel security and its average latency. Each time $x_{\text{rand},i}$ is larger than $r$ the randomness is rejected and not used to set a bit in $h_i$. This, however, does not create an attack surface because the algorithm finishes in constant time with respect to the set bit positions in the polynomial. An attacker observing the sampling process would not gain any information about the actual sampled bit position in $h_i$ and no confidential information is revealed [DG19].

However, just recently, Guo et *al.* presented an attack that targets the non-constant time behavior of rejection sampling [GHJ+22]. This attack does not exploit information of the rejection sampling in the key generation but targets a sampling routine used in the encapsulation and decapsulation. To this end, the corresponding sample routines were adapted in the latest specifications of BIKE [ABB+22].

The probability of not getting rejected, i.e., the success probability is $s = \frac{r}{2^{\lceil \log(r) \rceil}}$. However, this term needs to be adjusted as collisions get more likely with an increased number of bits already set in $h_i$ which is done by $(1 - \frac{j-1}{r})$ where $j$ indicates the number of bits that already have been set. Finally, Equation 12.1 is used to calculate the average clock cycles $N_{\text{sample,avg}}$

$$
\begin{aligned}
c_0 &= m_0 \cdot h_0 + m_1 \cdot h_9 + m_2 \cdot h_8 + m_3 \cdot h_7 + ... \\
c_1 &= m_0 \cdot h_1 + m_1 \cdot h_0 + m_2 \cdot h_9 + m_3 \cdot h_8 + ... \\
c_2 &= m_0 \cdot h_2 + m_1 \cdot h_1 + m_2 \cdot h_0 + m_3 \cdot h_9 + ... \\
\hline
c_3 &= m_0 \cdot h_3 + m_1 \cdot h_2 + m_2 \cdot h_1 + m_3 \cdot h_0 + ... \\
c_4 &= m_0 \cdot h_4 + m_1 \cdot h_3 + m_2 \cdot h_2 + m_3 \cdot h_1 + ... \\
c_5 &= m_0 \cdot h_5 + m_1 \cdot h_4 + m_2 \cdot h_3 + m_3 \cdot h_2 + ... \\
\hline
c_6 &= m_0 \cdot h_6 + m_1 \cdot h_5 + m_2 \cdot h_4 + m_3 \cdot h_3 + ... \\
c_7 &= m_0 \cdot h_7 + m_1 \cdot h_6 + m_2 \cdot h_5 + m_3 \cdot h_4 + ... \\
c_8 &= m_0 \cdot h_8 + m_1 \cdot h_7 + m_2 \cdot h_6 + m_3 \cdot h_5 + ... \\
\hline
c_9 &= m_0 \cdot h_9 + m_1 \cdot h_8 + m_2 \cdot h_7 + m_3 \cdot h_6 + ...
\end{aligned}
$$

Figure 12.1: Exemplary decomposition of the partial products for a multiplication with $r = 10$ and $b = 3$.

required to finish the sampling process for the polynomials $h_i$. The leading factor of two is due to the read and write accesses to the BRAM mentioned above.

$$N_{\text{sample,avg}} = 2 \cdot \sum_{j=1}^{w/2} \frac{1}{s \cdot (1 - \frac{j-1}{r})} \tag{12.1}$$

For the lowest security level $N_{\text{sample,avg}} = 189.34$.

**Uniform Sampler.** The sampling process of the secret key $\sigma$ and $m$ is done in a straightforward way by using a 32-bit input providing fresh randomness. The 256 random bits are stored in registers as explained in Section 12.2.1.

### 12.2.3 Multiplication

Polynomial multiplication is a basic building block for each of the three algorithms involved in BIKE. Our multiplier focuses on minimal BRAM usage as well as a good area-time product and is formally defined in the appendix in Algorithm 15 using the vector-matrix representation. Although the runtime of our multiplication is $\mathcal{O}(\lceil r/b \rceil^2)$, we benefit from carry-less and reduction-less multiplication in $\mathbb{F}_2$.

We also considered using Karatsuba multiplication and reviewed the literature for implementations. The authors in [ZGF20] provide one such implementation but due to the high area costs (cf. Section 12.3.3 for more details) we do not follow their approach. Instead, we compute columns block-wise which fits well with our design philosophy of processing $b$ bits in parallel and integrates well with other components.

A multiplication $c = m \cdot h$ requires the constant overhang $O = r \bmod b$ (cf. Algorithm 15), that is the number of bits in the polynomial's most significant word. The multiplier reads $b$ bits of $m$ and $b$ bits of $h$ such that $b \cdot b$ partial products are computed at the same time. This leads to the previously mentioned column-wise multiplication, i.e., all partial products including the message's bits $m[i]$ are calculated before the next $b$ bits of $m$ are read from the BRAM.

As an example, we graphically depict the multiplication process for $r = 10$ and $b = 3$ in Figure 12.1. For every column consisting of $r \cdot b$ partial products, there are two initial steps: the first step computes the partial products of the upper triangle (in our example $m_2 \cdot h_8$), the

**Algorithm 11** Inversion based on the classic ITA [IT88, HGWC15].

**Require:** $r-2=(r_{q-1},...,r_0)$ with $r_i \in 0,1$ and $a \in \mathcal{R}^*$
**Ensure:** $a^{-1}$
1: $f \leftarrow a$, $t \leftarrow 1$
2: **for** $i \leftarrow q-2$ **to** $0$ **do**
3: $g \leftarrow f^{2^t}$
4: $f \leftarrow f \cdot g$
5: $t \leftarrow 2t$
6: **if** $r_i = 1$ **then** $g \leftarrow f^2$ $f \leftarrow a \cdot g$ $t \leftarrow t+1$
7: **end if**
8: **end for**
9: **return** $f^2$

second step computes all partial products that include the current most $O$ significant bits of $h$ and all bits from $m[i]$ excluding the first bit (in our example $m_1 \cdot h_9$ and $m_2 \cdot h_9$).

Afterwards, the algorithm proceeds with a regular flow. In each clock cycle, the multiplier reads $h[j]$ and $c[j]$ from the BRAMs and computes the related partial products in the next clock cycle (illustrated by connected background colors). The lower $b$ bits of the result are added to the intermediate result which was gained by the upper $b-1$ bits of the previous multiplication's result. These intermediate results are stored in registers in order to have direct access.

As the authors in [vMG14], we also use the *read-first* setting of the BRAMs enabling to read a result and write a new value to a specific address in one clock cycles. Hence, new results from the multiplication engine, which are added to the current intermediate result $c[j]$, are stored in the BRAM at position $(j+1) \bmod r$. Since there are $\lceil r/b \rceil$ columns, the final result $c$ is stored in the correct layout, i.e., $c[0]$ contains the LSBs of the final polynomial. The polynomial $h$ is also rotated in the BRAM. This is tracked in the implementation including special cases such as determining $h[0]$ as it consists partly of $h[r-1]$ and partly of $h[r-2]$ (in our example $h[0]=(h_7,h_8,h_9)$ for the second column).

The multiplier performs a multiplication within $\lceil r/b \rceil \cdot (\lceil r/b \rceil + 3) + 1$ clock cycles. The additional three clock cycles in every column originate from the two initial steps described above and one additional clock cycle to read $h[0]$. The last clock cycle is required to switch to a `DONE` state.

### 12.2.4 Inversion

With the decision of the BIKE team to only rely on the BIKE version being built upon the Niederreiter framework, a new challenge of implementing a polynomial inversion in hardware arose. Since BIKE is also designed to work with ephemeral keys, an efficient implementation of an inversion algorithm is even more critical to achieve reasonable throughput. To this end, we decided to implement the inversion of a polynomial $a$ in $\mathcal{R}$ using Fermat's Little Theorem as

$$a^{-1} = a^{2^{r-1}-2} \tag{12.2}$$

holds for every $a \in \mathcal{R}^*$ with $\mathrm{ord}(a) \mid 2^{r-1}-2$.

To exponentiate a target polynomial $a$ with $2^{r-1}-2$, we first rewrite the exponent as $2\,(2^{r-2}-1)$. Eventually, the exponentiation is accomplished by Algorithm 11 which is based

Table 12.1: Comparison between Algorithm 11 and Algorithm 2 from [DGK20b] indicating the amount of squaring operations.

| $\mathcal{K}$ | $k=1$ | $k=2$ | $k=3$ | $k=4$ | Sum |
|---|---|---|---|---|---|
| *Algorithm 2 from [DGK20b]* | | | | | |
| $\{1\}$ | 12 355 | 0 | 0 | 0 | 12 355 |
| $\{1,2\}$ | 5 | 6 175 | 0 | 0 | 6 180 |
| $\{1,2,3\}$ | 10 | 6 | 4 111 | 0 | 4 127 |
| $\{1,2,3,4\}$ | 5 | 1 | 0 | 3 087 | 3 093 |
| *Algorithm 11 (used in **this chapter**)* | | | | | |
| $\{1\}$ | 12 321 | 0 | 0 | 0 | 12 321 |
| $\{1,2\}$ | 7 | 6 157 | 0 | 0 | 6 164 |
| $\{1,2,3\}$ | 8 | 2 | 4 103 | 0 | 4 113 |
| $\{1,2,3,4\}$ | 6 | 2 | 1 | 3 077 | 3 086 |

on the classic ITA [IT88] and a slightly adapted version of Algorithm 1 defined in [HGWC15]. Note that we do not follow the recently proposed algorithm by Drucker et *al.* [DGK20b] (which is used in the additional software implementation of BIKE [DGK20a]) as it performs slightly worse in hardware. The number of required multiplications is the same for both algorithms but the number of squarings differs. Assuming that an exponentiation $f^{2^t}$ is divided into a chain of operations of the form $f^{2^k}$ with $k \in \mathcal{K}$ and $k \leq t$ where each operation has the same runtime (more details are given below), Algorithm 11 requires less of these operations than Algorithm 2 from [DGK20b] as shown in Table 12.1 for different sets of $\mathcal{K}$. Additionally, the proposed algorithm by Drucker et *al.* would require one additional BRAM to hold the intermediate results $res$ (cf. Algorithm 2, line 8 in [DGK20b]).

However, Algorithm 11 executes the exponentiation of $(2^{r-2}-1)$ described by lines 2-11 first and eventually the final squaring from line 12. To this end, the inversion consists of exponentiations of the form $f^{2^t}$, of polynomial squarings, and of polynomial multiplications. The latter operation is realized by using the multiplier described in Section 12.2.3. The strategies to implement a squaring module and to realize the exponentiation with $2^t$ are described in the following.

**Squaring Module for Fixed k.** An exponentiation of a polynomial $f$ with $2^t$ for arbitrary $t$ can always be accomplished by dividing the exponentiation into a chain of $t$ squarings. One possibility to speed up the calculation is to implement a module which performs $k < t$ squarings within the same time as a single squaring. A squaring chain would consist of $\lfloor t/k \rfloor$ $k$-squarings and $t \bmod k$ single squarings.

The strategy implementing squaring modules with fixed $k$ pursues our global design consideration to achieve submodules which scales with $b$. A polynomial squaring $g = f^{2^k}$ for arbitrary $k$ can be realized by a simple bit-permutation and is mathematically described by

$$g_i = f_{i \cdot 2^{-k} \bmod r} \tag{12.3}$$

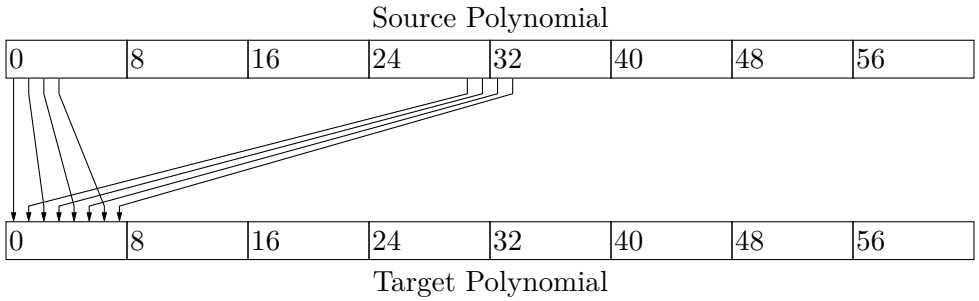



Figure 12.2: Exemplary permutation for a squaring module with $k = 1$, $r = 59$, and $b = 8$.

where $i$ denotes the $i$-th element in the target polynomial. Equation 12.3 indicates that for each $b$ bits of the target polynomial $g$, bits from at least $2^k$ different addresses of the source polynomial $f$ are required where the maximum number of different addresses is bounded to $2 \cdot 2^k - 1$. As an example, Figure 12.2 shows a draft of the permutation and corresponding memory pattern for a squaring with $k = 1$, $b = 8$, and $r = 59$. It is shown that bits from three different addresses are required in order to combine them to the correct result written to the first address implying that all necessary bits from $f$ need to be loaded from the BRAM first. This is done in an initial phase which is automatically calculated to be optimal by our scripts. Additionally, the scripts ensure that all upcoming results can be directly written to the BRAM containing the target polynomial by determining an optimal read sequence of bits from the source polynomial. The amount of clock cycles required for the initial phase also determines the number of $b$-bit registers holding the already read parts from the source polynomial. Note, after the initial phase, which depends on $k$ and $r$, the squaring finishes within $\lceil r/b \rceil$ clock cycles.

**Squaring Module for Arbitrary k.** Besides the above described strategy, we explore another approach implementing a squaring module which can accomplish a $k$-squaring (i.e., $g = f^{2^k}$) for arbitrary $k$ within $r$ clock cycles. For Algorithm 11, this approach is especially interesting for larger $t$ as the exponentiation has not to be decomposed into a squaring chain but rather can directly be carried out. Figure 12.3 shows a schematic drawing of the hardware implementation and the corresponding operations required to compute the addresses of the source and target polynomial and the output data for the target polynomial $g$. The bits of the target polynomial are determined in an ascending order so that the corresponding bits from the source polynomial need to be computed by the implementation. Therefore, the module requires an input `INC` which needs to be assigned to $2^{-k} \bmod r$. Starting with 0, the implementation adds (modulo $r$) every clock cycle `INC` to the current value where the upper bits determine the address and the lower $\log(b)$ bits are used as a selection signal for a $b$-to-1 multiplexer. The input of the multiplexer is the current $b$-bit chunk of the source polynomial. After selecting the desired bit from the input, a barrel shifter is used to shift the desired bit to the correct position. The resulting $b$ bits are then added (xored) to the current intermediate result destined for the target polynomial. After every $b$ bits for a target address of $g$ are collected and shifted to the correct position, the implementation writes the result to the BRAM.

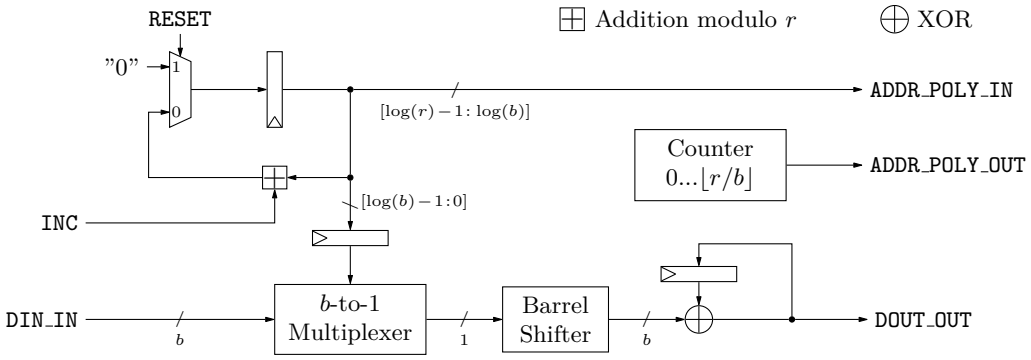
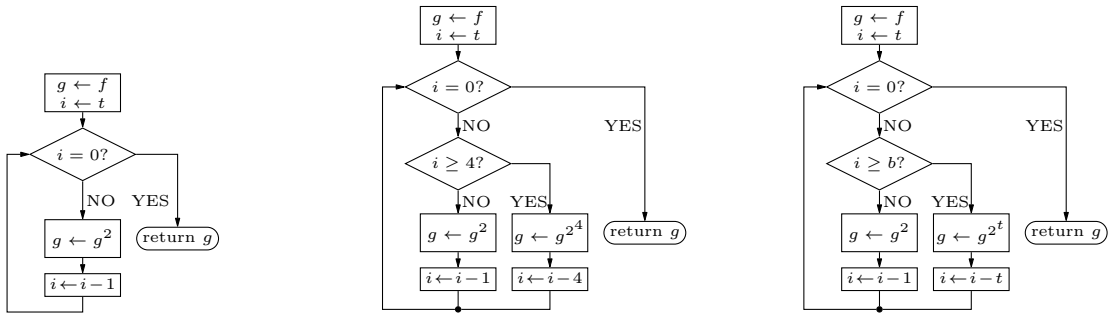



Figure 12.3: Schematic drawing of a $k$-squaring module for arbitrary $k$ in $r$ clock cycles.




(a) Strategy 1. (b) Strategy 2. (c) Strategy 3.

Figure 12.4: Different strategies to implement $g = f^{2^t}$ required for the polynomial inversion.

**Squaring Strategies.** Given the two different modules to compute a $k$-squaring, we investigate three optimization strategies to implement the exponentiation $g = f^{2^t}$ in Algorithm 11, line 3. The three approaches are depicted in flow charts in Figure 12.4. The first strategy only utilizes a squaring module for a fixed $k = 1$. In this case, all exponentiations are carried out by chains of simple squarings. The second strategy implements two different but fixed squaring modules: one with $k = 1$ and the other one with $k = 4$. Hence, as long as $t$ and the remaining exponent of the squaring chain is larger or equal to four, the faster module is used. If the remaining exponent is smaller the squaring module with $k = 1$ is applied. The last strategy uses a combination of a fixed squaring module with $k = 1$ and the module being able to perform arbitrary $k$-squarings. In this way, all $k$-squarings with $k \geq b$ are executed by the latter module.

Note that all strategies have implemented a fixed squaring module with $k = 1$ because of two reasons: (1) simple squarings are always needed in the inversion process (cf. Algorithm 11, line 7 and line 12), and (2) it consumes just a few hardware resources and speeds up the computation notably (more information will be given in Section 12.3.1).

Independently of the strategy, the inversion process requires four BRAMs. One BRAM stores the private key, i.e., $(h_0, h_1)$. The other three BRAM modules are interchangeably used to perform a squaring chain (two BRAMs are used in alternation as source and target polynomial) and a subsequent multiplication by the squaring chain's input polynomial (cf. Algorithm 11, line 3 and line 4).

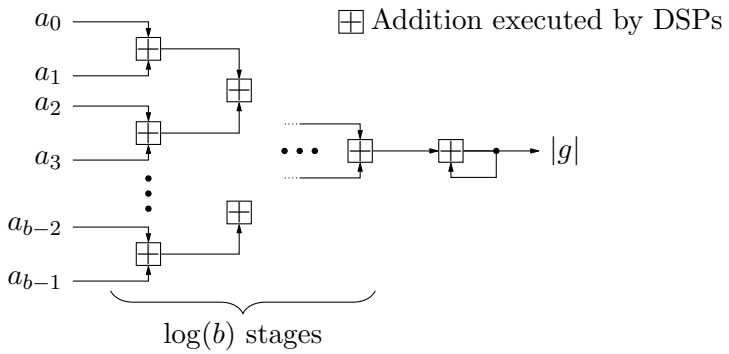



Figure 12.5: Hamming weight computation of a polynomial $g$ divided into $b$-bit chunks $a$. In each stage, as many as possible additions are carried out by one DSP.

### 12.2.5 Decoder

The BGF decoder mainly consists of three submodules. The first module is the threshold function described in Equation 4.1. Its argument $|(s+eH^{\mathrm{T}})|$ is computed by the second module. The third module flips the bits of the error vector $e$ and generates the black and gray lists. In the following, we describe our implementations of these three modules.

**Threshold Function.** The threshold for flipping a bit in the error vector is calculated with a multiplication followed by an addition with a constant term. We use Digital Signal Processors (DSPs) instantiated with an output register stage as a straightforward implementation choice. In order to ensure that the bus widths of the input ports are used as optimally as possible, the corresponding VHDL-code is generated by a Sage script producing binary representations of the constants $f_0$ and $f_1$. The floor function is realized by omitting all fractional digits from the result. As this procedure sustains a loss of precision, the script also checks that the result is still correct for all possible inputs $x$.

**Hamming Weight.** The implementation of the Hamming weight module follows our design strategy to scale submodules with the parameter $b$. Again, we utilize DSPs with one register stage to add up all non-zero bits. To do so, each $b$-bit chunk $a = g[i]$ of a target polynomial $g$ is separately fed into the module depicted in Figure 12.5. In $\log(b)$ stages, all bits are accumulated where each stage consists of $\left\lceil \frac{b/2^j \cdot (j+1)}{b_{\mathrm{DSP}}} \right\rceil$ DSPs where $b_{\mathrm{DSP}}$ denotes the input bit width of the applied DSP and $1 \leq j \leq \log(b)$. Hence, for each stage, the full width of each DSP is utilized. In total, the Hamming weight computation requires

$$1 + \sum_{j=1}^{\log(b)} \left\lceil \frac{b/2^j \cdot (j+1)}{b_{\mathrm{DSP}}} \right\rceil \tag{12.4}$$

DSPs where the additional DSP is used to accumulate all intermediate results at the end.

**Bit-Flipping.** The last module of the decoder is responsible for the bit-flipping of the error vector's bits, i.e., the functions `BFIter` and `BFMIter` from Algorithm 5. In our implementation,

we realize both functions in one module and select the modes of operations (i.e., `BFIter` producing the black and gray lists, `BFIter` without producing the lists, `BFMIter` processing the black mask, and `BFMIter` processing the gray mask) with a control signal `MODE`. The most interesting part is the process of counting the UPC equations which is depicted in Figure 12.6. We follow our design strategy and instantiate $b$ counters in parallel where the `ENABLE` (`EN`) signals depend on the current part of the syndrome and the secret key. For storing the secret key, we decided to rely on a compact representation, i.e., only the positions of non-zero bits are stored instead of the entire polynomial. Hence, to determine the enable signals of all $b$ counters in the same clock cycle, we compute the positions of the currently considered non-zero bit for the next $b-1$ columns (considering the secret key in its matrix representation) by adding the corresponding offsets (white adders) which would be gained when shifting the polynomial to the right. The position of the non-zero bit of the secret key is also used to read the corresponding chunk of the syndrome (depicted at the top in Figure 12.6). Here, we decide to duplicate the syndrome $s$ and store a copy in a separate BRAM. This is necessary since we need $b$ successive bits from $s$ starting at the bit position determined by the current non-zero bit of the secret key which is not aligned with the layout of the BRAMs. For $r = 17$ and $b = 4$ this behavior is shown in the following example.

$$s_2\ s_1\ s_0\ s_{16} \mid s_{15}\ s_{14}\ s_{13}\ s_{12} \mid s_{11}\ s_{10}\ \underline{s_9\ s_8 \mid s_7\ \overset{\downarrow}{s_6}\ s_5\ s_4} \mid s_3\ s_2\ s_1\ s_0$$

The arrow indicates the position of the current non-zero bit of the secret key and the underlined bits are required to determine the enable signals of the $b$ counters. As we can only read one chunk within one clock cycle, we decided to create the aforementioned copy of the BRAM storing the syndrome to achieve a lower latency and read both chunks within one clock cycle from two different memories. The careful reader may notice that the least significant bits of the syndrome in the example are also stored in the most significant chunk such that the chunk is completely filled with data. The least significant bits from $s$ are copied to the most significant chunk in an initial phase each time the `BFIter` module is evoked. This is necessary in case the non-zero bit of the secret key (the arrow in the example) would point for example to $s_{15}$.

After a non-zero bit position is read from the BRAM, $b$ is added and the result is written back to the memory for the next iteration, i.e., the next $b$ columns. At the end of each `BFIter` execution, the original secret key is restored from a copy as it is required for the next execution.

However, after each non-zero bit position is read once from the BRAM, the counter values can be evaluated and compared to the threshold $T$. In case a counter value exceeds $T$ the corresponding bit is set. The resulting $b$ bit vector is added to the current chunk of the error vector or is used to set the bits in the black list. The same procedure is applied for the gray list but with a threshold reduced by $\tau$.

The `BFIter` function finishes in constant time and only depends on $r$, $w$, and $b$ as shown in Equation 12.5.

$$N_{\text{BFIter}} = \frac{w}{2} \cdot 2 \cdot \left\lceil \frac{r}{b} \right\rceil + 6 \cdot 2 \cdot \left\lceil \frac{r}{b} \right\rceil + 5 \tag{12.5}$$

### 12.2.6 Random Oracles

The BIKE specification defines the three functions $\mathbf{H}, \mathbf{K}$, and $\mathbf{L}$ as random oracles [ABB+20b]. $\mathbf{K}$ and $\mathbf{L}$ rely on a standard SHA384 core hashing $m$ concatenated with $C$ and hashing $(e_0, e_1)$,

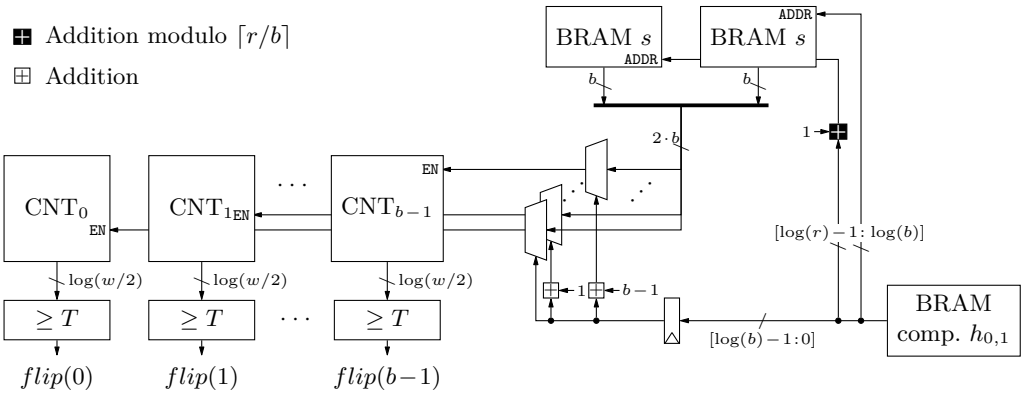



Figure 12.6: Extract of the bit-flipping module.

respectively. It is assumed that all data is stored in byte arrays so that the input size to the SHA function is a multiple of eight. For our hardware design, we implemented the SHA core in a straightforward way, i.e., as a round-based approach including retiming.

The **H** function relies on an AES256 core (instantiated in counter mode) where the input to **H** serves as 256-bit key. After one execution of AES, the resulting ciphertext is used as randomness generating the error vectors. More precisely, the 128-bit output is divided into four 32-bit words which serve as inputs to the sampler described in Section 12.2.2.

## 12.3 Implementation and Analysis

Before we cover the composition of the key generation, encapsulation, and decapsulation, we provide analyses of the above described submodules. Finally, we compare our approaches to related work.

### 12.3.1 Analysis of Submodules

**Sampler.** In order to verify our hardware implementation of the rejection sampler, we performed 100 000 simulations setting $r = 12\,323$. Figure 12.7 shows a histogram of the required clock cycles to finish the sampling process. The results confirm the correct functionality of our implemented sampler and show the expected average number of clock cycles which we deduced in Equation 12.1.

One sampler generating a single polynomial consumes 25 slices partitioned into 66 LUTs and 19 registers. For $r = 12\,323$ a half (i.e., a 18 KB) BRAM tile is required to store the polynomial. Our final implementation instantiates two samplers to generate $(h_0, h_1)$ in parallel.

**Multiplier.** Here, we just report the implementation results for the multiplier setting $b = 32$ and $r = 12\,323$ which are summarized in Table 12.2. A more detailed analysis and a comparison to related work are presented in Section 12.3.3.

**Squaring Modules.** In Section 12.2.4 we introduced two different squaring modules. The first module was designed to perform the operation $f^{2^k}$ for a fixed $k$ and a target polynomial $f$ in

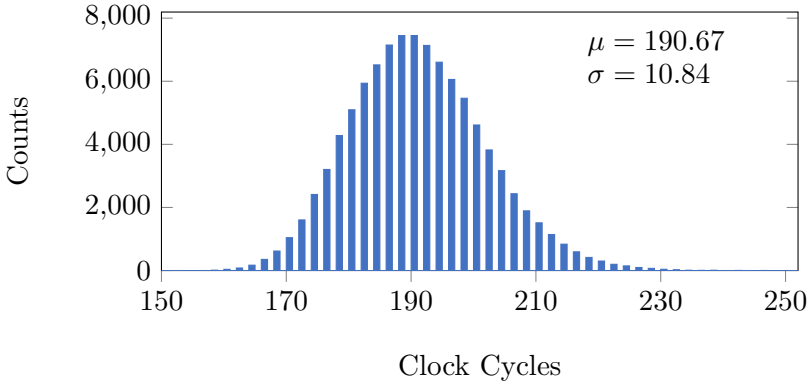



Figure 12.7: Distribution of required clock cycles to sample one polynomial of the secret key for $r = 12\,323$ and $w = 142$ based on 100 000 simulations.

approximately $\lceil r/b \rceil$ clock cycles. The implementation results for $k = 1$ and $k = 4$ are shown in Table 12.2. Increasing $k$ significantly increases the amount of required hardware resources which can be explained by more complex control logic and more intermediate values that need to be buffered in registers. Note that the gain in terms of throughput only increases linearly.

Due to these exploding implementation costs, we investigated a second squaring strategy which performs squarings of arbitrary $k$ in approximately $r$ clock cycles. For $r = 12\,323$ and $b = 32$ this approach requires just 45 slices partitioned into 96 LUTs and 80 registers. The utilization is very similar to that of the squaring module working with a fixed $k = 1$ which makes it especially beneficial for larger $k$.

However, both modules require two 18 KB BRAM tiles which hold the source and the target polynomial.

**Decoder.** The decoder can be divided into three parts: the threshold computation, the Hamming weight module, and the `BFIter` function. The threshold computation is realized by one DSP configured as a multiplier with a subsequent addition. Therefore, it consumes one DSP (independent of the security level and $b$) and a few LUTs for control logic.

The Hamming weight module also uses DSPs as described in Section 12.2.5 while the number of required DSPs depends on $b$ (cf. Equation 12.4). Note, for Artix-7 FPGAs $b_{\text{DSP}} = 28$. No additional logic is required.

The hardware utilization for the `BFIter` function for $r = 12\,323$ and $b = 32$ adds up to 355 slices composed of 280 registers and 1 125 LUTs. Altogether, the module needs to be connected to 4.5 BRAMs to store two times the syndrome, the compact representation of the secret key (a half memory is sufficient), the error vectors, and the black and gray lists.

**Random Oracles.** Both, **K** and **L**, use a SHA384 which consumes 1 171 slices (3 636 LUTs and 2 110 registers). The wrapper to realize **K** consumes additional 114 slices while the wrapper for **L** only requires 45 additional slices.

The realization of **H** utilizes additional 614 slices which includes the AES256 and the wrapper logic.

Table 12.2: Implementation results of the required submodules to assemble the BIKE algorithms ($r = 12\,323$, $b = 32$).

| | **Logic** | | **Memory** | | **Area** |
|---|---|---|---|---|---|
| | LUT | DSP | FF | BRAM | Slices |
| *Sampler* | 66 | 0 | 19 | 0.5 | 25 |
| *Multiplier* | 886 | 0 | 119 | 1.5 | 274 |
| *Squaring* $k = 1$ | 81 | 0 | 105 | 1 | 38 |
| *Squaring* $k = 4$ | 4 070 | 0 | 820 | 1 | 1 124 |
| *Squaring arbitrary* | 96 | 0 | 80 | 1 | 45 |
| *Threshold Function* | 6 | 1 | 0 | 0 | 5 |
| *Hamming Weight* | 0 | 6 | 0 | 0 | 0 |
| *Bit-Flipping* | 1 125 | 0 | 280 | 4.5 | 355 |
| *SHA384* | 3 636 | 0 | 2 110 | 0 | 1 171 |
| *Wrapper for* **K** | 220 | 0 | 29 | 0 | 114 |
| *Wrapper for* **L** | 45 | 0 | 22 | 0 | 45 |
| **H** *Function* | 1 879 | 0 | 457 | 0 | 614 |

Comparing these implementation results with those of the other submodules in Table 12.2, it can clearly be seen that the hardware resources to realize the three random oracles dominate the total utilization costs (especially for $b = 32$). Hence, from the hardware implementation's point of view switching to another cryptographic primitive like Keccak (used as SHAKE and SHA-3) could reduce this overhead.

### 12.3.2 Composed Key Encapsulation Mechanism

Now, we present implementation results of the composed designs of the three algorithms involved in BIKE.

**Key Generation.** Given all the submodules, we now describe the assembly of the key generation module. On the top level, it consists of two samplers generating the private key $(h_0, h_1)$. The resulting key is written to a generic BRAM module which automatically picks and connects the minimum number of required BRAM tiles based on the selected parameters $r$ and $b$. The private key $\sigma$ is generated by the sampler described in Section 12.2.2 and is stored in a 256-bit register. In order to generate the public key $h = h_1 h_0^{-1}$, one of the above introduced inversion modules is instantiated. The multiplication is also performed inside the inversion module as it already contains a multiplication engine.

Table 12.3 summarizes the implementation results for the key generation for all three introduced design strategies. Starting with Strategy 1, which utilizes only one squaring module, the implementation requires in average for $b = 32$ 7.37 million clock cycles[2] which corresponds to a

[2]The average number of clock cycles was determined by performing a simulation and applying Equation 12.1.

Table 12.3: Implementation results for Level 1 ($r = 12\,323$).

| | Resources | | | | | Performance | | |
|---|---|---|---|---|---|---|---|---|
| | Logic | | Memory | | Area | Cycles | Freq. | Latency |
| | LUT | DSP | FF | BRAM | Slices | Cycles | MHz | ms |
| **Key Generation** | | | | | | | | |
| *Strategy 1* | | | | | | | | |
| 32 bit | 2 092 | 0 | 589 | 4 | 669 | 7 370 429 | 129.87 | 56.75 |
| 64 bit | 3 607 | 0 | 631 | 5 | 1 046 | 3 070 613 | 125 | 24.56 |
| 128 bit | 11 838 | 0 | 861 | 10 | 3 354 | 1 409 621 | 104 | 13.53 |
| *Strategy 2* | | | | | | | | |
| 32 bit | 6 982 | 0 | 1 396 | 4 | 1 986 | 3 804 192 | 131.58 | 28.91 |
| 64 bit | 9 140 | 0 | 2 303 | 5 | 2 570 | 1 295 190 | 123.46 | 10.49 |
| 128 bit | 23 801 | 0 | 4 567 | 10 | 6 742 | 520 374 | 106.38 | 4.89 |
| *Strategy 3* | | | | | | | | |
| 32 bit | 2 074 | 0 | 659 | 4 | 649 | 2 671 076 | 131.58 | 20.30 |
| 64 bit | 4 432 | 0 | 735 | 5 | 1 285 | 748 964 | 113.64 | 6.59 |
| 128 bit | 12 654 | 0 | 1 044 | 10 | 3 554 | 258 750 | 96.15 | 2.69 |
| **Encapsulation** | | | | | | | | |
| 32 bit | 6 730 | 0 | 3 298 | 3 | 2 143 | 152 694 | 121.95 | 1.25 |
| 64 bit | 8 253 | 0 | 3 327 | 5 | 2 538 | 40 368 | 121.95 | 0.33 |
| 128 bit | 14 829 | 0 | 3 471 | 10 | 4 540 | 12 240 | 121.95 | 0.10 |
| **Decapsulation** | | | | | | | | |
| 32 bit | 9 380 | 7 | 3 943 | 10 | 2 971 | 1 626 674 | 125 | 13.01 |
| 64 bit | 16 140 | 9 | 4 307 | 15 | 4 942 | 518 105 | 116.28 | 4.46 |
| 128 bit | 30 430 | 13 | 5 063 | 29 | 8 785 | 188 646 | 100 | 1.89 |

latency of 56.75 ms for a maximum possible frequency of 129.87 MHz. The latency can roughly be decreased by a factor of four setting $b = 128$. However, the hardware utilization scales with a factor of five resulting in an area footprint of 3 354 slices. A better ratio between latency and resource utilization is achieved with Strategy 3. The utilization is very similar to the first strategy but the latency is notably decreased so that the implementation for $b = 128$ requires just 2.69 ms to finish one key generation by consuming 3 554 slices and 10 BRAMs. Hence, a distinct superiority is clearly visible.

**Encapsulation.** Figure 12.8 shows a schematic of the encapsulation. To sample and store $m$, an uniform sampler and a 256-bit register is instantiated. The message $m$ is used as input to $\mathbf{H}$ generating the error vector $e = (e_0, e_1)$. Afterwards, $c_0 = e_0 + e_1 h$ and $c_1 = m \oplus \mathbf{L}(e_0, e_1)$

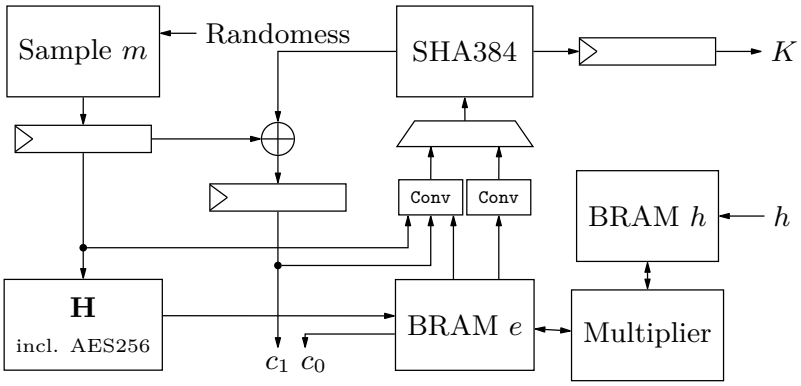



Figure 12.8: Top level view of the encapsulation module.

are computed in parallel. A parallel computation is only possible due to an additional BRAM which is placed in the conversion module and stores a copy of $e$ serving as input to $\mathbf{L}$. The final result of the multiplication is stored in the part of the BRAM which initially holds $e_0$. The cryptogram and the message $m$ are fed into a conversion module `Conv` to generate the input to the SHA core realizing $\mathbf{K}$.

Again, Table 12.3 summarizes the implementation results for the encapsulation module for $b \in \mathcal{B}$. Since the main part of the encapsulation is the multiplication to generate $c_0$, the implementation perfectly scales with $b$. For $b = 32$ the design requires 3 BRAMs and 2 133 slices while performing one encapsulation within 1.25 ms. Switching to $b = 128$, increases the hardware utilization roughly by a factor of two while the latency is decreased by a factor of twelve. The small increase of the hardware utilization originates from the relatively large footprints of the SHA384 and the AES256 which stay constant for each $b$. Both modules consume together roughly 1 800 slices (cf. Table 12.2) which are 83 % of the whole design when setting $b = 32$.

**Decapsulation.** The decapsulation uses most of the submodules including two multipliers, the decoder, and all three random oracles (see Figure 12.9). After transmitting the private key $(h_0, h_1, \sigma)$ and the ciphertext $C = (c_0, c_1)$, the decapsulation is started by computing the syndrome $s = c_0 h_0$. Afterwards, the algorithm invokes the decoder enabling the `BFIter` module which forms the center of the decapsulation. After each iteration, the two multipliers compute the updated syndrome by $s' = e''_0 \cdot h_0 + e''_1 \cdot h_1 + s$ where $s$ is the initial syndrome determined in the first step. Next, the content of $e''$ is converted, forwarded to the SHA core, and added to $c_1$. The resulting message $m'$ serves as key for the AES256 core generating an error vector which is stored in $e'$ and compared to the content in $e''$. In case the polynomials are equal, the implementation forwards $m'$ to $\mathbf{K}$ determining the shared key $k$. Otherwise, $\sigma$ is used as input to $\mathbf{K}$.

Again, Table 12.3 summarizes the implementation results for the decapsulation. The main parts, i.e., the multipliers and the bit-flipping module, perfectly scale with the parameter $b$. Hence, increasing $b$ from 32 bits to 128 bits lowers the latency from 13.02 ms to 1.89 ms by spending roughly three times more hardware recourses.

The presented results for the key generation, encapsulation, and decapsulation are for

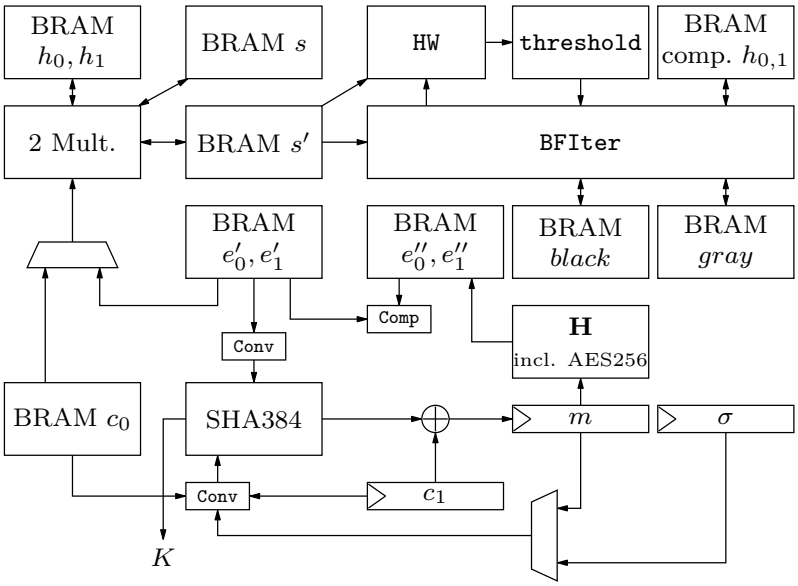



Figure 12.9: Top level view of the decapsulation module.

the level 1 security level, i.e., for a polynomial length of $r = 12\,323$. Additionally, we present implementation results for the third security level ($r = 24\,659$) in Table 12.4.

### 12.3.3 Comparison to Related Work and Discussion

In all three algorithms, the multiplier represents an important part. Therefore, we first compare our multiplier to designs from the literature. Afterwards, we provide a comparison to other code-based PQC schemes and briefly discuss the advantages and disadvantages of BIKE.

**Multiplier.** In Table 12.5, we first compare our approach for the multiplication with the Karatsuba implementation from [ZGF20] to reason the choice of our design. Note that the corresponding results are generated for $r = 24\,533$ as it is a valid polynomial size for LEDAcrypt used as case study in [ZGF20] and is very similar to the parameter set for the third security level of BIKE. Hence, we synthesized our multiplier for the same $r$ in order to allow a fair comparison. Our design achieves a better time-area product while consuming considerably less BRAMs. As one design target of our work is to implement BIKE also for low-cost FPGAs, we decided to use the multiplier design presented in Section 12.2.3.

In the second part of Table 12.5, we compare our multiplier to the recently proposed design by Hu et *al.* [HWCW19] whose implementation conducts a multiplication within $\lceil \frac{r}{b} \rceil^2 + 18\lceil \frac{r}{b} \rceil - 9$ clock cycles. Our multiplier achieves a latency of $\lceil \frac{r}{b} \rceil^2 + 3\lceil \frac{r}{b} \rceil + 1$ clock cycles with a slightly decreased linear part. Additionally, we included the design from the Round-2 submission of the BIKE specifications [ABB+19].

These results were generated for $r = 10\,163$ since Hu et *al.* reported their results for the parameter set of the second round submission of BIKE. While our implementation consumes slightly more hardware resources, the latency clearly decreases. However, the area-time product only shows better results for $b = 32$ and $b = 64$. We cannot explain the difference in the uti-

Table 12.4: Implementation results for Level 3 ($r = 24\,659$).

| | Resources | | | | | Performance | | |
|---|---|---|---|---|---|---|---|---|
| | Logic | | Memory | | Area | Cycles | Freq. | Latency |
| | LUT | DSP | FF | BRAM | Slices | Cycles | MHz | ms |
| *Key Generation (Strategy 3)* | | | | | | | | |
| 32 bit | 1 757 | 0 | 628 | 5 | 561 | 11 600 207 | 135.14 | 85.84 |
| 64 bit | 4 580 | 0 | 801 | 5 | 1 303 | 3 089 329 | 111.11 | 27.80 |
| 128 bit | 12 193 | 0 | 970 | 10 | 3 491 | 930 179 | 96.15 | 9.67 |
| *Encapsulation* | | | | | | | | |
| 32 bit | 6 436 | 0 | 3 305 | 5 | 1 982 | 601 099 | 121.95 | 4.93 |
| 64 bit | 8 329 | 0 | 3 366 | 5 | 2 508 | 154 499 | 119.05 | 1.30 |
| 128 bit | 15 004 | 0 | 3 441 | 10 | 4 376 | 42 173 | 125 | 0.34 |
| *Decapsulation* | | | | | | | | |
| 32 bit | 8 515 | 7 | 3 978 | 16 | 2 912 | 5 969 105 | 125 | 47.75 |
| 64 bit | 13 424 | 9 | 4 359 | 16 | 4 324 | 1 804 958 | 116.28 | 15.52 |
| 128 bit | 30 635 | 13 | 5 127 | 30 | 9 727 | 609 915 | 96.15 | 6.34 |

lization of slices for $b = 128$. As Hu et *al.* mentioned in their work, the required area increases quadratically with the scaling parameter $b$ [HWCW19, Table IV]. This roughly holds for our design but we cannot explain why Hu et *al.* achieve much better results.

**Complete BIKE Design.** In this paragraph, we compare the complete hardware implementation of BIKE to related works that present hardware designs of code-based cryptography. Recently, Dang et *al.* published a paper comparing round 2 candidates of the NIST PQC standardization process [DFA+20]. The only code-based scheme reported in their work is the Classic McEliece Public-Key Encryption (PKE) scheme whose hardware implementation was originally proposed in [WSN18]. Their design can also be configured and instantiated as a lightweight or high-speed implementation. The corresponding implementation results are listed in Table 12.6 while also showing estimations of a composed BIKE design using our introduced modules. Here, we assume that the AES and SHA cores are only instantiated once on the chip such that the encapsulation and decapsulation share them. Note, that this, however, still results in a very conservative estimation since memory, registers, and the multiplier could be shared as well. Nevertheless, in terms of latency, the Classic McEliece scheme clearly outperforms BIKE for all three operations and for both implementation strategies (lightweight and high-speed). In return, the resource utilization is considerably higher than for BIKE so that the Classic McEliece scheme is not particularly suitable for implementations on low-cost devices. Considering the Artix-7 device family from Xilinx (recommended by the NIST), Classic McEliece could only be implemented on the largest FPGAs (i.e., on XC7A200T devices) due to the high amount of required BRAMs. Even if the huge amount of BRAM is neglected, the design would still

Table 12.5: Comparison between different multipliers on Artix-7 FPGAs.

| | Resources | | | | Performance | | | |
|---|---|---|---|---|---|---|---|---|
| | Logic | Memory | | Area | Cycles | Freq. | Latency | Area-Time |
| $b$ [bit] | LUT | FF | BRAM | Slices | Cycles | MHz | ms | Slices × ms |
| *Karatsuba [ZGF20]* [a] | | | | | | | | |
| 64 | 67 300 | 13 440 | 165 | 16 825 | 5 715 | 143 | 0.04 | 673 |
| ***This work*** [a] | | | | | | | | |
| 64 | 2 377 | 152 | 3 | 704 | 148 609 | 163 | 0.565 | 397.76 |
| *Round-2 Implementation [ABB+19]* [b] | | | | | | | | |
| 32 | 87 | 53 | 3 | 40 | 3 252 161 | 416 | 7.818 | 312.72 |
| *Multiplier by Hu et al. [HWCW19]* [b] | | | | | | | | |
| 32 | N/A | N/A | 2.5 | 219 | 106 839 | 205 | 0.521 | 114.099 |
| 64 | N/A | N/A | 5 | 654 | 28 134 | 180 | 0.156 | 102.024 |
| 128 | N/A | N/A | 7.5 | 1 596 | 7 831 | 150 | 0.052 | 82.992 |
| ***This work*** [b] | | | | | | | | |
| 32 | 886 | 90 | 1.5 | 274 | 102 079 | 312 | 0.327 | 89.598 |
| 64 | 2 384 | 119 | 3 | 740 | 25 759 | 277 | 0.093 | 68.82 |
| 128 | 8 864 | 248 | 6 | 2 519 | 6 641 | 147 | 0.045 | 113.355 |

[a] $r = 24\,533$ [b] $r = 10\,163$

require a XC7A50T or XC7A200T for the lightweight and high-speed versions, respectively. In comparison, our lightweight design can be instantiated on a low-cost XC7A35T device while the high-speed design requires a XC7A100T FPGA. At the time of writing this article, a XC7A200T FPGA costs around 196 $ while a low-cost XC7A35T device can be purchased for roughly 35 $. This makes our design also suitable for low-cost applications.

In Table 12.6 we additionally compare our design to the key generation approach from [HWCW19] which was designed for an old parameter set with $r = 10\,163$. Even though our design uses a slightly larger $r$, it clearly outperforms the implementation by Hu et *al.*. Setting $b = 64$, our key generation implementation consumes roughly the same amount of slices but is as twice as fast (cf. Table 12.3).

Note, that we do not compare our hardware design to the implementation reported in the Round-2 submission of BIKE as it was based on the older algorithm BIKE-1.

### 12.3.4 Discussion

In case a hardware implementation of BIKE does not have to perform the key generation, encapsulation, and decapsulation in parallel, a composed design could further be optimized. Besides instantiating the AES and SHA core only once, a shared multiplier, shared register banks and shared BRAMs could be used as well.

Table 12.6: Comparison to other code-based schemes.

| Design | LUT | FF | Slices | DSP | BRAM | Freq.* | KeyGen cycles† | KeyGen $\mu s$ | Enc. cycles† | Enc. $\mu s$ | Dec. cycles† | Dec. $\mu s$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| mceliece348864$^{pke}$ (LW) [WSN18] | 25 327 | 49 383 | 6 332[a] | 0 | 168 | 108 | 1 600 | 14 800 | 2.7 | 25.2 | 18.3 | 169.8 |
| mceliece348864$^{pke}$ (HS) [WSN18] | 81 339 | 132 190 | 16 524[a] | 0 | 236 | 106 | 202.7 | 1 920.3 | 2.7 | 25.8 | 12.7 | 120.7 |
| BIKE-2 [HWCW19] | 3 874 | 2 141 | 1 312 | 0 | 10 | 160 | 2 150 | 13 437 | – | – | – | – |
| **This work (LW)** | 12 868 | 5 354 | 4 078 | 7 | 17 | 121 | 2 671 | 21 903 | 153 | 1 252 | 1 628 | 13 349 |
| **This work (HS)** | 52 967 | 7 035 | 15 187 | 13 | 49 | 96 | 259 | 2 691 | 12 | 127 | 189 | 1 972 |

[pke] Results are only for the PKE and not for the KEM. [LW] Lightweight implementation.
[HS] High-speed implementation.
[*] in MHz. [†] in thousand. [a] Estimation (assuming all slices are completely utilized).

In Section 12.3.1 we already discussed the huge footprints of the random oracles. Hence, the choice of using AES and SHA as underlying building blocks appears not to be optimal for hardware implementations. To this end, we would suggest using other standardized cores like Keccak which could be used as hash function (for **K** and **L**) and as random number generator (for **H**). This should reduce the overall footprint of a BIKE hardware implementation.

## 12.4 Conclusion

In this section, we present a complete hardware implementation of BIKE selected as an alternate candidate in the NIST PQC standardization process. Our implementation is scalable with respect to the used hardware resources and the corresponding latency while performing all operations in constant time (i.e., there is no dependency on secret values). As polynomial multiplications mainly determine the speed of the key generation and encapsulation, we use carry-less vector-matrix-multiplication with a short feedback path. For the key generation, we investigate three different implementation strategies resulting in one outstanding design. Additionally, we propose the first hardware implementation of the BGF decoder required in the decapsulation. With all these improvements and optimizations we are able to implement a key generation that only takes 2.69 ms, an encapsulation that can be accomplished in 0.1 ms, and a decapsulation that finishes in 1.89 ms. Since multiplication is the most important operation with respect to performance, we suggest to investigate other approaches for high-speed implementations in future work.

# Chapter 13

# Racing BIKE – Optimized Hardware Design

*This chapter presents a new and improved FPGA implementation of BIKE. We optimize two key arithmetic operations, which are the sparse polynomial multiplication and the polynomial inversion. Our sparse multiplier achieves time-constancy for sparse polynomials of indefinite Hamming weight used in BIKE's encapsulation. The polynomial inversion is based on the extended Euclidean algorithm, which is unprecedented in current BIKE implementations. Our optimized design results in a* 5.5 *times faster key generation compared to our previous implementations based on Fermat's little theorem.*

*Besides the arithmetic optimizations, we present a united hardware design of BIKE with shared resources and shared sub-modules among KEM functionalities. On Xilinx Artix-7 FPGAs, our lightweight implementation consumes only* 3 777 *slices and performs a key generation, encapsulation, and decapsulation in* 3 797 µs*,* 443 µs*, and* 6 896 µs*, respectively. Our high-speed design requires* 7 332 *slices and performs the three KEM operations in* 1 672 µs*,* 132 µs*, and* 1 892 µs*, respectively. The results presented in this chapter emerged from a collaboration with Ming-Shing Chen, Santosh Ghosh, and Tim Güneysu [RCGG22].*

**Contents of this Chapter**



## 13.1 Introduction

In this chapter, we target to improve the efficiency of the KEM functionalities of BIKE by an FPGA hardware design. Since NIST announced that performance plays an important role in their PQC standardization efforts [NIS20], researchers presented several optimization techniques for BIKE on the suggested platforms including the AVX2 instruction set on x86, embedded microprocessors, and FPGAs. For example, Drucker et *al.* [DGK20a] optimized BIKE for x86 CPUs. Chen et *al.* [CCK21] presented optimization techniques for x86 and Arm Cortex

M4. In the previous chapter, we proposed an optimized scalable hardware implementation for reconfigurable devices. Now, we present new optimization techniques for efficient FPGA implementations of BIKE and report significant improvements compared to previous works.

### 13.1.1 Related Work

Although there were several early works implementing QC-MDPC codes on hardware devices for variants of the McEliece cryptosystem [vMG14, HvMG13] and for the Niederreiter framework [HC17], the first hardware implementation of BIKE was presented with the round-two submission of NIST's PQC standardization process [ABB+19]. The implementation was designed for an older version of BIKE (called BIKE-1) and only supported the key generation and encapsulation.

In 2020, Reinders et *al.* [RMGS20] proposed a complete hardware design which, however, targets the older parameters of BIKE. Besides, they presented an efficient hardware implementation for a novel constant-time decoder.

In Chapter 12, we presented the first complete hardware design of the BIKE version submitted to the third round of the NIST standardization process [ABB+20a]. We implemented for the first time the BGF decoder on hardware, introduced an optimized polynomial inversion module (based on Fermat's little theorem), and proposed a scalable multiplier.

In further detail, BIKE poses several challenges on the arithmetic level. For improving the polynomial multipliers in code-based schemes, Hu et *al.* [HWCW19] presented two different approaches. While the first design is based on a schoolbook multiplication, the second multiplier improves multiplications by exploiting the sparseness of the polynomials used in QC-MDPC codes. Additionally, they instantiated their designs to create a key generation module based on previous parameter sets of BIKE.

Barenghi et *al.* [BFG+19] presented similar approaches to implement polynomial multiplications for the code-based scheme LEDAcrypt [BBC+19a]. They explored different configurations of schoolbook and sparse multipliers for Xilinx FPGAs.

### 13.1.2 Contribution

In this chapter, we revise previous concepts and identify significant improvements and systematic explorations of the hardware implementation of BIKE on FPGAs. Specifically, we introduce an optimized polynomial multiplier that exploits the sparseness of QC-MDPC codes while performing all multiplications applied in BIKE in constant time. In addition to that, we present a novel component for polynomial inversion based on the Extended Euclidean Algorithm (extGCD) accelerating the key generation in hardware. For that we adapt the extGCD from the constant-time algorithm recently proposed by Bernstein and Yang [BY19], and demonstrate that this approach clearly outperforms previous implementations based on Fermat's little theorem in the specific case of BIKE. As a design constraint, our implementation is highly scalable to instantiate specifically tailored cryptographic components for any use case.

Besides these major arithmetic-oriented optimizations, we also substitute symmetric cryptography from encapsulation and decapsulation implementations presented in Chapter 12 with a single Keccak core to demonstrate our assertion of achieving a lower footprint by applying this modification. Additionally, we present a combined hardware implementation of BIKE that

consolidates all three KEM algorithms in one single, united design. This approach enables resource and module sharing between the KEM algorithms achieving a design that reduces the overall implementation costs.

Our implementations are written in Verilog and are publicly available at `https://github.com/Chair-for-Security-Engineering/RacingBIKE`.

## 13.2 Arithmetic Operations

In this section, we summarize important polynomial arithmetic. More precisely, we briefly explain the multiplication of spares polynomials and introduce the polynomial inversion by using the extended euclidean algorithm.

### 13.2.1 Sparse Polynomial Multiplication

In BIKE, all multiplications in $\mathcal{R}$ comprise a sparse operand $f \in \mathcal{R}$ with $|f| \ll r$. For the key generation, $h_1$ is the sparse polynomial in the multiplication $h_1 \cdot h_0^{-1}$. For the encapsulation, $e_1$ is sparse in $e_1 \cdot h$. For the decapsulation, $h_0$ is sparse in $c_0 \cdot h_0$. The `decoder` contains some additional multiplications by the sparse polynomials $(h_0, h_1)$ which are part of the private key.

We represent a sparse polynomial as a set of indexes corresponding to its non-zero terms. For example, the set $I_f = \{i_1, \ldots, i_t\}$ represents the sparse polynomial $f = X^{i_1} + \cdots + X^{i_t}$ with the Hamming weight $|f| = t$. Multiplying a dense polynomial $g$ by the sparse polynomial $f$ simply accumulates $t$ products of multiplications $g \cdot X^i$ for $i \in I_f$. Since $g$ is represented as a bit sequence, multiplication by $X^i$ shifts the bit sequence $i$-bit to the left and modulo by $X^r - 1$ moves the shifted bit segment exceeding the $r$-th bit to the empty bit segment starting from the 0-th bit. In other words, the multiplication simply accumulates $t$ rotated $g$ by $i_1, \ldots, i_t$ bits.

### 13.2.2 Polynomial Inversion with the Extended Euclidean Algorithm

The key generation (cf. Algorithm 2) computes the multiplicative inverse of a secret polynomial $h_0 \in \mathcal{R}$. Previous works, e.g., our implementation from Chapter 12 and [ABB+20a, HWCW19], computed the inversion by raising $h_0$ to the power of $2^{r-1} - 2$ (Fermat's little theorem).

In this chapter, we compute the inversion with the extGCD which takes two input polynomials $(f, g)$ and outputs three polynomials $(\gcd(f, g), u, v)$, where $\gcd(f, g)$ is the greatest common divisor of $f$ and $g$ and $\gcd(f, g) = u \cdot f - v \cdot g$. All polynomials are in $\mathbb{F}_2[X]$ in the context of BIKE.

In a nutshell, we compute $\text{extGCD}(X^r - 1, h_0)$ for the inverse $h_0^{-1}$. Under the parameters of BIKE, the polynomial $X^r - 1$ has two factors $X^r - 1 = (X - 1)(\sum_{i=0}^{r-1} X^i)$. Since $|h_0| = w/2$ is an odd number, $h_0$ is not a multiple of $X - 1$. Since $|h_0| \neq r$, $h_0$ is also not the polynomial $\sum_{i=0}^{r-1} X^i$. Hence, the $\text{extGCD}(X^r - 1, h_0)$ outputs $(1, u, v)$ s.t. $1 = u \cdot (X^r - 1) - v \cdot h_0$, and $v$ is the inverse $h_0^{-1}$ since $v \cdot h_0 \equiv 1 \mod (X^r - 1)$.

However, a traditional extGCD is unsuitable for cryptographic applications because it usually contains branches that depend on the inputs. While the inputs are secret, an attacker can collect information about the inputs through running-time differences. Hence, we have to apply a constant-time extGCD to prevent the leakage of timing side-channel information.

In this chapter, we adopt the constant-time version of the extGCD proposed by Bernstein and Yang [BY19]. In contrast to the traditional extGCD that eliminates the head coefficients of polynomials at any degree, the constant-time extGCD in [BY19] always eliminates the 0-th bit of polynomials. This leads to extra coefficient reversal processes for inputs to move its head coefficient to the 0-th bit position and before output for recovering the polynomial to its original coefficient order.[1] Considering for example an input polynomial $f$, the coefficient process is equivalent to perform the $f' \leftarrow f(1/X) \cdot X^{\deg(f)}$ operation. This operation moves the original head coefficients of $f$ to a new position of degree 0, which is accessed by $f'[0]$. Thus the extGCD always eliminates the head coefficients at the 0-th bit.

**Division Steps and Transition Matrix.** In this chapter, we simplify the extGCD in [BY19] regarding $\mathbb{F}_2[X]$ for the BIKE application. The algorithm consists of a constant number of simple *division steps* (`divstep`s) for the two input polynomials. Define $\texttt{divstep} : \mathbb{Z} \times \mathbb{F}_2[X] \times \mathbb{F}_2[X] \to \mathbb{Z} \times \mathbb{F}_2[X] \times \mathbb{F}_2[X]$ as

$$\texttt{divstep}(\delta, f, g) = \begin{cases} (1-\delta, g, (g(0)f - f(0)g)/X) & \text{if } \delta > 0 \text{ and } g(0) \neq 0, \\ (1+\delta, f, (f(0)g - g(0)f)/X) & \text{otherwise.} \end{cases}$$

Here, $\delta$ means the degree difference between $f$ and $g$. The `divstep` outputs two polynomials. The first polynomial aims for the polynomial of the higher degree among two input polynomials. The other is the result of the subtraction of two polynomials for eliminating one head term, and it adjusts the new head term to the degree-0 coefficient by the division of $X$.

Since the division of $X$ causes negative degrees, we adjust the representation of polynomials to prevent negative degrees. If the polynomial $f$ contains a monomial of negative degree, e.g., $1/X^i$, we will store $f$ as an alternative polynomial $f'$ s.t. $f = f' \cdot (1/X)^i$ and degrees of all monomials of $f'$ are non-negative. For applying `divstep` multiple times, define $(\delta_n, f_n, g_n) = \texttt{divstep}^n(\delta, f, g)$, i.e., applying the `divstep` to inputs $(\delta, f, g)$ for $n$ times.

Bernstein and Yang describe the transition of the two polynomials $(f, g)$ under the `divstep` operation as a matrix-vector multiplication. Let $T(\delta, f, g)$ be a $2 \times 2$ transition matrix which performs the transition $(f, g) \to (f_1, g_1)$ as matrix multiplication:

$$\begin{pmatrix} f_1 \\ g_1 \end{pmatrix} = T(\delta, f, g) \begin{pmatrix} f \\ g \end{pmatrix}, \text{ where } T(\delta, f, g) = \begin{cases} \begin{pmatrix} 0 & 1 \\ \frac{g(0)}{X} & \frac{-f(0)}{X} \end{pmatrix} & \text{if } \delta > 0 \text{ and } g(0) \neq 0, \\ \begin{pmatrix} 1 & 0 \\ \frac{-g(0)}{X} & \frac{f(0)}{X} \end{pmatrix} & \text{otherwise.} \end{cases}$$

Define the transition matrix of $i$-th step as $T_i = T(\delta_i, f_i, g_i)$. After $n$ steps, the input polynomials $(f, g)$ become

$$\begin{pmatrix} f_n \\ g_n \end{pmatrix} = T_{n-1} \cdots T_0 \cdot \begin{pmatrix} f \\ g \end{pmatrix} = \begin{pmatrix} u_n & v_n \\ q_n & w_n \end{pmatrix} \begin{pmatrix} f \\ g \end{pmatrix}.$$

Note that we use $w$ instead of the original $r$ in [BY19] to avoid the symbol conflict.

[1] See [BY19, Section 6.5] for an alternative method of skipping the reversal. It requests a post-process for polynomials before output.

Since we aim for the polynomial inversion in BIKE, we keep only two vectors $(f, g)$ and $(v, w)$ in our storage space for storing all $(f_i, g_i)$ and $(v_i, w_i)$ for $i$ in $0, \ldots n$, instead of tracking full transition matrices. The polynomials $(f, g)$ and $(v, w)$ are stored in different formats. Since $(v_i, w_i)^T$ is part of the transition matrix, they are polynomials with monomials of negative degrees. Hence, we store the vector $(v_i, w_i)$ in a form of $(v'_i, w'_i) \cdot (1/X)^i$ and $i$ increases with steps to keep the polynomials $(v'_i, w'_i)$ with non-negative degrees. Since $(f_i, g_i)^T$ and $(v_i, w_i)^T$ are multiplied by the same transition matrix, we update the two vectors with similar operations except for the degree adjustment. We remove the coefficient of the constant term of $g$ for the division by $X$ but increase the coefficients of $v$ by one degree to keep the correct form of $(v'_i, w'_i) \cdot (1/X)^i$.

Last, we describe the overall algorithm for the polynomial inversion in BIKE. We initialize the two input polynomials $f = X^r - 1$, $g = h_0(1/X) \cdot X^r$, and their degree difference $\delta = 1$. Note that $g$ is initialized as a bit-reversal form. The $(v, w)$ polynomials are initialized to $(0, 1)$ as the right column of an identity matrix. Then we perform $2r - 1$ `divstep`s to update $(\delta, f, g)$ and $(v, w)$ as well. After `divstep`s, we reverse the coefficients of the polynomial $v$ and output it as the inverse $h_0^{-1}$.

## 13.3 Optimization Strategies

In this section, we propose several optimization strategies to improve the hardware implementation of BIKE. We start by describing the exchange of the symmetric cryptographic building blocks, i.e., AES-256 and SHA2-384 with a single Keccak core. Then, we introduce a new design of a multiplier exploiting the sparseness of QC-MDPC polynomials. Afterwards, we present an improved inversion module based on the algorithm proposed by Bernstein and Yang [BY19]. We conclude this section with an united hardware design that consolidates all three KEM algorithms of BIKE in one implementation.

### 13.3.1 Design Considerations

We start with our design considerations. First, our implementations utilize the framework presented in Chapter 12 while we modify and optimize several hardware modules described in the following sections. Besides these modifications, the main structure is based on the original implementation. However, we translate all modules to Verilog.

Second, we keep the same bandwidth parameter $b$ in our modified modules as proposed in our implementations from Chapter 12. Hence, our design is scalable with $b$ as well, and we benchmark our designs with the same instantiations of $b \in \mathcal{B} = \{32, 64, 128\}$. Larger $b$ generally improve the latency of the corresponding computation since $b$-bit chunks of polynomials can be accessed and processed in parallel.

### 13.3.2 Random Oracles

The BIKE team updated the random oracles **H**, **K**, and **L** with the version 4.3 specifications [ABB+21]. They adapted the core components of these functions from AES256 and SHA2-384 to SHAKE256 and SHA3-384 with an unified Keccak core, respectively. While

we already suggested a unified symmetric core would be beneficial for a hardware implementation in Chapter 12, we, however, did not validate our suggestions with a concrete hardware implementation.

In this chapter, we modify the implementations presented in Chapter 12 to the updated specification of hash functions and report the comparisons in Section 13.4.1. Therefore, we implement a simple KECCAK core which only contains the round function and a controlling interface. In the following, we describe the implementations of wrappers that are connected to the KECCAK core and form the random oracles.

First, for the **H** function, we instantiate a SHAKE256 from the KECCAK's round function. As in Algorithm 3, **H** uses a 256-bit message $m$ as seed for SHAKE256 which is requested by a dedicated interface in our implementation. Then, with correct padding and controlling of the KECCAK core, the wrapper divides the 1 088 output bits into 32-bit chunks. The integrated sampler uses the chunks to generate the indexes of error polynomials $(e_0, e_1)$ and rejects illegal samplings. If the sampler has consumed all randomness, the wrapper initiates an additional squeezing phase of SHAKE256.

Second, for generating the private key $(h_0, h_1)$ in the key generation (cf. Algorithm 2), our wrapper operates similarly to the **H** function besides different Hamming weights.

Third, for the **L** function, the wrapper uses the error polynomials $(e_0, e_1)$ and provides them in the absorbing phases to the KECCAK core. In this case, it performs a SHA3-384 hashing operation. Besides the correct padding, the wrapper ensures to concatenate the error polynomials by eight-bit blocks. Last, it truncates the 384-bit hash value to a 256-bit value and adds it to $m$.

Fourth, our wrapper for the **K** function is realized similarly to the **L** function. However, the input to the SHA3-384 slightly differs since a 256-bit string needs to be concatenated with an $r$-bit polynomial and with another 256-bit string. Nevertheless, it truncates the 384-bit output to 256 bits in the same way.

### 13.3.3 Sparse Polynomial Multiplier

In this section, we present the hardware design of the sparse polynomial multiplier for BIKE. In 2019, Hu et *al.* [HWCW19] already applied the approach of sparse multiplications to BIKE. However, compared to their design, our optimized implementation achieves a better area-time product and reduces the latency (for detailed information see Section 13.4.2). Additionally, our design keeps the time-constancy for the encapsulation while computing $e_0 + e_1 \cdot h$ with the indefinite Hamming weight of $e_1$.

As in Section 13.2.1, given a multiplication $p_{\text{res}} = p_{\text{sparse}} \cdot p_{\text{arb.}}$ where $p_{\text{sparse}}, p_{\text{arb.}} \in \mathcal{R}$ and $|p_{\text{sparse}}| \ll r$. Further, the polynomial $p_{\text{sparse}}$ is represented as a set of indexes of non-zero terms and $p_{\text{arb.}}$ is a $r$-bit sequence divided into $\lceil \frac{r}{b} \rceil$ chunks. Then, we conduct the multiplication by reading the non-zero indexes of $p_{\text{sparse}}$, rotating $p_{\text{arb.}}$ by the indexes to the left, and accumulating the rotated results to the product $p_{\text{res}}$.

**General Sparse Multiplier.** Figure 13.1 shows a simplified architecture of the general sparse multiplier which iterates over the indexes of the sparse polynomial $p_{\text{sparse}}$. Each iteration is initiated by reading a non-zero index from $p_{\text{sparse}}$. Meanwhile, it starts to access the values of the polynomial $p_{\text{arb.}}$ in an ascending order starting at the second uppermost address, proceeding

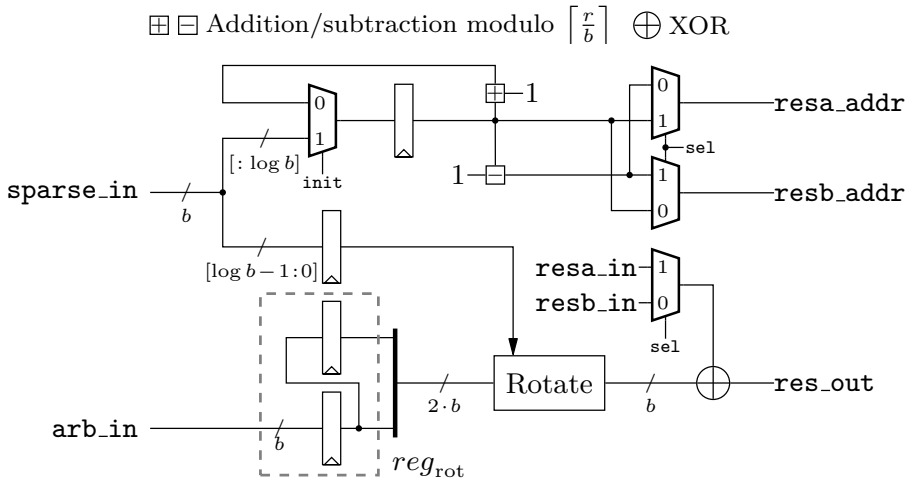



Figure 13.1: Schematic architecture of the general sparse multiplier.

with the uppermost, and then keep going from address zero. This procedure simplifies to deal with the most-significant bits in $p_{\text{arb.}}$ since $r \bmod b \neq 0$ ($r$ is always prime). Figure 13.1 neglects the hardware to deal with this exception (mostly multiplexers) for clarity.

While processing a particular index from $p_{\text{sparse}}$, the lower $\log b$ bits of the index determine the number of bits to shift the input from $p_{\text{arb.}}$ to the left. The shifted output is added to the current intermediate result depicted by the xor-gates in Figure 13.1. We instantiate two memories to store the intermediate results of the multiplication. This allows us to read the current intermediate result from one memory and write the new result to the other one in the same clock cycle.

The upper part of the schematic in Figure 13.1 determines the addresses for both memories. When an index of the sparse polynomial is read, the upper bits are sampled in a register used as initial value for a counter. To handle the jump from the highest address (i.e., $\lfloor \frac{r}{b} \rfloor$) to zero, our final design contains slightly more logic. Again, Figure 13.1 neglects this logic for the sake of clarity. However, the output of the counter is subtracted by one, and two multiplexers decide which of the address values are used to access which of the memories. The decision signal `sel` is determined based on the LSB from the address counter used to read out the indexes of the sparse polynomial.

For each index of the sparse polynomial, our multiplier spends $\lceil \frac{r}{b} \rceil + 4$ clock cycles for shifting and accumulating the intermediate results. The total latency is given by

$$L_{\text{mult}}(th) = \left( \left\lceil \frac{r}{b} \right\rceil + 4 \right) \cdot th + 1 \tag{13.1}$$

where $th$ denotes the weight of the sparse polynomial (e.g., for the key generation in BIKE $th = \frac{w}{2}$). The circuit switches to the `DONE` state in the additional clock cycle.

This design iterates over a fixed number of indexes of the sparse polynomial. While this approach is capable of processing the secret polynomials $(h_0, h_1)$, it cannot process the multiplication $e_1 \cdot h$ in the encapsulation with a constant latency since the Hamming weight of $e_1$ is unknown. Therefore, we modify the design of the general sparse multiplier into a dedicated multiplier for BIKE in the next paragraph.

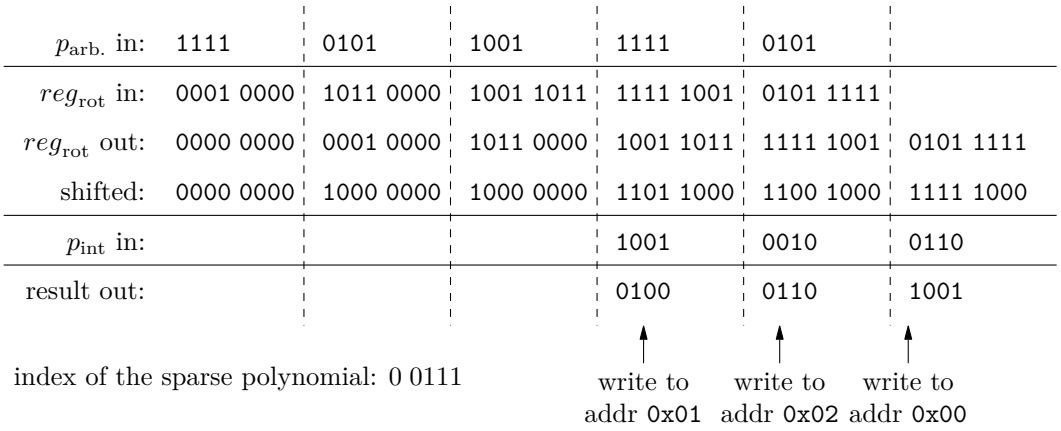



Figure 13.2: Example for a multiplication with an index from $e_1$.

**Tailored Constant-time Multiplier for BIKE.** To deal with the indefinite weight of $e_1$ in the encapsulation, we utilize the relation $|e_0| + |e_1| = t$ defined by BIKE. It allows to rephrase the encoding operation as an addition of two multiplications

$$c_0 = e_0 \cdot 1 + e_1 \cdot h. \tag{13.2}$$

For computing $c_0$, we modify the general sparse multiplier introduced above and add a multiplexer choosing $h$ or 1 as input for $p_{\text{arb.}}$ depending on $e_0$ or $e_1$. To indicate whether $e_0$ or $e_1$ is processed, we add an additional leading bit to the indexes and set the MSB of the indexes belonging to $e_0$ to '1'. We embed this operation directly into the sampling function **H**. Hence, the multiplexer selects its output according to the MSB of the indexes of the sparse polynomial.

In order to illustrate the two modes of the multiplication engine, we provide a small example for $r = 11$, $b = 4$, and $p_{\text{arb.}} = X^{10} + X^8 + X^7 + X^6 + X^5 + X^4 + X^3 + 1 =$ 101 1111 1001 (corresponds to $h$ in Equation 13.2). For the error polynomials, we exemplary assume $e_0 = X^5$ and $e_1 = X^7$ and their corresponding indexes $e_{0,\text{idx}} =$ 1 0101 and $e_{1,\text{idx}} =$ 0 0111, respectively. For both modes, we assume that the current intermediate result is $p_{\text{int}} =$ 010 1001 0110.

Figure 13.2 visualizes the multiplication $e_1 \cdot p_{\text{arb.}}$ where each dashed line separates the data flow between the clock cycles. In this case, the expected result is

$$X^7 \cdot \texttt{101 1111 1001} \oplus \texttt{010 1001 0110} = \texttt{100 1101 1111} \oplus \texttt{010 1001 0110} = \texttt{110 0100 1001}.$$

As described above, the module first reads the second uppermost chunk from the input polynomial which is 1111 in our example. Since $r = 11$ and $b = 4$, only the most significant bit from this chunk is required and stored in the register $reg_{\text{rot}}$ (cf. Figure 13.1). The remaining three bits are taken from the uppermost chunk. Afterwards, the process proceeds in a regular pattern by reading a new chunk and moving the old chunk to the lower part of $reg_{\text{rot}}$. The multiplier determines the starting address to read the first chunk from the intermediate result by the upper bits of the error index, i.e., 0x01 in our example. This describes the required shift on word level. The output of the register is shifted to the left by 3 bit which are the least $\log(b)$ bits from index $e_{1,\text{idx}}$ and describe the required shift on bit level. Hence, the first chunk of the new intermediate result is written to address 0x01. Note, when the multiplier writes the

| | | | | | | |
|---|---|---|---|---|---|---|
| $p_{\text{one}}$ in: | 0000 | 0000 | 0001 | 0000 | 0000 | |
| $reg_{\text{rot}}$ in: | 0000 0000 | 0000 0000 | 0001 0000 | 0000 0001 | 0000 0000 | |
| $reg_{\text{rot}}$ out: | 0000 0000 | 0000 0000 | 0000 0000 | 0001 0000 | 0000 0001 | 0000 0000 |
| shifted: | 0000 0000 | 0000 0000 | 0000 0000 | 0010 0000 | 0000 0010 | 0000 0000 |
| $p_{\text{int}}$ in: | | | | 1001 | 0010 | 0110 |
| result out: | | | | 1011 | 0010 | 0110 |

index of the sparse polynomial: 1 0101

write to addr 0x01 | write to addr 0x02 | write to addr 0x00

Figure 13.3: Example for a multiplication with an index from $e_0$.

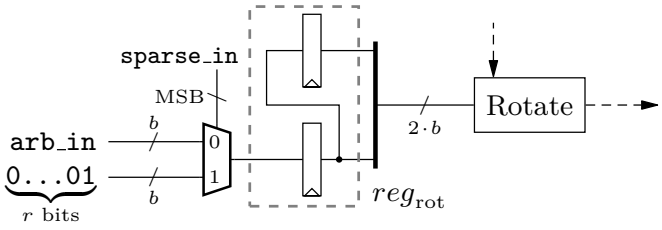



Figure 13.4: Modifications to the input operand of the tailored multiplier.

result to address 0x02, the most significant bit is set to 0 since it does not belong to a valid polynomial of size $r = 11$.

The procedure for a multiplication with the index $e_{0,\text{idx}}$ is similar. Instead of providing the polynomial $p_{\text{arb.}}$ to the multiplier, the polynomial $p_{\text{one}} = 1 =$ 000 0000 0001 is selected by the most signification bit of $e_{0,\text{idx}}$. The corresponding data flow is visualized in Figure 13.3. It is clearly visible that the multiplication with an index from $e_0$ requires the same amount of clock cycles such that a constant-time operation is guaranteed.

To this end, Figure 13.4 shows the adjustment for processing the operand $p_{\text{arb.}}$ in the multiplier. Note, the polynomial of one does not require extra memory but is generated on the fly. While accessing the 0-th chunk of $p_{\text{arb.}}$, the circuit feeds a $b$-bit chunk of 0...01 to the multiplexer. Otherwise, the multiplexer always gets a zero $b$-bit chunk. Hence, the multiplier always finishes the multiplication from Equation 13.2 in $L_{\text{mult}}(t)$ clock cycles.

Last, we add an additional input to the multiplier design determining the number of non-zero indexes of the sparse polynomial for the two possible weights $(t, w/2)$ of the sparse input polynomials.

### 13.3.4 Polynomial Inversion

We present our hardware design and optimization for the polynomial inversion in this section. In 2020, Marotzke [Mar20] reported an implementation for the polynomial inversion required in NTRU Prime, a post-quantum KEM. The inversion module utilizes Bernstein and Yang's

extGCD algorithm [BY19] optimized to perform inversions of polynomials of degree 760 with coefficients in prime fields, where the arithmetic takes place in DSP units. Since our design targets to invert polynomials in $\mathcal{R}$ with large degrees (i.e., $\geq 12\,323$), the two implementations pursue different purposes and are not directly comparable.

In the following, we first divide the computation of `divstep` into two subroutines. Then, we introduce the main framework of the inversion and the two subroutines followed by our hardware designs.

**Performing the `divstep`.** Recalling Section 13.2.2, an extGCD for polynomial inversion computes $2r - 1$ `divstep`s. In [BY19], based on the shape of the transition matrix, Bernstein and Yang optimized the multiplication by the transition matrix in a single `divstep` as two simple functions:

(1) a conditional swap: replacing $(\delta, f, g)$ with $(-\delta, g, f)$ if $\delta > 0$ and $g(0) \neq 0$.

(2) an elimination: replacing $(\delta, f, g)$ with $(1 + \delta, f, (f(0)g - g(0)f)/X)$.

Since the head coefficient $f(0)$ is always one for computing the inversion in BIKE, we need only two information bits deduced from $(\delta, g(0))$ in each `divstep` as instructions for updating $(f, g)$ and $(v, w)$. The first bit indicates the swap operation and the second bit is $g(0)$ used in the elimination operation. We refer to the two information bits as *control bits* of one `divstep` in this paper. Furthermore, we split one `divstep` into two operations:

(1) `get_control_bits()`: calculates the control bits based on the values of $\delta$ and the necessary coefficients of the polynomials $(f, g)$, and

(2) `update_fg_or_vw()`: updates the polynomials $(f, g)$ and $(v, w)$ based on the computed control bits.

**Main Framework.** Algorithm 12 describes the main framework of the polynomial inversion. As introduced above, the algorithm uses four temporary polynomials $f$, $g$, $v$, and $w$ while $g$ is initialized with the bit-reversed input polynomial $g_{\text{in}}$. The main parts of the algorithm are $2r - 1$ `divstep`s, which are decoupled into series of `get_control_bits()` and `update_fg_or_vw()` subroutines. Last, the algorithm shifts $v$ one bit to the right, reverses its coefficients, and returns $v$ as the inverse of the input polynomial.

In the algorithm, we introduce a parameter $s$ to control the *step size*, allowing to proceed $s$ `divstep`s in each iteration in parallel (cf. line 7). The `get_control_bits()` and the `update_fg_or_vw()` take the parameter as well and proceed $s$ steps accordingly. Therefore, the subroutine `get_control_bits()` determines $2s$ control bits and updates $\delta$ based on the state of $(\delta, f[0], g[0])$. Afterwards, a loop iterates over all four polynomials $f$, $g$, $v$, and $w$ and updates them by `update_fg_or_vw()` for $s$ steps in each call. Starting from line 21, the algorithm covers the remaining steps and updates only $(v, w)$ accordingly.

Besides the step size $s$, the execution time of Algorithm 12 scales with the bandwidth parameter $b$ as well. Enlarging $b$ decreases the number of chunks $N$ and therefore, less iterations are executed in the inner loop since `update_fg_or_vw()` updates one chunk in each execution. In our design, the choice for $s$ is also limited by $s \leq b$ since `get_control_bits()` takes inputs of one polynomial chunk only. We describe the details of `get_control_bits()` and `update_fg_or_vw()` in the following paragraphs.

**Algorithm 12** Main framework for the polynomial inversion.

**Require:** Input polynomial $g_{\text{in}}$ and step size $s$.
**Ensure:** Inverted polynomial $g_{\text{out}} = g_{\text{in}}^{-1}$

1: $N \leftarrow \lceil \frac{r}{b} \rceil$
2: $f\,[N],\, g\,[N],\, v\,[N],\, w\,[N] \leftarrow 0$ ▷ Initialize polynomials (arrays)
3: $w[0] = 1;\ f[0] = 1;\ f[N-1] \leftarrow 2^{r \bmod b}$
4: $g \leftarrow \texttt{bitreverse}\,(g_{\text{in}})$ ▷ Reverse the bits of the input polynomial
5: $\delta \leftarrow 1$ ▷ Degree difference of polynomials $f$ and $g$
6: $\tau \leftarrow 2r - 1$ ▷ Number of `divstep`s to be executed
7: **while** $\tau \geq s$ **do** $\delta, c \leftarrow \texttt{get\_control\_bits}(\delta, f[0], g[0], s)$
8: **for** $j = 0$ **to** $N$ **do**
9: $f_0, f_1 \leftarrow f[j], ((j+1) > N\ ?\ 0 : f[j+1])$
10: $g_0, g_1 \leftarrow g[j], ((j+1) > N\ ?\ 0 : g[j+1])$
11: $f[j], g[j] \leftarrow \texttt{update\_fg\_or\_vw}(c, f_1, f_0, g_1, g_0, s, 1)$
12: **end for**
13: **for** $j = N$ **to** $0$ **do**
14: $v_0, v_1 \leftarrow (j == 0\ ?\ 0 : v[j-1]), v[j]$
15: $w_0, w_1 \leftarrow (j == 0\ ?\ 0 : w[j-1]), w[j]$
16: $v[j], w[j] \leftarrow \texttt{update\_fg\_or\_vw}(c, v_1, v_0, w_1, w_0, s, 0)$
17: **end for**
18: $\tau \leftarrow \tau - s$
19: **end while**
20: **if** $\tau > 0$ **then**
21: $\delta, c \leftarrow \texttt{get\_control\_bits}(\delta, f[0], g[0], \tau)$
22: **for** $j = N$ **to** $0$ **do**
23: $v_0, v_1 \leftarrow (j == 0\ ?\ 0 : v[j-1]), v[j]$
24: $w_0, w_1 \leftarrow (j == 0\ ?\ 0 : w[j-1]), w[j]$
25: $v[j], w[j] \leftarrow \texttt{update\_fg\_or\_vw}(c, v_1, v_0, w_1, w_0, s, 0)$
26: **end for**
27: **end if**
28: $v \leftarrow \texttt{shift\_right}(v)$ ▷ Shift one bit to the right
29: **return** `bitreverse` $(v)$

**Determining Control Bits.** Algorithm 13 details the process of `get_control_bits()`. The algorithm takes four inputs, which are the degree difference $\delta$, two $b$-bit chunks $(f[0], g[0])$ from the polynomials $(f, g)$, and the step size $s$. The algorithm outputs the updated $\delta$ and $2s$ control bits $c$ for $s$ `divstep`s. For generating the control bits for the $s$ `divstep`s, the algorithm uses only $s$ bits from each input polynomial instead of the full coefficients. Note, however, the algorithm is a sequential process where the control bits of iteration $i$ depend on the results of the previous iterations.

Our hardware design for `get_control_bits()` incorporates this characteristic such that we aim to fully utilize the computational capacity and hence execute $d$ iterations of the loop shown in Algorithm 13 in one clock cycle. Therefore, Figure 13.5 shows a schematic draft of this approach where one iteration is highlighted by the red dashed border. For larger step sizes $s$,

**Algorithm 13** Compute control bits.

**Require:** Current $\delta$, $f[0]$, $g[0]$, and the step size $s$.
**Ensure:** Updated $\delta$ and an array of control bits $c[2s]$

$f, g \leftarrow f[0], g[0]$
**for** $i = 0$ **to** $s - 1$ **do**
  $swap \leftarrow ((-\delta < 0)\ ?\ 1 : 0)\ \&\ (g \wedge 1)$
  $\alpha \leftarrow g \wedge 1$
  $c[i \cdot 2] \leftarrow swap$
  $c[i \cdot 2 + 1] \leftarrow \alpha$
  $\delta \leftarrow swap\ ?\ -\delta + 1 : \delta + 1$
  $f, g \leftarrow (swap\ ?\ g : f), (g \oplus (f \cdot \alpha))/2$
**end for**
**return** $\delta$, $c$

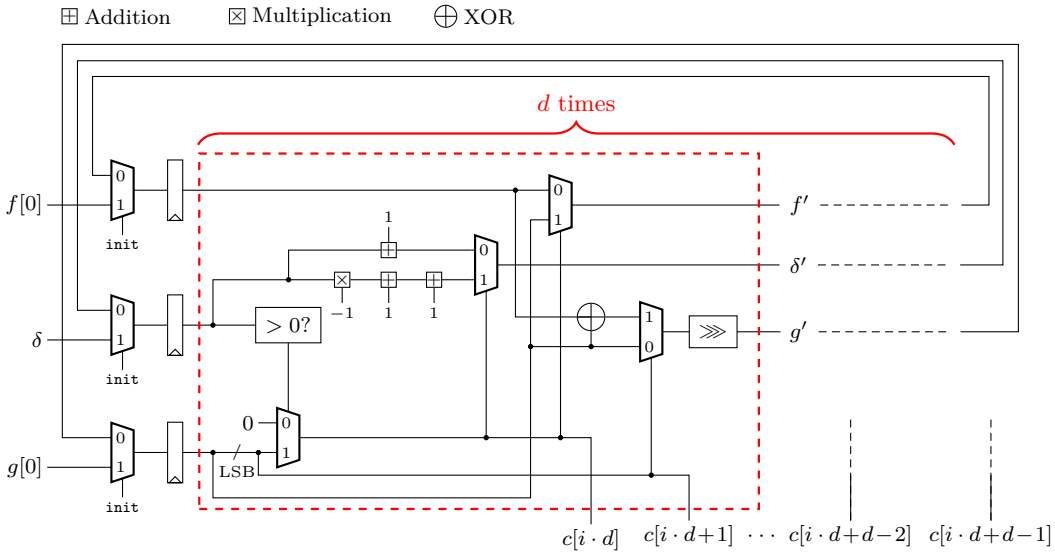



Figure 13.5: Hardware design for the computation of the control bits.

however, unrolling the whole loop in a hardware implementation would result in a long critical path. Hence, we introduce a round-based circuit that is executed $\lceil \frac{s}{d} \rceil$ times since $d \cdot \lceil \frac{s}{d} \rceil \geq s$. We store the generated control bits in registers to use them immediately for updating the polynomials $(f, g)$ and $(v, w)$ by `update_fg_or_vw()`.

**Updating Polynomials.** We summarize the details of `update_fg_or_vw()` in Algorithm 14. The algorithm expects as inputs the control bits $c$, two $2b$-bit chunks of the polynomials $(f, g)$ or $(v, w)$, the step size $s$, and one bit specifying whether the input chunks originating from the pairs $(f, g)$ or $(v, w)$. The algorithm updates the given chunks for $s$ `divstep`s according to the control bits $c$. Since $(f, g)$ and $(v, w)$ are multiplied by the same transition matrix in the same `divstep`, the arithmetic for updating the polynomials is identical. The different formats of

**Algorithm 14** `update_fg_or_vw()`

**Require:** Control bits $c$, two $2b$-bit chunks of $f$ and $g$, and the step size $s$ .
**Ensure:** Updated $b$-bit chunks $r_0$ and $r_1$

```
for i = 0 to s − 1 do
    f, g ← (c[i · 2] ? g : f), (c[i · 2 + 1] ? g ⊕ f : g)
    if is_updating_fg then
        g ← g/2                                  ▷ Shift right, i.e., dividing by X
    else
        f ← f · 2                                ▷ Shift left, i.e., multiplying by X
    end if
end for
if is_updating_fg then
    r0, r1 ← f[0 : b], g[0 : b]                  ▷ lower b bits
else
    r0, r1 ← f[b : 2b], g[b : 2b]                ▷ higher b bits
end if
return r0, r1
```

storing polynomials (see Section 13.2.2) cause the difference between the two operating modes, which shift polynomials in different directions and output different chunks of polynomials.

Figure 13.6 shows our hardware design for updating the polynomials $(f, g)$. The *basic block* (highlighted by the red dashed border) updates the polynomials for one `divstep`, consisting of simple shifts, an addition (xor), and multiplexing operations. The whole submodule can finish the computation with $s$ consecutive basic blocks which, however, would result in a long critical path without any further modifications. Therefore, to control the length of the critical path, we introduce pipeline registers after $u$ basic blocks. Hence, there are $\lfloor \frac{s}{u} \rfloor$ pipeline stages in the module. Note, we implement a similar module to update $(v, w)$.

Although, Figure 13.6 depicts two full $b$-bit chunks for each input associated with the different polynomials, the algorithm actually only requires $b + s$ bits of data from the input polynomials. The algorithm inputs the $2b$-bit chunks because it accesses polynomials in chunks of $b$ bits from the memory. However, the module only instantiates logic for processing $s + b$ data such that no area overhead occurs.

**Overall Design of the Polynomial Inversion.** The entire polynomial inversion module consists of two counters controlling the reversion of the bits and the final right shift (cf. Algorithm 12). Additionally, we instantiate `get_control_bits()` and two versions of `update_fg_or_vw()` (updating $(f, g)$ and $(u, w)$ in parallel) as described above. Since the algorithm works on four temporary polynomials, the inversion module utilizes eight BRAMs allowing to read and write the intermediate results in the same clock cycle. Nevertheless, the latency of the proposed design depends on several parameters, i.e., $r$, $b$, $s$, $d$, and $u$. It is determined by

$$L_{\text{inv}}(s,d,u) = \underbrace{\lambda \cdot \left(3 + \left\lceil \frac{s}{d} \right\rceil + \left\lceil \frac{s}{u} \right\rceil + \left\lceil \frac{r}{b} \right\rceil\right)}_{\text{main computation}} + \underbrace{\rho + \left\lceil \frac{s}{d} \right\rceil + \left\lceil \frac{r}{b} \right\rceil}_{\text{remainder}} + \underbrace{3 \cdot \left\lceil \frac{r}{b} \right\rceil + 13}_{\text{bitreverse \& shift}} \tag{13.3}$$

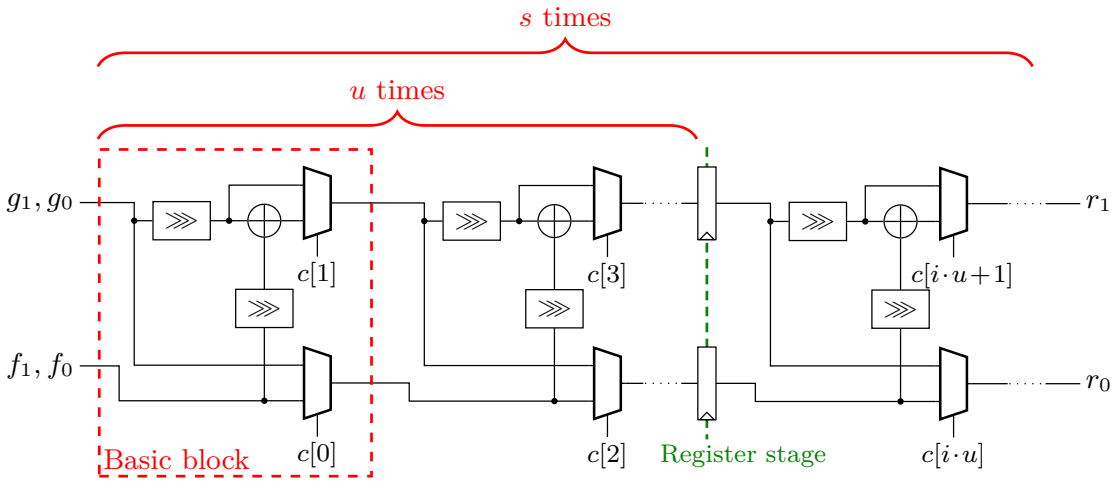



Figure 13.6: Hardware implementation of the update process for the $f$ and $g$ polynomial.

where $\lambda = \left\lfloor \frac{2 \cdot r - 1}{s} \right\rfloor$ and $\rho = \left\lceil \frac{2 \cdot r - 1 - \lambda \cdot s}{u} \right\rceil$. Note, our design for $s = 1$ does not follow Equation 13.3 since it is a handcrafted and optimized design which achieves a slightly smaller latency and requires only seven BRAMs instead of eight.

### 13.3.5 United Hardware Design

Given the optimized modules for the polynomial arithmetic and the modifications for the random oracles, we now present an *united hardware design* of BIKE consolidating the key generation, encapsulation and decapsulation in one module. Such a design allows to share resources between the different KEM operations. For example, we only instantiate one single multiplier, one Keccak core with the corresponding wrappers described in Section 13.3.2, and a limited number of BRAM modules. The number of required BRAMs is given by the decapsulation since its implementation utilizes the most memories (cf. Chapter 12). However, this design decision implies that only one of the three KEM algorithms of BIKE can be executed at the same time. Therefore, we implement a control interface that allows to enable the desired algorithm by a three-bit instruction, load and read data (polynomials and 256-bit strings), and request randomness used as seed for the PRNG. A top-level draft of this implementation is shown in Figure 13.7. While all building blocks that are used by more than one KEM algorithms are marked by a green border, the black modules are only required for a single KEM operation (the inversion module and sampler are used only in the key generation, and the BFIter module together with the Hamming weight and threshold computation only in the decapsulation).

The FSM on the right side manages all input/output operations and the control flow of the three KEM algorithms. The input interface expects a six-bit instruction identifying which data should be loaded. For the key generation, no initial data is required. The encapsulation requires the public key $h$ which needs to be loaded to a BRAM before the computation can be started. To perform a decapsulation, the implementation assumes that the user load the two parts of the cryptogram $(c_0, c_1)$, the two polynomials of the private key $(h_0, h_1)$, and $\sigma$. The output interface returns the same data and additionally the shared key $K$. After the required data has been accessed, all memories are reset by overwriting the content with zero.

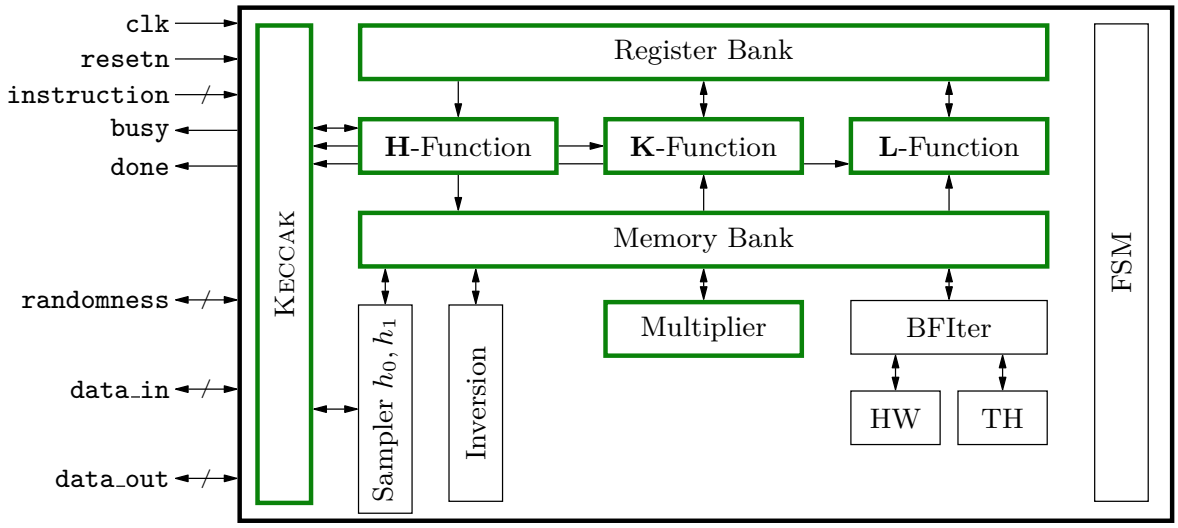



Figure 13.7: Top-level view of the united hardware design.

## 13.4 Implementation Results

In this chapter, we evaluate the proposed optimizations and modifications for a hardware implementation of BIKE. First, we show that the modifications of the random oracles are beneficial for a hardware design of BIKE. Second, we report implementation results for the proposed sparse multipliers and compare them to designs from the literature. Third, we demonstrate the scalability of our inversion module by presenting implementation results for different configurations. Fourth, since both – the multiplication and inversion – influence the footprint and performance of the key generation, we provide dedicated implementation results for a standalone key generation design. Fifth, we present the implementation results of the united hardware design and compare it to other implementations of code-based PQC schemes. We generate all results for an Artix-7 XC7A200T FPGA manufactured by Xilinx.

### 13.4.1 New Random Oracles

As described in Section 13.3.2, BIKE's new specification [ABB+21] updates the random oracles from AES-256 and SHA2 to an unified Keccak core. To test how the design choice of cryptographic primitives affects the performance of hardware implementations, we compare the implementations of the original VHDL code[2] from Chapter 12 with our adapted version applying the new specification with a replaced Keccak core. We performed no other optimizations for a fair comparison.

Table 13.1 reports the comparisons for the encapsulation and decapsulation. For both KEM algorithms and all hardware configurations, the adapted versions achieve slightly better results in terms of area and latency. Especially the number of required registers decreases by roughly 880 in the adapted implementation for all designs. To this end, these implementation results show that the modifications of the random oracles are indeed beneficial for hardware implementations of BIKE.

[2]The authors published their code at `https://github.com/Chair-for-Security-Engineering/BIKE/`

Table 13.1: Comparison of KEM functions w.r.t. different random oracle settings ($r = 12\,323$).

| | **Resources** | | | | | **Performance** | | |
|---|---|---|---|---|---|---|---|---|
| | Logic | | Memory | | Area | Cycles | Freq. | Latency |
| $b$ | LUT | DSP | FF | BRAM | Slices | Cycles | MHz | ms |
| *Encapsulation with adapted random oracles* | | | | | | | | |
| 32 bit | 6 604 | 0 | 2 409 | 3 | 1 906 | 151 587 | 121.95 | 1.24 |
| 64 bit | 8 388 | 0 | 2 444 | 5 | 2 408 | 39 264 | 121.95 | 0.32 |
| 128 bit | 15 135 | 0 | 2 625 | 10 | 4 268 | 11 136 | 119.05 | 0.094 |
| *Encapsulation of the previous specification from Chapter 12* | | | | | | | | |
| 32 bit | 6 730 | 0 | 3 298 | 3 | 2 143 | 152 694 | 121.95 | 1.25 |
| 64 bit | 8 253 | 0 | 3 327 | 5 | 2 538 | 40 368 | 121.95 | 0.33 |
| 128 bit | 14 829 | 0 | 3 471 | 10 | 4 540 | 12 240 | 121.95 | 0.10 |
| *Decapsulation with adapted random oracles* | | | | | | | | |
| 32 bit | 9 070 | 7 | 3 055 | 10 | 2 570 | 1 624 402 | 125 | 13 |
| 64 bit | 14 011 | 9 | 3 415 | 15 | 3 933 | 515 823 | 116.28 | 4.44 |
| 128 bit | 29 697 | 13 | 4 170 | 29 | 8 234 | 186 364 | 100 | 1.86 |
| *Decapsulation of the previous specification from Chapter 12* | | | | | | | | |
| 32 bit | 9 380 | 7 | 3 943 | 10 | 2 971 | 1 626 674 | 125 | 13.01 |
| 64 bit | 16 140 | 9 | 4 307 | 15 | 4 942 | 518 105 | 116.28 | 4.46 |
| 128 bit | 30 430 | 13 | 5 063 | 29 | 8 785 | 188 646 | 100 | 1.89 |

### 13.4.2 Multiplier

Table 13.2 shows the implementation results for our two multiplier designs configured for the lowest security level of BIKE, i.e., for $r = 12\,323$. The first design is the general sparse multiplier where the sparse polynomial always has a fixed Hamming weight, i.e., the Hamming weight is determined before synthesis. In BIKE, such cases occur in the key generation and decapsulation where $|p_{\text{sparse}}| = w/2$. The second design reads the Hamming weight of the sparse polynomial via an input interface. Hence, it can be used for all multiplications required in BIKE. Additionally, the design performs the encoding in the encapsulation in constant time. To this end, the hardware utilization is slightly higher than for the general sparse multiplier. Note, for the second multiplier design, we report performance numbers for the multiplication performed in the encapsulation, i.e., $|p_{\text{sparse}}| = t = 134$. The number of clock cycles for different Hamming weights follows Equation 13.1.

Table 13.2 also lists the results of the schoolbook-based (dense) multiplier from Chapter 12 and of the sparse multiplier design from [HWCW19]. Since we only reported implementation results for $r = 10\,163$ in Chapter 12, we synthesized the design for $r = 12\,323$, again. As expected, the sparse multiplier clearly outperforms the schoolbook-based design with respect to area. For a fixed Hamming weight of 71, the sparse multiplier also achieves better performance results. However, for $b = 128$, the schoolbook multiplier achieves slightly better performance results

Table 13.2: Comparison of sparse polynomial multipliers for $r = 12\,323$.

| | **Resources** | | | | **Performance** | | |
|---|---|---|---|---|---|---|---|
| | Logic | Memory | | Area | Cycles | Frequency | Latency |
| $b$ | LUT | FF | BRAM | Slices | Cycles | MHz | $\mu$s |
| *General sparse multiplier ($\|p_{sparse}\| = w/2 = 71$)* | | | | | | | |
| 32 | 319 | 127 | 2 | 132 | 27 691 | 234.36 | 118.16 |
| 64 | 549 | 190 | 4 | 197 | 13 988 | 222.22 | 62.94 |
| 128 | 1 136 | 381 | 8 | 378 | 7 172 | 184.95 | 38.78 |
| *Tailored sparse multiplier for BIKE ($\|p_{sparse}\| = t = 134$)* | | | | | | | |
| 32 | 349 | 135 | 2 | 151 | 52 261 | 238.15 | 219.5 |
| 64 | 629 | 204 | 4 | 245 | 26 399 | 222.52 | 118.8 |
| 128 | 1 249 | 386 | 8 | 437 | 13 535 | 184.3 | 73.44 |
| *Sparse multiplier from [HWCW19] ($r = 10\,163$, $\|p_{sparse}\| = 71$)* | | | | | | | |
| 32 | – | – | 2 | 100 | 158 614 | 240 | 660.89 |
| 64 | – | – | 3 | 157 | 90 880 | 220 | 413.09 |
| 128 | – | – | 5 | 292 | 51 688 | 210 | 246.13 |
| *Dense polynomial multiplier from Chapter 12* | | | | | | | |
| 32 | 697 | 105 | 1.5 | 220 | 150 155 | 201.37 | 745.67 |
| 64 | 2 595 | 137 | 3 | 864 | 37 829 | 173.82 | 230.91 |
| 128 | 9 539 | 293 | 6 | 3 332 | 9 701 | 183.66 | 52.82 |

than the tailored sparse multiplier which it trades with a huge area footprint. Therefore, the sparse multiplier is clearly superior with respect to the Area-Time (AT) product.

Compared to the multiplier from [HWCW19], our design achieves a considerably lower latency albeit our results were generated for a larger parameter set. Our design mainly differentiates from their implementation in two parts. First, we decided to instantiate two memories to store the intermediate results of the multiplication's product. This allows us to perform a read and write access in the same clock cycle while the implementation by Hu et *al.* requires two clock cycles. Note, for Xilinx FPGAs one could exploit the *read-then-write* option allowing to perform a read and write access in the same clock cycle to the same address reducing the amount of required BRAM modules. However, we decided not to use this option but rather instantiate two memories since it is a more generic approach that is universally applicable to other hardware devices as well. Second, our rotation unit performs the whole rotation within one clock cycle while the design by [HWCW19] requires $\lceil \log b \rceil$ clock cycles. Even though our multiplier architectures consume slightly more slices, it clearly improves the AT product.

We also tried to compare our results to the design proposed in [BFG+19] but we were not able to figure out which value the authors applied for the parameter $BW$ (corresponds to our bandwidth parameter $b$) so that a fair comparison is difficult. However, we assume that their design is similar to our multiplier design which uses fixed Hamming weights.

### 13.4.3 Inversion Module

In this section, we first evaluate the polynomial inversion module described in Section 13.3.4 for $b \in \mathcal{B}$ and for $r = 12\,323$ and compare our approach afterwards to the design presented in Chapter 12 which is based on Fermat's little theorem. Note, in all experiments, we fix the maximum number of basic blocks instantiated between two register stages for the updating process of $(f, g)$, and $(v, w)$ to $u = 8$ achieving a critical path that is smaller than 10 ns. Additionally, we generate all results in this subsection for a target frequency of 100 MHz.

**Detailed Evaluation of the Inversion Module.** Figure 13.8a shows the number of required slices and the latency in clock cycles for $b = 32$, $1 \leq s \leq 32$, and $d = 2$. The area footprint linearly increases with the step size parameter $s$ while the number of clock cycles follows Equation 13.3. Moreover, we include the configuration for the best AT product (slices $\times$ cycles/$10^6$) visualized by the green dashed line. The configuration for $s = 23$ achieves the best result with an AT product of 432. A more detailed evaluation of the implementations can be found in the appendix in Table 15.2.

Figure 13.8b shows the implementation results for different step sizes $s$ for $b = 64$. The trends for the required clock cycles and for the area utilization are very similar to the configurations for $b = 32$. The smallest configuration requires 4 880 299 clock cycles but only consumes 377 slices while the fastest design performs one inversion within 91 678 clock cycles by consuming 5 457 slices. The design with the best AT product is obtained for $s = 31$ (a detailed evaluation can be found in the appendix in Table 15.3).

The implementation results for $b = 128$ are plotted in Figure 13.8c where the best AT product is obtained for $s = 16$. To achieve reasonable critical paths (maximum possible frequency larger than 100 MHz), we reduce the number of unrolled rounds to compute the control bits $c$ to $d = 1$. With $s = 128$ we can instantiate our fastest inversion module which finishes one polynomial inversion in only 47 386 clock cycles. However, the implementation costs drastically increase to 21 435 slices. Again, a detailed evaluation is given in the appendix in Table 15.4 and Table 15.5.

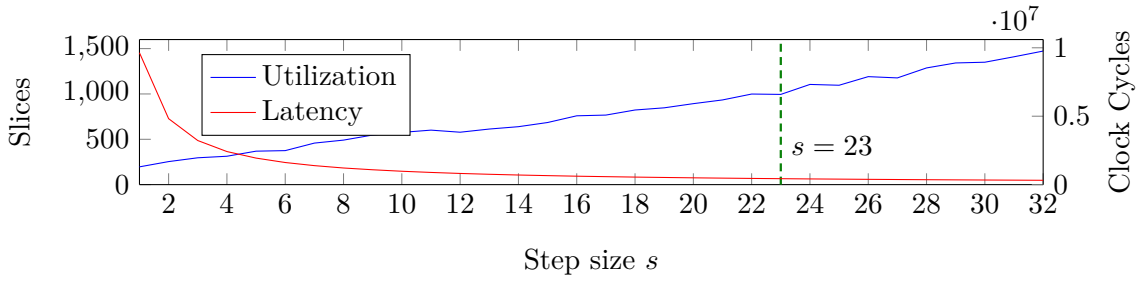
**Comparison to Related Work.** We compare our inversion module to the approach presented in Chapter 12 which is based on Fermat's little theorem in Table 13.3. The corresponding numbers are extracted from their implementation of the key generation.

With Fermat's little theorem, given a $g \in \mathcal{R}$, we compute the inverse as $g^{2^{r-1}-1}$ in Chapter 12. To efficiently raise the degree of $g$, we used a square-and-multiply chain from the ITA [IT88] achieving a latency of

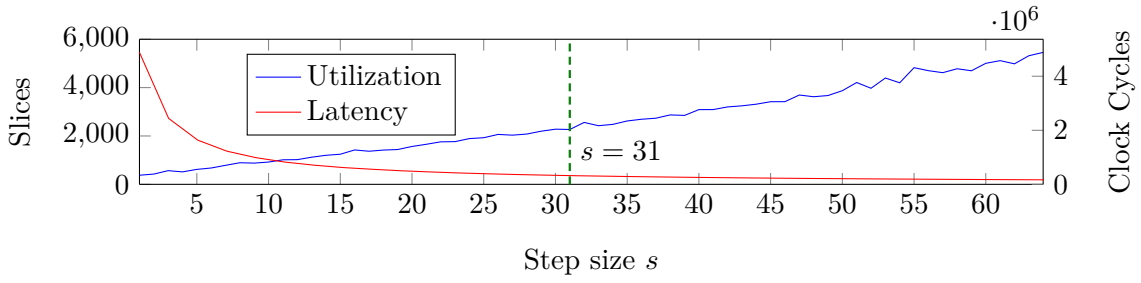
$$L_{\text{inv-Fermat}} \approx \log(r) \cdot (r + L_{\text{school}}) + |r_{\text{bin}}| \cdot \left( \left\lceil \frac{r}{b} \right\rceil + L_{\text{school}} \right) \tag{13.4}$$

where $r_{\text{bin}} = r - 2$ and $L_{\text{school}} = \lceil \frac{r}{b} \rceil \cdot (\lceil \frac{r}{b} \rceil + 3) + 1$. Note, Equation 13.4 describes just an approximation of the required clock cycles since the implementation from Chapter 12 is highly optimized to the use-case of BIKE. However, compared to the dominant term $\left\lfloor \frac{2 \cdot r - 1}{s} \right\rfloor \cdot \lceil \frac{r}{b} \rceil$ from Equation 13.3, our inversion module has an extra parameter $s$, allowing to achieve more optimized configurations.

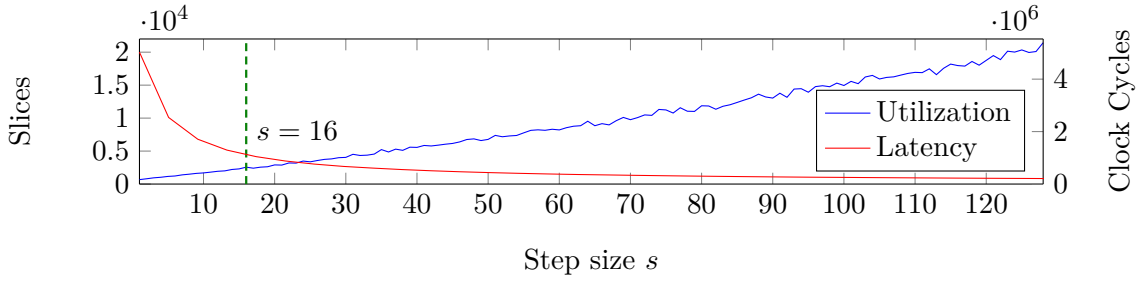
In Table 13.3, we present results for the lightweight ($s = 1$) and high-speed ($s = b$) configuration as well as the design with the best area-time product. For comparison with the area




(a) Implementation results for $b = 32$, and $d = 2$.




(b) Implementation results for $b = 64$ and $d = 2$.




(c) Implementation results for $b = 128$ and $d = 1$.

Figure 13.8: Implementation results for the polynomial inversion for a Xilinx Artix-7 FPGA and a target frequency of 100 MHz setting $r = 12\,323$. The green dashed lines indicate the configurations with the best area-time product.

cost, we report a configuration targeting the number of clock cycles of the approach from Chapter 12. While finishing the inversion with the same amount of clock cycles, Table 13.3 shows that the inversion module based on the extGCD achieves a smaller footprint. This implies that the extGCD implementation results in a better area-time product. We note that the inversion based on Fermat's little theorem always requires a dense polynomial multiplier, which increases the area cost notably. For the design with the best area-time product, our approach consumes roughly twice the amount of logic but finishes the inversion with only one sixth clock cycles setting $b = 32$.

Recently, Deshpande et *al.* [DPM+21] presented a hardware implementation of Bernstein and Yang's inversion algorithm for computing the modular inverse for integers. Their implementation targets integer sizes of 255 bits to 2 048 bits which requires units for integer additions with carry logic. Since we compute the inverse of bit polynomials of at least 12 323 bits and

Table 13.3: Comparison of our inversion module to related work for $r = 12\,323$.

| | **Resources** | | | | **Performance** | | |
|---|---|---|---|---|---|---|---|
| | Logic | Memory | | Area | Cycles | Frequency | Latency |
| $b$ | LUT | FF | BRAM | Slices | Cycles | MHz | $\mu$s |
| *Our lightweight designs ($s = 1$)* | | | | | | | |
| 32 | 580 | 117 | 7 | 196 | 9 637 363 | 100 | 96 637 |
| 64 | 1 020 | 183 | 7 | 377 | 4 880 299 | 100 | 48 803 |
| 128 | 1 805 | 247 | 14 | 671 | 2 514 091 | 100 | 25 141 |
| *Our high-speed design ($s = b$)* | | | | | | | |
| 32 | 5 038 | 943 | 8 | 1 473 | 316 504 | 100 | 3 165 |
| 64 | 18 610 | 3 563 | 8 | 5 457 | 91 678 | 100 | 917 |
| 128 | 75 269 | 14 028 | 16 | 21 435 | 47 386 | 100 | 474 |
| *Our design with the best area-time product ($s = 23$, $s = 31$, $s = 16$)* | | | | | | | |
| 32 | 3 359 | 643 | 8 | 995 | 434 255 | 100 | 4 343 |
| 64 | 7 801 | 1 473 | 8 | 2 269 | 172 522 | 100 | 1 725 |
| 128 | 8 322 | 1 245 | 16 | 2 560 | 182 138 | 100 | 1 821 |
| *Our design targeting the clock cycles Chapter 12 ($s = 4$, $s = 7$, $s = 11$)* | | | | | | | |
| 32 | 905 | 179 | 8 | 313 | 2 416 672 | 100 | 24 167 |
| 64 | 2 391 | 334 | 8 | 786 | 708 310 | 100 | 7 083 |
| 128 | 5 615 | 1 157 | 16 | 1 807 | 253 533 | 100 | 2 535 |
| *Inversion Module used in Chapter 12* | | | | | | | |
| 32 | 1 721 | 343 | 5 | 495 | 2 670 881 | 131.58 | 20 299 |
| 64 | 3 597 | 419 | 5 | 994 | 748 769 | 113.64 | 6 589 |
| 128 | 11 878 | 722 | 10 | 3 352 | 258 555 | 96.15 | 2 689 |

perform carry-less additions, i.e., the `XOR` operation, the two implementations target different applications, and a comparison of performance numbers would be misleading.

Additionally, referring to the sequential design of [DPM+21], they always compute the control bits for only one `divstep` and update the integers with one `divstep`. This corresponds to the configuration of $s = 1$ in our design introduced in Section 13.3.4. Hence, our inversion module provides more configurations allowing to finely adapt to various circumstances.

### 13.4.4 Key Generation

We report implementation results for stand-alone key generation modules in Table 13.4 and compare them to the key generation module from Chapter 12. We evaluate our designs only on the key generation because the polynomial inversion module is used solely in this KEM operation. Because our design is based on the extGCD instead on Fermat's little theorem, we do not install a dense polynomial multiplier that is required for the inversion with Fermat's little theorem. Instead, we use a sparse multiplier which is far more efficient (in both area and latency) than the dense multiplier in the key generation (cf. Table 13.2). Although the module

of key generation consists of various components, including the main operations occur in the inversion module and the multiplier.

As described before, both designs perfectly scale with the bandwidth parameter $b$ while the inversion module provides an additional configuration via the step size $s$. Nevertheless, for each $b \in \mathcal{B}$, we only pick two configurations for the inversion: (1) setting $s = b$ which results in the fastest configurations we can achieve, and (2) instantiating the inversion module with the lowest AT product determined in Section 13.4.3.

The fastest key generation, that we can implement with our approaches, is obtained for $b = s = 128$. The key generation only takes 484 µs but requires over 25 000 slices. The maximum frequencies for the designs with $b = 128$ are slightly higher than for $b = 64$ because the parameter $d$ is decreased to $d = 1$. We decided to synthesize these designs for $d = 1$ since otherwise the critical path for the computation of the control bits would drastically increase. Note, the results for $b = 64$ and $b = 128$ for the designs adjusted to the best AT product achieve roughly the same performance because $b$ is doubled while $s$ is halved. Therefore, the design for $b = 64$ is more efficient due to the lower footprint.

Since our proposed inversion module is highly scalable, there are many other possible configurations. An estimation of the expected footprint and clock cycles can be obtained by using the results provided in the appendix (see Section 15.4).

In Chapter 12 we did not implement a PRNG to provide randomness to the sampler which makes a comparison in this chapter more difficult. Therefore, we determined the hardware utilization of our Keccak core which consumes roughly 800 slices. Considering these additional costs, our design adjusted to the AT products of the inversion modules is roughly 5.5 times faster while it only consumes 3.6 more slices for $b = 32$.

### 13.4.5 United Design

We present the implementation results of the united hardware design of BIKE, introduced in Section 13.3.5, in Table 13.5 for the lowest security level. Results for Level 3 and Level 5 can be found in the appendix in Table 15.6. We created three different implementations where the first one is a lightweight design ($b = 32$), the second one is a design with a trade-off between hardware resources and performance ($b = 64$), and the last one is a high-speed design with $b = 128$. The instantiations of the inversion module are the designs with the best AT product identified in Section 13.4.3.

Table 13.5 also contains the estimated implementation results for a united hardware design of BIKE from Chapter 12. For the lightweight configuration, our design clearly outperforms the previous design with respect to hardware resources and performance. This improvement is mainly due to the new multiplier design and inversion module.

For the high-speed design, our proposed implementation consumes only half the amount of slices while achieving comparable performance results. Particularly, the latency of the key generation is significantly improved due to the inversion module. However, the number of clock cycles for the encapsulation and decapsulation slightly increased. This slight increase is due to the sparse polynomial multiplier.

Since the latency of the sparse multiplier is proportional to the Hamming weight of the sparse polynomial (cf. Equation 13.1), the schoolbook multiplier achieves a better performance when

Table 13.4: Comparison of stand-alone key generation modules for $r = 12\,323$.

| Configuration | | | | Utilization | | | | Performance | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Logic | Memory | | Area | Cycles | Frequency | Latency |
| $b$ | $s$ | $d$ | PRNG* | LUT | FF | BRAM | Slices | Cycles | MHz | ms |
| *This work – High Speed ($s = b$)* | | | | | | | | | | |
| 32 | 32 | 2 | ✓ | 9 880 | 3 321 | 5 | 3 070 | 344 777 | 130.91 | 2.63 |
| 64 | 64 | 2 | ✓ | 24 564 | 6 255 | 10 | 7 776 | 106 243 | 104.65 | 1.02 |
| 128 | 128 | 1 | ✓ | 82 457 | 17 510 | 19 | 25 009 | 55 135 | 113.95 | 0.484 |
| *This work – Best AT product for inversion* | | | | | | | | | | |
| 32 | 23 | 2 | ✓ | 7 791 | 3 004 | 5 | 2 179 | 462 533 | 125 | 3.7 |
| 64 | 31 | 2 | ✓ | 12 741 | 4 169 | 10 | 3 694 | 187 097 | 98.04 | 1.91 |
| 128 | 16 | 1 | ✓ | 14 705 | 4 709 | 19 | 4 121 | 189 897 | 113.64 | 1.67 |
| *KeyGen from Chapter 12* | | | | | | | | | | |
| 32 | – | – | ✗ | 2 074 | 659 | 4 | 649 | 2 671 076 | 131.58 | 20.30 |
| 64 | – | – | ✗ | 4 432 | 1 285 | 5 | 1 285 | 748 964 | 113.64 | 6.59 |
| 128 | – | – | ✗ | 12 654 | 3 554 | 10 | 3 554 | 258 750 | 96.15 | 2.69 |

* The PRNG (Keccak) is used to sample $(h_0, h_1)$ (the core consumes roughly 800 slices).

the Hamming weight of the sparse polynomial exceeds a certain value. More precisely, the latency of the schoolbook multiplier introduced in Chapter 12 is defined by

$$L_{\text{school}} = \left\lceil \frac{r}{b} \right\rceil^2 + 3 \cdot \left\lceil \frac{r}{b} \right\rceil + 1. \tag{13.5}$$

In case $L_{\text{mult}}(th)$ results in a larger latency than $L_{\text{school}}$ for a Hamming weight $th$ and a fixed $\lceil r/b \rceil$, the schoolbook multiplier finishes the corresponding multiplication in fewer clock cycles. In BIKE, this phenomenon only appears for $b = 128$ and for the parameter sets of the security levels 1 and 3. However, especially for $b = 128$ the sparse multiplier achieves a considerably better AT product as shown in Table 13.2.

Besides implementation results for BIKE, Table 13.5 also provides implementation costs and performance values for other code-based cryptographic schemes submitted to the NIST standardization process. As already pointed out in Chapter 12, the comparison to the Classic McEliece implementation is difficult. On the one hand, the reported numbers are only for the PKE scheme and not for the KEM. On the other hand, the Classic McEliece design consumes a huge amount of BRAMs which requires to use larger and more expensive FPGAs.

The hardware design for HQC was recently presented in the latest specification [MAB+21] and is based on a high-level synthesis. While our hardware design of BIKE achieves similar performance results for the encapsulation and decapsulation, HQC has a faster key generation since no polynomial inversion is required.

Eventually, the last part of Table 13.5 reports recent hardware implementation results from other post-quantum schemes which were selected as finalists in the NIST standardization pro-

Table 13.5: Comparison of hardware implementations of post-quantum schemes.

| | Utilization | | | | | Performance | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Logic | | Memory | | Area | Freq. | Key Gen | | Encaps | | Decaps | |
| **Design** | LUT | DSP | FF | BRAM | Slices | MHz | cycles† | $\mu s$ | cycles† | $\mu s$ | cycles† | $\mu s$ |
| *This work, united design* | | | | | | | | | | | | |
| Light weight | 12 319 | 7 | 3 896 | 9 | 3 777 | 121 | 463 | 3 797 | 54 | 443 | 841 | 6 896 |
| Trade-off | 19 607 | 9 | 5 008 | 17 | 5 617 | 100 | 187 | 1 870 | 28 | 280 | 421 | 4 210 |
| High speed | 25 549 | 13 | 5 462 | 34 | 7 332 | 113 | 190 | 1 672 | 15 | 132 | 215 | 1 892 |
| *Chapter 12* | | | | | | | | | | | | |
| Light weight | 12 868 | 7 | 5 354 | 17 | 4 078 | 121 | 2 671 | 21 903 | 153 | 1 252 | 1 628 | 13 349 |
| High speed | 52 967 | 13 | 7 035 | 49 | 15 187 | 96 | 259 | 2 691 | 12 | 127 | 189 | 1 972 |
| *HQC [MAB+21]* | | | | | | | | | | | | |
| Light weight | 8 900 | 0 | 6 400 | 14 | 3 100 | 132 | 630 | 4 773 | 1 500 | 11 364 | 2 100 | 15 909 |
| High speed | 20 000 | 0 | 16 000 | 12.5 | 6 600 | 148 | 40 | 270 | 89 | 601 | 190 | 1 284 |
| *mceliece348864[pke] [WSN18]* | | | | | | | | | | | | |
| Light weight | 25 327 | 0 | 49 383 | 168 | – | 108 | 1 600 | 14 800 | 2.7 | 25.2 | 18.3 | 169.8 |
| High speed | 81 339 | 0 | 132 190 | 236 | – | 106 | 203 | 1 920 | 2.7 | 25.8 | 12.7 | 120.7 |
| *CRYSTALS-KYBER* | | | | | | | | | | | | |
| [XL21] | 7 412 | 2 | 4 644 | 3 | 2 126 | 161 | 3.8 | 23.4 | 5.1 | 30.5 | 6.7 | 41.3 |
| [DMG21] | 9 457 | 4 | 8 543 | 4.5 | – | 220 | 2.2 | 10 | 3.2 | 14.7 | 4.5 | 20.5 |
| *LightSaber [DFA+20]* | | | | | | | | | | | | |
| Light weight | 24 688 | 0 | 14 785 | 1.5 | – | 370 | 1.6 | 4.3 | 2.2 | 5.8 | 2.8 | 7.6 |
| High speed | 65 890 | 0 | 28 230 | 1.5 | – | 310 | 0.9 | 2.9 | 1 | 3.3 | 1.3 | 4.2 |
| *NTRU Prime [Mar20]* | | | | | | | | | | | | |
| – | 9 538 | 19 | 7 803 | 14 | 1 841 | 271 | 1 305 | 4 815 | 142 | 524 | 260 | 958 |

[pke] Results are only for the PKE and not for the KEM. † in thousand.

cess. We list the corresponding implementation costs and performance numbers from lattice-based schemes including CRYSTALS-KYBER, LightSaber, and NTRU Prime. In general, the comparison shows that lattice-based schemes cost less area and achieve lower latencies than the code-based KEM operations.

## 13.5 Discussion

In this section, we briefly discuss the resistance of our implementations against side-channel attacks and address the transferability of our optimization approaches to software implementations.

### 13.5.1 Resistance against Side Channels

In this chapter, we present a constant-time hardware implementation of BIKE which prevents the timing side-channel leakage. However, we did not apply any specific countermeasure against power SCA. In [RMGS20], the authors briefly discussed the resistance of their BIKE hardware implementation against power side channels. They suggested that a parallel processing of $b = 128$ bit chunks makes it hard to identify single-bit dependencies in the power trace. Since our implementation also supports a 128 bit bandwidth, it follows the same argumentation. Additionally, using BIKE with ephemeral keys (suggested as one operation mode in the BIKE specification [ABB+21]), makes a side-channel attack even harder since the attacker can only use single traces.

Nevertheless, this is not a guarantee for resisting power side-channel attacks. For example, analyzing a power trace of our proposed multiplication engine from Section 13.3.3 would probably reveal if an index of $e_0$ or $e_1$ is processed due to the Hamming weight difference of $|1|$ and $|h|$. The multiplication with an index from $e_0$ probably generates different power traces than a multiplication with $e_1$ such that the Hamming weights of $|e_0|$ and $|e_1|$ are leaked. It requires further research to investigate the effect with respect to security from leaking $|e_0|$ and $|e_1|$. The leakage can be avoided by using two sparse multipliers, where one is dedicated to $e_1 \cdot 1$ and the other is dedicated to $e_2 \cdot h$ running in parallel.

### 13.5.2 Transferability to Software

In this section, we discuss the possibility of transferring the presented approaches to software implementations for polynomial inversions and spare polynomial multiplications targeting various platforms.

When considering the inversion algorithms for the key generation, given the latency of extGCD inversion (Equation 13.3) and Fermat's inversion (Equation 13.4), the key issue is the latency of the exponentiation and multiplication ($L_{\text{school}}$) operations in the ITA algorithm on the target platforms. Although the multiplication involves complicated hardware circuits, it is a sunk cost in software when the underlying platform supports related instructions. Therefore, for platforms with native instructions of bit-polynomial multiplication, e.g., the `pclmulqdq` instruction in `x86`, we believe $L_{\text{inv-Fermat}}$ is smaller than $L_{\text{inv}}$. For platforms without instructions for bit-polynomial multiplication, $L_{\text{inv}}$ is likely to be smaller than $L_{\text{inv-Fermat}}$. However, besides the platform, the latency of the multiplication also depends on the implemented algorithms. Recently, Chen et *al.* [CCK21] reported an efficient FFT-based bit-polynomial multiplication on the 32-bit Arm Cortex-M4 platform. Hence, we expect extGCD based inversion outperforms Fermat's inversion in even smaller platforms without efficient multiplication implementations, e.g., 8-bit AVR microcontrollers.

Regarding the sparse polynomial multiplication in BIKE, we mainly consider the side-channel leakage of the degrees of sparse terms. If a software implements the sparse-dense multiplication by accumulating the shifted dense polynomial with the degrees of sparse terms, then it might leak the degrees of sparse terms through a cache-time attack. This is a reason that recent software implementations, e.g., [CCK21, DGK20a], implemented the multiplication with algorithms for dense polynomial multiplication. Thus, we believe that the spare polynomial multiplication will be useful for small microcontrollers without data cache.

## 13.6 Conclusion

In this chapter, we propose various optimization strategies and present an improved hardware design for BIKE, one of the NIST's alternate KEM candidates.

For arithmetic optimizations, we implement a constant-time sparse polynomial multiplier for all three KEM algorithms of BIKE. Compared to a schoolbook implementation, our design improves the area-time product by at least five times for all design parameters. Our implementation also achieves a better latency except for the high-speed design (i.e., $b = 128$) for the encapsulation and the decapsulation. Additionally, we propose a hardware implementation of the polynomial inversion based on the extended Euclidean algorithm. Compared to previous results based on Fermat's little theorem, our new design not only achieves better latency but also provides smaller area-time products for the key generation in BIKE. Moreover, due to its scalable design, the instantiation of the inversion module can be tailored to various circumstances providing higher throughput or smaller area footprints.

Besides these arithmetic optimizations, we show that the random oracles of a unified Keccak core in the new specification of BIKE indeed result in a more efficient hardware design compared to the design using versions of both AES256 and SHA2. Based on our improvements, we developed a united hardware design with shared resources and sub-modules, achieving a better latency with less area compared to previous BIKE implementations. Altogether, our high-speed implementation performs a key generation in 1 672 µs, an encapsulation in 132 µs, and a decapsulation in 1 802 µs on Xilinx Artix-7 FPGAs.

# Part VI

# Conclusion

# Chapter 14

# Conclusion and Future Work

*In this chapter, we briefly summarize our research contributions and conclude this work. Additionally, we discuss future research directions with respect to combined countermeasures, combined gadgets, formal verification, and protected implementations of BIKE.*

**Contents of this Chapter**



## 14.1 Conclusion

In this thesis, we present in the first part countermeasures improving the protection against physical attacks. More precisely, we combine existing countermeasures against SCA with protection mechanisms against FIA. To this end, we first introduce a novel layout of linear ECCs that is adjusted orthogonal to the state-matrix of AES. Additionally, we increase the protection of our approach against SCA by adding LMDPL as state-of-the-art countermeasure. In the second chapter, we utilize the inherent structure of linear ECCs as an opportunity to introduce noise to a cipher's hardware implementation in order to achieve increased protection against SCA. Hence, we attach linear ECCs to a first-order protected TI and dynamically exchange the generator matrices of the underlying codes. Therefore, we increase the protection level given by the first-order TI to higher orders.

In the second part, we introduce adversary models and security notions abstracting and describing physical attacks. We start by revisiting existing fault-injection mechanisms to elaborate underlying physical behavior. Afterwards, we introduce a simple, generic, and consolidated fault-injection adversary model that is perfectly tailored to the physical behavior. Moreover, we revisit composability notions for gadgets against FIA and present a new security notion inspired by the SCA notion PINI. Using existing SCA and FIA composability notions as baseline, we transfer existing composability notions for combined attacks from software to hardware implementations and introduce CINI and ICINI as new combined security notions.

These theoretical considerations are used as fundamental foundation to design computer-aided verification frameworks analyzing the security of countermeasures against physical attacks. Therefore, we first introduce FIVER capable of evaluating countermeasures against FIA. Due to the underlying data structure of BDDs, we can use symbolic simulation while covering all

possible assignments of input variables avoiding false positives. In several case studies, we demonstrate the application of FIVER on real-world designs implementing well-established countermeasures against FIA. Additionally, we introduce the first verification framework that can validate side-channel security, fault-injection resistance, and protection against combined attacks. In extensive case studies, we demonstrate that VERICA can assist a designer of countermeasures by revealing flaws in existing countermeasures from the literature. Moreover, we confirm by practical side-channel measurements that precisely injected faults can decrease the order of the side-channel security of cryptographic primitives constructed from CINI gadgets.

Eventually, we present efficient hardware implementations of the PQC scheme BIKE. We start by presenting the first complete hardware design of BIKE including cores for the key generation, encapsulation, and decapsulation. To this end, we explore different strategies to implement the polynomial inversion based on Fermat's little theorem, introduce an improved polynomial multiplication, and implement the BGF decoder for the first time on hardware. In the subsequent chapter, we improve our previous results and present an optimized polynomial multiplication utilizing the sparseness of one operand and propose a polynomial inversion unit based on the extended Euclidean algorithm.

## 14.2 Future Research Directions

Eventually, we discuss future research directions built upon the findings and contributions of this thesis. Hence, we first present further ideas with respect to countermeasures against physical attacks. Afterwards, we discuss improvements and extensions for our formal verification frameworks. Finally, we briefly explore future work with respect to secure implementations of PQC schemes.

### 14.2.1 Countermeasures against Physical Attacks

In Part II, we discuss two countermeasures that provide protection against SCA and FIA. In this section, we briefly discuss future work addressing similar research directions.

**Combining Provable Secure Masking Schemes with Hiding Techniques for Combined Protection.** In Chapter 7, we present a countermeasure that dynamically exchanges the generator matrices of linear ECCs to achieve higher-order protection against SCA based on a first-order provable secure TI. Implementing provable higher-order countermeasure against SCA is generally connected to a huge overhead with respect to area, latency, and power consumption. Additionally, modern cryptographic implementations should not only be protected against SCA but also against FIA which introduces additional costs. To this end, future work could further focus on combined countermeasures based on provable first-order schemes that provide higher-order protection by dynamic reconfiguration. Here, interesting research directions appear with respect to the quality of the required randomness. For example, for many schemes smaller PRNGs producing randomness of less quality may be good enough to provide the desired protection against SCA.

### 14.2.2 Design of Composable Gadgets

Here, we briefly introduce ideas for new composable gadgets based on findings from Chapter 9 and Chapter 11.

**SIFA Secure Gadgets.** With the introduction of VERICA, we present several case studies where one experiment is dedicated to the analysis of SIFA protected implementations. The resistance in these implementations is achieved by using shared Toffoli gates. However, as pointed out in [HPB21], the generation of protected circuits using shared Toffoli gates is very challenging for larger and more complex designs, e.g., AES S-boxes. To this end, future research could focus on the design of gadgets applying the underlying ideas of the shared Toffoli gates to protect arbitrary circuits against SIFA-based attacks. Please note, our CINI and ICINI gadgets introduced in Chapter 9 also provide protection against SIFA (they correct faults inside the gadget), however, they introduce a huge overhead.

**Improved Combined Gadgets.** Especially this larger overhead introduced by combined gadgets, in general, can serve as a foundation for further future research. In order to be applicable to real-world hardware, gadgets providing protection against combined attacks need further improvements with respect to their required hardware resources.

### 14.2.3 Formal Verification of Hardware Circuits

In this paragraph, we discuss further features and improvements for our formal verification frameworks presented in Part IV.

**Transitional Leakage.** As discussed in Chapter 11, VERICA only supports verifications in the glitch-extended $d$-probing model. However, on hardware devices exist more physical defects like transitions and couplings (cf. Section 2.1.2). Hence, future research could investigate how countermeasures in the transition-extended $d$-probing model could be verified. Therefore, VERICA has to be extended such that round-based designs can be supported which includes to define a new circuit model that is not based on a DAG structure.

**Considering Annotations from Physical Properties.** In order to support verifications considering couplings, VERICA requires more information from the user. Couplings are highly connected to the physical layout of the target design such that routing and layout information are required to perform an appropriate verification. Hence, extending VERICA with the possibility to parse annotated netlist files containing information about the routing or timing, could further improve the verification results with respect to the abstraction of the real-world behavior.

**Apply Concept to Software Designs.** In Chapter 10 and Chapter 11, we present two verification concepts for fault-injection countermeasures and combined countermeasures targeting hardware implementations. A next step could identify how these concepts can be transferred to countermeasures implemented in software targeting microcontroller designs.

**Performance Optimization.** Eventually, as discussed in our case studies, our formal verification frameworks are currently limited with respect to circuit sizes and higher-order verifications (for SCA, FIA, and combined analyses). Therefore, interesting directions for further research are the optimization of our concepts allowing to verify larger and more complex circuits. We already started to increase the performance of VERICA by applying a new approach for SCA verification based on indistinguishability analyses. This work shifts the complexity to the breadth of the target design instead of being limited by the depths of the circuit. The corresponding paper is currently under review [FGG+22].

### 14.2.4 Protected PQC Implementations

Finally, we address future research in the area of secure and protected PQC implementations with respect to physical attacks.

**Protecting BIKE against SCA.** In Part V, we present optimized hardware implementations of BIKE. However, we do not cover the protection against SCA. Hence, future research should address countermeasures against SCA for BIKE. This includes hardware and software implementations likewise. With respect to secure software implementations, we already investigated efficient masked polynomial inversion [KLRG22b] and secure fixed-weight sampling [KLRG22a]. Nevertheless, masking remaining parts of BIKE is still an open challenge.

**Fault-injection Attacks on BIKE.** So far, no work addresses fault-injection attacks on BIKE. Hence, investigating how faults can be used to reveal secret key material in BIKE is still an open research question. In case a successful attack is discovered, corresponding protection mechanisms need to be identified and implemented.

# Part VII

# Appendix

# Chapter 15

# Supplementary Material

## 15.1 Detailed Reports from VerFI Case Study

Table 15.1: Detailed results of the VerFI case study.

| Fault Model | Detected | Non-detected | Ineffective | Scenarios (sum) |
|---|---|---|---|---|
| $\zeta(1, \tau_{bf}, \mathrm{mc}_\infty)$ | 40 | 0 | 0 | 40 |
| $\zeta(2, \tau_{bf}, \mathrm{mc}_\infty)$ | 772 | 0 | 48 | 820 |
| $\zeta(3, \tau_{bf}, \mathrm{mc}_\infty)$ | 10 652 | 0 | 48 | 10 700 |
| $\zeta(4, \tau_{bf}, \mathrm{mc}_\infty)$ | 97 428 | 3 598 | 1 064 | 102 090 |
| $\zeta(1, \tau_{s}, \mathrm{mc}_\infty)$ | 24 | 0 | 16 | 40 |
| $\zeta(2, \tau_{s}, \mathrm{mc}_\infty)$ | 666 | 0 | 154 | 820 |
| $\zeta(3, \tau_{s}, \mathrm{mc}_\infty)$ | 9 710 | 0 | 990 | 10 700 |
| $\zeta(4, \tau_{s}, \mathrm{mc}_\infty)$ | 96 660 | 497 | 4 933 | 102 090 |
| $\zeta(1, \tau_{r}, \mathrm{mc}_\infty)$ | 16 | 0 | 24 | 40 |
| $\zeta(2, \tau_{r}, \mathrm{mc}_\infty)$ | 514 | 0 | 306 | 820 |
| $\zeta(3, \tau_{r}, \mathrm{mc}_\infty)$ | 8 230 | 0 | 2 470 | 10 700 |
| $\zeta(4, \tau_{r}, \mathrm{mc}_\infty)$ | 87 372 | 49 | 14 669 | 102 090 |
| $\zeta(1, \tau_{sr}, \mathrm{c}_7)$ | 8 | 0 | 8 | 16 |
| $\zeta(2, \tau_{sr}, \mathrm{c}_7)$ | 92 | 0 | 36 | 128 |
| $\zeta(3, \tau_{sr}, \mathrm{c}_7)$ | 484 | 0 | 92 | 576 |
| $\zeta(4, \tau_{sr}, \mathrm{c}_7)$ | 1 520 | 14 | 162 | 1 696 |
| $\zeta(1, \tau_{sr}, \mathrm{m})$ | 8 | 0 | 8 | 16 |
| $\zeta(2, \tau_{sr}, \mathrm{m})$ | 92 | 0 | 36 | 128 |
| $\zeta(3, \tau_{sr}, \mathrm{m})$ | 484 | 0 | 92 | 576 |
| $\zeta(4, \tau_{sr}, \mathrm{m})$ | 1 520 | 14 | 162 | 1 696 |
| $\zeta(1, \tau_{nang15}, \mathrm{mc}_\infty)$ | 76 | 0 | 68 | 144 |
| $\zeta(2, \tau_{nang15}, \mathrm{mc}_\infty)$ | 7 720 | 0 | 2 520 | 10 240 |
| $\zeta(3, \tau_{nang15}, \mathrm{mc}_\infty)$ | 405 616 | 0 | 63 824 | 469 440 |
| $\zeta(4, \tau_{nang15}, \mathrm{mc}_\infty)$ | 14 383 842 | 72 462 | 1 245 232 | 15 701 536 |

## 15.2 Performance Results of FIVER

Figure 15.1 shows the evaluation times for different number of cores used by our tool. The results were obtained for a single-round CRAFT design with four bit redundancy under the fault model $\zeta(3, \tau_{bf}, cs)$ and enabled complexity reduction. The memory limit for each CUDD manager was set to 8 GB.

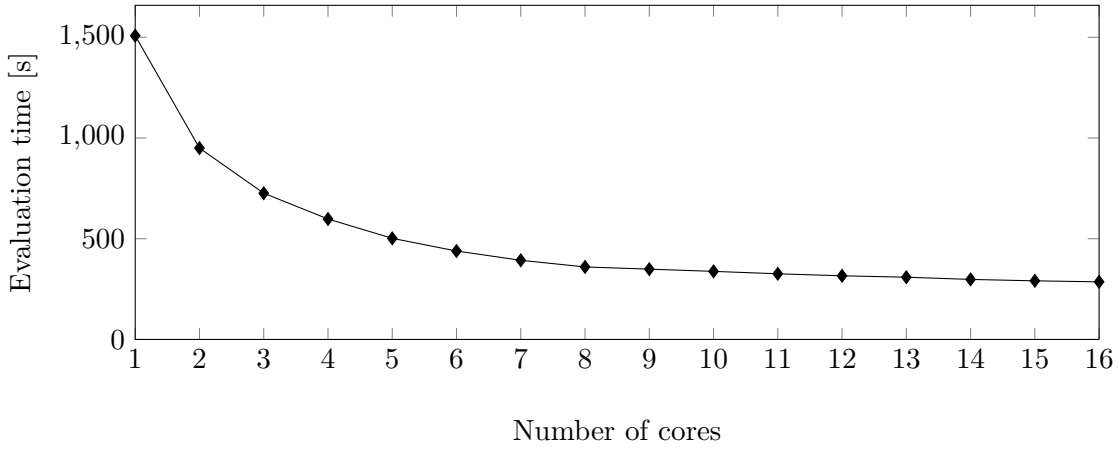



Figure 15.1: Multithreading performance for a single-round CRAFT design with four bit redundancy.

Figure 15.2 shows the evaluation performances for different settings of the memory limit for each BDD manager. The results were obtained for the same design and same fault model as in Figure 15.1 while in this case the number of threads was fixed to eight.

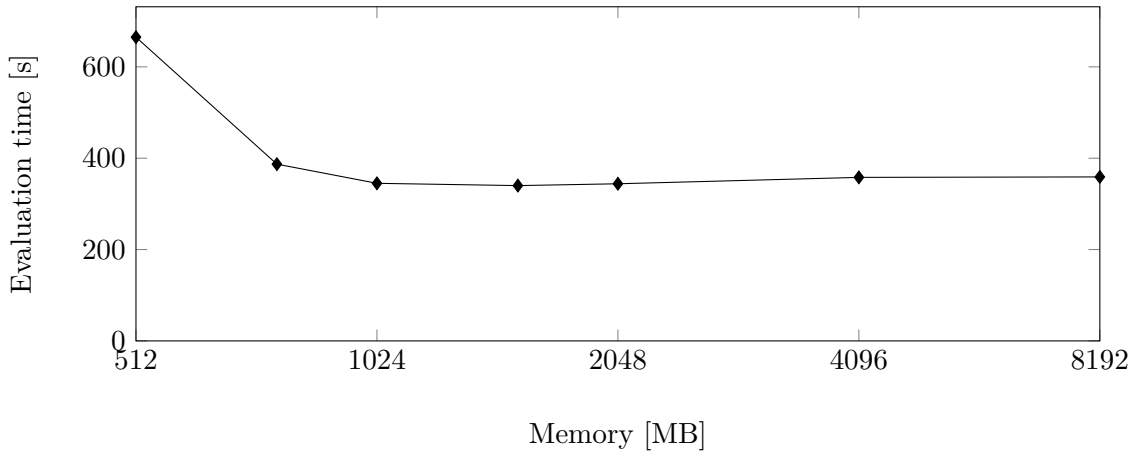



Figure 15.2: Dependency of the memory limit on the performance for an single-round CRAFT design with four bit redundancy.

## 15.3 Multiplication Algorithm for Folding BIKE

Algorithm 15 formally describes our approach to implement the polynomial multiplication. The two initialization phases require each one clock cycle. Everything inside the *for*-loop iterating over $j$ is executed in parallel.

**Algorithm 15** Polynomial Multiplication.

**Require:** Input polynomials $h, m \in \mathcal{R}$.
**Ensure:** Product $c = m \cdot h \in \mathcal{R}$ which is written to a BRAM.

1: $O \leftarrow r \bmod b$, $mask \leftarrow (2^b - 1)$, $addr \leftarrow \lceil r/b \rceil$
2: **for** $i \leftarrow 0$ **to** $addr - 1$ **do**
3: $temp \leftarrow 0$
4: **for** $u \leftarrow O + 1$ **to** $b - 1$ **do** ▷ Initialization Phase 1
5: $temp \leftarrow temp \oplus ((m[i] >> u)\ \&\ 1) \cdot (h[addr - 2] >> (b + O - u))$
6: **end for**
7: $t \leftarrow (h[addr - 1]\ \&\ (2^O - 1)) << (b - O - 1)$; ▷ Initialization Phase 2
8: **for** $u \leftarrow 1$ **to** $b - 1$ **do**
9: $temp \leftarrow temp \oplus ((m[i] >> u)\ \&\ 1) \cdot (t >> (b - 1 - u))$
10: **end for**
11: $h' \leftarrow h[0]$, $tmp\_c\_add \leftarrow c[i]$ ▷ Regular Flow
12: **for** $j \leftarrow 0$ **to** $addr - 1$ **do**
13: $temp2 \leftarrow temp$ ▷ Parallel execution.
14: $temp \leftarrow 0$
15: **for** $u \leftarrow 0$ **to** $b - 1$ **do**
16: $p \leftarrow (((m[i] >> u)\ \&\ 1) \cdot h') << u$
17: $temp2 \leftarrow temp2 \oplus (p\ \&\ mask)$
18: $temp \leftarrow temp \oplus ((p >> b)\ \&\ mask)$
19: **end for**
20: $tmp\_c \leftarrow c[(j + i + 1) \bmod addr]$
21: **if** $j = (addr - 1)$ **then**
22: $c[(j + i + 1) \bmod addr] \leftarrow tmp\_c\_add \oplus \left(temp2\ \&\ \left(2^O - 1\right)\right)$
23: $h[0] \leftarrow ((h' << (b - O))\ |\ (h[j] >> O))\ \&\ mask$
24: **else**
25: $c[(j + i + 1) \bmod addr] \leftarrow tmp\_c\_add \oplus temp2$
26: $tmp\_h \leftarrow h'$
27: $h' \leftarrow h[j + 1]$
28: $h[j + 1] \leftarrow tmp\_h$
29: **end if**
30: $tmp\_c\_add \leftarrow tmp\_c$
31: **end for**
32: **end for**
33: **return** $c$

## 15.4 Additional Implementation Results for Racing BIKE

Table 15.2: Implementation results for the polynomial inversion for $r = 12\,323$, $b = 32$, and $d = 2$. We fixed the frequency to 100 MHz and selected an Artix-7 XC7A200T FPGA as target platform.

| | **Utilization** | | | **Performance** | | |
|---|---|---|---|---|---|---|
| Step Size $s$ | LUT | FF | Slices | Clock Cycles | Latency [ms] | Area-Time |
| $s = 1$ | 580 | 117 | 196 | 9 637 363 | 96.37 | 1 888.92 |
| $s = 2$ | 732 | 168 | 254 | 4 819 461 | 48.19 | 1 224.14 |
| $s = 3$ | 852 | 175 | 296 | 3 221 840 | 32.22 | 953.66 |
| $s = 4$ | 905 | 179 | 313 | 2 416 672 | 24.17 | 756.42 |
| $s = 5$ | 1 131 | 187 | 369 | 1 938 658 | 19.39 | 715.36 |
| $s = 6$ | 1 166 | 193 | 375 | 1 615 612 | 16.16 | 605.85 |
| $s = 7$ | 1 389 | 199 | 458 | 1 388 442 | 13.88 | 635.91 |
| $s = 8$ | 1 493 | 206 | 491 | 1 215 082 | 12.15 | 596.61 |
| $s = 9$ | 1 697 | 346 | 547 | 1 085 811 | 10.86 | 593.94 |
| $s = 10$ | 1 788 | 355 | 576 | 977 307 | 9.77 | 562.93 |
| $s = 11$ | 1 929 | 366 | 601 | 890 844 | 8.91 | 535.40 |
| $s = 12$ | 1 808 | 373 | 578 | 816 606 | 8.17 | 472.00 |
| $s = 13$ | 1 977 | 383 | 613 | 755 776 | 7.56 | 463.29 |
| $s = 14$ | 2 063 | 393 | 639 | 702 045 | 7.02 | 448.61 |
| $s = 15$ | 2 245 | 403 | 685 | 657 123 | 6.57 | 450.13 |
| $s = 16$ | 2 479 | 414 | 759 | 616 026 | 6.16 | 467.56 |
| $s = 17$ | 2 526 | 557 | 767 | 582 617 | 5.83 | 446.87 |
| $s = 18$ | 2 619 | 570 | 823 | 550 536 | 5.51 | 453.09 |
| $s = 19$ | 2 765 | 585 | 847 | 522 962 | 5.23 | 442.95 |
| $s = 20$ | 2 905 | 601 | 893 | 496 832 | 4.97 | 443.67 |
| $s = 21$ | 3 068 | 613 | 934 | 474 289 | 4.74 | 442.99 |
| $s = 22$ | 3 232 | 631 | 999 | 452 929 | 4.53 | 452.48 |
| $s = 23$ | 3 359 | 643 | 995 | 434 255 | 4.34 | 432.08 |
| $s = 24$ | 3 679 | 655 | 1 105 | 416 076 | 4.16 | 459.76 |
| $s = 25$ | 3 705 | 802 | 1 096 | 401 483 | 4.01 | 440.03 |
| $s = 26$ | 3 869 | 819 | 1 191 | 386 055 | 3.86 | 459.79 |
| $s = 27$ | 3 998 | 840 | 1 177 | 372 758 | 3.73 | 438.74 |
| $s = 28$ | 4 149 | 865 | 1 287 | 359 732 | 3.60 | 462.98 |
| $s = 29$ | 4 411 | 877 | 1 342 | 347 967 | 3.48 | 466.97 |
| $s = 30$ | 4 549 | 900 | 1 350 | 336 542 | 3.37 | 454.33 |
| $s = 31$ | 4 735 | 921 | 1 410 | 326 729 | 3.27 | 460.69 |
| $s = 32$ | 5 038 | 943 | 1 473 | 316 504 | 3.17 | 466.21 |

Table 15.3: Implementation results for the polynomial inversion for $r = 12\,323$, $b = 64$, and $d = 2$. We fixed the frequency to 100 MHz and selected an Artix-7 XC7A200T FPGA as target platform.

| | **Utilization** | | | **Performance** | | |
|---|---|---|---|---|---|---|
| Step Size $s$ | LUT | FF | Slices | Clock Cycles | Latency [ms] | Area-Time |
| $s = 1$ | 1 020 | 183 | 377 | 4 880 299 | 48.80 | 3 679.75 |
| $s = 2$ | 1 245 | 296 | 425 | 2 440 543 | 24.41 | 1 037.23 |
| $s = 3$ | 1 662 | 306 | 566 | 1 635 573 | 16.36 | 925.73 |
| $s = 4$ | 1 540 | 312 | 515 | 1 226 827 | 12.27 | 631.82 |
| $s = 5$ | 1 890 | 322 | 618 | 986 589 | 9.87 | 609.71 |
| $s = 6$ | 1 947 | 327 | 676 | 822 189 | 8.22 | 555.80 |
| $s = 7$ | 2 391 | 334 | 786 | 708 310 | 7.08 | 556.73 |
| $s = 8$ | 2 637 | 348 | 893 | 619 870 | 6.20 | 553.54 |
| $s = 9$ | 2 633 | 613 | 878 | 556 605 | 5.57 | 488.70 |
| $s = 10$ | 2 865 | 629 | 922 | 500 983 | 5.01 | 461.91 |
| $s = 11$ | 3 045 | 632 | 1 015 | 457 752 | 4.58 | 464.62 |
| $s = 12$ | 3 208 | 645 | 1 021 | 419 605 | 4.20 | 428.42 |
| $s = 13$ | 3 444 | 658 | 1 122 | 389 269 | 3.89 | 436.76 |
| $s = 14$ | 3 623 | 665 | 1 201 | 361 593 | 3.62 | 434.27 |
| $s = 15$ | 3 964 | 685 | 1 245 | 339 252 | 3.39 | 422.37 |
| $s = 16$ | 4 506 | 704 | 1 422 | 318 034 | 3.18 | 452.24 |
| $s = 17$ | 4 429 | 969 | 1 369 | 302 188 | 3.02 | 413.70 |
| $s = 18$ | 4 537 | 985 | 1 417 | 285 547 | 2.86 | 404.62 |
| $s = 19$ | 4 697 | 992 | 1 440 | 271 869 | 2.72 | 391.49 |
| $s = 20$ | 4 975 | 1 010 | 1 566 | 258 284 | 2.58 | 404.47 |
| $s = 21$ | 5 439 | 1 030 | 1 657 | 247 128 | 2.47 | 409.49 |
| $s = 22$ | 5 476 | 1 046 | 1 758 | 235 997 | 2.36 | 414.88 |
| $s = 23$ | 5 804 | 1 064 | 1 766 | 226 780 | 2.27 | 400.49 |
| $s = 24$ | 6 323 | 1 095 | 1 892 | 217 286 | 2.17 | 411.11 |
| $s = 25$ | 6 280 | 1 353 | 1 928 | 210 606 | 2.11 | 406.05 |
| $s = 26$ | 6 724 | 1 383 | 2 064 | 202 512 | 2.03 | 417.98 |
| $s = 27$ | 6 769 | 1 390 | 2 034 | 195 970 | 1.96 | 398.60 |
| $s = 28$ | 6 862 | 1 407 | 2 080 | 189 120 | 1.89 | 393.37 |
| $s = 29$ | 7 517 | 1 442 | 2 197 | 183 338 | 1.83 | 402.79 |
| $s = 30$ | 7 733 | 1 462 | 2 281 | 177 317 | 1.77 | 404.46 |
| $s = 31$ | 7 801 | 1 473 | 2 269 | 172 522 | 1.73 | 391.45 |
| $s = 32$ | 8 379 | 1 500 | 2 560 | 167 122 | 1.67 | 427.83 |
| $s = 33$ | 8 324 | 1 772 | 2 425 | 163 434 | 1.63 | 396.33 |
| $s = 34$ | 8 339 | 1 799 | 2 480 | 158 638 | 1.59 | 393.42 |
| $s = 35$ | 8 687 | 1 814 | 2 621 | 154 980 | 1.55 | 406.20 |
| $s = 36$ | 9 016 | 1 836 | 2 692 | 150 602 | 1.51 | 405.42 |
| $s = 37$ | 9 288 | 1 860 | 2 735 | 147 325 | 1.47 | 402.93 |
| $s = 38$ | 9 552 | 1 882 | 2 871 | 143 367 | 1.43 | 411.61 |
| $s = 39$ | 9 909 | 1 911 | 2 851 | 140 261 | 1.40 | 399.88 |
| $s = 40$ | 10 426 | 1 945 | 3 090 | 136 942 | 1.37 | 423.15 |
| $s = 41$ | 10 374 | 2 227 | 3 092 | 134 830 | 1.35 | 416.89 |
| $s = 42$ | 10 751 | 2 260 | 3 202 | 131 489 | 1.31 | 421.03 |
| $s = 43$ | 10 989 | 2 282 | 3 247 | 129 160 | 1.29 | 419.38 |
| $s = 44$ | 11 233 | 2 310 | 3 315 | 126 248 | 1.26 | 418.51 |
| $s = 45$ | 11 737 | 2 357 | 3 417 | 123 887 | 1.24 | 423.32 |
| $s = 46$ | 11 853 | 2 379 | 3 419 | 121 188 | 1.21 | 414.34 |
| $s = 47$ | 12 516 | 2 430 | 3 694 | 119 236 | 1.19 | 440.46 |
| $s = 48$ | 12 758 | 2 459 | 3 623 | 116 750 | 1.17 | 422.99 |
| $s = 49$ | 12 960 | 2 740 | 3 673 | 115 272 | 1.15 | 423.39 |
| $s = 50$ | 13 305 | 2 777 | 3 873 | 112 992 | 1.13 | 437.62 |
| $s = 51$ | 14 176 | 2 818 | 4 211 | 111 420 | 1.11 | 469.19 |
| $s = 52$ | 13 762 | 2 837 | 3 973 | 109 135 | 1.09 | 433.59 |
| $s = 53$ | 15 295 | 2 882 | 4 397 | 107 763 | 1.08 | 473.83 |
| $s = 54$ | 14 616 | 2 903 | 4 200 | 105 695 | 1.06 | 443.92 |
| $s = 55$ | 16 613 | 2 975 | 4 825 | 104 302 | 1.04 | 503.26 |
| $s = 56$ | 16 031 | 3 002 | 4 701 | 102 454 | 1.02 | 481.64 |
| $s = 57$ | 15 994 | 3 289 | 4 618 | 101 473 | 1.01 | 468.60 |
| $s = 58$ | 16 445 | 3 326 | 4 780 | 99 613 | 1.00 | 476.15 |
| $s = 59$ | 16 491 | 3 345 | 4 697 | 98 399 | 0.98 | 462.18 |
| $s = 60$ | 17 232 | 3 397 | 5 005 | 96 761 | 0.97 | 484.29 |
| $s = 61$ | 17 711 | 3 435 | 5 116 | 95 757 | 0.96 | 489.89 |
| $s = 62$ | 17 171 | 3 462 | 4 984 | 94 115 | 0.94 | 469.07 |
| $s = 63$ | 18 103 | 3 510 | 5 318 | 93 095 | 0.93 | 494.15 |
| $s = 64$ | 18 610 | 3 563 | 5 457 | 91 678 | 0.92 | 500.29 |

Table 15.4: Implementation results for the polynomial inversion for $r = 12\,323$, $b = 128$, and $d = 1$. We fixed the frequency to 100 MHz and selected an Artix-7 XC7A200T FPGA as target platform.

| | **Utilization** | | | **Performance** | | |
|---|---|---|---|---|---|---|
| Step Size $s$ | LUT | FF | Slices | Clock Cycles | Latency [ms] | Area-Time |
| $s = 1$ | 1 805 | 247 | 671 | 2 514 091 | 25.14 | 16 869.55 |
| $s = 2$ | 2 134 | 517 | 801 | 1 269 570 | 12.70 | 1 016.93 |
| $s = 3$ | 2 519 | 527 | 944 | 854 765 | 8.55 | 806.90 |
| $s = 4$ | 2 880 | 542 | 1 030 | 647 311 | 6.47 | 666.73 |
| $s = 5$ | 3 257 | 553 | 1 158 | 522 881 | 5.23 | 605.50 |
| $s = 6$ | 3 616 | 565 | 1 224 | 439 857 | 4.40 | 538.38 |
| $s = 7$ | 4 239 | 578 | 1 393 | 380 569 | 3.81 | 530.13 |
| $s = 8$ | 4 496 | 587 | 1 498 | 336 130 | 3.36 | 503.52 |
| $s = 9$ | 4 857 | 1 117 | 1 618 | 304 329 | 3.04 | 492.40 |
| $s = 10$ | 5 315 | 1 139 | 1 689 | 276 380 | 2.76 | 466.81 |
| $s = 11$ | 5 615 | 1 157 | 1 807 | 253 533 | 2.54 | 458.13 |
| $s = 12$ | 6 083 | 1 170 | 1 936 | 234 457 | 2.34 | 453.91 |
| $s = 13$ | 6 286 | 1 183 | 2 001 | 218 341 | 2.18 | 436.90 |
| $s = 14$ | 6 866 | 1 202 | 2 211 | 204 576 | 2.05 | 452.32 |
| $s = 15$ | 7 284 | 1 215 | 2 299 | 192 648 | 1.93 | 442.90 |
| $s = 16$ | 8 322 | 1 245 | 2 560 | 182 138 | 1.82 | 466.27 |
| $s = 17$ | 7 860 | 1 758 | 2 407 | 174 300 | 1.74 | 419.54 |
| $s = 18$ | 8 398 | 1 787 | 2 561 | 166 069 | 1.66 | 425.30 |
| $s = 19$ | 8 553 | 1 792 | 2 623 | 158 655 | 1.59 | 416.15 |
| $s = 20$ | 9 161 | 1 824 | 2 903 | 151 958 | 1.52 | 441.13 |
| $s = 21$ | 9 392 | 1 834 | 2 877 | 145 876 | 1.46 | 419.69 |
| $s = 22$ | 10 000 | 1 866 | 3 197 | 140 424 | 1.40 | 448.94 |
| $s = 23$ | 10 510 | 1 881 | 3 174 | 135 372 | 1.35 | 429.67 |
| $s = 24$ | 11 576 | 1 930 | 3 487 | 130 730 | 1.31 | 455.86 |
| $s = 25$ | 11 088 | 2 427 | 3 370 | 127 494 | 1.27 | 429.65 |
| $s = 26$ | 11 836 | 2 471 | 3 569 | 123 540 | 1.24 | 440.91 |
| $s = 27$ | 12 205 | 2 488 | 3 742 | 119 903 | 1.20 | 448.68 |
| $s = 28$ | 12 346 | 2 507 | 3 816 | 116 590 | 1.17 | 444.91 |
| $s = 29$ | 13 172 | 2 550 | 3 999 | 113 350 | 1.13 | 453.29 |
| $s = 30$ | 13 565 | 2 567 | 4 046 | 110 447 | 1.10 | 446.87 |
| $s = 31$ | 14 739 | 2 612 | 4 512 | 107 758 | 1.08 | 486.20 |
| $s = 32$ | 14 632 | 2 624 | 4 334 | 105 154 | 1.05 | 455.74 |
| $s = 33$ | 14 854 | 3 162 | 4 382 | 103 386 | 1.03 | 453.04 |
| $s = 34$ | 15 394 | 3 201 | 4 547 | 101 075 | 1.01 | 459.59 |
| $s = 35$ | 17 161 | 3 249 | 5 225 | 98 997 | 0.99 | 517.26 |
| $s = 36$ | 16 025 | 3 238 | 4 864 | 96 884 | 0.97 | 471.24 |
| $s = 37$ | 17 789 | 3 314 | 5 295 | 95 011 | 0.95 | 503.08 |
| $s = 38$ | 17 007 | 3 295 | 5 113 | 93 106 | 0.93 | 476.05 |
| $s = 39$ | 18 682 | 3 365 | 5 590 | 91 309 | 0.91 | 510.42 |
| $s = 40$ | 18 997 | 3 386 | 5 560 | 89 762 | 0.90 | 499.08 |
| $s = 41$ | 20 059 | 3 956 | 5 853 | 88 790 | 0.89 | 519.69 |
| $s = 42$ | 19 581 | 3 953 | 5 776 | 87 176 | 0.87 | 503.53 |
| $s = 43$ | 20 206 | 4 000 | 5 901 | 85 822 | 0.86 | 506.44 |
| $s = 44$ | 20 562 | 4 037 | 6 044 | 84 446 | 0.84 | 510.39 |
| $s = 45$ | 21 092 | 4 052 | 6 158 | 83 047 | 0.83 | 511.40 |
| $s = 46$ | 21 587 | 4 076 | 6 390 | 81 772 | 0.82 | 522.52 |
| $s = 47$ | 23 411 | 4 157 | 6 754 | 80 623 | 0.81 | 544.53 |
| $s = 48$ | 23 283 | 4 189 | 6 851 | 79 454 | 0.79 | 544.34 |
| $s = 49$ | 22 459 | 4 701 | 6 597 | 78 768 | 0.79 | 519.63 |
| $s = 50$ | 23 571 | 4 742 | 6 772 | 77 701 | 0.78 | 526.19 |
| $s = 51$ | 24 722 | 4 799 | 7 379 | 76 768 | 0.77 | 566.47 |
| $s = 52$ | 24 364 | 4 821 | 7 173 | 75 667 | 0.76 | 542.76 |
| $s = 53$ | 25 137 | 4 864 | 7 296 | 74 855 | 0.75 | 546.14 |
| $s = 54$ | 25 516 | 4 887 | 7 366 | 73 874 | 0.74 | 544.16 |
| $s = 55$ | 26 330 | 4 932 | 7 765 | 73 033 | 0.73 | 567.10 |
| $s = 56$ | 27 413 | 4 995 | 8 144 | 72 178 | 0.72 | 587.82 |
| $s = 57$ | 28 143 | 5 522 | 8 223 | 71 741 | 0.72 | 589.93 |
| $s = 58$ | 28 594 | 5 594 | 8 151 | 70 850 | 0.71 | 577.50 |
| $s = 59$ | 28 171 | 5 587 | 8 310 | 70 105 | 0.70 | 582.57 |
| $s = 60$ | 28 393 | 5 640 | 8 209 | 69 347 | 0.69 | 569.27 |
| $s = 61$ | 29 925 | 5 696 | 8 545 | 68 739 | 0.69 | 587.37 |
| $s = 62$ | 30 319 | 5 734 | 8 769 | 67 957 | 0.68 | 595.91 |
| $s = 63$ | 30 852 | 5 775 | 8 847 | 67 327 | 0.67 | 595.64 |
| $s = 64$ | 32 695 | 5 864 | 9 527 | 66 686 | 0.67 | 635.32 |

Table 15.5: Implementation results for the polynomial inversion for $r = 12\,323$, $b = 128$, and $d = 1$. We fixed the frequency to 100 MHz and selected an Artix-7 XC7A200T FPGA as target platform.

| | **Utilization** | | | **Performance** | | |
|---|---|---|---|---|---|---|
| Step Size $s$ | LUT | FF | Slices | Clock Cycles | Latency [ms] | Area-Time |
| $s = 65$ | 31 221 | 6 352 | 8 847 | 66 414 | 0.66 | 587.56 |
| $s = 66$ | 31 436 | 6 389 | 9 158 | 65 746 | 0.66 | 602.10 |
| $s = 67$ | 30 577 | 6 397 | 8 972 | 65 067 | 0.65 | 583.78 |
| $s = 68$ | 32 594 | 6 503 | 9 607 | 64 547 | 0.65 | 620.10 |
| $s = 69$ | 35 630 | 6 620 | 10 104 | 64 018 | 0.64 | 646.84 |
| $s = 70$ | 33 656 | 6 572 | 9 761 | 63 480 | 0.63 | 619.63 |
| $s = 71$ | 35 061 | 6 638 | 10 077 | 62 933 | 0.63 | 634.18 |
| $s = 72$ | 36 133 | 6 719 | 10 485 | 62 378 | 0.62 | 654.03 |
| $s = 73$ | 36 613 | 7 276 | 10 416 | 62 151 | 0.62 | 647.36 |
| $s = 74$ | 38 766 | 7 388 | 11 314 | 61 748 | 0.62 | 698.62 |
| $s = 75$ | 38 813 | 7 410 | 11 239 | 61 162 | 0.61 | 687.40 |
| $s = 76$ | 37 402 | 7 400 | 10 770 | 60 744 | 0.61 | 654.21 |
| $s = 77$ | 39 655 | 7 480 | 11 572 | 60 319 | 0.60 | 698.01 |
| $s = 78$ | 38 221 | 7 473 | 11 045 | 59 709 | 0.60 | 659.49 |
| $s = 79$ | 39 225 | 7 561 | 11 016 | 59 269 | 0.59 | 652.91 |
| $s = 80$ | 41 260 | 7 653 | 11 871 | 59 002 | 0.59 | 700.41 |
| $s = 81$ | 41 334 | 8 206 | 11 838 | 58 853 | 0.59 | 696.70 |
| $s = 82$ | 40 519 | 8 219 | 11 344 | 58 389 | 0.58 | 662.36 |
| $s = 83$ | 41 921 | 8 315 | 11 794 | 57 918 | 0.58 | 683.08 |
| $s = 84$ | 42 303 | 8 364 | 12 025 | 57 625 | 0.58 | 692.94 |
| $s = 85$ | 43 112 | 8 427 | 12 378 | 57 140 | 0.57 | 707.28 |
| $s = 86$ | 44 150 | 8 471 | 12 741 | 56 836 | 0.57 | 724.15 |
| $s = 87$ | 45 026 | 8 552 | 13 084 | 56 525 | 0.57 | 739.57 |
| $s = 88$ | 46 817 | 8 654 | 13 632 | 56 210 | 0.56 | 766.25 |
| $s = 89$ | 46 236 | 9 187 | 13 206 | 55 977 | 0.56 | 739.23 |
| $s = 90$ | 46 659 | 9 248 | 13 038 | 55 647 | 0.56 | 725.53 |
| $s = 91$ | 48 035 | 9 326 | 13 776 | 55 312 | 0.55 | 761.98 |
| $s = 92$ | 46 928 | 9 348 | 13 154 | 54 972 | 0.55 | 723.10 |
| $s = 93$ | 49 552 | 9 461 | 14 406 | 54 820 | 0.55 | 789.74 |
| $s = 94$ | 51 236 | 9 514 | 14 461 | 54 470 | 0.54 | 787.69 |
| $s = 95$ | 48 958 | 9 518 | 13 927 | 54 114 | 0.54 | 753.65 |
| $s = 96$ | 51 575 | 9 644 | 14 776 | 53 754 | 0.54 | 794.27 |
| $s = 97$ | 52 032 | 10 221 | 14 911 | 53 839 | 0.54 | 802.79 |
| $s = 98$ | 52 507 | 10 290 | 14 760 | 53 466 | 0.53 | 789.16 |
| $s = 99$ | 54 072 | 10 373 | 15 308 | 53 088 | 0.53 | 812.67 |
| $s = 100$ | 52 470 | 10 375 | 14 946 | 52 905 | 0.53 | 790.72 |
| $s = 101$ | 54 968 | 10 517 | 15 569 | 52 719 | 0.53 | 820.78 |
| $s = 102$ | 54 428 | 10 506 | 15 225 | 52 326 | 0.52 | 796.66 |
| $s = 103$ | 57 494 | 10 651 | 16 235 | 52 132 | 0.52 | 846.36 |
| $s = 104$ | 58 278 | 10 710 | 16 475 | 51 730 | 0.52 | 852.25 |
| $s = 105$ | 57 104 | 11 241 | 15 935 | 51 762 | 0.52 | 824.83 |
| $s = 106$ | 57 155 | 11 314 | 16 171 | 51 554 | 0.52 | 833.68 |
| $s = 107$ | 58 343 | 11 416 | 16 257 | 51 343 | 0.51 | 834.68 |
| $s = 108$ | 58 391 | 11 443 | 16 525 | 51 128 | 0.51 | 844.89 |
| $s = 109$ | 59 464 | 11 547 | 16 778 | 50 910 | 0.51 | 854.17 |
| $s = 110$ | 59 938 | 11 563 | 16 937 | 50 688 | 0.51 | 858.50 |
| $s = 111$ | 60 228 | 11 677 | 16 895 | 50 463 | 0.50 | 852.57 |
| $s = 112$ | 63 192 | 11 787 | 17 473 | 50 234 | 0.50 | 877.74 |
| $s = 113$ | 60 735 | 12 262 | 16 590 | 50 220 | 0.50 | 833.15 |
| $s = 114$ | 61 661 | 12 364 | 17 550 | 49 982 | 0.50 | 877.18 |
| $s = 115$ | 64 408 | 12 476 | 18 190 | 49 741 | 0.50 | 904.79 |
| $s = 116$ | 63 069 | 12 475 | 17 966 | 49 496 | 0.49 | 889.25 |
| $s = 117$ | 63 678 | 12 579 | 17 887 | 49 248 | 0.49 | 880.90 |
| $s = 118$ | 67 589 | 12 727 | 18 588 | 48 996 | 0.49 | 910.74 |
| $s = 119$ | 64 989 | 12 706 | 18 025 | 48 960 | 0.49 | 882.50 |
| $s = 120$ | 66 836 | 12 784 | 18 754 | 48 702 | 0.49 | 913.36 |
| $s = 121$ | 69 566 | 13 467 | 19 483 | 48 644 | 0.49 | 947.73 |
| $s = 122$ | 67 738 | 13 479 | 18 871 | 48 600 | 0.49 | 917.13 |
| $s = 123$ | 72 170 | 13 641 | 20 145 | 48 330 | 0.48 | 973.61 |
| $s = 124$ | 71 966 | 13 736 | 20 013 | 48 057 | 0.48 | 961.76 |
| $s = 125$ | 73 447 | 13 795 | 20 338 | 48 006 | 0.48 | 976.35 |
| $s = 126$ | 72 494 | 13 814 | 19 953 | 47 727 | 0.48 | 952.30 |
| $s = 127$ | 72 596 | 13 900 | 20 096 | 47 671 | 0.48 | 958.00 |
| $s = 128$ | 75 269 | 14 028 | 21 435 | 47 386 | 0.47 | 1 015.72 |

Table 15.6: Implementation results of the united hardware design presented in Chapter 13 for Level 3 and Level 5.

| | **Utilization** | | | | | **Performance** | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Logic | | Memory | | Area | Frequency | Key Gen | | Encaps | | Decaps | |
| **Design** | LUT | DSP | FF | BRAM | Slices | MHz | cycles† | $\mu s$ | cycles† | $\mu s$ | cycles† | $\mu s$ |
| *United design for* $r = 24\,659$ | | | | | | | | | | | | |
| Low weight | 13 850 | 7 | 4 010 | 15 | 4 152 | 116 | 1 775 | 15 268 | 157 | 1 348 | 2 381 | 20 479 |
| Trade-off | 20 049 | 9 | 5 039 | 17 | 5 688 | 100 | 693 | 6 929 | 80 | 801 | 1 198 | 11 982 |
| High speed | 25 811 | 13 | 5 460 | 34 | 7 242 | 113 | 681 | 5 997 | 42 | 367 | 605 | 5 325 |
| *United design for* $r = 40\,973$ | | | | | | | | | | | | |
| Low weight | 13 973 | 7 | 4 002 | 34 | 4 192 | 113 | 4 809 | 42 324 | 343 | 3 020 | 5 217 | 45 911 |
| Trade-off | 21 373 | 9 | 5 160 | 34 | 6 145 | 94 | 1 847 | 19 580 | 174 | 1 847 | 2 620 | 27 770 |
| High speed | 26 441 | 13 | 5 601 | 34 | 7 288 | 111 | 1 798 | 16 186 | 90 | 808 | 1 321 | 11 885 |

† in thousand.

# Bibliography


[AASA+19] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. *Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process*. US Department of Commerce, National Institute of Standards and Technology, 2019. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=927303.

[ABB+19] Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, France Worldline, Loïc Bidoux, Olivier Blazy, Philippe Gaborit, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, et al. BIKE: Bit Flipping Key Encapsulation - Round 2 Submission. 2019. https://bikesuite.org/files/round2/spec/BIKE-Spec-Round2.2019.03.30.pdf.

[ABB+20a] Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, France Worldline, Loïc Bidoux, Olivier Blazy, Philippe Gaborit, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, et al. BIKE: Bit Flipping Key Encapsulation - Round 3 Submission. 2020. https://bikesuite.org/files/round2/spec/BIKE-Spec-2020.02.07.1.pdf.

[ABB+20b] Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, France Worldline, Loïc Bidoux, Olivier Blazy, Philippe Gaborit, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, et al. BIKE: Bit Flipping Key Encapsulation - Submission for Round 3 Consideration. 2020. https://bikesuite.org/files/v4.0/BIKE_Spec.2020.05.03.1.pdf.

[ABB+21] Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, France Worldline, Loïc Bidoux, Olivier Blazy, Philippe Gaborit, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, et al. BIKE: Bit Flipping Key Encapsulation. 2021. https://bikesuite.org/files/v4.2/BIKE_Spec.2021.07.26.1.pdf.

[ABB+22] Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, France Worldline, Loïc Bidoux, Olivier Blazy, Philippe Gaborit, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, et al. BIKE: Bit Flipping Key Encapsulation - Round 4 Submission. 2022. https://bikesuite.org/files/v5.0/BIKE_Spec.2022.10.10.1.pdf.

[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 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2017.

[ADN+10] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When Clocks Fail: On Critical Paths and Clock Faults. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau, Germany, April 14-16, 2010. Proceedings*, volume 6035 of *Lecture Notes in Computer Science*, pages 182–193. Springer, 2010.

[AIS18] Prabhanjan Ananth, Yuval Ishai, and Amit Sahai. Private Circuits: A Modular Approach. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 427–455. Springer, 2018.

[AM11] Subidh Ali and Debdeep Mukhopadhyay. A Differential Fault Analysis on AES Key Schedule Using Single Fault. In Luca Breveglieri, Sylvain Guilley, Israel Koren, David Naccache, and Junko Takahashi, editors, *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011*, pages 35–42. IEEE Computer Society, 2011.

[AMR+20] Anita Aghaie, Amir Moradi, Shahram Rasoolzadeh, Aein Rezaei Shahmirzadi, Falk Schellenberg, and Tobias Schneider. Impeccable Circuits. *IEEE Trans. Computers*, 69(3):361–376, 2020.

[ANR18] Victor Arribas, Svetla Nikova, and Vincent Rijmen. VerMI: Verification Tool for Masked Implementations. In *25th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2018, Bordeaux, France, December 9-12, 2018*, pages 381–384. IEEE, 2018.

[AWMN20] Victor Arribas, Felix Wegener, Amir Moradi, and Svetla Nikova. Cryptographic Fault Diagnosis using VerFI. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020*, pages 229–240. IEEE, 2020.

[Bau04] Robert C Baumann. Soft Errors in Commercial Integrated Circuits. *International Journal of High Speed Electronics and Systems*, 14(02):299–309, 2004.

[BBB+21] Anubhab Baksi, Shivam Bhasin, Jakub Breier, Anupam Chattopadhyay, and Vinay B. Y. Kumar. Feeding Three Birds With One Scone: A Generic Duplication Based Countermeasure To Fault Attacks. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*, pages 561–564. IEEE, 2021.

[BBC+19a] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. LEDAcrypt: QC-LDPC Code-Based Cryptosystems with Bounded Decryption Failure Rate. In *Code-Based Cryptography - 7th International Workshop*, volume 11666 of *Lecture Notes in Computer Science*, pages 11–43. Springer, 2019.

[BBC+19b] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskVerif: Automated Verification of

Higher-Order Masking in Presence of Physical Defaults. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019.

[BBD+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified Proofs of Higher-Order Masking. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 457–485. Springer, 2015.

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong Non-Interference and Type-Directed Higher-Order Masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129. ACM, 2016.

[BBK+03] Guido Bertoni, Luca Breveglieri, Israel Koren, Paolo Maistri, and Vincenzo Piuri. Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard. *IEEE Trans. Computers*, 52(4):492–505, 2003.

[BBKM04] Guido Bertoni, Luca Breveglieri, Israel Koren, and Paolo Maistri. An Efficient Hardware-Based Fault Diagnosis Scheme for AES: Performances and Cost. In *19th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2004), 10-13 October 2004, Cannes, France, Proceedings*, pages 130–138. IEEE Computer Society, 2004.

[BCC+14] Julien Bringer, Claude Carlet, Hervé Chabanne, Sylvain Guilley, and Houssem Maghrebi. Orthogonal Direct Sum Masking - A Smartcard Friendly Computation Paradigm in a Code, with Builtin Protection against Side-Channel and Fault Attacks. In David Naccache and Damien Sauveron, editors, *Information Security Theory and Practice. Securing the Internet of Things - 8th IFIP WG 11.2 International Workshop, WISTP 2014, Heraklion, Crete, Greece, June 30 - July 2, 2014. Proceedings*, volume 8501 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2014.

[BCN+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proc. IEEE*, 94(2):370–382, 2006.

[BCP+20] Sonia Belaïd, Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Abdul Rahman Taleb. Random Probing Security: Verification, Composition, Expansion and New Constructions. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020,*

*Proceedings, Part I*, volume 12170 of *Lecture Notes in Computer Science*, pages 339–368. Springer, 2020.

[BCPZ16] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal Side-Channel Attacks and Countermeasures on the ISW Masking Scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 23–39. Springer, 2016.

[BDM+20] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part III*, volume 12107 of *Lecture Notes in Computer Science*, pages 311–341. Springer, 2020.

[BDPA13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 313–314. Springer, 2013.

[BFG+19] Alessandro Barenghi, William Fornaciari, Andrea Galimberti, Gerardo Pelosi, and Davide Zoni. Evaluating the Trade-offs in the Hardware Design of the LEDAcrypt Encryption Functions. In *26th IEEE International Conference on Electronics, Circuits and Systems*, pages 739–742. IEEE, 2019.

[BGE+17] Jan Burchard, Mael Gay, Ange-Salomé Messeng Ekossono, Jan Horácek, Bernd Becker, Tobias Schubert, Martin Kreuzer, and Ilia Polian. AutoFault: Towards Automatic Construction of Algebraic Fault Attacks. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017*, pages 65–72. IEEE Computer Society, 2017.

[BGI+18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal Verification of Masked Hardware Implementations in the Presence of Glitches. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 321–353. Springer, 2018.

[BGR18] Sonia Belaïd, Dahmun Goudarzi, and Matthieu Rivain. Tight Private Circuits: Achieving Probing Security with the Least Refreshing. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th*

*International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*, volume 11273 of *Lecture Notes in Computer Science*, pages 343–372. Springer, 2018.

[BK18] Raik Brinkmann and Dave Kelf. *Formal System Verification*, chapter Formal Verification - The Industrial Perspective, pages 155–182. Springer, 2018.

[BKH+19] Arthur Beckers, Masahiro Kinugawa, Yu-ichi Hayashi, Daisuke Fujimoto, Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. Design Considerations for EM Pulse Fault Injection. In Sonia Belaïd and Tim Güneysu, editors, *Smart Card Research and Advanced Applications - 18th International Conference, CARDIS 2019, Prague, Czech Republic, November 11-13, 2019, Revised Selected Papers*, volume 11833 of *Lecture Notes in Computer Science*, pages 176–192. Springer, 2019.

[BKHL20] Jakub Breier, Mustafa Khairallah, Xiaolu Hou, and Yang Liu. A Countermeasure Against Statistical Ineffective Fault Analysis. *IEEE Trans. Circuits Syst.*, 67-II(12):3322–3326, 2020.

[BKL+07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[Bla03] Richard E. Blahut. *Algebraic Codes for Data Transmission*. Cambridge University Press, 2003.

[BLMR19] Christof Beierle, Gregor Leander, Amir Moradi, and Shahram Rasoolzadeh. CRAFT: lightweight tweakable block cipher with efficient protection against DFA attacks. *IACR Trans. Symmetric Cryptol.*, 2019(1):5–45, 2019.

[BMRT21] Sonia Belaïd, Darius Mercadier, Matthieu Rivain, and Abdul Rahman Taleb. IronMask: Versatile Verification of Masking Security. *IACR Cryptol. ePrint Arch.*, page 1671, 2021.

[BN08] Alberto Bosio and Giorgio Di Natale. LIFTING: A Flexible Open-Source Fault Simulator. In *17th IEEE Asian Test Symposium, ATS 2008, Sapporo, Japan, November 24-27, 2008*, pages 35–40. IEEE Computer Society, 2008.

[BOG19] Mario Bischof, Tobias Oder, and Tim Güneysu. Efficient Microcontroller Implementation of BIKE. In Emil Simion and Rémi Géraud-Stewart, editors, *Innovative Security Solutions for Information Technology and Communications - 12th International Conference, SecITC 2019, Bucharest, Romania, November 14-15, 2019, Revised Selected Papers*, volume 12001 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2019.

[BPG18] Florian Bache, Christina Plump, and Tim Güneysu. Confident leakage assessment - A side-channel evaluation framework based on confidence intervals. In Jan Madsen and Ayse K. Coskun, editors, *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, pages 1117–1122. IEEE, 2018.

[BPW+19] Florian Bache, Christina Plump, Jonas Wloka, Tim Güneysu, and Rolf Drechsler. Evaluation of (power) side-channels in cryptographic implementations. *it Inf. Technol.*, 61(1):15–28, 2019.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[BS97] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, pages 513–525, 1997.

[BS03] Johannes Blömer and Jean-Pierre Seifert. Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In Rebecca N. Wright, editor, *Financial Cryptography, 7th International Conference, FC 2003, Guadeloupe, French West Indies, January 27-30, 2003, Revised Papers*, volume 2742 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 2003.

[BV18] Stephen Boyd and Lieven Vandenberghe. *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*. Cambridge university press, 2018.

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):340–398, 2019.

[Can01] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001.

[CBR+15] Thomas De Cnudde, Begül Bilgin, Oscar Reparaz, Ventzislav Nikov, and Svetla Nikova. Higher-Order Threshold Implementation of the AES S-Box. In Naofumi Homma and Marcel Medwed, editors, *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, volume 9514 of *Lecture Notes in Computer Science*, pages 259–272. Springer, 2015.

[CCGR99] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model verifier. In *International conference on computer aided verification*, pages 495–499. Springer, 1999.

[CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing BIKE for the Intel Haswell and ARM Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):97–124, 2021.

[CGLS21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware Private Circuits: From Trivial Composition to Full Verification. *IEEE Trans. Computers*, 70(10):1677–1690, 2021.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[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, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 181–194. Springer, 2007.

[CLFT14] Franck Courbon, Philippe Loubet-Moundi, Jacques J. A. Fournier, and Assia Tria. Adjusting Laser Injections for Fully Controlled Faults. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*, volume 8622 of *Lecture Notes in Computer Science*, pages 229–242. Springer, 2014.

[CMD+19] Brice Colombier, Alexandre Menu, Jean-Max Dutertre, Pierre-Alain Moëllic, Jean-Baptiste Rigaud, and Jean-Luc Danger. Laser-induced Single-bit Faults in Flash Memory: Instructions Corruption on a 32-bit Microcontroller. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2019, McLean, VA, USA, May 5-10, 2019*, pages 1–10. IEEE, 2019.

[CML+11] Gaetan Canivet, Paolo Maistri, Régis Leveugle, Jessy Clédière, Florent Valette, and Marc Renaudin. Glitch and Laser Fault Attacks onto a Secure AES Implementation on a SRAM-Based FPGA. *J. Cryptol.*, 24(2):247–268, 2011.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference. *IEEE Trans. Inf. Forensics Secur.*, 15:2542–2555, 2020.

[DBC+18] Jean-Max Dutertre, Vincent Beroulle, Philippe Candelier, Stephan De Castro, Louis-Barthelemy Faber, Marie-Lise Flottes, Philippe Gendrier, David Hély, Régis Leveugle, Paolo Maistri, Giorgio Di Natale, Athanasios Papadimitriou, and Bruno Rouzeyre. Laser Fault Injection at the CMOS 28 nm Technology Node: an Analysis of the Fault Model. In *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2018, Amsterdam, The Netherlands, September 13, 2018*, pages 1–6. IEEE Computer Society, 2018.

[DDE+20] Joan Daemen, Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Florian Mendel, and Robert Primas. Protecting against Statistical Ineffective Fault Attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):508–543, 2020.

[DDRT12] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES. In Guido Bertoni and Benedikt Gierlichs, editors, *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012*, pages 7–15. IEEE Computer Society, 2012.

[DEG+18] Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Stefan Mangard, Florian Mendel, and Robert Primas. Statistical Ineffective Fault Attacks on Masked AES with Fault Countermeasures. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*, volume 11273 of *Lecture Notes in Computer Science*, pages 315–342. Springer, 2018.

[DEK+18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):547–572, 2018.

[DEMS16] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. ASCON v1. 2, 2016. `https://competitions.cr.yp.to/round3/asconv12.pdf`.

[DFA+20] Viet Ba Dang, Farnoud Farahmand, Michal Andrzejczak, Kamyar Mohajerani, Duc Tri Nguyen, and Kris Gaj. Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using Hardware and Software/Hardware Co-design Approaches. *IACR Cryptol. ePrint Arch.*, page 795, 2020.

[DG19] Nir Drucker and Shay Gueron. A toolbox for software optimization of QC-MDPC code-based cryptosystems. *J. Cryptogr. Eng.*, 9(4):341–357, 2019.

[DGK20a] Nir Drucker, Shay Gueron, and Dusan Kostic. Additional Implementation of BIKE (Bit Flipping Key Encapsulation). github, 2020. `https://github.com/awslabs/bike-kem`.

[DGK20b] Nir Drucker, Shay Gueron, and Dusan Kostic. Fast Polynomial Inversion for Post Quantum QC-MDPC Cryptography. In Shlomi Dolev, Vladimir Kolesnikov, Sachin Lodha, and Gera Weiss, editors, *Cyber Security Cryptography and Machine Learning - Fourth International Symposium, CSCML 2020, Be'er Sheva, Israel, July 2-3, 2020, Proceedings*, volume 12161 of *Lecture Notes in Computer Science*, pages 110–127. Springer, 2020.

[DGK20c] Nir Drucker, Shay Gueron, and Dusan Kostic. QC-MDPC Decoders with Several Shades of Gray. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020, Paris, France, April 15-17, 2020, Proceedings*, volume 12100 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2020.

[DHAK18] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. The design of Xoodoo and Xoofff. *IACR Trans. Symmetric Cryptol.*, 2018(4):1–38, 2018.

[DLM19] Mathieu Dumont, Mathieu Lisart, and Philippe Maurine. Electromagnetic Fault Injection : How Faults Occur. In *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2019, Atlanta, GA, USA, August 24, 2019*, pages 9–16. IEEE, 2019.

[DLM21] Mathieu Dumont, Mathieu Lisart, and Philippe Maurine. Modeling and Simulating Electromagnetic Fault Injection. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 40(4):680–693, 2021.

[DLV03] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential fault analysis on AES. In *International Conference on Applied Cryptography and Network Security*, pages 293–306. Springer, 2003.

[DMG21] Viet Ba Dang, Kamyar Mohajerani, and Kris Gaj. High-Speed Hardware Architectures and Fair FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber. 2021.

[DN20] Siemen Dhooghe and Svetla Nikova. My Gadget Just Cares for Me - How NINA Can Prove Security Against Combined Attacks. In Stanislaw Jarecki, editor, *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings*, volume 12006 of *Lecture Notes in Computer Science*, pages 35–55. Springer, 2020.

[DPM+21] Sanjay Deshpande, Santos Merino Del Pozo, Víctor Mateu, Marc Manzano, Najwa Aaraj, and Jakub Szefer. Modular Inverse for Integers using Fast Constant Time GCD Algorithm and its Applications. In *31st International Conference on Field-Programmable Logic and Applications, FPL 2021, Dresden, Germany, August 30 - Sept. 3, 2021*, pages 122–129. IEEE, 2021.

[DZD+17] A. Adam Ding, Liwei Zhang, François Durvaux, François-Xavier Standaert, and Yunsi Fei. Towards Sound and Optimal Leakage Detection Procedure. In *CARDIS*, volume 10728 of *Lecture Notes in Computer Science*, pages 105–122. Springer, 2017.

[ESH+11] Sho Endo, Takeshi Sugawara, Naofumi Homma, Takafumi Aoki, and Akashi Satoh. An on-chip glitchy-clock generator for testing fault injection attacks. *J. Cryptogr. Eng.*, 1(4):265–270, 2011.

[FGG+22] Jakob Feldtkeller, Anna Guinet, Tim Güneysu, Jan Richter-Brockmann, and Pascal Sasdrich. INDIANA - Verifying Probing Security through Indistinguishability Analysis. In submission, 2022.

[FGP+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.

[FJLT13] Thomas Fuhr, Éliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault Attacks on AES with Faulty Ciphertexts Only. In Wieland Fischer and Jörn-Marc Schmidt, editors, *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013*, pages 108–118. IEEE Computer Society, 2013.

[FRSG22] Jakob Feldtkeller, Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. CINI MINIS: Domain Isolation for Fault and Combined Security. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1023–1036. ACM, 2022.

[Gal62] Robert G. Gallager. Low-density parity-check codes. *IRE Trans. Inf. Theory*, 8(1):21–28, 1962.

[GGJR+11] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[GHJ+22] Qian Guo, Clemens Hlauschek, Thomas Johansson, Norman Lahr, Alexander Nilsson, and Robin Leander Schröder. Don't Reject This: Key-Recovery Timing Attacks Due to Rejection-Sampling in HQC and BIKE. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(3):223–263, 2022.

[GHP+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs. In Michael Bailey and Rachel Greenstadt, editors, *USENIX*, pages 1469–1468. USENIX Association, 2021.

[Gir04] Christophe Giraud. DFA on AES. In Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa, editors, *Advanced Encryption Standard - AES, 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers*, volume 3373 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2004.

[GLH18] Tomás Grimm, Djones Lettnin, and Michael Hübner. A survey on formal verification techniques for safety-critical systems-on-chip. *Electronics*, 7(6):81, 2018.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. In Begül Bilgin, Svetla Nikova, and Vincent Rijmen, editors, *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*, page 3. ACM, 2016.

[GPK+21] Michael Gruber, Matthias Probst, Patrick Karl, Thomas Schamberger, Lars Tebelmann, Michael Tempelmeier, and Georg Sigl. DOMREP-An Orthogonal Countermeasure for Arbitrary Order Side-Channel and Fault Attack Protection. *IEEE Trans. Inf. Forensics Secur.*, 16:4321–4335, 2021.

[GPPR11] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The LED Block Cipher. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2011.

[GST12] Benedikt Gierlichs, Jörn-Marc Schmidt, and Michael Tunstall. Infective Computation and Dummy Rounds: Fault Protection for Block Ciphers without Check-before-Output. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, volume 7533 of *Lecture Notes in Computer Science*, pages 305–321. Springer, 2012.

[GYTS14] Nahid Farhady Ghalaty, Bilgiday Yuce, Mostafa M. I. Taha, and Patrick Schaumont. Differential Fault Intensity Analysis. In Assia Tria and Dooho Choi, editors, *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014, Busan, South Korea, September 23, 2014*, pages 49–58. IEEE Computer Society, 2014.

[HB21] Vedad Hadzic and Roderick Bloem. COCOALMA: A Versatile Masking Verifier. In *FMCAD*, pages 1–10. IEEE, 2021.

[HBZL19] Xiaolu Hou, Jakub Breier, Fuyuan Zhang, and Yang Liu. Fully Automated Differential Fault Analysis on Software Implementations of Block Ciphers. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):1–29, 2019.

[HC17] Jingwei Hu and Ray C. C. Cheung. Area-Time Efficient Computation of Niederreiter Encryption on QC-MDPC Codes for Embedded Hardware. *IEEE Trans. Computers*, 66(8):1313–1325, 2017.

[HDL+20] Benjamin Hettwer, Kallyan Das, Sebastien Leger, Stefan Gehrer, and Tim Güneysu. Lightweight Side-Channel Protection using Dynamic Clock Randomization. In Nele Mentens, Leonel Sousa, Pedro Trancoso, Miquel Pericàs, and Ioannis Sourdis, editors, *30th International Conference on Field-Programmable Logic and Applications, FPL 2020, Gothenburg, Sweden, August 31 - September 4, 2020*, pages 200–207. IEEE, 2020.

[HFL+20] Benjamin Hettwer, Daniel Fennes, Sebastien Leger, Jan Richter-Brockmann, Stefan Gehrer, and Tim Güneysu. Deep Learning Multi-Channel Fusion Attack Against Side-Channel Protected Hardware. In *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*, pages 1–6. IEEE, 2020.

[HGWC15] Jingwei Hu, Wei Guo, Jizeng Wei, and Ray C. C. Cheung. Fast and Generic Inversion Architectures Over GF($2^{\mathrm{m}}$) Using Modified Itoh-Tsujii Algorithms. *IEEE Trans. Circuits Syst. II Express Briefs*, 62-II(4):367–371, 2015.

[HPB21] Vedad Hadzic, Robert Primas, and Roderick Bloem. Proving SIFA Protection of Masked Redundant Circuits. In Zhe Hou and Vijay Ganesh, editors, *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings*, volume 12971 of *Lecture Notes in Computer Science*, pages 249–265. Springer, 2021.

[HS13] Michael Hutter and Jörn-Marc Schmidt. The Temperature Side Channel and Heating Fault Attacks. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, volume 8419 of *Lecture Notes in Computer Science*, pages 219–235. Springer, 2013.

[HvMG13] Stefan Heyse, Ingo von Maurich, and Tim Güneysu. Smaller Keys for Code-Based Cryptography: QC-MDPC McEliece Implementations on Embedded Devices. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, volume 8086 of *Lecture Notes in Computer Science*, pages 273–292. Springer, 2013.

[HWCW19] Jingwei Hu, Wen Wang, Ray C. C. Cheung, and Huaxiong Wang. Optimized Polynomial Multiplier Over Commutative Rings on FPGAs: A Case Study on BIKE. In *International Conference on Field-Programmable Technology, FPT 2019, Tianjin, China, December 9-13, 2019*, pages 231–234. IEEE, 2019.

[Inc] NewAE Technology Inc. ChipSHOUTER Kit.

[IPSW06] Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and David A. Wagner. Private Circuits II: Keeping Secrets in Tamperable Circuits. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, pages 308–327. Springer, 2006.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[IT88] Toshiya Itoh and Shigeo Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in GF(2ˆm) Using Normal Bases. *Inf. Comput.*, 78(3):171–177, 1988.

[Jr.78] Sheldon B. Akers Jr. Binary Decision Diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.

[KHEB14] Thomas Korak, Michael Hutter, Baris Ege, and Lejla Batina. Clock Glitch Attacks in the Presence of Heating. In Assia Tria and Dooho Choi, editors, *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014, Busan, South Korea, September 23, 2014*, pages 104–114. IEEE Computer Society, 2014.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[KKG03] Ramesh Karri, Grigori Kuznetsov, and Michael Gössel. Parity-Based Concurrent Error Detection of Substitution-Permutation Network Block Ciphers. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 113–124. Springer, 2003.

[KLRG22a] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. A Holistic Approach Towards Side-Channel Secure Fixed-Weight Polynomial Sampling. In submission, 2022.

[KLRG22b] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. Efficiently Masking Polynomial Inversion at Arbitrary Order. In Jung Hee Cheon and Thomas Johansson, editors, *Post-Quantum Cryptography - 13th International Workshop, PQCrypto 2022, Virtual Event, September 28-30, 2022, Proceedings*, volume 13512 of *Lecture Notes in Computer Science*, pages 309–326. Springer, 2022.

[KM22] David Knichel and Amir Moradi. Low-Latency Hardware Private Circuits. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1799–1812. ACM, 2022.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, pages 104–113, 1996.

[KRH17] Punit Khanna, Chester Rebeiro, and Aritra Hazra. XFC: A Framework for eXploitable Fault Characterization in Block Ciphers. In *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*, pages 8:1–8:6. ACM, 2017.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - Statistical Independence and Leakage Verification. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, 2020.

[KWMK02] Ramesh Karri, Kaijie Wu, Piyush Mishra, and Yongkook Kim. Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 21(12):1509–1517, 2002.

[LH96] Hyung Ki Lee and Dong Sam Ha. HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 15(9):1048–1058, 1996.

[LMRG22] Georg Land, Adrian Marotzke, Jan Richter-Brockmann, and Tim Güneysu. Masking Streamlined NTRU Prime Decapsulation at Gate Level on Reconfigurable Hardware. In submission, 2022.

[LMW14] Andrew J. Leiserson, Mark E. Marson, and Megan A. Wachs. Gate-Level Masking under a Path-Based Leakage Metric. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 580–597. Springer, 2014.

[MAB+21] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loıc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and IC Bourges. Hamming Quasi-Cyclic (HQC) – third round version, 2021.

[MAN+19] Lauren De Meyer, Victor Arribas, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. M&M: Masks and Macs against Physical Attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(1):25–50, 2019.

[Mar20] Adrian Marotzke. A Constant Time Full Hardware Implementation of Streamlined NTRU Prime. In *CARDIS 2020*, volume 12609 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2020.

[Mau11] Ueli Maurer. Constructive Cryptography - A New Paradigm for Security Definitions and Proofs. In Sebastian Mödersheim and Catuscia Palamidessi, editors, *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*, volume 6993 of *Lecture Notes in Computer Science*, pages 33–56. Springer, 2011.

[McE78] Robert J. McEliece. A public-key cryptosystem based on algebraic. *Coding Thv*, 4244:114–116, 1978.

[MSY06] Tal Malkin, François-Xavier Standaert, and Moti Yung. A Comparative Cost/Security Analysis of Fault Attack Countermeasures. In Luca Breveglieri, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2006, Yokohama, Japan, October 10, 2006, Proceedings*, volume 4236 of *Lecture Notes in Computer Science*, pages 159–172. Springer, 2006.

[MTOL12] Philippe Maurine, Karim Tobich, Thomas Ordas, and Pierre Yvan Liardet. Yet another fault injection technique: by forward body biasing injection. In *YACC'2012: Yet Another Conference on Cryptography*, 2012.

[MTSB13] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo S. L. M. Barreto. MDPC-McEliece: New McEliece variants from Moderate Density Parity-Check codes. In *Proceedings of the 2013 IEEE International Symposium on Information Theory, Istanbul, Turkey, July 7-12, 2013*, pages 2069–2073. IEEE, 2013.

[MW15] Amir Moradi and Alexander Wild. Assessment of Hiding the Higher-Order Leakages in Hardware - What Are the Achievements Versus Overheads? In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 453–474. Springer, 2015.

[NCP92] Thomas M. Niermann, Wu-Tung Cheng, and Janak H. Patel. PROOFS: a fast, memory-efficient sequential circuit fault simulator. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 11(2):198–207, 1992.

[Nie86] Harald Niederreiter. Knapsack-type Cryptosystems and Algebraic Coding Theory. *Prob. Control and Inf. Theory*, 15(2):159–166, 1986.

[NIS20] NIST. Guidelines for submitting tweaks for Third Round Finalists and Candidates, 2020. `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/LPuZKGNyQJ0/m/O6UBanYbDAAJ`.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer, 2006.

[O'F20] Colin O'Flynn. Low-Cost Body Biasing Injection (BBI) Attacks on WLCSP Devices. In Pierre-Yvan Liardet and Nele Mentens, editors, *Smart Card Research and Advanced Applications - 19th International Conference, CARDIS 2020, Virtual Event, November 18-19, 2020, Revised Selected Papers*, volume 12609 of *Lecture Notes in Computer Science*, pages 166–180. Springer, 2020.

[OGM15] Sébastien Ordas, Ludovic Guillaume-Sage, and Philippe Maurine. EM Injection: Fault Model and Locality. In Naofumi Homma and Victor Lomné, editors, *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2015, Saint Malo, France, September 13, 2015*, pages 3–13. IEEE Computer Society, 2015.

[OGM17] Sébastien Ordas, Ludovic Guillaume-Sage, and Philippe Maurine. Electromagnetic fault injection: the curse of flip-flops. *J. Cryptogr. Eng.*, 7(3):183–197, 2017.

[OGT+14] Sébastien Ordas, Ludovic Guillaume-Sage, Karim Tobich, Jean-Max Dutertre, and Philippe Maurine. Evidence of a Larger EM-Induced Fault Model. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, volume 8968 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2014.

[oST22] National Institute of Standards and Technology. Pqc standardization process: Announcing four candidates to be standardized, plus fourth round candidates. Information Technology Laboratory - Computer Security Resource Center, July 2022. `https://csrc.nist.gov/news/2022/pqc-candidates-to-be-standardized-and-round-4`.

[Pet11] Edward Petersen. *Single Event Effects in Aerospace*. John Wiley & Sons, 2011.

[RAD20] Keyvan Ramezanpour, Paul Ampadu, and William Diehl. RS-Mask: Random Space Masking as an Integrated Countermeasure against Power and Fault Analysis. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020*, pages 176–187. IEEE, 2020.

[Raz08] Behzad Razavi. *Fundamentals of Microelectronics*. Wiley, 2008.

[RBN+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating Masking Schemes. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 764–783. Springer, 2015.

[RBSG22] Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. Revisiting Fault Adversary Models – Hardware Faults in Theory and Practice. *IEEE Transactions on Computers*, pages 1–1, 2022.

[RCGG22] Jan Richter-Brockmann, Ming-Shing Chen, Santosh Ghosh, and Tim Güneysu. Racing BIKE: Improved Polynomial Multiplication and Inversion in Hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):557–588, 2022.

[RFSG22] Jan Richter-Brockmann, Jakob Feldtkeller, Pascal Sasdrich, and Tim Güneysu. VERICA - Verification of Combined Attacks: Automated formal verification of security against simultaneous information leakage and tampering. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4), 2022.

[RG20] Jan Richter-Brockmann and Tim Güneysu. Improved Side-Channel Resistance by Dynamic Fault-Injection Countermeasures. In *31st IEEE International Conference on Application-specific Systems, Architectures and Processors , ASAP 2020, Manchester, United Kingdom, July 6-8, 2020*, pages 117–124. IEEE, 2020.

[Ris21] Riscure. Inspector Fault Injection, Oct 2021.

[RMB+18] Oscar Reparaz, Lauren De Meyer, Begül Bilgin, Victor Arribas, Svetla Nikova, Ventzislav Nikov, and Nigel P. Smart. CAPA: The Spirit of Beaver Against Physical Attacks. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 121–151. Springer, 2018.

[RMG22] Jan Richter-Brockmann, Johannes Mono, and Tim Güneysu. Folding BIKE: Scalable Hardware Implementation for Reconfigurable Devices. *IEEE Transactions on Computers*, 71(5):1204–1215, 2022.

[RMGS20] Andrew H. Reinders, Rafael Misoczki, Santosh Ghosh, and Manoj R. Sastry. Efficient BIKE Hardware Design with Constant-Time Decoder. In *IEEE International Conference on Quantum Computing and Engineering, QCE 2020, Denver, CO, USA, October 12-16, 2020*, pages 197–204. IEEE, 2020.

[RNK19] Hila Rabii, Yaara Neumeier, and Osnat Keren. High Rate Robust Codes with Low Implementation Complexity. *IEEE Trans. Dependable Secur. Comput.*, 16(3):511–520, 2019.

[RRHB20] Indrani Roy, Chester Rebeiro, Aritra Hazra, and Swarup Bhunia. SAFARI: Automatic Synthesis of Fault-Attack Resistant Block Cipher Implementations. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(4):752–765, 2020.

[RSBG20] Jan Richter-Brockmann, Pascal Sasdrich, Florian Bache, and Tim Güneysu. Concurrent Error Detection Revisited: Hardware Protection against Fault and Side-channel Attacks. In Melanie Volkamer and Christian Wressnegger, editors, *ARES 2020: The 15th International Conference on Availability, Reliability and Security, Virtual Event, Ireland, August 25-28, 2020*, pages 20:1–20:11. ACM, 2020.

[RSDT13] Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre, and Assia Tria. Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells. In Wieland Fischer and Jörn-Marc Schmidt, editors, *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013*, pages 89–98. IEEE Computer Society, 2013.

[RSM21] Shahram Rasoolzadeh, Aein Rezaei Shahmirzadi, and Amir Moradi. Impeccable Circuits III. In *IEEE International Test Conference, ITC 2021, Anaheim, CA, USA, October 10-15, 2021*, pages 163–169. IEEE, 2021.

[RSS+21] Jan Richter-Brockmann, Aein Rezaei Shahmirzadi, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. FIVER - Robust Verification of Countermeasures against Fault Injections. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):447–473, 2021.

[RU96] Teresa Riesgo and Javier Uceda. A fault model for VHDL descriptions at the register transfer level. In Graham Symonds and Wolfgang Nebel, editors, *Proceedings of the conference on European design automation, EURO-DAC '96/EURO-VHDL '96, Geneva, Switzerland, September 16-20, 1996*, pages 462–467. IEEE Computer Society Press, 1996.

[SA02] Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 2002.

[SBHM20] Pascal Sasdrich, Begül Bilgin, Michael Hutter, and Mark E. Marson. Low-Latency Hardware Masking with Application to AES. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):300–326, 2020.

[SBHS15] 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 - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, volume 9514 of *Lecture Notes in Computer Science*, pages 193–205. Springer, 2015.

[Sen21] Nicolas Sendrier. Secure Sampling of Constant-Weight Words â€" Application to BIKE. *IACR Cryptol. ePrint Arch.*, page 1631, 2021.

[SFG+16] Falk Schellenberg, Markus Finkeldey, Nils Gerhardt, Martin Hofmann, Amir Moradi, and Christof Paar. Large Laser Spots and Fault Sensitivity Analysis. In William H. Robinson, Swarup Bhunia, and Ryan Kastner, editors, *2016 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2016, McLean, VA, USA, May 3-5, 2016*, pages 203–208. IEEE Computer Society, 2016.

[SGD08] Nidhal Selmane, Sylvain Guilley, and Jean-Luc Danger. Practical Setup Time Violation Attacks on AES. In *Seventh European Dependable Computing Conference, EDCC-7 2008, Kaunas, Lithuania, 7-9 May 2008*, pages 91–96. IEEE Computer Society, 2008.

[Sho99] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Rev.*, 41(2):303–332, 1999.

[SHO19] Bodo Selmke, Florian Hauschild, and Johannes Obermaier. Peak Clock: Fault Injection into PLL-Based Systems via Clock Manipulation. In Chip-Hong Chang, Ulrich Rührmair, Daniel E. Holcomb, and Patrick Schaumont, editors, *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop, ASHES@CCS 2019, London, UK, November 15, 2019*, pages 85–94. ACM, 2019.

[SHS16] Bodo Selmke, Johann Heyszl, and Georg Sigl. Attack on a DFA Protected AES by Simultaneous Laser Fault Injections. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016*, pages 36–46. IEEE Computer Society, 2016.

[SJR+19] Sayandeep Saha, Dirmanto Jap, Debapriya Basu Roy, Avik Chakraborti, Shivam Bhasin, and Debdeep Mukhopadhyay. Transform-and-Encode: A Countermeasure Framework for Statistical Ineffective Fault Attacks on Block Ciphers. *IACR Cryptol. ePrint Arch.*, page 545, 2019.

[SJR+20] Sayandeep Saha, Dirmanto Jap, Debapriya Basu Roy, Avik Chakraborty, Shivam Bhasin, and Debdeep Mukhopadhyay. A Framework to Counter Statistical Ineffective Fault Analysis of Block Ciphers Using Domain Transformation and Error Correction. *IEEE Trans. Inf. Forensics Secur.*, 15:1905–1919, 2020.

[Sko09] Sergei P. Skorobogatov. Local Heating Attacks on Flash Memory Devices. In Mohammad Tehranipoor and Jim Plusquellic, editors, *IEEE International Workshop on Hardware-Oriented Security and Trust, HOST 2009, San Francisco, CA, USA, July 27, 2009. Proceedings*, pages 1–6. IEEE Computer Society, 2009.

[SM15] Tobias Schneider and Amir Moradi. Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 495–513, 2015.

[SMC21] Albert Spruyt, Alyssa Milburn, and Lukasz Chmielewski. Fault Injection as an Oscilloscope: Fault Correlation Analysis. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):192–216, 2021.

[SMD18] Sayandeep Saha, Debdeep Mukhopadhyay, and Pallab Dasgupta. ExpFault: An Automated Framework for Exploitable Fault Characterization in Block Ciphers. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):242–276, 2018.

[SMG15] Pascal Sasdrich, Amir Moradi, and Tim Güneysu. Affine Equivalence and Its Application to Tightening Threshold Implementations. In Orr Dunkelman and Liam Keliher, editors, *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*, volume 9566 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 2015.

[SMG16] Tobias Schneider, Amir Moradi, and Tim Güneysu. ParTI - Towards Combined Hardware Countermeasures Against Side-Channel and Fault-Injection Attacks. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 302–332. Springer, 2016.

[SMG17] Pascal Sasdrich, Amir Moradi, and Tim Güneysu. Hiding Higher-Order Side-Channel Leakage - Randomizing Cryptographic Implementations in Reconfigurable Hardware. In Helena Handschuh, editor, *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings*, volume 10159 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2017.

[Som99] Fabio Somenzi. Binary Decision Diagrams. In *Calculational System Design*, pages 303–366. IOS, 1999.

[Som18] Fabio Somenzi. CUDD: CU decision diagram package-release 2.7.0. 2018.

[SRM20] Aein Rezaei Shahmirzadi, Shahram Rasoolzadeh, and Amir Moradi. Impeccable Circuits II. In *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*, pages 1–6. IEEE, 2020.

[SSR+18] Pasquale Davide Schiavone, Ernesto Sánchez, Annachiara Ruospo, Francesco Minervini, Florian Zaruba, Germain Haugou, and Luca Benini. An Open-Source Verification Framework for Open-Source Cores: A RISC-V Case Study. In *IFIP/IEEE International Conference on Very Large Scale Integration, VLSI-SoC 2018, Verona, Italy, October 8-10, 2018*, pages 43–48. IEEE, 2018.

[TBM14] Harshal Tupsamudre, Shikha Bisht, and Debdeep Mukhopadhyay. Destroying Fault Invariant with Randomization - A Countermeasure for AES Against Differential Fault Attacks. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 93–111. Springer, 2014.

[Tof80] Tommaso Toffoli. Reversible Computing. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherlands, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 632–644. Springer, 1980.

[vMG14] Ingo von Maurich and Tim Güneysu. Lightweight code-based cryptography: QC-MDPC McEliece encryption on reconfigurable devices. In Gerhard P. Fettweis and Wolfgang Nebel, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6. European Design and Automation Association, 2014.

[vT93] Henk van Tilborg. Coding Theory, a first course. 1993.

[WA08] Fan Wang and Vishwani D. Agrawal. Single Event Upset: An Embedded Tutorial. In *21st International Conference on VLSI Design (VLSI Design 2008), 4-8 January 2008, Hyderabad, India*, pages 429–434. IEEE Computer Society, 2008.

[WRS+20] Jonas Wloka, Jan Richter-Brockmann, Colin Stahlke, Thorsten Kleinjung, Christine Priplata, and Tim Güneysu. Revisiting ECM on GPUs. In Stephan Krenn, Haya Shulman, and Serge Vaudenay, editors, *Cryptology and Network Security - 19th International Conference, CANS 2020, Vienna, Austria, December 14-16, 2020, Proceedings*, volume 12579 of *Lecture Notes in Computer Science*, pages 299–319. Springer, 2020.

[WSN18] Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-Based Niederreiter Cryptosystem Using Binary Goppa Codes. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*, volume 10786 of *Lecture Notes in Computer Science*, pages 77–98. Springer, 2018.

[XL21] Yufei Xing and Shuguo Li. A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):328–356, 2021.

[ZDCT13] Loïc Zussa, Jean-Max Dutertre, Jessy Clédière, and Assia Tria. Power supply glitch induced faults on FPGA: An in-depth analysis of the injection mechanism.

In *2013 IEEE 19th International On-Line Testing Symposium (IOLTS), Chania, Crete, Greece, July 8-10, 2013*, pages 110–115. IEEE, 2013.

[ZGF20] Davide Zoni, Andrea Galimberti, and William Fornaciari. Flexible and Scalable FPGA-Oriented Design of Multipliers for Large Binary Polynomials. *IEEE Access*, 8:75809–75821, 2020.

# List of Abbreviations

**AES** Advanced Encryption Standard

**ASIC** Application-Specific Integrated Circuit

**AT** Area-Time

**BDD** Binary Decision Diagram

**BGF** Black-Gray-Flip

**BIKE** Bit Flipping Key Encapsulation

**BRAM** Block-RAM

**CA** Combined Analysis

**CEC** Concurrent Error Correction

**CED** Concurrent Error Detection

**CINI** Combined-Isolating Non-Interference

**CLB** Configurable Logic Block

**CMOS** Complementary Metal-Oxide-Semiconductor

**CNI** Combined Non-Interference

**CPU** Central Processing Unit

**CSNI** Combined Strong Non-Interference

**CUDD** Colorado University Decision Diagram

**DAG** Direct Acyclic Graph

**DAT** Data Arrival Time

**DES** Data Encryption Standard

**DFA** Differential Fault Analysis

**DFF** D Flip-Flop

**DFIA** Differential Fault Intensity Analysis

**DLL** Delay Locked Loop

**DOM** Domain-Oriented Masking

**DRT** Data Required Time

**DSP** Digital Signal Processor

**ECC** Error-Correcting Code

**EMFI** Electromagnetic Fault Injection

**EMP** Electromagnetic Pulse

**extGCD** Extended Euclidean Algorithm

**FF** Flip-Flop

**FIA** Fault-Injection Analysis

**FINI** Fault-Isolating Non-Interference

**FNI** Fault Non-Interference

**FPGA** Field-Programmable Gate Array

**FSNI** Fault Strong Non-Interference

**FSM** Finite State Machine

**HD** Hamming Distance

**HPC** Hardware Private Circuit

**IC** Integrated Circuit

**ICINI** Independent Combined-Isolating Non-Interference

**ICSNI** Indpendet Combined Strong Non-Interference

**IFA** Ineffective Fault Analysis

**IoT** Internet of Things

**ISW** Ishai-Sahai-Wagner

**ITA** Itoh-Tsujii Algorithm

**KEM** Key Encapsulation Mechanism

**LDPC** Low-Density Parity-Check

**LFI** Laser Fault Injection

**LFSR** Linear Feedback Shift Register

**LMDPL** LUT-Masked Dual-rail with Precharge Logic

**LNA** Low Noise Amplifier

**LUT** Look-Up Table

**LSB** Least Significant Bit

**MAC** Message Authentication Code

**MDC** Multi Duplication with Comparison

**MDPC** Moderate-Density Parity-Check

**MPC** Multi-Party Computation

**NA** Non-Accumulation

**NI** Non-Interference

**NINA** Non-Interference Non-Accumulative

**NIST** National Institute of Standards and Technology

**NMOS** N-type Metal-Oxide Semiconductor

**OCL** Open Cell Library

**OS** Operating System

**PCB** Printed Circuit Board

**PINI** Probe-Isolating Non-Interference

**PKC** Public-Key Cryptography

**PKE** Public-Key Encryption

**PLL** Phase-Locked Loop

**PMOS** P-type Metal-Oxide Semiconductor

**PNI** Probe Non-Interference

**PQC** Post-Quantum Cryptography

**PRNG** Pseudorandom Number Generator

**PSNI** Probe Strong Non-Interference

**QC** Quasi-Cyclic

**QC-MDPC** Quasi-Cyclic Moderate-Density Parity-Check

**RNG** Random Number Generator

**ROBDD** Reduced Ordered Binary Decision Diagram

**SCA** Side-Channel Analysis

**SEI** Squared Euclidean Imbalance

**SET** Single Event Transient

**SEU** Single Event Upset

**SFA** Statistical Fault Attack

**SIFA** Statistical Ineffective Fault Analysis

**SININA** Strong Independent Non-Interference Non-Accumulation

**SNA** Strong Non-Accumulation

**SNINA** Strong Non-Interference Non-Accumulation

**SNR** Signal-to-Noise Ratio

**SRD** Shared Redundancy Domain

**TI** Threshold Implementation

**TMR** Triple Modular Redundancy

**TVLA** Test Vector Leakage Assessment

**UPC** Unsatisfied-Parity-Check

**VLSI** Very Large Scale Integration

**XOR** Exclusive OR

# List of Figures

# List of Tables

# About the Author

Author information as of December 2022.

## Personal Data


| | |
|---|---|
| **Name** | Jan Richter-Brockmann |
| **Address** | Chair for Security Engineering<br>Universitätsstr. 150, ID 2/653<br>44801 Bochum, Germany |
| **E-Mail** | jan.richter-brockmann@rub.de |
| **Date of birth** | March 11, 1993 |
| **Place of birth** | Datteln, Germany |


## Education

| | |
|---|---|
| Since 05/2022 | **PhD-student**, *Ruhr-Universität Bochum*, Computer Science. |
| 02/2018 - 04/2022 | **PhD-student**, *Ruhr-Universität Bochum*, Electrical and Information Engineering. |
| 10/2015 - 09/2017 | **M.Sc.**, *RWTH Aachen University*, Electrical Engineering |
| 10/2012 - 09/2015 | **B.Sc.**, *RWTH Aachen University*, Electrical Engineering |

## Professional Experience

| | |
|---|---|
| Since 02/2018 | **Research Assistant**, *Ruhr-Universität Bochum.* |
| 10/2020 - 00/2021 | **Intern**, *Intel Labs*, Munich. |
| 10/2016 - 03/2017 | **Intern**, *Intel*, Munich. |
| 11/2015 - 09/2016 | **Student Assistant**, *Chair of Integrated Analog Circuits, RWTH Aachen.* |
| 11/2014 - 03/2015 | **Student Assistant**, *Chair of Electrical Engineering and Computer Systems, RWTH Aachen.* |
| 04/2014 - 06/2014 | **Student Assistant**, *Institute for Power Electronics and Electrical Drives, RWTH Aachen.* |
| 12/2013 - 01/2014 | **Student Assistant**, *Institute for Power Electronics and Electrical Drives, RWTH Aachen.* |

# Publications and Academic Activities

## Peer-Reviewed Journal Papers

- Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. Revisiting Fault Adversary Models – Hardware Faults in Theory and Practice. *IEEE Transactions on Computers*, pages 1–1, 2022
- Jan Richter-Brockmann, Johannes Mono, and Tim Güneysu. Folding BIKE: Scalable Hardware Implementation for Reconfigurable Devices. *IEEE Transactions on Computers*, 71(5):1204–1215, 2022

## Peer-Reviewed Conference Proceeding

- Jakob Feldtkeller, Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. CINI MINIS: Domain Isolation for Fault and Combined Security. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1023–1036. ACM, 2022
- Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. Efficiently Masking Polynomial Inversion at Arbitrary Order. In Jung Hee Cheon and Thomas Johansson, editors, *Post-Quantum Cryptography - 13th International Workshop, PQCrypto 2022, Virtual Event, September 28-30, 2022, Proceedings*, volume 13512 of *Lecture Notes in Computer Science*, pages 309–326. Springer, 2022
- Jan Richter-Brockmann, Jakob Feldtkeller, Pascal Sasdrich, and Tim Güneysu. VERICA - Verification of Combined Attacks: Automated formal verification of security against simultaneous information leakage and tampering. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4), 2022
- Jan Richter-Brockmann, Ming-Shing Chen, Santosh Ghosh, and Tim Güneysu. Racing BIKE: Improved Polynomial Multiplication and Inversion in Hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):557–588, 2022
- Jan Richter-Brockmann, Aein Rezaei Shahmirzadi, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. FIVER - Robust Verification of Countermeasures against Fault Injections. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):447–473, 2021
- Jan Richter-Brockmann, Pascal Sasdrich, Florian Bache, and Tim Güneysu. Concurrent Error Detection Revisited: Hardware Protection against Fault and Side-channel Attacks.

In Melanie Volkamer and Christian Wressnegger, editors, *ARES 2020: The 15th International Conference on Availability, Reliability and Security, Virtual Event, Ireland, August 25-28, 2020*, pages 20:1–20:11. ACM, 2020

- Jonas Wloka, Jan Richter-Brockmann, Colin Stahlke, Thorsten Kleinjung, Christine Priplata, and Tim Güneysu. Revisiting ECM on GPUs. In Stephan Krenn, Haya Shulman, and Serge Vaudenay, editors, *Cryptology and Network Security - 19th International Conference, CANS 2020, Vienna, Austria, December 14-16, 2020, Proceedings*, volume 12579 of *Lecture Notes in Computer Science*, pages 299–319. Springer, 2020

- Benjamin Hettwer, Daniel Fennes, Sebastien Leger, Jan Richter-Brockmann, Stefan Gehrer, and Tim Güneysu. Deep Learning Multi-Channel Fusion Attack Against Side-Channel Protected Hardware. In *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*, pages 1–6. IEEE, 2020

- Jan Richter-Brockmann and Tim Güneysu. Improved Side-Channel Resistance by Dynamic Fault-Injection Countermeasures. In *31st IEEE International Conference on Application-specific Systems, Architectures and Processors , ASAP 2020, Manchester, United Kingdom, July 6-8, 2020*, pages 117–124. IEEE, 2020

## Submissions Under Review

- Georg Land, Adrian Marotzke, Jan Richter-Brockmann, and Tim Güneysu. Masking Streamlined NTRU Prime Decapsulation at Gate Level on Reconfigurable Hardware. In submission, 2022

- Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. A Holistic Approach Towards Side-Channel Secure Fixed-Weight Polynomial Sampling. In submission, 2022

- Jakob Feldtkeller, Anna Guinet, Tim Güneysu, Jan Richter-Brockmann, and Pascal Sasdrich. INDIANA - Verifying Probing Security through Indistinguishability Analysis. In submission, 2022

## Technical Reports

- Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, France Worldline, Loïc Bidoux, Olivier Blazy, Philippe Gaborit, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, et al. BIKE: Bit Flipping Key Encapsulation. 2021. `https://bikesuite.org/files/v4.2/BIKE_Spec.2021.07.26.1.pdf`

- Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, France Worldline, Loïc Bidoux, Olivier Blazy, Philippe Gaborit, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, et al. BIKE: Bit Flipping Key Encapsulation - Round 4 Submission. 2022. `https://bikesuite.org/files/v5.0/BIKE_Spec.2022.10.10.1.pdf`

## Participation in Selected Conferences and Workshops

- CCS, 2022, *Los Angeles, California*
- CHES, 2022, *Leuven, Belgium*
- CARDIS, 2021, *Lübeck, Germany*
- Fall School on Nano-Electronics for Secure Systems, 2021, *Lübeck, Germany*
- CHES, 2021, *online*
- ASAP, 2020, *online*
- ARES, 2020, *online*
- CHES, 2020, *online*
- CHES, 2018, *Amsterdam, The Netherlands*
- Summer School on Real-World Crypto and Privacy, 2018, *Sibenik, Croatia*