Robert Primas

# Side-Channel and Fault Analysis of Cryptographic Implementations

**DOCTORAL THESIS**

to achieve the university degree of

Doctor of Technical Sciences; Dr. techn.

submitted to

**Graz University of Technology**

**Assessors**

Stefan Mangard
Graz University of Technology

Joan Daemen
Radboud University Nijmegen

Graz, February 2023

＊＊＊

# Abstract

Cryptographic algorithms are commonly used to achieve security properties such as confidentiality and authenticity of messages that are being sent over an untrusted communication channel like the Internet. This is generally realized by using a cryptographic key to (1) transform a plaintext into encrypted ciphertext and (2) append an authentication tag to it.

The security of cryptographic algorithms is generally measured by the cost of the best-known method for breaking its security properties in a (mostly) black-box setting. In this setting, an attacker, besides having knowledge of the algorithmic specification, can only observe its inputs/outputs that are not required to be kept secret. An attacker may then use various analysis techniques trying to break the algorithm's security properties. In the most drastic case, an attack achieves full key recovery, which usually breaks all security properties of a cryptographic algorithm.

In the real world, cryptographic algorithms are deployed as implementations on electronic devices that, depending on the concrete scenario, may allow an attacker to gain physical access. In this setting, cryptographic implementations find themselves in a more like gray-box setting that gives attackers the additional capability to perform physical manipulations or to observe a device's physical properties. These improved capabilities can greatly simplify attacks on these implementations, thus breaking their security properties. Providing effective protection against such implementation attacks is of high importance for many practical applications of cryptography and requires a deep understanding of the possible attack vectors and defense techniques. Exactly this understanding is constantly challenged by the evolution of cryptographic algorithms, implementation techniques, and attack strategies. In this thesis, we address this problem by analyzing and advancing the current state of implementation security with respect to both passive and active implementation attacks.

In the context of passive implementation attacks, we study power analysis attacks on fairly new lattice-based cryptographic schemes, which serve as a potential future replacement for currently used RSA/ECC-based schemes. We present a power analysis attack on a lattice-based decryption scheme that allows the extraction of entire private keys from a single power trace, even though the construction of lattice-based cryptography makes such attacks particularly challenging. We then show an improvement of this method that, while being limited to certain applications of lattice-based encryption, requires a significantly less powerful power analysis adversary.

In the context of active implementation attacks, we present exploitation techniques for fault attacks that can circumvent various algorithmic defense techniques that were previously believed to offer sufficient protection against such attacks. We then further show the applicability of these attacks in the context of authenticated encryption, a setting in which the successful application of fault attacks is usually particularly difficult. Finally, we present a novel defense technique that can provide effective and efficient protection against a large class of active (and passive) implementation attacks.

# Acknowledgements

My first expression of gratitude is devoted to all the people at university and in school who helped to shape my later scientific path by passing both their knowledge and passion on to me. Starting with my school teachers, I would like to thank Nicole Bizjak, Leander Brandl, and Siegfried Patz, who first developed my interest in computer science in middle/high school, amongst others things, by showing me how to construct football-playing robots.

During my bachelor studies, I would like to thank Mario Lamberger for giving me a smooth start into the world of information security, and Florian Mendel and Martin Schläffer for supervising me during my Bachelor's thesis. During my Master's studies, I would like to additionally thank Maria Eichlseder for excellent in-depth lectures on information security, as well as Stefan Mangard and Peter Pessl for waking my interest in implementation security and supervising me during my Master's thesis.

As far as I can remember, I had never talked to Christoph Dobraunig before I started my PhD. This did, however, change quickly and he soon become one of my most important supporters during my PhD. I would also like to thank many other of my co-authors or colleagues at the IAIK institute that not only contributed to many fruitful scientific discussions but also to just as many goofy ones. Being part of the IAIK institute fills me with gratitude, not only due to the encouraging and employee-friendly working environment but also because of the people, professors and students alike, that introduced me to scientific publishing as well as personal and group leadership! As I am approaching the end of my PhD journey, I would also like to thank my external examiner Joan Daemen for many detailed comments that helped improve the quality of this thesis.

Last but not least, I want to sincerely thank my family for the supportive and healthy environment I grew up and lived in and my dog Joy for spreading delight independent of the current circumstances.

＊＊＊

# Table of Contents

# 1 Introduction

Be alone with my thoughts? Near on unachievable these days.

*V - Cyberpunk 2077*

Many aspects of today's life are determined by information technology. Not too long ago, globalized instantaneous communication, with all of its current applications, was unthinkable. Yet today, most of us take this ability for granted and do not necessarily (want to) consider the potential risks that come with it. What is often a bit overlooked is the fact that technology does serve not only the optional purpose of entertainment and convenience but also much more vital purposes. Nowadays, critical services such as communication networks or power grids are examples of complex systems consisting of a network of interconnected devices, each of which plays a vital role in a system's overall functionality. Consequently, the integrity of such systems is not only determined by a certain central processing endpoint but also relies on the integrity of all of its components, very much in line with the famous quote: "*A chain is only as strong as its weakest link*" – as it appears in Thomas Reid's "*Essays on the Intellectual Powers of Man*" from 1786.

One of the key measures to secure systems against outside threats is the usage of secure communication mechanisms that can be realized via the proper use of cryptography and key management. Given a shared key between communication parties, cryptography can provide solutions that ensure confidentiality and authenticity of messages that are being sent between the parties. Key management is concerned with the proper generation, exchange, storage, or replacement of keys during the lifetime of a system.

Yet, even if a system features proper usage of cryptographic algorithms/keys and is free of implementation errors, there are still additional threats. So-called implementation attacks, including techniques such as physical side-channel or fault analysis, allow attackers to extract sensitive information, such as cryptographic keys, from physically accessible devices, which may then allow compromising the system in that they play a role. A classic example of such an attack scenario are smart cards that are used, e.g., to realize electronic payment or physical/digital access control to critical infrastructure. In situations like these, cryptographic secrets stored on these cards, if revealed to an attacker, could be misused to issue fraudulent money transactions or to disturb the operation of critical infrastructures like power grids or telecommunication. Additional examples include electronic passports containing cryptographic keys for copy protection, or access control mechanisms on personal devices (laptops or smartphones) that, if circumvented, can result in an infringement of personal security and privacy. The ability to implement cryptography offering security not only on cryptographic level but also in the presence of implementation attacks is hence of high importance for many practical applications of cryptography. To this end, in this thesis, we analyze and advance the current state of implementation security with respect to both, passive and active implementation attacks.

## 1.1 Contributions and Outline

This thesis starts with a brief introduction to cryptography and implementation attacks. The core chapters of this thesis are split into two parts, covering aspects of passive and active implementation attacks, respectively. We preface each part with a more in-depth introduction on passive/active implementation attacks that also covers some preliminaries of the presented contributions. The scientific contributions of this thesis have all been published as peer-reviewed articles at conferences of the field. In this thesis, the presentation of these contributions mostly corresponds to the published versions. Their original introduction and preliminary sections are adapted/modified to avoid redundant text and to underline the connections between them. Additionally, the general write-up was improved by revising various formulations and notations. We want to point out that the order of authors on the corresponding published articles does not necessarily reflect the order of contribution. Instead, and in line with a statement of the American Mathematical Society[1], the order of authors often simply follows alphabetical order. The contribution of the author to each scientific article is discussed in the following outline of the thesis. A conclusion, as well as a complete list of publications and further collaborations, are given at the end of this thesis.

[1]From: `https://www.ams.org/profession/leaders/CultureStatement04.pdf`: "In most areas of mathematics, joint research is a sharing of ideas and skills that cannot be attributed to the individuals separately. The roles of researchers are seldom differentiated (in the way they are in laboratory sciences, for example). Determining which person contributed which ideas is often meaningless because the ideas grow from complex discussions among all partners."

**Part I** of this thesis is devoted to passive implementation attacks and begins with a more in-depth introduction to the topic in Chapter 2.

In Chapter 3, we study the applicability of profiled power analysis on lattice-based cryptography, which serves as a potential future replacement for current RSA/ECC-based cryptography. More concretely, we demonstrate a power analysis attack on implementations of lattice-based decryption operations that allow the extraction of entire private keys, even if an attacker only gets to observe the power trace of a single decryption operation. Compared to previous attacks targeting RSA/ECC implementations, our attack setting is much more challenging since the key is processed in larger chunks, which leads to much less information in the power side channel.

This work was published at CHES 2017, and the author of this thesis has mainly contributed to developing/improving the attack methodology, the execution of practical experiments, and the paper write-up.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. “Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption.” In: *CHES*. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 513–533.

In Chapter 4, we show an improvement of our previously developed attack method that can recover ephemeral secrets during lattice-based encryption while requiring a significantly less powerful power-analysis adversary. We also demonstrate the practicality of this work on an off-the-shelf standard microprocessor.

This work was published at LATINCRYPT 2019, and the author of this thesis has mainly contributed to developing/improving the attack methodology and the paper write-up.

[PP19] Peter Pessl and Robert Primas. “More Practical Single-Trace Attacks on the Number Theoretic Transform.” In: *LATINCRYPT*. Vol. 11774. Lecture Notes in Computer Science. Springer, 2019, pp. 130–149.

**Part II** of this thesis is devoted to active implementation attacks and begins with a more in-depth introduction to the topic in Chapter 5.

In Chapter 6, we present statistical ineffective fault attacks (SIFA), an exploitation technique for fault attacks that is applicable in a wide variety of cryptographic applications and particularly hard to protect against. SIFA allows an attacker to infer information about cryptographic keys solely by analyzing the input/output of faulted but ultimately still correct cryptographic computations. Consequently, standard fault countermeasures like redundant computation cannot prevent this type of attack. Due to its statistical nature, SIFA is also convenient from an attacker’s perspective since only very limited knowledge about the attacked device is required.

This work was published at CHES 2018, and the author of this thesis has mainly contributed to finding a theoretical model for the attack, the execution of practical experiments, and the paper write-up.

[Dob+18b] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. "SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 547–572.

In Chapter 7, we analyze the applicability of SIFA in the context of nonce-based authenticated encryption schemes. While the application of many traditional fault attack techniques is prohibited by the uniqueness of the nonce during encryption and the built-in validation of data authenticity during decryption, we show how SIFA can be used to mount successful key recovery attacks in such a setting. In particular, we extend the idea of SIFA to target the initialization performed in nonce-based authenticated encryption schemes, which provides the attacker with an oracle on whether a fault was ineffective or not. We demonstrate the practicality of our analysis by presenting concrete exemplary attack strategies of SIFA targeting implementations of Keyak and Ketje.

This work was published at SAC 2018, and the author of this thesis has mainly contributed to improving the attack methodology, the execution of practical experiments, and the paper write-up.

[Dob+18c] Christoph Dobraunig, Stefan Mangard, Florian Mendel, and Robert Primas. "Fault Attacks on Nonce-Based Authenticated Encryption: Application to Keyak and Ketje." In: *SAC*. Vol. 11349. Lecture Notes in Computer Science. Springer, 2018, pp. 257–277.

In Chapter 8, we explore the capabilities of SIFA in the presence of masking countermeasures that were previously believed to offer effective protection against such attacks. More concretely, we show that fault inductions in the nonlinear layer of masked cryptographic operations can indeed have the required effect to allow key recovery via SIFA. These observations make SIFA an especially interesting attack technique against real-world cryptographic devices that feature protection mechanisms against both power and fault analysis.

This work was published at ASIACRYPT 2018, and the author of this thesis has mainly contributed to the general idea of the paper, the execution of practical experiments, and the paper write-up.

[Dob+18a] Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Stefan Mangard, Florian Mendel, and Robert Primas. "Statistical Ineffective Fault Attacks on Masked AES with Fault Countermeasures." In: *ASIACRYPT (2)*. Vol. 11273. Lecture Notes in Computer Science. Springer, 2018, pp. 315–342.

In Chapter 9, we present a novel algorithmic defense technique based on a specific combination of masking, redundancy, and reversible computing, that can provide effective protection from SIFA, amongst other common fault attacks. On top of that, our technique also allows the construction of masking schemes that are efficient in terms of runtime and energy consumption, which makes them particularly interesting for embedded applications where implementation attacks are of great concern.

This work was published at CHES 2020, and the author of this thesis has mainly contributed to the general idea of the paper, the execution of practical experiments, and the paper write-up.

[Dae+20a] Joan Daemen, Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Florian Mendel, and Robert Primas. "Protecting against Statistical Ineffective Fault Attacks." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.3 (2020), pp. 508–543.

**Other Contributions.** Besides the contributions presented in this thesis, the author of this thesis has also contributed to various other scientific works. These other works include cryptographic hardware designs [SP20; Nag+22], as well as further attack/defense techniques against implementation attacks [DMP22; Ham+21; Vaf+22; KPP20]. Moreover, the author has contributed to various works on formal verification of defense techniques [Gig+21; GPM21; GPM23; HPB21a; Blo+22] or analysis of cryptographic algorithms/modes [DMP22; ENP19].

Some of the works could have also been additionally used as part of this thesis. However, they were left out to allow this thesis to have a more specific scope. Last but not least, the author was involved in the submission of the authenticated encryption scheme ISAP to the standardization effort of lightweight cryptography by NIST [Dob+20], where it is currently competing in the final round [NIS18]. A full list of the (so far) 20 peer-reviewed articles that have been written in collaboration with 35 different co-authors can be found in Chapter 10.

## 1.2 A Brief Introduction to Cryptography

Cryptography is the theory and study of techniques behind secure communication and is one of the key building blocks of all products, services, and devices that build on information and communication technology. One of the most widely known applications of cryptography is the encryption of messages using a secret key, thereby ensuring their confidentiality while being transmitted over an untrusted communication channel. Besides encryption, there exist many more applications that make use of cryptography in one way or another. For example, hash functions can be used to ensure message integrity, while digital signatures or message authentication codes additionally ensure message authenticity. We now give a brief overview of prominent cryptographic algorithms and their applications. We organize our overview into two sections, symmetric cryptography, and asymmetric cryptography, based on how cryptographic keys are used

### 1.2.1 Symmetric Cryptography

Symmetric cryptography provides solutions for classical communication scenarios in which communicating parties share a secret key that typically corresponds to a string of 128 to 256 bits. For symmetric encryption, this key can be used in a *cipher* by either party to (1) encrypt a message into ciphertext and (2) decrypt a received ciphertext into the original message. The term *symmetric* refers to the fact that anyone in possession of the key can use it to perform both encryption and decryption, as illustrated in Figure 1.1a. One shortcoming of encryption schemes is that they do not provide a way to check the integrity or authenticity of messages. Hence, if an adversary on a communication channel does not only observe but also modifies data, a receiver might not be able to tell if a ciphertext (and the corresponding message) is trustworthy.

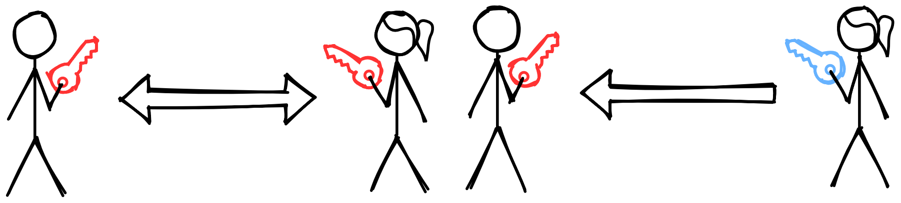
To overcome this problem, in practice, encryption is usually combined with message authentication code (MAC) functions. A MAC function calculates an additional check value, often also referred to as a *tag*, based on a message and a secret key. This tag can then be appended to a message and allows the receiver to verify if the message is unaltered and created by someone in possession of a particular secret key. Since, in practice, encryption is almost always used in tandem with MACs, there also exist dedicated *authenticated encryption* schemes that offer an interface for this combined functionality. Some prominent examples of authenticated encryption schemes that are frequently used for encrypting Internet traffic are AES-GCM, AES-CCM and ChaCha20-Poly1305 [For07; For15].

So far, all algorithms are described to be fully deterministic, i.e., same inputs give the same outputs. In practice, however, it is desirable to prevent repeated calls of cryptographic algorithms on the same inputs to result in the same outputs. This is due to the fact that, e.g., repeated observation of the same ciphertext/tag does leak some information about the plaintext. Hence, cryptographic algorithms often also take an additional *nonce* as input that differs for every call and can be sent along with a message/tag in plain.

If an application only requires message integrity, one can make use of a cryptographic *hash function*, i.e., an unkeyed and one-way function that compresses an arbitrarily large message into a fix-length hash (checksum). This hash can then be sent alongside the original message to be recomputed and compared by the receiver.

Over the last two decades, many different symmetric building blocks (ciphers, MACs, compression functions, ...) have been proposed for various cryptographic applications. One downside of this variety is the fact that usually, several of these building blocks are needed to realize classical communication scenarios. This comes with the cost of increased code size in software or chip area in hardware. While these costs are not necessarily problematic for higher-performance devices, they do impact the applicability of cryptography on lower-end devices.

A more recent research direction explores the usage of so-called *cryptographic permutations*, an unkeyed bijective function, together with keyed/unkeyed modes of operations to realize classical communication scenarios using just a single



**(a)** Symmetric encryption. **(b)** Asymmetric encryption.

**Figure 1.1:** A comparison of symmetric and asymmetric encryption schemes. **Red keys** need to be kept private while **blue keys** can be published.

building block. As of now, the most prominent family of cryptographic permutations is KECCAK-$p$ which is used in the NIST standardized hash function SHA-3 [Ber+11b; NIS15]. Permutation-based cryptography has also inspired the designs of authenticated encryption schemes such as ASCON [Dob+21], ELEPHANT [Bey+20], ISAP [Dob+20], PHOTON-BEETLE [Bao+21], SPARKLE [Bei+20], and XOODYAK [Dae+20b], which currently compete in the final round of the standardization effort for lightweight authenticated encryption by NIST [NIS18].

### 1.2.2 Asymmetric Cryptography

Asymmetric cryptography provides mechanisms for message authentication that can be verified using just public information and establishing a secret (symmetric) key over an untrusted channel. When compared to symmetric cryptography, this requires the usage of different types of keys and interfaces. Asymmetric cryptography operates on a pair of two keys, one of which needs to be kept private while the other one can be published. In this context, a private key should only be accessible by an individual person/entity, while secret keys in symmetric cryptography are shared with all communication parties. Given such a key pair, asymmetric cryptography can achieve security properties such as confidentiality and authenticity of messages without the need for sharing secret keys. In the case of asymmetric encryption, anyone in possession of the public key can use it to encrypt a message for the holder of the private key but not the other way around, as illustrated in Figure 1.1b. In the case of asymmetric authentication, more commonly known as *digital signature* generation, the roles of the keys are essentially reversed. Here, the private key is used to create a publishable signature, whereas the signer's public key can be used by anyone to verify the validity of the signature. An example of a cryptographic scheme for asymmetric encryption is the Rivest–Shamir–Adleman (RSA) cryptosystem [RSA78], while asymmetric authentication can be realized by the digital signature algorithm (DSA) [NIS94] or its elliptic-curve equivalent (ECDSA) [JMV01].

One prominent use case of asymmetric cryptography is the exchange of secret (symmetric) keys for secure communication on the Internet. A simple procedure for doing that starts with a client that requests a public (encryption) key from a web server. This key can be verified by the client for authenticity using another public (authentication) key that is stored directly on the client's machine (browser). To avoid the need for storing different authentication keys for every web server, so-called *certification authorities* act as a central root of trust that binds the identity of many entities (servers) to their respective public (encryption) keys using digital signatures. Once validated for authenticity, the webserver's public (encryption) key can then be used to send a newly generated symmetric key to the server for later use in faster symmetric authenticated encryption. This last step is commonly referred to as a key encapsulation mechanism (KEM) [CS03].

Symmetric and asymmetric cryptography differ not only in terms of key handling and interfaces but also in the way these schemes are constructed. Symmetric cryptography typically decomposes inputs (including keys) into small chunks of bit or byte granularity, and then applies a number of computation steps that ultimately result in highly nonlinear dependencies between chunks, which prevents the successful application of statistical analysis. In contrast, asymmetric cryptography typically relies on the difficulty of solving a more high-level mathematical problem that prevents an attacker from learning about a private key, given a corresponding public key (and other domain parameters). A typical example of a mathematical problem that is assumed to be computationally hard and used in the RSA cryptosystem is integer factorization [RSA78]. In this context, a recent research direction is the design and study of cryptographic algorithms that are based on mathematical problems conjectured to resist the potential future threat of quantum computers [Sho94]. To facilitate the standardization of such post-quantum cryptography (PQC) algorithms, NIST put out a call to submit candidates in 2016 [NIS16b] that has so far brought to light four candidates that have been selected for standardization: CRYSTALS-KYBER [Bos+18a] in the KEM category as well as CRYSTALS-DILITHIUM [Duc+18], FALCON [Fou+], and SPHINCS+ [Ber+19] in the digital signature category. Three of these candidates are based on mathematical problems in high-dimensional lattices. These lattice-based algorithms also turned out to become one of the most popular choices when designing PQC for low-end devices that are of particular interest in the context of implementation attacks.

## 1.3 A Brief Introduction to Implementation Attacks

Cryptographic algorithms are primarily designed to withstand mathematical attacks such as linear or differential cryptanalysis [Mat93; BS90] in a so-called *black-box* setting. In the black-box setting, an attacker is assumed to be able to make a certain amount of queries to a cryptographic algorithm while only being able to observe its inputs and outputs (cf. Figure 1.2b). In other words, an attacker is assumed to have no prior knowledge about the used key and is unable to observe any internal state of the cryptographic algorithm. The security of a cryptographic algorithm is then measured by the most efficient method of breaking its security properties in this setting. For a cryptographic algorithm to be considered secure, the complexity of the best-known cryptanalytic attack method has to be higher than the claimed security level and often coincides with the complexity of exhaustive key search.

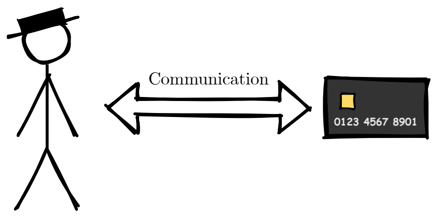
The black-box attack setting is reasonable for many applications of cryptography. For example, if we consider an attacker having access to public WiFi, they can usually only observe inputs/outputs of cryptographic computations, i.e., the ciphertexts corresponding to known/unknown messages, without having access to the communicating devices directly. However, there nowadays also exist many applications of cryptography that do involve physical devices that can rather easily fall into the hands of a potential attacker. A classic example of such an attack scenario are smart cards that are used, e.g., to realize electronic payments or physical/digital access control to critical infrastructure. In situations like these, cryptographic secrets stored on these cards, if revealed to an attacker, could be misused to issue fraudulent money transactions or to disturb the operation of critical infrastructure, like power grids or telecommunication. Additional examples include electronic passports that contain cryptographic keys for copy protection or access control mechanisms on personal devices (laptops or smartphones). Consequently, there nowadays exist mandatory certification procedures, such as Common Criteria [Com], for certain product categories that include lab testing against implementation attacks. Products are then only allowed to enter the market once they have successfully passed the certification.

As soon as a cryptographic algorithm, manifested in the form of a concrete implementation, falls in the hand of an attacker, it no longer finds itself in a black-box but rather in a *gray-box* setting. In the gray-box setting, an attacker has the additional capability to perform *implementation attacks* that involve physical manipulations or observations of physical properties (cf. Figure 1.2b). These improved capabilities can then be used to greatly simplify attacks, which can ultimately lead to key recovery with much higher efficiency than predicted in black-box settings.

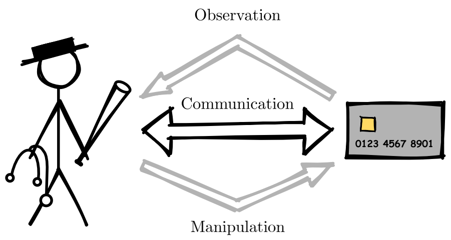
Depending on the attacker scenario and used equipment, implementation attacks can be roughly divided into two categories, passive and active implementation attacks.

### 1.3.1 Passive Implementation Attacks

Passive implementation attacks, on the other side, solely rely on exploiting observations of a device's physical properties. More concretely, by measuring so-called *physical side channels* of a device, one can again gain some knowledge about the internal state of a cryptographic computation, which can be used to significantly simplify breaking their security properties. These physical side channels are usually related to the behavior of semiconductor technology, such as metal-oxide-semiconductor (CMOS), that pretty much every current electronic device is based on. CMOS circuits have a primarily dynamic power consumption that depends on state transitions events, such as registers/wires changing their logical value. These events give away information about executed operations



**(a)** Black-box security: A typical attack setting for cryptanalytic attacks.



**(b)** Gray-box security: A typical attack setting for implementation attacks.

**Figure 1.2:** Black-box vs. gray-box security.

and processed data values via multiple physical side channels such as power consumption, electromagnetic radiation, photonic emission, or acoustic emission. We give a more detailed description of passive implementation attacks at the beginning of **Part I** .

### 1.3.2 Active Implementation Attacks

Active implementation attacks manipulate the execution of cryptographic computations in such a way that the resulting misbehavior simplifies breaking their security properties. A classic example involves an attacker repeatedly querying a device to perform an encryption of a constant plaintext while trying to force an erroneous device behavior via fault induction at a certain point in time. The attacker may then analyze the resulting valid/faulty ciphertexts to learn information about the cryptographic computation's internal state and, thus, about the used key. The fault induction itself can, for example, be realized by putting the attacked device outside of its specification ranges (clock speed, temperature, supply voltage) to produce faulty behavior. More sophisticated or invasive ways of disturbing the correct computation of a device can involve the usage of electromagnetic pulses, body-biasing induction, or laser fault induction. We give a more detailed description of active implementation attacks at the beginning of **Part II** .

# Part I

# Passive Implementation Attacks

＊＊＊

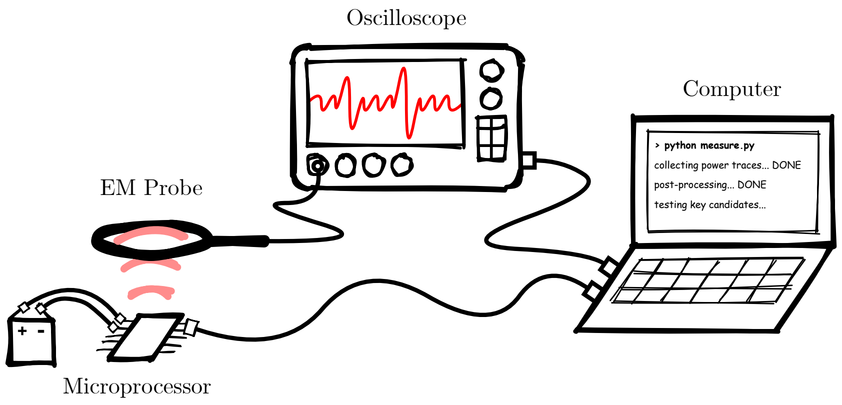
# 2
# A Primer on Passive Implementation Attacks

I have approximate knowledge of many things.

*Demon Cat - Adventure Time*

Passive implementation attacks extract sensitive information from electronic devices by analyzing information that is unintentionally leaked via their physical properties. More concretely, they measure so-called *physical side channels* that leak information about the internal state of (in most cases) cryptographic operations running on target devices, which can then be used to significantly simplify breaking their security properties. Physical side channels can range from power consumption, electromagnetic radiation, photonic emission, to acoustic emission and are related to the behavior of semiconductor technology. One example of such a technology that forms the base of almost every electronic device today is complementary metal-oxide-semiconductor (CMOS). During the computation of a CMOS device, the chip mostly draws power whenever state transitions events (i.e., changes in logical values) occur in logic gates, registers, or wires. Consequently, the instantaneous power consumption of a device carries information about currently executed operations and processed data values. This side channel can then be exploited by an attacker in various ways.

Electromagnetic radiation is produced by changes in the electric current flowing through the circuit (gates, registers, wires) that produce changes in the electromagnetic field and can be sensed by special probes in the form of small induced currents. The produced electromagnetic radiation is thus a result of the



```
> python measure.py
collecting power traces... DONE
post-processing... DONE
testing key candidates...
```

**Figure 2.1:** A typical EM analysis setup.

changes in the power consumption of the chip and depends on (sensitive) data that is processed by a device. These recorded *power traces* can then be evaluated using statistical analysis, machine learning algorithms, or pattern recognition methods.

A typical measurement setup for electromagnetic radiation is depicted in Figure 2.1. In this scenario, a microprocessor is connected to both a power supply and a computer. The computer uses a common communication interface to (1) send data to be processed by the microprocessor using a cryptographic operation with an unknown key and (2) receive the corresponding computation results. A physical attacker can now place a probe, suitable for recording electromagnetic radiation, close to the surface of the microprocessor. The probe itself is connected to an oscilloscope which allows the constant acquisition of side-channel information in the form of power traces. This allows an attacker to observe side-channel information about the cryptographic computations that are performed on the device when processing known inputs using an unknown key. In a similar spirit, an attacker can use different measurement equipment to capture e.g. photonic or acoustic emission of a chip which are also related to the chip power consumption. Independent of the concrete measurement method, the used exploitation strategies for extracting information from measurements are usually the same. In the following, we describe several well-known methods of extracting information from power traces, as well as corresponding defense techniques.

## 2.1 Overview of Passive Implementation Attacks

In this section, we provide descriptions of various commonly known techniques for performing passive implementation attacks that are particularly relevant to this thesis. These techniques mostly differ in the required sophistication of laboratory equipment, the required knowledge of an attacker about the attacked device, the cryptographic algorithm/implementation running on the device, and the communication interface to the device.

### 2.1.1 Simple Power Analysis (SPA)

SPA attacks exploit key-dependent patterns in the power trace of a cryptographic computation without the need to compare it to other traces/computations [MOP07; KJJ99]. These patterns can either stem from key-dependent sequences of operations or key-dependent processed data values. Even though SPA exploits only power traces corresponding to the processing of single inputs, an attacker may still repeat and average measurements corresponding to that input. This increases the ratio between exploitable signal and electronic/measurement noise. The term “simple” hence refers to the fact that in SPA, an attacker does not exploit the relation between multiple power traces corresponding to different inputs. This restriction is usually not by choice, but rather enforced by certain attack scenarios. For example, in the case of symmetric cryptography, SPA attacks can be used for attacking implementations of key expansion operations that expand a fix-sized secret seed into an arbitrarily long cryptographic key. Another classical application of SPA attacks is the exploitation of patterns in square-and-multiply algorithms, as they are used in unprotected implementations of RSA decryption. In such cases, one can usually distinguish rather easily whether a squaring or multiplication operation is performed, allowing a direct extraction of the secret exponent from a power trace.

Given that SPA attacks are rather simple in nature, their successful execution highly depends on appropriate measurement setups and a sufficient amount of side-channel information in the power trace. For example, the information content of the power side channel generally decreases with increasing word sizes of processors or increasing parallelism in hardware implementation. While this can be counteracted, e.g., with more localized measurement methods such as EM probes, EM analysis often requires a decapsulation of the attacked device, which makes the attack much more costly. Hence, when facing cryptographic code running on larger processors or parallel cryptographic hardware designs, exploitation techniques more sophisticated than SPA are usually required to perform successful attacks.

### 2.1.2 Template Attacks

Template attacks, first proposed by Chari et al. [CRR02a], represent a more sophisticated variant of SPA attacks that consist of two phases, an initial template building phase, and a later template matching phase.

During the template building, an attacker is assumed to gain full control over the attacked device. They may set arbitrary inputs to precisely characterize side-channel leakage, i.e., the dependencies between the computation of specific data and the power side channel. In practice, the leakage analysis is often only performed on one or very few specific operations that are executed by the device and are assumed to leak a sufficient amount of information. The goal of the attacker is to determine the data value which is processed by the attacked operation, solely based on measured power traces and the leakage analysis obtained during the template building phase. More concretely, during template building, a set of power traces is recorded for every possible data value $t \in T$ that can be processed by the target operation (sequence). Each set of power traces is then characterized by a multivariate normal distribution that is described by a tuple $(m_t, C_t)$ where $m_t$ is the mean vector and $C_t$ is the covariance matrix.

During template matching, the goal of the attacker is to determine the data value which is processed by the targeted operation (sequence), solely based on measured power traces and the templates obtained from the template building phase. More concretely, given a power trace $l$ corresponding to an unknown processed data value and a template $(m_t, C_t)$ corresponding to one possible processed value $t$, we can evaluate the probability density function of the $k$-variate normal distribution and calculate the probability:

$$\Pr(l|t) = \frac{\exp\left(-\frac{1}{2} \cdot (l - m_t)^{\mathrm{T}} \cdot {C_t}^{-1} \cdot (l - m_t)\right)}{\sqrt{(2\pi)^k \cdot \det(C_t)}}$$

By repeating this step for every possible $t \in T$, we can determine the most likely data value, which is given by the template with the highest probability. In some applications, one is not only interested in the template of maximum likelihood but instead in the probability distribution over all templates. Given an observed power trace $l$, the probability $\Pr(t|l)$ of data value $t$ and corresponding $(m_t, C_t)$ can be calculated using Bayes' theorem as follows:

$$\Pr(t|l) = \frac{\Pr(l|t) \cdot \Pr(t)}{\sum_{x=1}^{|T|} \left(\Pr(l|t_x) \cdot \Pr(t_x)\right)},$$

where $|T|$ denotes the number of possible data values and the prior probability $\Pr(t)$ is set to $(1/|T|)$ since its probability distribution is usually uniform.

### 2.1.3 Soft-Analytical Side-Channel Attacks (SASCA)

In SASCA, first proposed by Veyrat-Charvillon et al. [VGS14], one first performs template building for certain intermediate variables of a cryptographic implementation. In other words, one gets $\Pr(T = t|\ell)$, where $T$ is an attacked intermediate, $t$ runs through all of the possible values of $T$, and $\ell$ is the observed side-channel leakage. Then, one constructs a so-called *factor graph* that models the attacked algorithm and its specific implementation. A factor graph is a graphical model of the entire algorithm, including all relations between intermediate variables. After including the conditioned probabilities into this graph, the belief propagation (BP) algorithm is run, which returns marginal probabilities for all inputs (including the involved key components), outputs, and intermediates processed by the algorithm. We now give a more thorough descriptions of BP that is based on MacKay [Mac03, Chapter 26].

BP allows efficient marginalization in certain probabilistic models. Given is a function

$$P^*(\mathbf{x}) = \prod_{m=1}^{M} f_m(\mathbf{x}_m),$$

which is defined over a set of $N$ variables $\mathbf{x} \equiv \{x_n\}_{n=1}^{N}$ and the product of $M$ factors. Each of the factors $f_m(\mathbf{x}_m)$ is a function of a subset $\mathbf{x}_m$ of $\mathbf{x}$. The problem of marginalization is then defined as computing the marginal function

$$Z_n(x_n) = \sum_{\{x_{n'}\}, n' \neq n} P^*(\mathbf{x}),$$

or the normalized version $P_n(x_n) = Z_n(x_n)/Z$ , with $Z = \sum_{\mathbf{x}} \prod_{m=1}^{M} f_m(\mathbf{x})$.

BP solves this task efficiently by exploiting the known factorization of $P^*$. First, it represents the factorization in a probabilistic graphical model called factor graph (FG). Factor graphs are comprised of variable nodes, each representing one variable $x_n \in \mathbf{x}$, and factor nodes, each representing one $f_m$. Factor $f_m$ and variable $x_n$ are connected in the graph if $f_m$ depends on $x_n$. Second, it performs message-passing on the factor graph. Concretely, it iteratively runs the following two steps until convergence is reached:

1) from variable to factor:

$$u_{n \to m}(x_n) \;=\; \prod_{m' \in \mathcal{M}(n) \setminus \{m\}} v_{m' \to n}(x_n), \tag{2.1}$$

where $\mathcal{M}(n)$ denotes the set of factors in which $n$ participates.

2) from factor to variable:

$$v_{m \to n}(x_n) \;=\; \sum_{x_m \setminus n} \left( f_m(\mathbf{x}_m) \prod_{n' \in \mathcal{N}(m) \setminus m} u_{n' \to m}(x'_n) \right), \tag{2.2}$$

where $\mathcal{N}(m)$ denotes the indices of the variables that the $m$-th factor depends on and $\mathbf{x}_{m \setminus n}$ denotes the set of variables in $\mathbf{x}_m$ without $x_n$.

After convergence, the marginal function $Z_n(x_n)$ can be computed by multiplying all incoming messages at each node: $Z_n(x_n) = \prod_{m \in \mathcal{M}(n)} v_{m \to n}(x_n)$. The normalized marginals are given by $P_n(x_n) = Z_n(x_n)/Z$, where $Z = \sum_{x_n} Z_n(x_n)$.

BP is guaranteed to return the correct marginals only if the factor graph is acyclic. If this is not the case, then the same update rules can still be used in what is then called loopy BP. This variant might not even converge, but when it does, it often gives sufficiently precise approximations to the true marginals. The performance of loopy BP, i.e., the quality of the approximations and converge properties, is inversely proportional to the length of the loops. Put simply, loops in the factor graph introduce positive feedback, which can cause overconfidence in certain beliefs and subsequently even oscillations, especially when deterministic factors are involved. Longer loops are less susceptible to this effect. Note that there do exist approaches, such as generalized belief propagation [YFW00], aiming at significantly improving the quality of the marginal probabilities in loopy graphs. They, however, can come with significantly increased computational runtime.

### 2.1.4 Differential Power Analysis (DPA)

DPA, first proposed by Kocher et al., represents one of the most powerful classes of passive implementation attacks [KJJ99]. As the name suggests, during a DPA, one analyzes the differences in a device's power consumption when performing cryptographic computations on multiple different inputs. A classical DPA can be divided into three phases: measurement phase, hypotheses building phase, and hypotheses matching phase.

During the measurement phase, one collects power traces of a device when processing varying known inputs (plaintexts/nonces) using the same secret key. The second phase consists of generating hypotheses on parts of the secret key. Since this is, by design, infeasible for an entire cryptographic key, this is usually performed in a divide-and-conquer manner and requires identifying a point in the attacked implementation that depends both on known input and a key chunk that is enumerable. The hypotheses contain the potential values of the attacked intermediate result of the algorithm that is calculated for each input. As the dynamic power consumption of a CMOS circuit depends on changes of a signal rather than the absolute values of the intermediate value, the hypothetical intermediate values are then mapped to a power consumption model. The mapping to a power model can again be rather simple, for example, just calculating the number of bits that are nonzero (Hamming-weight model) or the number of bits that changed their value (Hamming-distance model) but can also become more complex and use power consumption characteristics of the attacked device. Finally, a hypothesis evaluation function tells us which of our hypotheses fits best to a given set of power traces. In DPA, a hypothesis evaluation corresponding to one guessed key value is usually performed for multiple known operation inputs. For each of those inputs a hypothesis (i.e., power consumption prediction) is created based on the used power model. These hypotheses are then evaluated using, e.g., the Pearson correlation coefficient to the recorded power traces at each point in time.

The biggest advantage of DPA over the previously discussed methods is that for DPA, an attacker does not need precise knowledge about the actual implementation or the attacked device. Additionally, since a hypothesis is evaluated over multiple traces, a DPA is usually more resistant to noise and therefore yields better results.

## 2.2 Countermeasures

In order to prevent attackers from performing successful power analysis attacks, extensive research on countermeasures has been conducted in the last two decades. In the research community, most focus is put on algorithmic countermeasures such as masking/hiding, or the design of cryptographic modes/protocols that reduce the attack surface of power analysis attacks.

### 2.2.1 Masking

The goal of masking is to randomize the representation of security-sensitive variables during the execution of cryptographic computations. The resulting side-channel leakage is then independent of the underlying data, which counteracts power analysis techniques like DPA. The most popular masking approaches are Boolean masking schemes, which are formed over finite field arithmetic in $\mathrm{GF}(2^n)$ and represent a natural fit for many symmetric cryptographic schemes.

In Boolean masking, a sensitive variable $x$ is split into a number of so-called *shares* (denoted $x_i$) which, when considered on their own or in conjunction of up to $d$ shares, are statistically independent of the corresponding *native* (non-shared) variable $x$. This degree of independence is usually referred to as the protection order $d$ and requires to split each variable with sensitive information into at least $d + 1$ shares. The shares are uniformly random in each execution, but at any time, it is ensured that the sum over all shares again results in the native variable $x$:

$$x = \bigoplus_i x_i = x_0 \oplus x_1 \oplus \cdots \oplus x_d \,.$$

In a similar manner, functions over shared variables are split into component functions $\mathrm{f}_i(\dots)$ such that again a correct and secure sharing of the original function is established:

$$\mathrm{f}(x, y) = \bigoplus_i \mathrm{f}_i(\dots) = \mathrm{f}_0(\dots) \oplus \mathrm{f}_1(\dots) \oplus \cdots \oplus \mathrm{f}_d(\dots) \,.$$

Throughout the entire implementation, a proper separation of shares and of the output of the component functions needs to be ensured in order to not violate the $d^{\text{th}}$-order independence, which is commonly expressed in the probing model of Ishai et al. [ISW03]. In the probing model, an attacker is modeled with the ability to probe up to $d$ intermediate results of the masked implementation. An implementation is said to be secure if the probing attacker cannot gain any statistical advantage in guessing any secret variable by combining the probed results in an arbitrary manner. While this share separation can be easily ensured for functions which are linear over $\mathrm{GF}(2^n)$ – for example, the masked calculation of $x \oplus y$ can be performed share-wise $(x_i \oplus y_i)$ –, the secure implementation of nonlinear functions usually requires the introduction of fresh randomness [ISW03; Bel+17; Rep+15a; Cnu+16; GIB18; GM17; GMK16; Bar+17].

As an example for a shared implementation of a nonlinear function, we can consider the generic masked multiplication algorithm by Ishai et al. [ISW03]. In

**Algorithm 2.1** : Masked GF($2^n$) multiplication according to Ishai et al. [ISW03] (ISW)

```
Input: $x_0, \dots x_d, y_0, \dots y_d \in \mathrm{GF}(2^n)$
Output: $q_0, \dots q_d \in \mathrm{GF}(2^n)$
 1: for $i = 0$ to $d$ do
 2:     for $j = i + 1$ to $d$ do
 3:         $r_{i,j} \overset{?}{\leftarrow} \mathrm{GF}(2^n)$
 4:         $t_{i,j} \leftarrow r_{i,j}$
 5:         $t_{j,i} \leftarrow r_{i,j} \oplus x_i y_j \oplus x_j y_i$
 6: for $i = 0$ to $d$ do
 7:     $q_i \leftarrow x_i y_i$
 8:     for $j = 0$ to $d$ do
 9:         if $i \neq j$ then
10:             $q_i \leftarrow q_i \oplus t_{i,j}$
```

order to securely calculate $q = x \cdot y$, each of the $d+1$ shares of $x$ is multiplied with each of the shares of $y$, resulting in $(d+1)^2$ multiplication terms. Subsequently, the multiplication terms are summed up together with fresh random variables denoted $r_{i,j}$, and distributed to the output shares $q_i$ (Algorithm 2.1).

A first-order masked GF(2) multiplication, which corresponds to the calculation of an AND gate, is given in the following:

$$\begin{aligned} q_0 &= x_0 y_0 \oplus r_{0,1} \\ q_1 &= x_1 y_1 \oplus (r_{0,1} \oplus x_0 y_1 \oplus x_1 y_0) \,. \end{aligned}$$

A uniform distribution of each of the shares of $q$ is ensured by the random $r$ shares. In general, the joint distribution of any $d$ shares of $q$ in the masked multiplication algorithm is uniform, or in other words, any $d$ shares are independently and identically (uniformly) distributed.

In case of asymmetric (i.e. lattice-based) cryptography, algorithms typically operate on elements in larger groups, which is why modular addition of shares is a more natural fit:

$$x = \sum_i x_i = x_0 + x_1 + \dots + x_d \bmod q \,.$$

### 2.2.2 Hiding

While masking can generally be seen as a data randomization technique, hiding techniques perform randomization in the time domain to reduce the dependency between processed data values and corresponding power consumption. While such approaches generally cannot be implemented to an extent where, e.g., performing a DPA becomes infeasible, they can still be used in addition to masking to further increase the practical difficulty of power analysis attacks. Hiding countermeasures are especially effective against profiling attacks like

SASCA that accumulate leakage over longer instruction sequences and hence require quite precise knowledge about the instruction order. Shuffling is one of the most popular hiding countermeasures and attempts to reduce the correlation between processed data and power consumption by randomizing the execution sequence of certain operations for a given algorithm. Then, an attacker cannot reliably distinguish power traces of the attacked operation from power traces of irrelevant operations. Consequently, the performance of DPA and template attack decreases since the hypothesis is partially evaluated on wrong data sets. Since the implementation of shuffling is rather easy, both in software and hardware, it is used in many real-world cryptographic implementations.

### 2.2.3 Mode-level Protection

A quite different approach to counteract power analysis attacks is to use cryptographic modes/protocols that can either reduce the cost of certain algorithmic countermeasures significantly or prevent certain attacks entirely.

In the former case, one can make use of so-called *leveled implementations*, a technique that was first used in the authenticated encryption scheme Ascon [Dob+21] and later formalized in [PSV15]. The basic idea behind leveled implementations is to restrict the need for algorithmic countermeasures to only certain parts of a cryptographic computation. In the context of authenticated encryption, this can be achieved, e.g., by making the initialization/finalization hard to invert, which prevents key recovery if an attacker recovers the cipher state during message processing. Consequently, if one is mainly concerned about DPA protection of the key, one does not need additional algorithmic DPA countermeasures during message processing which improves throughput significantly.

In the latter case, one can make use of a mode-level countermeasure against DPA attacks that is based on the Goldreich-Goldwasser-Micali (GGM) construction [GGM86] and essentially allows to transform a pseudo-random bit-sequence generator into a pseudo-random function. In the context of authenticated encryption, a GGM construction can be used to restrict the attacker to only observe the processing of two different (attacker-controlled) inputs under the same key, which prevents attacks like DPA. One example of an authenticated encryption scheme making use of a sponge-based equivalent of the GGM construction to achieve DPA protection is Isap [Dob+20].

* * *

# 3 Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption

The current public-key infrastructure is threatened by progress towards large-scale quantum computing. Public key algorithms whose security relies on the hardness of integer factorization or discrete logarithm problem, will succumb to Shor's algorithm [Sho99]. While estimates on the availability of quantum computers large/efficient enough to pose a threat for currently used public key cryptography vary greatly – they range from 15 years [Mar14] to never [Sha16] – the threat is still taken very seriously. This is demonstrated by, e.g., NIST's current call for post-quantum secure proposals [NIS16a] and official recommendations regarding post-quantum security from the NSA [NSA16].

When it comes to candidates for post-quantum secure algorithms, lattice-based cryptography appears to be a promising option and has garnered a lot of attention over the past decade. It proved to be versatile and efficient, as there already exist practical lattice-based constructions offering basic services such as public-key encryption, digital signatures, and key exchange. Furthermore, lattices also serve as the basis for new constructions such as homomorphic encryption.

A popular building block for lattice-based cryptography is a problem called learning with errors (LWE) [Reg05] and it's ring-variant (RLWE) [LPR10]. Given a system of random linear equations, all of which contain a small additive error, the LWE problem asks to find a solution for the (error-free) equation system. In matrix-vector notation, this problem can be stated as recovering secret $\mathbf{r_2}$ (or error $\mathbf{r_1}$) from $\mathbf{p} = \mathbf{r_1} - \mathbf{A}\mathbf{r_2}$. The decision version of this problem, which is the one actually used to build various cryptographic constructions, asks to distinguish between elements of $(\mathbf{A}, \mathbf{p})$ and $(\mathbf{A}, \mathbf{p}')$, where $\mathbf{p}'$ is sampled from a uniformly random distribution. The ring-variant of this problem introduces some additional cyclic structure into the coefficients of the equations ($\mathbf{A}$) that allows a more compact representation and hence also smaller key sizes.

Recent implementations of RLWE-based public-key encryption, e.g., [Cle+15; POG15; Roy+14], have shown that its performance is roughly on par with RSA and ECC-based systems on a large range of platforms. While these results demonstrate practicality, the implementation security aspect of lattice-based cryptography is still an fairly unexplored topic. Just like any other cryptographic algorithm, an unprotected implementation of RLWE-based encryption will succumb to side-channel attacks such as Kocher's differential power analysis (DPA) [KJJ99]. Due to the large number of linear operations in the en- and decryption process, masking [Cha+99] appears to be a natural fit for protecting lattice-based cryptosystems against DPA. In fact, there already exist several works showing that (first-order) masked implementations of lattice-based encryption can be implemented with (relatively) little $3 - 5\times$ runtime overhead [Ode+18; Rep+16a; Rep+16b; Rep+15b].

However, especially for public-key algorithms the simple power analysis (SPA) resistance is an attention point. This is demonstrated by, e.g., the large number of single-trace attacks targeting implementations of RSA and ECC. Yet, for lattice-based cryptography this aspect has not been analyzed before. As implementation techniques for RLWE-based schemes differ drastically from those of established public-key constructions, there are new potential venues for such single-trace attacks.

**Our Contribution.** In this chapter, we show that single-trace attacks are indeed a threat to implementations of lattice-based cryptography. We present a new side-channel attack on implementations of lattice-based encryption that can, given a practically realistic amount of leakage, recover the private key using just the side-channel observation of a single decryption. Hence, it can also be applied to masked implementations to recover each individual share, recombine them, and still perform full decryption-key recovery.

Our attack targets the computation of the number theoretic transform (NTT), which is an essential building block for almost all efficient implementations of lattice-based cryptography. Thus, the attack can be ported to not only different implementations of encryption, but also to implementations of other lattice-based constructions. Furthermore, the NTT is not the first target for a DPA attack and was thus less protected in earlier works [Ode+18].

Our attack is comprised of three main steps. First, we perform a side-channel template matching [CRR02b] on each modular operation performed during the inverse NTT in the decryption process. In the second step, we combine the information (probabilities of intermediate variables) of every operation in the entire NTT. We do so by representing the FFT-like structure of the NTT as a graph and then applying the belief propagation algorithm (BP). While the use of BP in context of side-channel attacks is not new [VGS14; GS15; GS18], it hasn't been used in the context of public-key encryption yet. In our setting, a simple implementation of BP would require an impractical amount of time. Thus, we designed several optimizations that are targeted specifically at the NTT analysis. In our third and final step, we combine the knowledge of some secret intermediate variables with the public key in order to reveal the private key. Concretely, we recover the full decryption key by first reducing the size of the public key and then performing a lattice decoding.

We evaluate our single-trace key recovery attack in two different settings. First, we determine the success rate in a generic Hamming-weight leakage model. There, our attack has a high success rate, i.e., $> 0.9$, with noise parameters of up to $\sigma = 0.4$. Second, to verify our findings in practice we use real traces from EM measurements of an ARM Cortex-M4F software implementation. In this latter scenario, we were always able to recover the decryption key. Finally, we also show that our attack performs similarly well even if masking is used.

**Outline.** In Section 3.1, we recall lattice-based encryption, efficient implementations, as well as proposed side-channel protection mechanisms. The three steps of the attack are then described in the following sections. The first step, a side-channel analysis of the NTT, is given in Section 3.2. Then, in Section 3.3 we efficiently combine all information using Belief Propagation. The third and final step, i.e., lattice decoding, is given in Section 3.4. We present and discuss the outcome and performance of our attack in Section 3.5.

## 3.1 Lattice-Based Encryption and Implementation

In this section, we recall efficient implementation techniques for lattice-based encryption as well as previous works on side-channel countermeasures. We only consider standard lattice-based encryption schemes (cf. Appendix A.2) that offer CPA security [Flu16]. While Oder et al. [Ode+18] recently presented an extension that also offers protection against adaptive chosen-ciphertext attacks (CCA2), the core encryption and decryption algorithms are identical, which is why we do not further discuss their CCA2 transformation here.

### 3.1.1 Efficient Implementation

There already exists a somewhat large body of work targeting efficient implementation of the above encryption scheme. They range from FPGAs to low-resource microcontrollers and desktop CPUs (e.g., [Cle+15; Göt+12; Liu+15; PG13; POG15; Roy+14]).

In our work we use the parameter set ($n = 256, q = 7681, \sigma = 4.51$), which was introduced by Göttert et al. [Göt+12] and is used by all of the above implementations. The concrete security level provided by this instance is still under debate and estimates vary (see, e.g., [APS15; GVW17; Ode+18]). However, all our later analysis can be extended to other parameters.

**Number Theoretic Transform (NTT).** If $q$ is prime, $n$ a power of two, and $q \equiv 1 \bmod 2n$ (which is the case for virtually all previously proposed parameter sets), then there exist primitive $n$-th roots of unity $\omega_n$ in $\mathbb{Z}_q$. This fact allows to efficiently compute polynomial multiplication in $\mathcal{R}_q$ by means of the number theoretic transform (NTT).

The NTT is essentially a discrete fourier transform (DFT) in a prime field $\mathbb{Z}_q$ instead of over the complex numbers $\mathbb{C}$. Thus, this transformation is efficiently computed using the same optimizations found in, e.g., the Cooley-Tukey FFT, and runs in time $\mathcal{O}(n \log n)$. The basic building block is a butterfly, which is comprised of a modular multiplication with a certain power of the chosen primitive root, a modular addition, and a modular subtraction. A total of $n \log_2(n)/2$ butterflies are computed during the NTT, as shown in Figure 3.1 with the example of a 4-coefficient NTT. The required powers of the primitive root, i.e., $\omega_n^0 \dots \omega_n^{n/2}$, are typically called *twiddle factors*. The inverse transformation (INTT) is computed by simply invoking the NTT with $\omega_n^{-1} \bmod q$. We denote $\hat{\mathbf{a}}$ as the NTT transformed of $\mathbf{a}$.

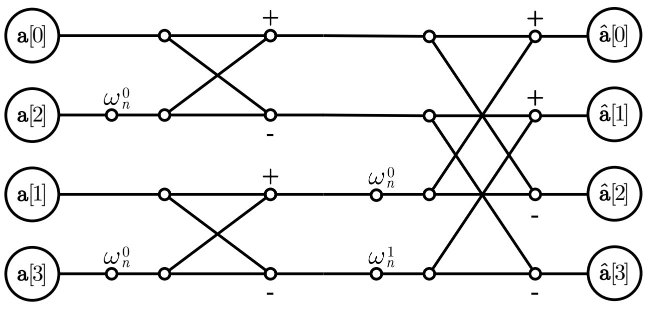


**Figure 3.1:** A 4-coefficient NTT network comprised of 4 butterflies.

Multiplication of two polynomials $\mathbf{a}, \mathbf{b}$ can now be implemented as $\mathbf{c} = \text{INTT}(\text{NTT}(\mathbf{a}) * \text{NTT}(\mathbf{b}))$, where $*$ denotes a pointwise multiplication[1]. Thus, a product can be computed in time complexity $\mathcal{O}(n \log n)$ (compared to $\mathcal{O}(n^2)$ for non-ring-based LWE constructions). This is one of the main arguments behind the choice of the particular ring[2] $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1 \rangle$.

As proposed by Roy et al. [Roy+14], the encryption scheme described in Appendix A.2 can be optimized by keeping the ciphertext in the NTT domain, i.e., transmitting $(\hat{\mathbf{c}}_1, \hat{\mathbf{c}}_2)$. This requires that the same primitive root $\omega_n$ is used for both encryption and decryption. Thus, it must be agreed upon and is public.

[1]This explanation is slightly simplified and omits, e.g., the scaling required for the negative-wrapped convolution. For a more thorough explanation, we refer to [Roy+14].

[2]There do exist proposals that are consciously avoiding the ring $\mathcal{R}_q$ and thus cannot use the NTT [Ber+16; Bos+16]. Still, NTT-enabled variants are the large majority.

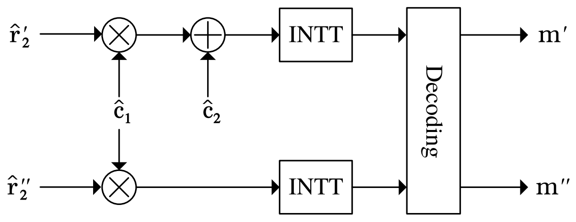


**Figure 3.2:** Basic masking scheme for decryption.

### 3.1.2 Side-Channel Protection of RLWE Encryption

Implementation security of lattice-based cryptography is still a fairly new topic. Nevertheless, there do exist several previous works that study potential protection mechanisms. We now discuss some of these proposals.

**Masking.** Due to the linear relationship between arithmetic masks for the main operations, i.e., polynomial addition and multiplication, arithmetic masking (cf. Section 2.2.1) is a natural fit for lattice-based encryption [Cha+99]. As proposed by Reparaz et al. [Rep+16b; Rep+15b] and shown in Figure 3.2, the private key $\mathbf{r}_2$ can be split into two shares $\mathbf{r}_2', \mathbf{r}_2''$ such that $\mathbf{r}_2 = \mathbf{r}_2' + \mathbf{r}_2'' \bmod q$. Then, polynomial multiplications, additions, and the inverse NTT can be computed on each share individually.

The final decoding step, i.e., recovering $\mathbf{m}$ from $\mathbf{m}^\star$, is nonlinear and requires more care. Reparaz et al. designed a masked decoder which outputs two binary shares of the message, i.e., $\mathbf{m} = \mathbf{m}' \oplus \mathbf{m}''$, which can then be used as a shared key in a protected implementation of, e.g., the AES.

**Shuffling and Blinding.** In addition to masking, Oder et al. [Ode+18] propose to use further countermeasures. First, they suggest to use shuffling to protect the pointwise operations, i.e., pointwise addition and multiplication. They state that these operations are the most likely target for a DPA attack. Hence, the NTT is still computed in an unshuffled manner.

And second, they also use a randomization technique previously proposed by Saarinen [Saa18]. They pick random values $a, b \in [1, q-1]$ and then multiply the coefficients $a \cdot \hat{\mathbf{c}}_1$, $b \cdot \hat{\mathbf{r}}_2$ and $ab \cdot \hat{\mathbf{c}}_2 \bmod q$. Due to the linearity of the NTT, the mask can be removed by multiplying the output of the INTT with $(ab)^{-1} \bmod q$.

**Additively Homomorphic Masking.** In a later work, Reparaz et al. present a different masking approach which exploits the additively homomorphic property of LWE [Rep+16a]. This, however, has some caveats. First, Reparaz et al. do not claim theoretical first-order security. And second, decoding errors are more likely. This makes their method incompatible with the CCA2-transformation presented by Oder et al. [Ode+18]. Due to these reasons, we do not further analyze the susceptibility of this approach to our attack.

## 3.2 Attack Step 1: Side Channels in an NTT Butterfly

After having covered all required preliminaries, we now start the description of our attack. As the first step of the attack, we exploit side-channel leakage during the computation of the inverse NTT in the decryption algorithm. Concretely, we first perform a profiling and then, for the actual attack, we match the recorded templates at each modular operation. As outcome, we obtain information in form of a probability vector for each such operation. In order to understand how much information a side-channel adversary can realistically expect in this first step, and to also allow attack evaluation in a realistic scenario, we performed a side-channel analysis of the NTT on a real device. We now discuss our targeted implementation and platform, the measurement setup, and some results of this analysis. We additionally introduce a generic and simpler Hamming-weight leakage model, which will later be used in addition to real traces. First, however, we explain the choice of the NTT as the primary target for our attack.

### 3.2.1 The NTT as Side-Channel Target

The number theoretic transform (NTT) is a main building block of virtually all efficient instantiations and implementations of lattice-based cryptography. Yet, thus far it has not been target of any side-channel analysis. One potential reason is that the pointwise operations, i.e., multiplications and additions while computing $\hat{\mathbf{c}}_1 * \hat{\mathbf{r}}_2 + \hat{\mathbf{c}}_2$, are the prime target for DPA attacks as they allow easy coefficient-wise prediction of intermediate variables [Ode+18]. However, this makes it tempting to use less protection in other parts, i.e., the NTT. Also, the NTT is an interesting target for algebraic side-channel attacks. As seen in Figure 3.1, it is comprised of many potentially leaking modular operations which are additionally connected by relatively simple algebraic rules. This makes it possible to combine the information of all leaking computations.

### 3.2.2 Measurement Setup and Implementation

We performed side-channel measurements on a Texas Instruments MSP432 (ARM Cortex-M4F) microcontroller on a MSP432P401R LaunchPad development board. A Cortex-M4F was also used by many other (protected) implementations of RLWE encryption [Cle+15; Ode+18; Rep+16b].

We exploit the EM side channel. As shown in Figure 3.3, we placed a Langer RF-B 3-2 near-field probe in proximity to the external core-voltage regulation circuitry. This setup does not require any on-chip spatial profiling. Also, we expect similar outcomes for a power analysis. Our microcontroller was clocked at its maximum possible frequency of 48 MHz.

We base our analysis on the implementation techniques used in the open-sourced Cortex-M4F implementation of de Clercq et al. [Cle+15], which is also the basis of the masked software implementation of Reparaz et al. [Rep+16b]. They

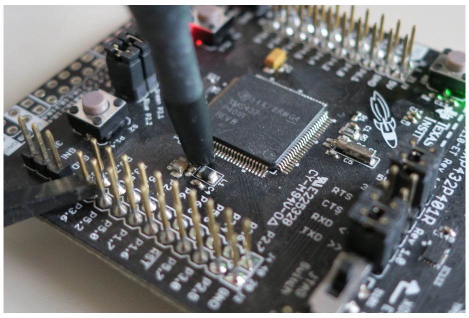


**Figure 3.3:** EM probe placed near the voltage-regulation circuitry of an ARM Cortex-M4F.

implemented modular multiplication with division, i.e., $a \bmod q = a - q\lfloor a/q \rfloor$, and use the integrated hardware multiplier and divider. On our platform, the multiplication runs in constant time, but the `DIV` instruction does not. Reduction after addition and subtraction is implemented using ARM conditional statements (`IT` instruction), which run in constant time.

### 3.2.3 Real-Device Side-Channel Analysis

The NTT is comprised of repeated applications of a butterfly. It is a reasonable assumption that all invocations utilize the same hardware, e.g., on-chip multiplier and divider, which results in loop-invariant leakage. To simplify our later analysis and attack evaluation, we thus opt for the following approach. We analyzed the butterfly operations, i.e., modular multiplication and addition/subtraction, independently. For the analysis, the operands were preloaded into registers and no leakage of loading and storing in memory was used. We prerecorded a set number of traces for each possible operand combination. For attack evaluation, we pick a random key, perform encryption/decryption, and for each of the $n \log_2(n)/2 = 1024$ butterflies invoked during decryption randomly pick one of the prerecorded traces that corresponds to the processed intermediate variable. We now describe our results for each operation in the butterfly.

**Modular addition and Subtraction.** de Clercq et al. implement modular addition and subtraction with conditional ARM statements. While these run in constant time, they still leak their state through other side channels. With a template matching, we were able to correctly classify virtually all, i.e., $> 0.99$, of the taken branches. In the following, we simply assume that an attacker can correctly detect whether a reduction happened or not. Alternatively, one could also include the probability that a reduction happened in the later analysis.

**Modular Multiplication.** In a butterfly, one of the inputs is multiplied by a known twiddle factor $\omega$. There are $qn/2 = 983\,168$ possible operand/twiddle factor combinations, for each of them we prerecorded 100 traces. Thus, we use roughly 100 million traces for evaluation. For the attack, for each multiplication we randomly pick one out of the 100 traces corresponding to the processed value.

In the analysis, we use two steps to recover information on the unknown input. First, we exploit that the runtime of division is data dependent. We found that it depends on the bit size of the dividend, i.e., the value that is reduced (the divisor is the constant $q$). By measuring this time, which we do with a simple thresholding in the side-channel trace, we can immediately assign the intermediate variable to one out of several disjoint sets.

In the second step, we perform a side-channel template matching [CRR02b] to further narrow down the operand. For each multiplication, we use 99 (remaining) traces to build templates for each currently possible operand. The points-of-interest used for template building were determined with a t-test [GLP06]. We then match all templates with the previously picked trace and compute the probability vector required for the next step of our attack.

In order to give a sense on the informativeness of our traces we use the metric proposed in [SMY09], i.e., give the average entropy left in the probability vectors conditioned on the leakage $\Pr(T = t|\ell)$. Without leakage, we have an entropy of $\log_2(q) \approx 12.9$ bit. After performing the template matching, the average entropy decreases to roughly 7 bit. However, we observed that the outcome somewhat correlates with the value of the used twiddle factor. With $\omega_n^0 = 1$ we have a remaining entropy of about 10 bits. With larger values, we generally achieve better results.

### 3.2.4 A Simplified Model

In order to allow reproducibility, we additionally analyze the performance of our attack with a more generic and simpler model, namely the common noisy Hamming weight leakage model. That is, apart from knowing if a reduction happened after addition/subtraction, for each modular multiplication an attacker gets two samples of the form:

$$l = (\mathrm{HW}(a) + \mathcal{N}(0, \sigma_l))||(\mathrm{HW}(a\omega_n^i \bmod q) + \mathcal{N}(0, \sigma_l))$$

$a$ is the unknown input and $\omega_n^i$ the used twiddle factor. HW denotes the Hamming weight function and $\mathcal{N}$ the Gaussian distribution with standard deviation $\sigma_l$. For the experiments, we perform a 2-variate template matching on these simulated traces.

## 3.3 Attack Step 2: Belief Propagation in the NTT

In the above template matching, the adversary obtains side-channel information on each computed butterfly. In the second step of the attack, we now combine all this information over the entire (I)NTT. We efficiently do so by using belief propagation (BP), described in Section 2.1.3. In our concrete scenario, we construct a factor-graph representation of the NTT, include the side-channel information in this graph, and then run BP until convergence is reached. With the constructed factor graph the runtime of a straightforward BP implementation is impractical. Thus, we present optimizations designed specifically for the NTT factor-graph, which decrease the runtime drastically.

### 3.3.1 Factor-Graph Construction

A factor graph is a bipartite graph containing variable nodes and factor nodes. For modeling the NTT, we add one variable node $x$ for each input/output of a butterfly. With $n = 256$, we thus have $n(\log_2(n) + 1) = 2\,304$ variable nodes.

We then add three types of factor nodes: $f_{\mathrm{ADD}}$, $f_{\mathrm{SUB}}$, and $f_{\mathrm{MUL}}$. As seen in Figure 3.4, each type of factor occurs once per butterfly. Thus, there are a total of $3n\log_2(n)/2 = 3072$ factor nodes in the NTT model. Evidently, there are cycles in the graph shown in Figure 3.4, so the loopy BP algorithm is needed.

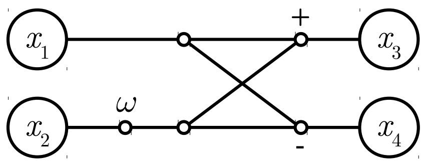
$f_{\mathrm{MUL}}$ is only connected to $x_2$ and thus has degree 1. Its purpose is to add the side-channel information gathered from the modular multiplication of $x_2$ with the known twiddle factor $\omega$. We performed a template matching in Step 1 and therefore are given vector of probabilities conditioned on the leakage $l$. Thus we have:

$$f_{\mathrm{MUL}}(x_{i_2}) = \Pr(x_2 = x_{i_2}|l)$$

The factors $f_{\mathrm{ADD}}$ and $f_{\mathrm{SUB}}$ represent the modular addition and subtraction, respectively. They are connected to both butterfly-input nodes $x_1$ and $x_2$, and one of the two output nodes $x_3$ or $x_4$. Thus, their degree is 3. These factors model how variable nodes inside a butterfly are related, e.g., that $x_3 = x_1 + x_2\omega \bmod q$. Furthermore, we use these factors to include whether a reduction happened after addition or subtraction, respectively. For addition with subsequent reduction step, we have:

$$f_{\mathrm{ADD}}(x_{i_1}, x_{i_2}, x_{i_3}) = \begin{cases} 1 \text{ if } x_{i_1} + x_{i_2}\omega \equiv x_{i_3} \bmod q \text{ and } x_{i_1} + (x_{i_2}\omega \bmod q) \geq q \\ 0 \text{ otherwise} \end{cases}$$

If no reduction happened, then the second clause $x_{i_1} + (x_{i_2}\omega \bmod q) > q$ is simply negated. For subtraction with subsequent reduction, we have:



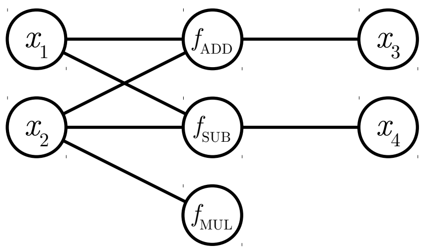


**Figure 3.4:** Butterfly network (left) and our corresponding factor graph (right)

$$f_{\text{SUB}}(x_{i_1}, x_{i_2}, x_{i_4}) = \begin{cases} 1 \text{ if } x_{i_1} - x_{i_2}\omega \equiv x_{i_4} \bmod q \text{ and } x_{i_1} - (x_{i_2}\omega \bmod q) < 0 \\ 0 \text{ otherwise} \end{cases}$$

**Other Leakage Points.** The factor-graph representation of the NTT is flexible, thus it can be modified to accommodate other leaking operations. One could, e.g., additionally include side-channel information of loading and storing in memory or leakage on operands of modular addition and subtraction.

### 3.3.2 BP Runtime Estimation without Optimization

As it turns out, the runtime of a straightforward implementation of BP on our constructed factor graph is impractically high. It depends on the number of iterations, the number of variable nodes and the size of their domain $\mathcal{D}$, as well as the number of factor nodes and their degree.

Each iteration of BP involves the invocation of the update rules $\mathfrak{q}$ (variable to factor, Equation 2.1) and $\mathfrak{r}$ (factor to variable, Equation 2.2) for all variable nodes and factor nodes, respectively. In our case the number of required iterations is small, e.g., $\leq 25$, and therefore does not have a significant impact on the asymptotic runtime. The runtime of $\mathfrak{q}$ is also fairly low.

However, the same cannot be said for $\mathfrak{r}$. For a factor $f$ with degree $\deg(f)$ and its inputs $x_1, \dots, x_{\deg(f)}$ with domain $\mathcal{D}$, one can compute the update rule given in Equation 2.2 by simply looping over all $|\mathcal{D}|^{\deg(f)}$ possible input combinations of $f$. In our scenario, we have factors $f_{\text{ADD}}$, $f_{\text{SUB}}$ with $\deg(f) = 3$ and variable nodes with domain size $|\mathcal{D}| = q = 7681$. When additionally multiplying with the number of $f_{\text{ADD}}$ and $f_{\text{SUB}}$ in our factor graph, then we reach a runtime of $\approx 2^{49}$ for a single iteration. Reducing from cubic to quadratic runtime can be done by only considering triplets where $f_{\text{ADD}}$, $f_{\text{SUB}}$ can be 1, but this still amounts to $\approx 2^{37}$ operations. Obviously, both numbers are not very practical and optimizations are needed.

### 3.3.3 Runtime Optimizations

In Algorithm 3.1, we show an optimization that can decrease the runtime of $\mathfrak{r}$ for all factor nodes of degree 3 in the factor graph, i.e., $f_{\text{ADD}}$ and $f_{\text{SUB}}$, drastically. We show it on the example of a factor node of type $f_{\text{ADD}}$. A slight variation of the presented algorithm can be used to optimize $f_{\text{SUB}}$.

Our optimization uses the fact that update rules for input/output distributions of modular additions/subtractions can be efficiently expressed in matrix-vector notation. Consider the addition $a + b = c \bmod q$, with $\mathfrak{q}_a, \mathfrak{q}_b, \mathfrak{q}_b$ the incoming messages from the corresponding variable nodes. Each such message is a $q$-dimensional vector assigning a probability to each value in $\mathcal{D}$, we say that $a_i = \mathfrak{q}_{a_i} = \Pr(a = i)$. The output $\mathfrak{r}_c$ depends on $\mathfrak{q}_a, \mathfrak{q}_b$ and an entry $c_k^* = \mathfrak{r}_{c_k}$ can be computed as the sum over all $a_i b_j$ with $i + j \equiv k \bmod q$. The whole update can be written in matrix-vector notation:

**Algorithm 3.1** Efficient BP for Modular Addition

```
Input:
    $\mathfrak{q}_a, \mathfrak{q}_b, \mathfrak{q}_c$    Incoming messages from summands and result node
    Reduction    True if a reduction step was executed
Output:
    $\mathfrak{r}_a, \mathfrak{r}_b, \mathfrak{r}_c$    Outgoing messages for summands and result node
1: $\hat{\mathbf{a}} = \mathrm{FFT}_{2q}(\mathfrak{q}_a)$, $\hat{\mathbf{b}} = \mathrm{FFT}_{2q}(\mathfrak{q}_b)$, $\hat{\mathbf{c}} = \mathrm{FFT}_{2q}(\mathfrak{q}_c)$
2: $\mathbf{t}_a = \mathrm{IFFT}_{2q}(\mathrm{CONJ}(\hat{\mathbf{b}}) * \hat{\mathbf{c}})$
3: $\mathbf{t}_b = \mathrm{IFFT}_{2q}(\mathrm{CONJ}(\hat{\mathbf{a}}) * \hat{\mathbf{c}})$
4: $\mathbf{t}_c = \mathrm{IFFT}_{2q}(\hat{\mathbf{a}} * \hat{\mathbf{b}})$
5: if Reduction then
6:     $\mathfrak{r}_a = \mathbf{t}_a[q \dots 2q-1]$, $\mathfrak{r}_b = \mathbf{t}_b[q \dots 2q-1]$, $\mathfrak{r}_c = \mathbf{t}_c[q \dots 2q-1]$
7: else
8:     $\mathfrak{r}_a = \mathbf{t}_a[0 \dots q-1]$, $\mathfrak{r}_b = \mathbf{t}_b[0 \dots q-1]$, $\mathfrak{r}_c = \mathbf{t}_c[0 \dots q-1]$
```

$$\begin{bmatrix} a_0 & a_{q-1} & \cdots & a_2 & a_1 \\ a_1 & a_0 & a_{q-1} & & a_2 \\ \vdots & a_1 & a_0 & \ddots & \vdots \\ a_{q-2} & & \ddots & \ddots & a_{q-1} \\ a_{q-1} & a_{q-2} & \cdots & a_1 & a_0 \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{q-2} \\ b_{q-1} \end{bmatrix} = \begin{bmatrix} c_0^* \\ c_1^* \\ \vdots \\ c_{q-2}^* \\ c_{q-1}^* \end{bmatrix},$$

where the columns of the left matrix are circular shifts of $\mathfrak{q}_a$. The above equation can be rewritten as a circular convolution $\mathfrak{q}_a \star \mathfrak{q}_b$, which can be efficiently computed using the FFT and the circular convolution theorem. Thus, we have :

$$\mathfrak{r}_c = \mathfrak{q}_a \star \mathfrak{q}_b = \mathrm{IFFT}_q(\mathrm{FFT}_q(\mathfrak{q}_a) * \mathrm{FFT}_q(\mathfrak{q}_b)).$$

The update rules for $\mathfrak{r}_a$ and $\mathfrak{r}_b$ can be obtained similarly by additionally using complex conjugations CONJ, as shown in Algorithm 3.1. Recall that we also include whether a reduction happened during modular addition and subtraction. This can be efficiently done by replacing the $q$-coefficient FFT with a $2q$-coefficient FFT and by using only either the upper or lower half of the IFFT output.

The runtime of computing $\mathfrak{r}$ for the degree-3 factor nodes is now reduced to $\mathcal{O}(q \log q)$, since the only runtime relevant operations are FFTs. This allows us to perform one iteration of the BP algorithm for our whole factor graph in about one minute using a single core of an Intel Core i7-5600U CPU.

### 3.3.4 BP on Subgraphs

In our experiments, we found that applying BP to the whole NTT factor graph does not yield satisfactory results. While we can narrow down values, the outcome was not sufficient for key recovery. Yet, we were able to identify two problems and show how to circumvent them by applying BP only to subgraphs:

- **Uneven distribution of side-channel information.** The template attack on multiplication is a primary source of information. Yet, multiplications are not spread evenly across the NTT, as illustrated in Figure 3.5a (also compare to Figure 3.1). Each cell of this figure corresponds to one variable node. White variables are multiplied with a twiddle factor, black ones are not. Due to the lack of multiplications and its side-channel information in the top-right corner, the BP algorithm cannot recover these variable with high-enough certainty.
- **Varying outcome of the template attack.** As already pointed out in Section 3.2.3, the performance of the template attack depends on the used twiddle factor. In the first NTT layer, one always multiplies with $\omega_n^0 = 1$. Even if this multiplication is not optimized out, the fact that no reduction is performed leads to little leakage.

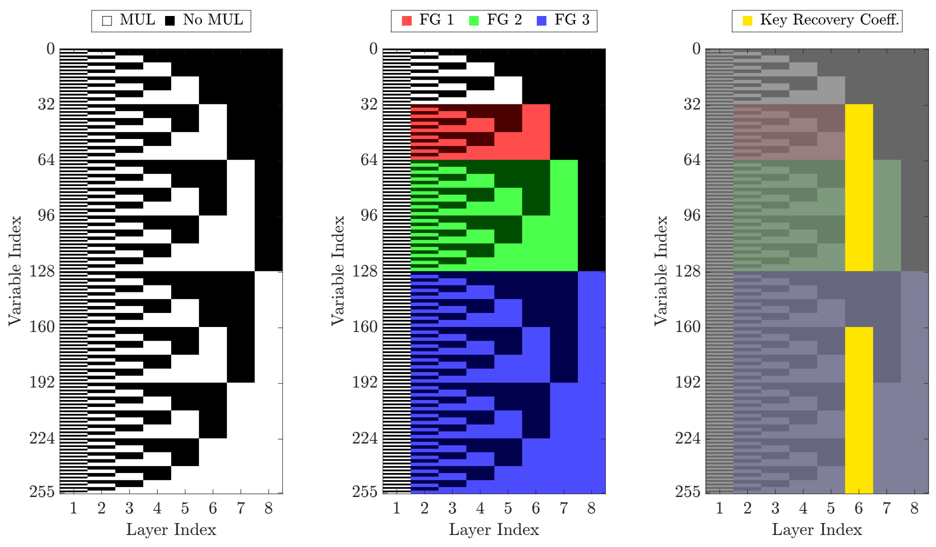
We circumvent these two problems by applying BP not on the whole NTT graph, but instead only on disjoint subgraphs. As depicted in Figure 3.5b, we have subgraphs FG 1, FG 2, and FG 3. These do not include the first layer and have a higher ratio of observed to unobserved variables (compared to the full graph). Thus, applying BP to these subgraphs gives significantly better results. After convergence is reached on all 3 graphs, we perform a classification, i.e., pick the most likely value, on certain variable nodes. Concretely, we use variables from layer 6 (output of layer 5 and input of layer 6). This is the last layer of FG 1 and variables in later layers are usually recovered with higher confidence. As shown in Figure 3.5c, we use the 192 variables with indices $32 \dots 128$ and $160 \dots 255$. If masking is used, then we have to perform BP twice to get the intermediate variables in both invocations of the INTT. The native intermediate variables can then be computed by simply adding the recovered variables of both INTTs.

## 3.4 Attack Step 3: Lattice Decoding

Due to applying BP only on subgraphs, we cannot recover the full INTT input $\hat{\mathbf{c}}_1 * \hat{\mathbf{r}}_2 + \hat{\mathbf{c}}_2$. Hence, the decryption key $\hat{\mathbf{r}}_2$ (or equivalently $\mathbf{r}_2$) cannot be determined with simple linear algebra and another step is needed. In this third and final attack step, we combine the recovered intermediate variables with the public key. First, we create linear equations in the variables and $\mathbf{r}_2$ and use them to decrease the rank of the lattice spanned by the public key $(\mathbf{a}, \mathbf{p})$. Then, we use lattice-basis reduction and decoding to find $\mathbf{r}_2$ in the reduced-rank lattice.

### 3.4.1 Generating Linear Equations in the Key

We use the recovered intermediate variables to construct linear equations in the private key $\mathbf{r}_2$. Polynomial multiplication in $\mathcal{R}_q$ can be written as a matrix-vector product. We write the INTT output as $\mathbf{m}^\star = \mathbf{c}_1\mathbf{r}_2 + \mathbf{c}_2 = \mathbf{C}_1\mathbf{r}_2 + \mathbf{c}_2$, where the columns of matrix $\mathbf{C}_1$ are nega-cyclic rotations of $\mathbf{c}_1$. All operations inside the



**(a)** Location of multiplications **(b)** Three used factor subgraphs **(c)** Intermediate variables used for key recovery

**Figure 3.5:** Representation of the NTT and used factors.

(I)NTT are linear, thus this system can be transformed to describe any of its intermediate variables. Concretely, we transform it such that it describes the recovered values of the sixth INTT layer.

We transform the system by performing a partial reversal of the INTT. We revert 3 butterfly stages by computing $x_1 = (x_3 + x_4)/2 \bmod q$ and $x_2 = (x_3 - x_4)/(2\omega)$ (cf. Figure 3.4). We end up with a system of form $\mathbf{C}_1'\mathbf{r}_2 + \mathbf{c}_2' = \mathbf{x}$, with $\mathbf{x}$ being the 192 recovered intermediate variables and $\mathbf{C}_1'$, $\mathbf{c}_2'$ the transformed coefficients.

## 3.4.2 Key Recovery using Lattice Reduction

The decryption key $\mathbf{r}_2$ is finally recovered by combining the above system with the information embedded in the public key $(\mathbf{a}, \mathbf{p})$. Recall that $\mathbf{p} = \mathbf{r}_1 - \mathbf{a}\mathbf{r}_2$. As $\mathbf{r}_1$ is small (it is sampled from a discrete Gaussian distribution with small $\sigma$), we have that $\mathbf{p} \approx -\mathbf{a}\mathbf{r}_2$. Thats is, $\mathbf{p}$ is close to the vector $-\mathbf{a}\mathbf{r}_2$ which is part of the $q$-ary lattice spanned by the columns of $\mathbf{A}$ (the matrix consisting of nega-cyclic rotations of $\mathbf{a}$). Hence, the recovery of $\mathbf{r}_2$ can be seen as a bounded-distance decoding problem. The chosen system-parameters $(n, q, \sigma)$ ensure that solving this decoding problem is not feasible without further information.

However, by incorporating the linear equations from above the problem can be reduced to a size that is solvable. We substitute the 192 equations $\mathbf{C}_1'\mathbf{r}_2 + \mathbf{c}_2' = \mathbf{x}$ into $\mathbf{p} = \mathbf{r}_1 - \mathbf{A}\mathbf{r}_2$ to get some $\mathbf{p}' = \mathbf{r}_1 - \mathbf{A}'\mathbf{r}_2'$. The number of columns of $\mathbf{A}'$, and hence the rank of the spanned lattice, is now reduced to $256 - 192 = 64$.

We then search for the closest vector to $\mathbf{p}'$ by solving a shortest-vector problem. Concretely, we search for the error term $\mathbf{r}_1$ (or $-\mathbf{r}_1$) as an unusually short vector in the lattice generated by $(\mathbf{A}'||\mathbf{p}')$. This approach of solving the lattice decoding problem is described by, e.g., Albrecht et al. [AFG13]. The short vector is recovered using the BKZ lattice basis reduction algorithm, we use the implementation provided by Shoup's NTL [Sho]. We invoke BKZ with a blocksize of 25, but abort reduction as soon as a candidate for $\mathbf{r}_1$, i.e., a vector with a small enough norm, is found.

After that, one can compute the private key $\mathbf{r}_2$ by solving the linear system $\mathbf{p} = \mathbf{r}_1 - \mathbf{a}\mathbf{r}_2$ for both recovered $\mathbf{r}_1$ and $-\mathbf{r}_1$. The correct $\mathbf{r}_2$ is the one that follows the distribution used for key generation. That is, we pick the smaller out of the two solutions.

**Performance of Decoding.** We tested the correctness and performance of this key recovery approach by performing well over 1 000 experiments. In each of them we use the correct intermediate variables (cf. Figure 3.5c) and only perform the decoding step. All our experiments were successful. The average runtime on a single core of a Xeon E5-2699v4 CPU is approximately 45 seconds.

This decoding approach is not limited to using exactly 192 recovered intermediate variables, it can be invoked with any number of coefficients. However, the runtime of decoding will increase if fewer values are available. For instance, with 160 recovered variables the average runtime is 5 minutes and thus still well within practicality. Below that, however, it increases drastically. With 150 values, it reaches multiple hours. Experiments with 146 or fewer coefficients were not successful after 1 full day of computation.

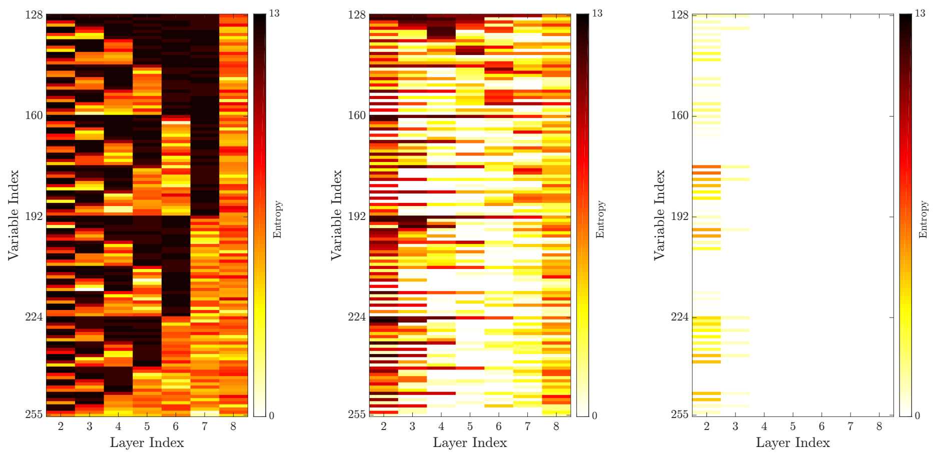
## 3.5 Attack Results and Conclusion

Our attack consists of subsequent execution of the three attack steps described in the previous sections. We now present the outcome. First, we evaluate the attack using real traces. We illustrate an exemplary outcome and give a success rate. Then, we give the success rate for the Hamming-weight model with varying noise-parameter $\sigma_l$, both with and without masking applied.

### 3.5.1 Real Device

With real traces obtained from the setup described in Section 3.2, we have the following results. Figure 3.6 illustrates an exemplary outcome of template-matching and the subsequent Belief Propagation on the subgraph FG 3 (cf. Section 3.3.4). For each variable node, we color-code the entropy of the probability vector. For black nodes, the probability distribution is close to uniform, whereas for white nodes one value has reached probability close to 1. After 1 iteration (Figure 3.6a), the probability distributions essentially correspond to the direct output of the template matching. After 20 iterations of BP (Figure 3.6c), the network has converged and almost all intermediate variables are determined with very high probability.

Lattice decoding is successful if all of the 192 variables used for key recovery are correct. After observing Figure 3.6, it should not come as a surprise that all our key recovery experiments in the real-trace setting were successful. The success rate, i.e., the probability that all used coefficients are correct, is 1.



**(a)** 1 iteration **(b)** 4 iteration **(c)** 20 iteration

**Figure 3.6:** FG 3: entropy after set number of iterations of BP.

### 3.5.2 Hamming-Weight Model

In order to get a broader and more generic analysis of our attack, we also tested it with a noisy Hamming-weight model (cf. Section 3.2.4). We rerun all tests with varying noise parameter $\sigma_l$. The outcome is illustrated in Figure 3.7, where we show the success rate and the average entropy (after template matching) for each tested value of $\sigma_l$. We give the entropy to allow at least a rough comparison to the real-trace setting.

In the non-masked case, we have a high single-trace success rate up to $\sigma_l = 0.4$ or 0.5, then it drops drastically. Note, however, that an attacker that can observe multiple decryptions can decrease the observed $\sigma_l$ by averaging the traces. In the masked setting, key recovery is successful if the correct intermediate variables are recovered in both invocations of the inverse NTT (see Figure 3.2). Only then their sum is equal to the native value. Thus, the expected success rate is squared, which is confirmed by our results. Obviously, averaging cannot be done if masking is used.

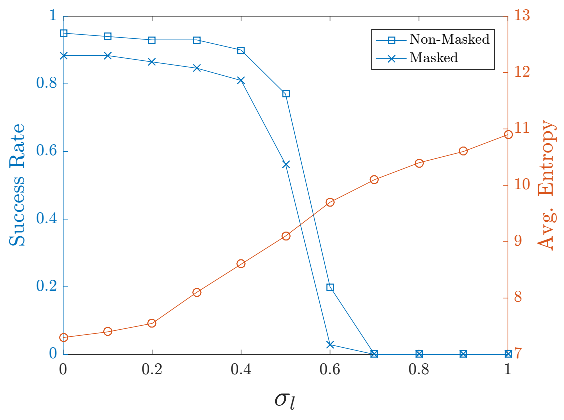


**Figure 3.7:** Success rates in the Hamming-weight leakage model.

### 3.5.3 Conclusion

Our attack clearly shows that SPA security of lattice-based schemes cannot be neglected and that relying on masking alone is not sufficient. Implementation techniques that are vastly different to established constructions such as RSA and ECC open up new venues in this regard. In fact, the regular structure of the NTT allows to efficiently combine leakage of the entire decryption process. Furthermore, each recovered intermediate variable can be used to decrease the difficulty of key recovery with the public key. And while this work focuses on lattice-based encryption, our attack can be adapted to any other implementation of lattice-based cryptography which employs the NTT.

When it comes to potential countermeasures, masking appears to be effective against DPA, yet it does not prevent our attack. Thus, additional countermeasures should be implemented and will now be discussed.

**Possible Countermeasures.** One of the first measures to strengthen an implementation against SPA attacks is to ensure a constant runtime and control flow. In our side-channel analysis of a real device, we exploit timing differences stemming from the `DIV` operation invoked during modular reduction. There do exist constant-time alternatives, as already shown by Oder et al. [Ode+18].

Like many other algebraic attacks, our key recovery can be thwarted by employing shuffling. Concretely, the operations inside the NTT, e.g., the order in which the butterflies are processed within one NTT layer, need to be shuffled. Shuffling only pointwise operations, as proposed by Oder et al., clearly does not hamper our attack. Other hiding countermeasures, such as the random insertion of dummy operations inside the NTT, can also make our attack harder.

Oder et al. also propose to use a blinding countermeasure (cf. Section 3.1.2). Our attack still applies, but needs an additional step and potentially a different selection of recovered intermediate variables. Concretely, it requires that a sufficient amount of the INTT output coefficients are recoverable or can be computed from the recovered variables. Then, one can test if the distribution of the unblinded INTT output, i.e., after multiplication with $ab^{-1} \bmod q$, corresponds to that of a valid $\mathbf{m}^{\star}$ (centered around 0 and $q/2$). For a non-masked implementation, or if the same blinding values $a, b$ are reused for both shares, then one can run through all $q-1$ possibilities of $ab \bmod q$. If different $a, b$ are used for both shares, then one needs to try all $(q-1)^2$ combinations. With our parameters, this can be easily done within a minute. When using 64 output coefficients, this always returned the correct blinding values in our tests. Hence, this countermeasure does not significantly increase single-trace security. It, however, prevents averaging in the non-masked scenario.

✳ ✳ ✳

# 4
# More Practical Single-Trace Attacks on the Number Theoretic Transform

In Chapter 3, we have proposed an attack on implementations of the number theoretic transform (NTT), which is an integral part of many lattice-based cryptographic schemes such as NewHope [Alk+17a] and Kyber [Ava+17]. The attack follows the path of soft-analytical side-channel attacks [VGS14]. That is, we first perform a side-channel template matching [CRR02b] on certain intermediate variables, construct a graph describing the NTT and all of its computation steps, include the observed leakage information in this graph, and finally run a message-passing algorithm known as belief propagation. The recovered secret input is either the key itself or can be used to recover said key.

However, while this attack can even bypass certain countermeasures, it does leave open the question of true practicality. For the evaluations on a real device, we need to build close to one million templates. Besides, we attack a variable-time implementation. While the attack, as such, does not require timing differences, it does benefit from them. And finally, we mainly focus on attacking the decryption process. This is the most apparent target and the involvement of long-term secrets makes the need for side-channel protections obvious. Encryption, however, only deals with ephemeral secrets and might thus see less care in side-channel protections. Still, a successful attack on encryption can lead to a compromise of the entire system.

**Our Contribution.** In this chapter, we address the above limitations and show that single-trace attacks on the NTT can be made truly practical. Several improvements to the attack, alongside the choice of a different target, allow us to attack a constant-time microcontroller implementation of the Kyber lattice-based key exchange [Ava+17], all while requiring only 213 univariate Hamming-weight templates.

More concretely, we include three improvements to belief propagation in the context of side-channel attacks on the NTT. We merge certain nodes in the graph representation of the NTT, make use of message damping, and introduce a new message schedule. These changes lead to higher accuracy of computed marginal probabilities and thus to better attack performance. The runtime of belief propagation, while increased, still stays very reasonable.

As already hinted above, we change the concrete target of our attack. In the previous work, we attacked the inverse NTT transformation during decryption. Decryption involves the private key, which makes not only attacks worthwhile, but also the need for careful side-channel protection obvious. We now target encryption instead. While this limits attacks to recovering the exchanged symmetric keys, it focuses on a part seemingly requiring less side-channel protection. Also, in encryption, the inputs of the NTT are confined to a narrow interval, which further aids attack performance.

These changes and performance improvements allow a simplification of the physical part of the attack. That is, by switching to Hamming-weight templates and targeting load/store operations instead of multiplications, the number of required templates and thus also traces for template building is cut down drastically.

We evaluate our attack for different noise levels using simulations. Furthermore, we study the effects of masking and recent implementation techniques, such as lazy reduction, on the attack performance. Finally, we demonstrate the attack using real power measurements of an STM32F4 microcontroller running a constant-time ASM-optimized Kyber implementation. Using just 213 univariate Hamming-weight templates, the entire secret NTT input can be recovered with a probability of up to 95%. Finally, we note that our attacks can be easily ported to many other implementations that make use of the NTT.

**Outline.** In Section 4.1, we briefly describe the concrete target of our attacks, namely the Kyber key exchange, and discuss its implementation aspects. In Section 4.2, we recall some details of our attack from Chapter 3 and also discuss its shortcomings. After having covered the necessary background, we show all our improvements and adaptations in Section 4.3. We then evaluate the attack using simulations in Section 4.4 and target a real device in Section 4.5. In Section 4.6, we discuss the applicability and effectiveness of previously proposed countermeasures.

## 4.1 Lattice-Based Cryptography

In this section, we briefly recall the lattice-based key-exchange Kyber. We also describe efficient implementation techniques, both for Kyber and lattice-based cryptography in general.

### 4.1.1 Kyber

In this work, we consider the specification of the KYBER key exchange [Ava+17] as used in the first round of the NIST standardization process [NIS]. In its core, KYBER resembles the RLWE encryption scheme proposed by Lyubashevsky, Peikert, and Regev [LPR10] as described in Appendix A.2 but it bases its security on the module learning-with-errors assumption (MLWE) [LS15]. This means that it operates with matrices/vectors containing polynomials defined over the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$. We use boldface letters to differentiate matrices/vectors of polynomials from single polynomials.

Already in its specification, KYBER prescribes usage of the NTT for efficient polynomial multiplication. Via pointwise multiplication of transformed polynomials, i.e., $ab = \mathsf{NTT}^{-1}(\mathsf{NTT}(a) \circ \mathsf{NTT}(b))$, multiplication can be performed in time $\mathcal{O}(n \log n)$. We use $\hat{a}$ as shorthand for the NTT-transformed of $a$, with $\hat{\mathbf{a}}$ we denote vectors where all component polynomials are transformed.

The core public-key encryption scheme (PKE) only offers IND-CPA security. For this reason, the KYBER authors apply a variant of the Fujisaki-Okamoto transform [FO99] to build an IND-CCA2 secure key-encapsulation mechanism (KEM). In essence, the transform requires a re-encryption of the decrypted message using the randomness seed used for the original encryption, which is embedded in the ciphertext. Only if the recomputed and the received ciphertexts match, the decrypted message is released. Since recovering the key/message used in the underlying PKE directly leads to key/message recovery of the KEM, we omit details of the transform and only focus on the PKE. We further omit aspects regarding, e.g., efficient packing, and give a simplified but conceptually identical description. For further details, we refer to the KYBER specification [Ava+17].

Algorithm 4.1 gives the key-generation procedure. The function $\mathsf{Sample}_\mathsf{U}$ samples the $(k \times k)$-matrix $\hat{\mathbf{A}}$ from uniform using the seed $\rho$, which is also part of the public key. The sampling is performed directly in the NTT domain. Then, the coefficients of $\mathbf{s}$ and $\mathbf{e}$ are sampled following a centered binomial distribution with support $[-\eta, \eta]$ using $\mathsf{Sample}_\mathsf{B}$ with seed $\sigma$. Afterward, the NTT is applied to all component polynomials of $\mathbf{s}$ independently to receive the secret key $\hat{\mathbf{s}}$. The result $\mathbf{t} := \mathbf{A}\mathbf{s} + \mathbf{e}$ is the public key.

**Algorithm 4.1** KYBER-PKE Key Generation (simplified)

```
Ensure: Public key $pk$, private key $sk$
1: Choose uniform seeds $\rho, \sigma$
2: $\hat{\mathbf{A}} \in R_q^{k \times k} := \mathsf{Sample}_\mathsf{U}(\rho)$          ▷ Generate uniform $\hat{\mathbf{A}}$ in NTT domain
3: $\mathbf{s} \in R_q^k := \mathsf{Sample}_\mathsf{B}(\sigma||0)$          ▷ Sample private key $\mathbf{s}$ (binomial distribution)
4: $\mathbf{e} \in R_q^k := \mathsf{Sample}_\mathsf{B}(\sigma||1)$          ▷ Sample error $\mathbf{e}$ (binomial distribution)
5: $\hat{\mathbf{s}} := \mathsf{NTT}(\mathbf{s})$          ▷ NTT for efficient multiplication
6: $\mathbf{t} := \mathsf{NTT}^{-1}(\hat{\mathbf{A}} \circ \hat{\mathbf{s}}) + \mathbf{e}$          ▷ $\mathbf{t} := \mathbf{A}\mathbf{s} + \mathbf{e}$
7: return $(pk := (\mathbf{t}, \rho), sk := \hat{\mathbf{s}})$
```

Encryption is shown in Algorithm 4.2. After recomputation of $\hat{\mathbf{A}}$ from the seed $\rho$, the variables $\mathbf{r}, \mathbf{e}_1, e_2$ are sampled. The seed $\tau$ used for this sampling is made explicit to allow the re-encryption required for the CCA2 transform. The ciphertext $c$ consists of two parts, where the second component $c_2$ contains $m$ encoded as an element in $\mathcal{R}_q$. The decryption process (Algorithm 4.3) requires to recover this $m$ from a noisy version.

**Algorithm 4.2** Kyber-PKE Encryption (simplified)

```
Require: Public key $pk = (\mathbf{t}, \rho)$, message $m$, seed $\tau$
Ensure: Ciphertext $c$
1: $\hat{\mathbf{A}} \in R_q^{k\times k} := \mathsf{Sample}_\mathsf{U}(\rho)$    ▷ Regenerate uniform $\hat{\mathbf{A}}$
2: $\mathbf{r} \in R_q^k := \mathsf{Sample}_\mathsf{B}(\tau||0)$
3: $\mathbf{e}_1 \in R_q^k := \mathsf{Sample}_\mathsf{B}(\tau||1)$    ▷ Sample noise $\mathbf{r}, \mathbf{e}_1, e_2$
4: $e_2 \in R_q := \mathsf{Sample}_\mathsf{B}(\tau||2)$
5: $\hat{\mathbf{r}} := \mathsf{NTT}(\mathbf{r})$    ▷ $\mathsf{NTT}$ for efficient multiplication
6: $\mathbf{c}_1 := \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$    ▷ $\mathbf{c}_1 := \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$
7: $c_2 := \mathsf{NTT}^{-1}(\mathsf{NTT}(\mathbf{t})^T \circ \hat{\mathbf{r}}) + e_2 + \mathsf{Encode}(m)$ ▷ $c_2 := \mathbf{t}^T\mathbf{r} + e_2 + \mathsf{Encode}(m)$
8: return $c := (\mathbf{c}_1, c_2)$
```

**Algorithm 4.3** Kyber-PKE Decryption (simplified)

```
Require: Public key $pk = (\mathbf{t}, \rho)$, secret key $sk = \hat{\mathbf{s}}$, ciphertext $c = (\mathbf{c}_1, c_2)$
Ensure: Message $m$
1: $m := \mathsf{Decode}(c_2 - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{c}_1)))$    ▷ $m := \mathsf{Decode}(c_2 - \mathbf{s}^T\mathbf{c}_1)$
2: return $m$
```

The Kyber authors originally specified three parameter sets. In this work, we primarily focus on the original Kyber768 set given by $(n = 256, k = 3, q = 7681, \eta = 4)$. Kyber512 and Kyber1024 mainly differ in the used $k$. Since we will target individual NTT executions, $k$ does not impact attack performance, at least as long the success probability on single NTTs is close to 1. We note that the Kyber parameters were tweaked for round 2 of the NIST standardization process. These parameter sets feature $(q = 3329, \eta = 2)$.[1] We will later show that this change is beneficial to our attack. Note that we will always use the original parameter set unless stated otherwise.

## 4.1.2 Efficient and Secure Implementation

The rising popularity of lattice-based cryptography in the last decade has also led to many efficient constant-time implementation techniques. As already stated above, the NTT allows efficient multiplication in $\mathcal{R}_q$ by pointwise multiplying two forward transformed polynomials and transforming the result back. A

[1] The new parameter set also requires some minor modifications to the NTT, such as different constants and omission of the last butterfly layer.

detailed description of the NTT is provided in Section 3.1. Besides that, there is opportunity to achieve more efficient and constant-time implementations by using device-specific functionality.

Early implementations of the NTT often used straightforward and variable-time modular reduction techniques. For instance, de Clercq et al. [Cle+15] use ARMs conditional operations for reductions after additions and subtractions, as well as integer divisions for reductions after multiplications. On most embedded devices, such divisions do not run in constant time.

More recent implementations, e.g., the Cortex-M optimized NewHope implementation by Alkim, Jakubeit, and Schwabe [AJS16], frequently make use of constant-time variants of the established Montgomery and Barrett reduction techniques. Constant-time is reached by omitting the final conditional subtractions. In other words, the result is not always reduced back to $[0, q-1]$, but can be larger. This can also be used for efficiency gains by, e.g., skipping reductions after additions (*lazy reduction*).

### 4.1.3 Protected Implementations

Constant-time operations mitigate timing attacks, both on small devices such as microcontrollers and large ones like PCs. Protecting against other types of side-channel attacks, e.g., differential power analysis (DPA), requires more effort. There do already exist works addressing this issue and proposing DPA-secured implementations of lattice-based cryptography; they use masking (cf. Section 2.2.1) as their main protection mechanism [Rep+15b; Ode+18; Bar+18].

Since the NTT is a linear transformation, it is trivial to mask. When $s$ is the sensitive input, then one can sample a uniformly random masking polynomial $m \in \mathcal{R}_q$, compute the NTT on $m$ and $(s - m)$ independently, and finally add the shares back again if needed. Masking the sampling of error polynomials and the decoding of the noisy message in decryption is much more intricate. Since we do not attack these operations, we refer to Oder et al. [Ode+18] for details.

Oder et al. [Ode+18] further employ hiding techniques. They shuffle the ordering of linear operations, such as the pointwise multiplication, and blind polynomials with a random scalar. The latter method was first introduced by Saarinen [Saa18].

## 4.2 Single-Trace Attacks on Lattice-Based Cryptography

Masking is very efficient in protecting against DPA-like attacks. Still, single-trace attacks are potentially able to bypass masking as well as other defenses. There do already exist earlier works showing the feasibility of such attacks in the context of lattice-based cryptography. Recently, horizontal side-channel attacks on matrix-vector multiplications found in schemes over unstructured lattices, such as Frodo [Alk+17b], were demonstrated [Ays+18; Bos+18b]. These attacks, however, do not carry over to schemes using structured lattices, as they typically use faster multiplication methods such as the NTT or Karatsuba's method.[2] In Chapter 3 we proposed a 3-step single-trace attack on the NTT that is also the basis of this work. First, we perform a side-channel template matching on each modular operation performed during the inverse NTT in the decryption process. In the second step, we combine the information of every operation in the entire NTT by representing the FFT-like structure of the NTT as a factor graph and then applying SASCA as described in Section 2.1.3. Finally, we combine the knowledge of some secret intermediate variables with the public key in order to reveal the private key. We now introduce additional background and then discuss some of the shortcomings of existing works.

**Single-Trace Attacks on the NTT.** As shown in Chapter 3, by running a soft-analytical side-channel attack (SASCA), one can recover the secret NTT inputs after observing just a single trace. Concretely, one can recover the inputs of the inverse NTT in decryption (Algorithm 4.3), and can then derive the key $\mathbf{s}$.[3] The NTT appears to be a fitting target for SASCA since each stored intermediate variable is computed using relatively simple combinations (additions and subtractions) of just two intermediate variables of the previous NTT layer.

Figure 4.1 demonstrates how one can construct a factor graph for the NTT. Figure 4.1a shows a single butterfly for reference. Note that since such a butterfly is equivalent to a length-2 NTT, we denote the outputs as $\hat{x}_0$ and $\hat{x}_1$. Figure 4.1b then depicts the corresponding factor graph that was used in the previous work. In this graph, variable nodes and factor nodes are represented by circles and squares, respectively. The factor nodes can be further split into two groups. Factor $f_\ell$ models the observed side-channel information, i.e., the outcome of the template matching. More concretely, we have $f_\ell(i) = \Pr(x = i|\ell)$, where $x$ is the matched intermediate variable. Template matching on the modular multiplication with $\omega$ can then be used to receive information on $x_1$.

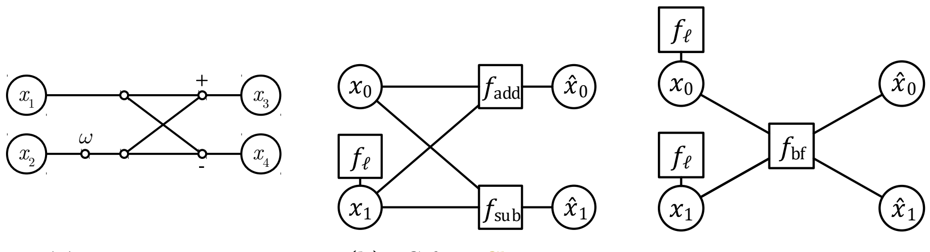
The second group of factors, consisting of $f_{\mathrm{add}}$ and $f_{\mathrm{sub}}$, then model the deterministic relationships between the variable nodes as specified by the NTT. For, e.g., the addition in the upper branch, we get:

$$f_{\mathrm{add}}(x_0, x_1, \hat{x}_0) = \begin{cases} 1 & \text{if } x_0 + x_1\omega = \hat{x}_0 \bmod q \\ 0 & \text{otherwise} \end{cases}$$

Due to the deterministic nature of $f_{\mathrm{add}}$ and $f_{\mathrm{sub}}$, the factor-to-variable update rule stated in Equation 2.2 can be computed in time $\mathcal{O}(q^2)$ by simply enumerating all $q^2$ possible input combinations. Our previous work can decrease the runtime to $\mathcal{O}(q \log q)$ by using cyclic properties of modular addition and FFTs of length $q$.

[2] Aysu et al. [Ays+18] do also run their attack for the RLWE-based scheme NewHope [Alk+17a]. However, their attacked implementation uses schoolbook multiplication instead of the NTT, resulting in a drastically increased runtime.

[3] The target is the original LPR scheme, which has very similar encryption and decryption routines.



**(a)** Single butterfly **(b)** FG from Chapter 3 **(c)** Our FG

**Figure 4.1:** Comparison of a single butterfly with possible factor-graph representations.

**Shortcomings.** While our previous work demonstrates the possibility of side-channel attacks on the NTT, the attack falls somewhat short of being fully practical.

We perform template matching on modular multiplications. This requires constructing templates for all possible combinations of $x_1$ and $\omega$ taking $q$ and $n/2$ possible values, respectively. For our evaluated parameter set, also featuring $(n = 256, q = 7681)$, close to a million templates are required. Each template is constructed using 100 traces, thus summing up to 100 million traces used for evaluation. We also assume time-invariance of leakage, which allows us to condense analysis to just that of butterflies without regarding its position in the trace. This assumption might not always hold (cf. Section 4.5.2).

In addition, we attack the variable-time NTT implementation by de Clercq et al. [Cle+15], which makes use of ARMs conditional instructions and variable-time integer division. Note that the attack as such does not require timing leakage, it can easily be adapted to constant-time implementations. However, the inclusion of timing information is beneficial to attack performance. Thus, the applicability to constant-time implementations is unknown.

Finally, we note that the butterfly factor graph shown in Figure 4.1b contains short loops, which is detrimental to the performance of BP. In fact, due to the bipartite and singly-connected nature of factor graphs, no shorter loops are possible in any such graph.

## 4.3 Reaching Practical Single-Trace Attacks

We now address these problems and show how single-trace attacks on the NTT can be made truly practical, even on constant-time implementations. First, in Section 4.3.1, we decrease the number of required templates. In combination with the lack of timing information, the attack now fails. For this reason, we adopt several improvements to the belief-propagation algorithm for our scenario, as explained in Section 4.3.2. Then, in Section 4.3.3, we explain why attacking encryption instead of decryption can boost performance even further.

### 4.3.1 Decreasing the Number of Templates

Our method to decrease the number of required templates and thus also traces for template building to a more considerate amount is relatively simple. Instead of performing a template matching on modular multiplication and constructing templates for each possible input combination, we target loading/storing of butterfly inputs/outputs to and from RAM (cf. Figure 4.1c using leakage of just loading inputs). In addition, we do not construct templates for every single possible value, but only for Hamming weights. Under ideal circumstances, this limits the number of templates to just $\lceil \log_2 q \rceil + 1 = 14$, which, compared to the previous attack, is a reduction by a factor of over 70 000. Apart from this reduction, (univariate) Hamming-weight templates are also significantly easier to port from one device to another (compared to multivariate value-based templates).

As it turns out, however, the information loss due to switching to such simpler templates and additionally losing all timing leakage–we only target constant-time implementations–is too high. The attack fails, even for the noise-free case. For this reason, we will now propose improvements to the attack, which allow successful message recovery for such a more constrained attacker.

### 4.3.2 Improving Belief Propagation for the NTT

There already exists a large body of work studying ways to improve the performance of belief propagation in cyclic factor graphs. We adopted three concrete methods for use with the NTT and will now describe them in depth.

**Butterfly Factors.** The factor graph shown in Figure 4.1b contains very short loops, which, especially in conjunction with deterministic factors, can lead to convergence problems and overall bad performance of loopy BP [Sto03]. Such network configurations and the resulting problems are however not exclusive to the NTT. As shown by Storkey [Sto03] and Yedidia [Yed03], similar problems also appear when applying BP to other FFT-like networks. Storkey analyzes BP in context of ordinary real-valued FFTs, whereas Yedidia focuses on Reed-Solomon codes, which can be represented by NTTs/FFTs over GF($q$).

To increase BP performance, both Stork and Yedidia propose to cluster the factors belonging to the same butterfly and thereby enforce all of its input/output relations at once. We follow their approach and replace factors $f_{\text{add}}$, $f_{\text{sub}}$ with a single *butterfly factor* $f_{\text{bf}}$. As seen in Figure 4.1c, this eliminates the loop inside each butterfly. The full factor graph of the NTT, built by connecting multiple instances of the butterfly FG (cf. Figure 3.1), will still contain loops. These loops, however, are longer, which will lead to increased performance. Yedidia notes that this clustering constitutes a simple form of generalized belief propagation [Yed03; YFW00]. Butterfly factors are specified as:

$$f_{\text{bf}}(x_0, x_1, \hat{x}_0, \hat{x}_1) = \begin{cases} 1 & \text{if } x_0 + x_1\omega = \hat{x}_1 \bmod q \text{ and } x_0 - x_1\omega = \hat{x}_1 \bmod q \\ 0 & \text{otherwise} \end{cases}$$

With the increased accuracy, however, comes also an increase in computational runtime. By making use of an FFT of length $q$, the update rules for $f_{\text{add}}$ and $f_{\text{sub}}$ can be evaluated in time $\mathcal{O}(q \log q)$. This does not carry over to $f_{\text{bf}}$. Instead, all input combinations need to be enumerated, thereby increasing the runtime to $\mathcal{O}(q^2)$.

For typical parameters of lattice-based encryption, with $q \approx 2^{13} - 2^{14}$ [Ava+17; Alk+17a], this is still very much practical, as will later be demonstrated. For the moduli used by lattice-based signatures, e.g., the schemes Dilithium [Lyu+17] and qTesla [Bin+17] both use $q \approx 2^{23}$, practicality cannot be claimed anymore. When also considering that the NTT in, e.g., Dilitihium, consists of $2^{10}$ butterfly invocations, then it becomes clear that the previous method with split factors needs to be used there.

**Optimized Message Schedule.** Another property that can influence convergence and accuracy in loopy BP is the chosen message schedule, i.e., the order in which messages are computed and passed between nodes. The most straightforward schedule is to update all variable or factor nodes simultaneously. This approach is followed in Chapter 3. When using the original representation of Figure 4.1b, then completing a loop requires just 2 iterations of evaluating Equation 2.1 and Equation 2.2 (for our proposed representation, this number increases to 4). It, however, can take up to $2 \log_2 n$ iterations for any two nodes in the full NTT graph to communicate.

This is clearly not ideal, which is why we adopt the schedule also used by Storkey [Sto03]. That is, we first pass messages from the NTT input to the output (layer by layer), and then back again. This does not affect the number of iterations required for completing a loop but allows any two nodes in the factor graph to communicate in just a single iteration. This becomes especially advantageous when changing the target to encryption, which features inputs with small support.

**Message Damping.** While the above two methods greatly increase accuracy, convergence is still not guaranteed. For this reason, we finally also adopt message damping. It aims to dampen oscillations by computing a weighted average of the new and the previous message. When denoting $\alpha$ as the damping factor and $u^{\text{prev}}_{n\to m}$ as the message sent from node to factor in the previous iteration, then the dampened version of Equation 2.1 is:

$$u_{n\to m}(x_n) = \alpha \left( \prod_{m' \in \mathcal{M}(n) \setminus \{m\}} v_{m' \to n}(x_n) \right) + (1-\alpha) u^{\text{prev}}_{n\to m}(x_n) \qquad (4.1)$$

For all of our later experiments, we set the damping factor $\alpha$ to 0.9.

### 4.3.3 Changing Targets

The decryption process described in Algorithm 4.3 involves the secret key $\mathbf{s}$ and is thus the obvious first target of a side-channel attack. We now argue that encryption, while not involving $\mathbf{s}$, can also be a very interesting target and is significantly easier to attack.

First, encryption only deals with ephemeral secrets. While this means that the side-channel attack has to be performed for each individual message, the lack of long-term secrets makes it very tempting to use implementations devoting fewer resources for side-channel protections, or maybe even an unprotected implementation. We want to prove this intuition wrong.

Second, the Fujisaki-Okamoto CCA2-transform employed by Kyber and many other lattice-based KEMs requires a re-encryption of the message. The message is then only released if the recomputed ciphertext matches the received one. This means that attacks are not restricted to sending devices, but can also be mounted on the receiving end, i.e., on devices having access to the secret key.

Third, encryption involves an NTT with inputs over a very narrow support: error polynomials follow a centered binomial distribution over $[-\eta, \eta]$, with $\eta = 4$ in our analyzed parameter set. The inputs to the inverse NTT in decryption, however, can be considered uniform in $\mathbb{Z}_q$. Information on the narrow support can easily be integrated into the factor graph; it suffices to apply Bayes' theorem to the leakage-likelihood obtained in the input layer and use the result in the factor nodes $f_\ell$. The first iteration of our forward-backward message passing then immediately spreads the information to all later layers and ensures, e.g., that in the second layer, only $(2\eta + 1)^2$ values have a non-zero probability. The narrow support allows BP to pick the correct value under much more noise, as we will later demonstrate.

Fourth and finally, we note that key generation (Algorithm 4.1) also features an NTT with small inputs, namely that of the private key $\mathbf{s}$. Thus, our attack also applies here. This becomes interesting on devices which either use the plain IND-CPA secure scheme described in Section 4.1.1 and thus only use ephemeral keys, or only store the seed $\sigma$ and regenerate $\mathbf{s}$ each time to save space in secure non-volatile storage.

**Attacking Encryption.** For the actual attack on encryption, we target the forward NTT of $\mathbf{r}$, found in Line 5 of Algorithm 4.2. We picked this invocation of the NTT, since all others in Algorithm 4.2 work on polynomials which are uniform over $\mathbb{Z}_q$. Since $\mathbf{r} \in \mathcal{R}_q^k$ is comprised of $k$ polynomials, we have to attack all $k$ independent invocations of the NTT. Then, we can compute the message $m$ by using Line 7. That is, $m = \mathsf{Decode}(c_2 - \mathbf{t}^T \mathbf{r}) = \mathsf{Decode}(e_2 + \mathsf{Encode}(m))$.

## 4.4 Evaluation and Additional Scenarios

After having described our attack in depth, we now evaluate it using leakage simulations and additionally explore more attack scenarios. First, in Section 4.4.1, we analyze the improvements proposed for BP in the context of the NTT. Then, we focus on implementations employing the masking countermeasure in Section 4.4.2. In Section 4.4.3, we finally study the effects of state-of-the-art implementation techniques.

**Leakage Simulation.** For all our simulations, we make use of the established *Hamming weight with additive Gaussian noise* model. Thus, when processing a value $x$, we receive simulated leakage $\ell = \mathsf{HW}(x) + \mathcal{N}(0, \sigma_{\mathsf{HW}})$. Here, $\mathsf{HW}$ denotes the Hamming-weight function, and $\mathcal{N}(0, \sigma_{\mathsf{HW}})$ describes a random sample from the normal distribution with zero mean and standard deviation $\sigma_{\mathsf{HW}}$.

For each simulation run, we generate one such sample for each intermediate state variable. Thus, we have $n$ samples in each layer (including inputs and outputs), this sums up to $n(\log_2(n) + 1) = 2\,304$ samples. Note that in practice, intermediate variables in inner layers of the NTT would leak twice: once during storing, once during loading. For the sake of simplicity, we only generate a single sample here. Finally, we perform a template matching on all generated samples and retrieve the corresponding conditioned probabilities $\Pr(X = x|\ell)$.

**Attack Implementation.** We implemented our evaluations and attacks, including the belief-propagation algorithm, in Matlab. The most time-critical component, namely the factor-to-variable update of butterfly factors $f_{\mathrm{bf}}$, was outsourced to multi-threaded C++ code. We performed up to 50 full message-passing iterations but abort as soon as convergence is reached for all variable nodes in the network. All experiments were run on an Intel Xeon E5-4669 v4 (2.2 GHz).

Evaluation of attack performance is done for varying values of $\sigma_{\mathsf{HW}}$. For each scenario and analyzed noise level, we performed at least 100 experiments. We computed the success rate by counting the experiments where belief propagation correctly classifies all $n$ NTT inputs, i.e., assigns the highest probability to the correct values.

### 4.4.1 Evaluating Improvements to BP

For our first evaluations, we analyze the effects of the improvements proposed in Section 4.3.2: the introduction of butterfly factor nodes, a changed message schedule, and the use of damping. This evaluation also doubles as a general analysis of the noise resistance of our attack.

We target a generic constant-time but otherwise unprotected implementation of the Kyber-NTT. The exact internal operations of this implementation are not important, at least as long the factor graph model and the actual implementation are consistent regarding the attacked intermediate variables. As we target the loads and stores at the inputs and outputs of butterflies, the exact methods used for, e.g., modular multiplication, are mostly irrelevant.

Figure 4.2 shows the outcome, for both the original ($q = 7681, \eta = 4$) and the tweaked ($q = 3329, \eta = 2$) Kyber parameter sets. Without the improvements, but still using the same load/store leakage, a success rate of more than 0.9 can be maintained up to and including $\sigma_{\mathsf{HW}} = 0.9$. With our changes, this threshold is increased to $\sigma_{\mathsf{HW}} = 1.5$. When computing an SNR and thus looking at the variance $\sigma^2_{\mathsf{HW}}$, then this difference corresponds to almost a tripling of the acceptable noise level. The smaller values used by the new parameter set lead to a slight improvement in the success rate. On a single core of our system,

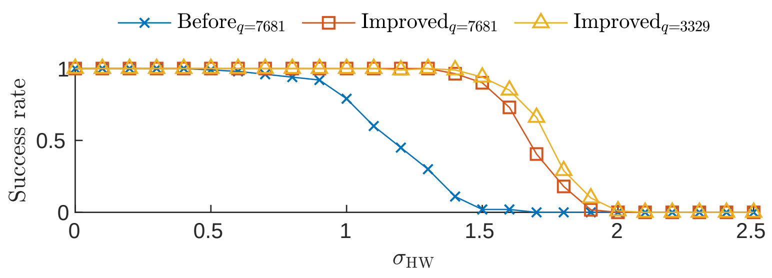


**Figure 4.2:** Comparison of the attack success rate, with and without the optimizations proposed in Section 4.3.2. The improved version was evaluated for both the original ($q = 7681$) and tweaked ($q = 3329$) parameter sets.

the runtime of a full forward-backward iteration of belief propagation using $q = 7681$ is roughly 2 minutes. In low-noise cases, 2 such iterations are already sufficient. For $\sigma_{\mathsf{HW}} = 1.5$, the average number of iterations (for convergent experiments) rises to 9. For this noise level, all failed experiments correspond to non-convergence of BP.

### 4.4.2 The Case of Masking

As noted in Section 4.1.3, masking the NTT is straightforward. So is, at least in theory, the adaptation of a single-trace attack to the masked case. One can simply recover each share individually and add them up to receive the unmasked input. This approach is used in Chapter 3. According to the evaluations, masking alone does not significantly decrease the attack success rate. In our scenario, this is no longer the case. We specifically target the NTT of **r** due to the small input coefficients. When using masking, this advantage is lost, as all input coefficients become uniformly distributed over $\mathbb{Z}_q$.

We reintroduce the information on the narrow support as follows. Instead of running belief propagation on two factor graphs corresponding to the two shares individually, we adjoin graphs at the input layer using factor nodes $f_{\mathrm{bino}}$. These nodes ensure that the sum of the two inputs is consistent with the centered binomial distribution over $[-\eta, \eta]$. When using $\mathcal{B}_\eta$ to denote the density of said distribution, and $x', x''$ as the two shares of the input, then we can write $f_{\mathrm{bino}}(x', x'') = \mathcal{B}_\eta(x' + x'' \bmod q)$.

Figure 4.3 shows the simulation results. Running BP on the shares independently does not yield satisfactory results, even for perfect Hamming-weight leakage. The introduction of $f_{\mathrm{bino}}$ brings the success rate close to 1, at least when $\sigma_{\mathsf{HW}} \leq 0.3$. While this is a drastic reduction compared to the unprotected case, it at least allows attacks in low noise scenarios. Here, also note that our attacker only uses Hamming-weight templates. A more powerful adversary might attack masked implementations even in high-noise settings.

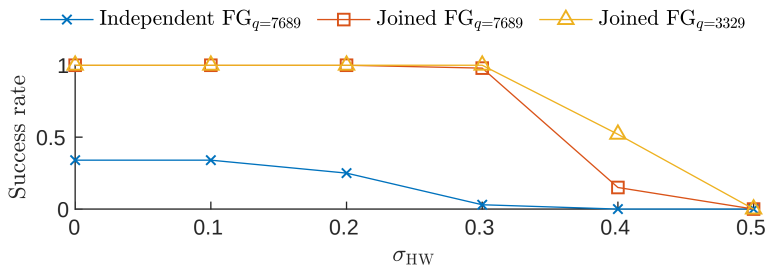


**Figure 4.3:** Success rate of our attack on a masked implementation, both using independent factor graphs and joined graphs.

## 4.4.3 The Case of Lazy Reductions

Up until now, we analyzed a relatively generic implementation of the NTT. There, operands are always in the range $[0, q-1]$. As mentioned in Section 4.1.2, this is not true for many recent implementations. They make use of constant-time variants of the Montgomery and Barrett reduction, both of which do not necessarily reduce operands down to the base range. Instead, they reduce to a representative (of the equivalence class) only guaranteed to be smaller than, e.g., $2^{16}$. Also, reductions after, e.g., additions, can be skipped for performance improvements (lazy reduction).

Such an implementation could, at least theoretically, be attacked using the same factor-graph representation. After performing the template matching on the now larger range of possible values, one could compute the probability of each equivalence class by summing up the probability of each possible representative. Due to just using Hamming-weight leakages, we do not think that such an approach is fruitful.

Instead, we modify the graph to directly model the changed operations. Concretely, we target the assembly-optimized ARM Cortex M4 implementation of Kyber provided by the PQM4 library [Kan+][4]. It uses Montgomery reductions after modular multiplications and Barrett reductions after additions and subtractions. Reductions after additions are skipped in each other layer. We integrate all that in the butterfly factors. When denoting $\mathsf{MRed}$ and $\mathsf{BRed}$ as the used reduction routines, and looking at a layer where no reduction is skipped, then

[4]Shortly after the initial publication of this article, the Kyber implementation in PQM4 was updated. For reference, we used the version found at `https://github.com/mupq/pqm4/releases/tag/Round1`.

we have

$$f_{\mathrm{bf}}(x_0, x_1, \hat{x}_0, \hat{x}_1) = \begin{cases} 1 & \text{if } \mathsf{BRed}(x_0 + \mathsf{MRed}(x_1\omega)) = \hat{x}_0 \text{ and} \\ & \quad \mathsf{BRed}(x_0 + 4q - \mathsf{MRed}(x_1\omega)) = \hat{x}_1 \\ 0 & \text{otherwise} \end{cases}$$

The results of the simulations using this model are shown in Figure 4.4. The analyzed implementation only supports the original parameter set with $q = 7681$, which is why we limit analysis to this scenario. Note, however, that our earlier results indicate better attack performance for the tweaked parameter set. Compared to our earlier results, the 90% success-rate threshold is now decreased to $\sigma_{\mathsf{HW}} = 1.3$. As an effect of the larger input ranges, the single-core runtime of a full iteration increases to approximately 8 minutes.

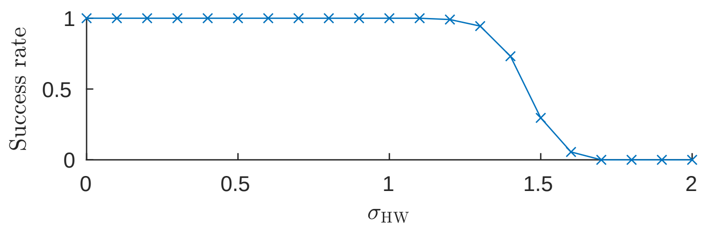


**Figure 4.4:** Success rate of our attack when modeling constant-time and lazy reductions.

## 4.5 Attacking a Real Device

The previous section already analyzed an optimized microcontroller implementation, but still resorts to leakage simulations. We now show that our attack carries over to an actual device.

### 4.5.1 Measurement Setup

The ARM Cortex M4 appears to be the standard target for embedded software implementations of post-quantum cryptography [AS18]. For this reason, we also performed our side-channel analysis on such a device. More concretely, we performed power measurements of an STM32F405 microcontroller atop the STM32F4 target for the ChipWhisperer UFO board [New]. The power consumption was measured using an AD8129A differential amplifier across an onboard shunt resistor. The 8 MHz device clock was externally generated using a function generator. This was done to simplify synchronization across traces.

This device then ran the optimized Kyber implementation of the PQM4 library [Kan+], which we already targeted in the previous section. More correctly, we only run the forward NTT of error polynomials. Each trace captures an entire NTT execution; we used a dedicated trigger pin to signal the start and end of this operation. We recorded 2 000 traces and then split this set into 1 900 templating traces and 100 attack traces. Note that, while both sets of traces were recorded on the same device, the use of simple univariate Hamming-weight templates makes porting of templates to similar but different devices much more realistic compared to the previous attack.

### 4.5.2 Trace Analysis and Attack

As we want to demonstrate practicality, we keep the trace analysis relatively simple by using univariate Hamming-weight templates. For determining the position of leaking operations, we run a correlation analysis along the entire length of the traces, for each of the 2 304 attacked intermediate variables. The single position of the highest peak (in absolute value) was then selected as point of interest.

The switch to Hamming-weight templates would ideally allow a reduction to just 14 templates. However, we found that the power leakage does show a certain degree of time dependence. Even after basic trace normalization, such as pointwise subtraction of the mean and normalization to a standard deviation of 1, the same operation showed slightly different behavior when executed at a different point in time. We suspect the different flow of instructions preceding our attacked operations, alongside the low-pass behavior of the power network, to be the cause.

Construction of templates for each position is still not required, as we follow an intermediate approach. We build two sets of templates for each NTT layer. The first set targets the upper branch in all butterflies of the respective layer, the second set targets the lower branch (cf. Figure 4.1a). Each template is thus used $n/2 = 128$ times. As not all Hamming weights are possible in all layers, one has to build a total of 213 templates.

After matching these templates on an attacked trace, we use the factor-graph representation already established in Section 4.4.3. Out of our 100 performed experiments, 83 yield the correct NTT input. Since we need to attack $k = 3$ independent NTTs, the total success rate can be estimated to be $0.83^3 \approx 0.57$.

### 4.5.3 Increasing the Success Rate

The stated success rate can be improved by making use of lattice-reduction techniques. Previously, we defined an attack to be successful if all $n$ NTT inputs are correctly recovered. For the full KYBER scheme, one needs to accomplish this $k$ times to recover all $nk$ coefficients of $\mathbf{r}$.

One can, however, also recover the message when only using the $nk - l$ most probable coefficients, for some small $l$. That is, one only picks the $nk - l$ coefficients where the final probability of the most likely value is closest to 1. These values are then plugged into the equation $\mathbf{c}_1 := \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$, the remaining $l$ coefficients can then be recovered using lattice-reduction techniques. Such an approach was also used by in our previous work Chapter 3, which is why we do not give further details here.

When using $l = 120$, a conversion to unique-SVP [Alb+17], and the BKZ lattice-reduction algorithm with block size 25, then the unknown coefficients can be recovered in approximately $5 - 10$ minutes. When picking $k = 3$ out of the 100 real experiments at random, then the success rate is increased to 0.95 when using this approach with the stated parameters.

## 4.6 Countermeasures

In the previous sections, we established that single-trace attacks on the NTT can be made truly practical. Several improvements to the underlying use of belief propagation in conjunction with the exploitation of small coefficients allow attacks even with simple Hamming-weight templates. This clearly shows that countermeasures are needed even for encryption where no long-term secrets are involved. We now discuss some possible options.

**Masking.** Masking is firstly a DPA countermeasure, but in our scenario also somewhat counteracts single-trace attacks. They are still possible, as shown in Section 4.4.2, but the acceptable noise level is drastically decreased. Nonetheless, further countermeasures are likely needed to protect against more sophisticated attackers.

**Blinding.** Saarinen [Saa18] and Oder et al. [Ode+18] make use of a blinding technique, which can be seen as a simple form of masking. They blind the two to-be-multiplied polynomials by first multiplying them with two random scalars $(a, b) \in \mathbb{Z}_q$. The product is then unblinded via a multiplication with $(ab)^{-1} \bmod q$. Note that this scalar blinding does keep the narrow support of, e.g., $\mathbf{r}$, intact. The concrete values of the support are however changed. Such a countermeasure might not be able to prevent attacks. First, one can mount a horizontal side-channel template attack, e.g., on all $n$ multiplications with $a$, to recover the fixed blinding value. Second, one can model the blinding value as an additional variable node in the factor graph and let belief propagation recover its value. We do not further study these scenarios here.

**Shuffling.** Similar to other algebraic side-channel attacks, shuffling is probably a very effective countermeasure. By randomizing the order of executed operations within each NTT layer, the leakage points cannot be trivially assigned to the correct variable nodes anymore. Note that shuffling linear operations, such as pointwise multiplications, was proposed by Oder et al. [Ode+18], but does not affect an attack on the NTT. We leave an analysis of the required granularity of shuffling and the overall cost of this countermeasure for future work.

# Part II

# Active Implementation Attacks

＊＊＊

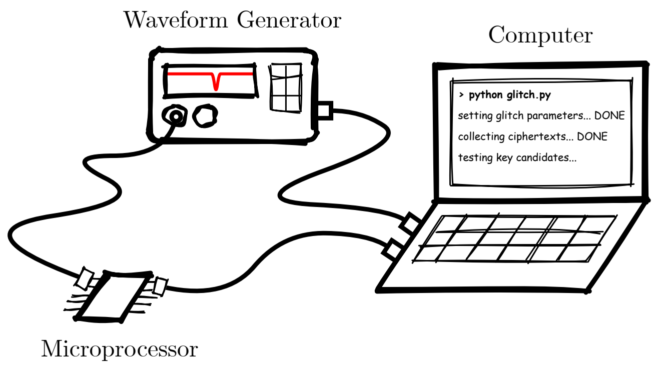
# 5 A Primer on Active Implementation Attacks

I don't know exactly what went wrong, but I know it's always my fault.

*Homer Simpson - The Simpsons*

Active implementation attacks circumvent security mechanisms on electronic devices by causing small hardware corruptions during the execution of normal device functions. A typical way of performing active implementation attacks is to put the target device outside of its operational specification (clock frequency, temperature, supply voltage) to force the occurrence of faulty behavior with, ideally, a specific effect and at a specific point in time. The resulting effect of an induced fault typically ranges from skipping entire (sequences of) instructions to setting the contents of registers or outcomes of computations to known/unknown values. These effects can then, if timed correctly, be used to bypass various security mechanisms present on the target device.

In practice, the reliability/precision of fault inductions highly depends on the available laboratory equipment, the target device, and the concrete exploitation method. Less precise fault induction setups may cause unintentional (side-)effects that differ with each fault induction attempt. The exploitation then becomes a much more time-consuming endeavor. In general, attackers try to minimize the number of fault induction attempts such that exploitation is possible within a reasonable amount of time.



```
> python glitch.py
setting glitch parameters... DONE
collecting ciphertexts... DONE
testing key candidates...
```

**Figure 5.1:** A simple fault attack setup using voltage glitching.

A simple example of a fault attack setup is shown in Figure 5.1. Here, a microprocessor is connected to a computer to provide a basic communication interface. The microprocessor is additionally connected to a waveform generator which serves as the power supply. In this scenario, an attacker can configure the waveform generator to insert a sudden voltage drop in the power supply at a specific point in time. The resulting temporary faulty behavior can then be potentially exploited by the attacker in various ways, e.g., depending on whether the faulty behavior affects the execution of firmware itself or the execution of a particular cryptographic computation.

In the former case, an attacker could attempt to boot the target device with a modified firmware image from an external storage chip that exposes sensitive internal memory contents to a public communication interface. Doing so is, however, not necessarily straightforward since secure boot procedures on the device typically verify the authenticity of a firmware image before use. In this situation, a fault induction can help an attacker, either by forcing a false positive outcome of the authenticity check or by simply skipping the authenticity check entirely.

Alternatively, an attacker may also choose not to target the firmware itself but instead extract a cryptographic key that is used by the device for sending/receiving sensitive information. The analysis of fault inductions on cryptographic computations already has a long history that reaches back about two decades to the seminal work of Boneh et al. [BDL97] in 1997. After they pointed out the importance of releasing only correctly computed RSA signatures, it quickly became clear that also symmetric cryptographic schemes are susceptible. Since then, the analysis of fault inductions on cryptographic algorithms has attracted a great amount of interest from industry and the academic research community, and many more different kinds of cryptanalytic methods have been developed to exploit erroneous computations for all kinds of cryptographic algorithms. In the following, we describe different methods of exploiting and counteracting fault inductions on implementations of symmetric cryptography that are particularly relevant to our later presented contributions.

## 5.1 Overview of Active Implementation Attacks

In the context of symmetric cryptography, a rich field of research emerged that focuses on techniques to recover the secret key from faulty ciphertexts, starting with the differential fault attacks (DFA). Since then, more exploitation methods have been proposed such as statistical fault attacks (SFA) and ineffective fault attacks (IFA) that are particularly relevant to this thesis.

### 5.1.1 Differential Fault Attacks (DFA)

DFA was introduced by Biham et al. and makes use of techniques from differential cryptanalysis to learn about key-dependent intermediate variables of cryptographic computations [BS97]. More concretely, DFA uses the observation that results of correct and faulty cryptographic computations of the same input contain information on the internal state of the computation, which allows learning information about the used key.

One often discussed application of DFA is on implementations of block ciphers such as AES [DR20] (described in Appendix A.1). In this scenario, an attacker performs two encryptions of the same plaintext and induces a fault into one of the computations. The location of the fault induction should be chosen such that (1) a partial key guess can be used to decrypt a faulty ciphertext to the location of the fault induction and (2) the partial decryption involves a nonlinear layer followed by a linear mixing layer. In the case of AES, a fault induction at or slightly before the penultimate round of the block cipher typically satisfies these requirements, as illustrated in Figure 5.2. The requirements on the physical effect of the fault induction on an intermediate variable are quite relaxed, i.e., any kind of modification can lead to a faulty ciphertext that is exploitable.

The exploitation itself consists of a partial decryption of both, the affected faulty bits of the ciphertext, and their correct counterpart, back to the location of fault induction using guesses of the involved key bits. If the partial key guess is correct, the difference between the partial decryption of valid and faulty ciphertext bits should only be somewhat small, e.g., affect only one intermediate value of the cipher state. If the partial key guess is incorrect, one would typically observe a larger difference between the partial decryption of valid and faulty ciphertext. In the best case, the observation of just a single pair of valid/faulty ciphertext that corresponds to the same plaintext may already lead to a full key recovery [TMA11a].

### 5.1.2 Statistical Fault Attacks (SFA)

SFA was introduced by Fuhr et al. [Fuh+13] as a method to recover the secret key from block cipher implementations and works in random and unknown plaintext

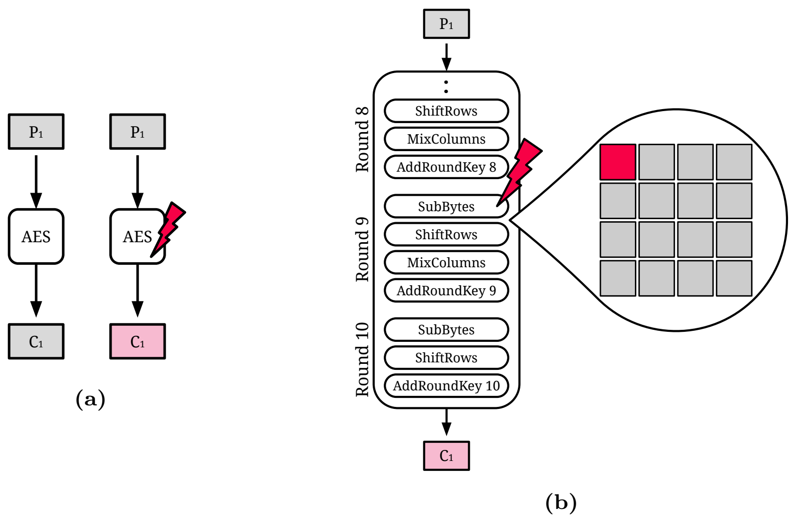


**Figure 5.2:** Simple illustration of DFA attacks on AES implementations. The attacker performs two encryptions of the same plaintext **(a)**, once with a fault induction before MixColumns in the penultimate round **(b)**.

scenarios. Consequently, while attacks like DFA require repeated computations of the same input under the same key, SFA is applicable even if inputs are constantly changing, e.g., in the case of nonce-based encryption. In SFA, the attacker collects faulty ciphertexts encrypted with the same key. Unlike other fault attacks like DFA, SFA requires fault inductions that introduce a bias in the distribution of certain intermediate values that would otherwise have followed a uniform distribution due to the cipher properties. The location of the fault induction should be chosen such that (1) a partial key guess can be used to decrypt a faulty ciphertext to the location of the fault induction and (2) the partial decryption involves a nonlinear layer followed by a linear mixing layer.

Given such a set of faulty ciphertexts, the attacker can partially decrypt every ciphertext back to the location of fault induction for each key candidate and measure the squared euclidean imbalance (SEI) of this value. The SEI is a measure of the distance between a given distribution and the uniform distribution. If we consider the case where we analyze the distribution of an 8-bit intermediate variable $x$ and its concrete observations $x_i$ with $1 \leq i \leq n$, the SEI can be calculated as:

$$\mathrm{SEI}(x) = \sum_{\delta=0}^{255} \left( \frac{\#\{i | x_i = \delta\}}{n} - \frac{1}{256} \right)^2 ,$$

where $\delta$ denotes one of the possible values of $x$. If the location and effect of the fault induction were chosen correctly, then a wrong partial key guess leads to a uniform distribution, while the correct one leads to a biased distribution. Then, the key candidate that gives the highest SEI is most likely the correct one.

If we consider SFA attacks on AES implementations, a suitable location for a fault induction is before MixColumns in the penultimate round, as illustrated in Figure 5.3. The attacker can then guess 4 bytes of the last round key $K_{10}$ and partially decrypt each faulty ciphertext $C$ to obtain a partial state $S_9$:

$$S_9 = \mathrm{MC}^{-1} \circ \mathrm{ARK}(K_9)^{-1} \circ \mathrm{SB}^{-1} \circ \mathrm{SR}^{-1} \circ \mathrm{ARK}(K_{10})^{-1} \circ C, \tag{5.1}$$

where SB, SR, MC and ARK represent the AES round functions (cf. Appendix A.1). Then, the attacker can evaluate the distribution of the bytes in $S_9$ and perform key recovery. Note that when using the SEI metric, no information on the penultimate round key $K_9$ has to be guessed because the constant key addition has no influence on the non-uniformity of the observed distribution.

While SFA was originally only evaluated on implementation of the AES block cipher, a more recent work has shown that SFA is in fact also applicable to a wide range of AES-based authenticated encryption schemes that are based on modes like GCM, CCM, and OCB [Dob+16].

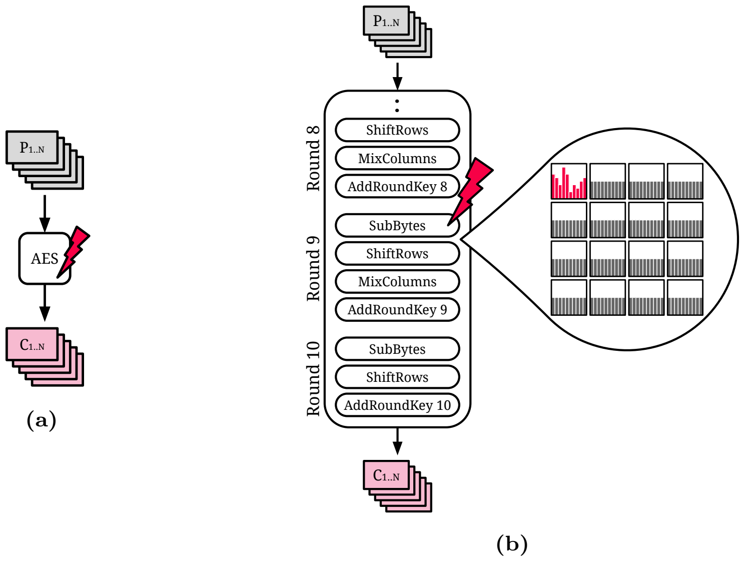


**Figure 5.3:** Simple illustration of SFA attacks on AES implementations. The attacker performs multiple encryptions with different plaintexts and faults each computations **(a)** before MixColumns in the penultimate round **(b)**.

## 5.1.3 Ineffective Fault Attacks (IFA)

The main observation of IFA by Clavier [Cla07] is that the behavior of a device in response to a fault induction may allow learning information on concrete intermediate variables of cryptographic algorithms running on the device. Consider an attacker that can set a certain intermediate variable during a cryptographic computation to a concrete value $x$ by means of fault induction. The location of the fault induction should be chosen such that a partial key guess can be used to decrypt a ciphertext to the location of fault induction. If the attacker nevertheless receives an correct output (ciphertext), which can be checked by repeating the computation without fault induction, the faulted variable must already have been $x$ before the fault induction. In such a situation, the attacker can make a partial key guess, partially decrypt the ciphertext back to the location of fault induction, and check if the resulting value is $x$.

If we consider attacks on AES implementations, an attacker could set one byte of the AES state before the last AddRoundKey operation to a known value $x$. If such a fault induction is ineffective, the corresponding key byte is the XOR between $x$ and the corresponding byte of the ciphertext.

One upside of IFA, compared to the previously discussed attacks, is the fact that IFA still works if the attacked implementation is equipped with a fault countermeasure that performs redundant computations and only returns a result if all of them match. On the other side, one downside of IFA is the strong assumption that an attacker is able to deterministically change the value of an intermediate variable to a known one.

## 5.2 Fault Attack Countermeasures

In order to prevent attackers from performing successful fault attacks, extensive research on countermeasures has already been conducted in the last two decades. These countermeasures can be roughly divided into two categories, depending on their realization as sensors or on algorithmic level.

### 5.2.1 Sensor-based Countermeasures

Sensor-based countermeasures generally aim to increase the resilience of an entire system, either by detecting fault inductions directly on the physical level, or by detecting/correcting resulting errors. Examples of sensor-based countermeasures include the usage of light, voltage, and temperature sensors to detect fault inductions via lasers [Sel+15], voltage glitches [Bar+06], or temperature variations [HS13]. Countermeasures like these already have a long tradition in the smart card industry. While they can significantly increase the difficulty of performing successful fault attacks, they should generally not be considered to offer sufficient protection as they can be bypassed with a certain additional reverse engineering effort. The focus of academic research mostly excludes sensor-based solutions as they are typically tailored to specific industrial devices, their design/analysis often requires large financial resources, and their concrete protection guarantees are often hard to quantify. Consequently, in practice, additional algorithmic countermeasures are typically used and serve as a second line of defense.

### 5.2.2 Algorithmic Countermeasures

Algorithmic fault countermeasures aim to detect or correct erroneous computations without reliance on hardware sensors. The most prominent example of a detection-based fault countermeasure performs redundant executions with subsequent consistency checks [Bar+06]. If applied to a cryptographic operation, such as a block cipher, the encryption of plaintext is performed twice (or more often). Then, the results of the redundant computations are compared, and the ciphertext is only released if all of them match (see also Algorithm 5.1).

**Algorithm 5.1** A Simple detection-based fault countermeasure.

```
Require: key $K$, plaintext $P$
Ensure: ciphertext $C = E_K(P)$, or $\perp$
1: $C_1 \leftarrow E_K(P)$
2: $C_2 \leftarrow E_K(P)$
3: if $C_1 \neq C_2$ return $\perp$
4: return $C_1$
```

Besides fault detection, there also exist other approaches to counteracting fault attacks, such as the usage of majority voting mechanisms for error correction [Bar+06]. The usage of error detection/correction techniques is not limited to cryptographic computations either. The resilience of a device against faults can be generally increased, e.g., by using error detection/correction in data memory[1] or control flow integrity mechanisms that can detect most implausible deviations of a program's intended execution [WWM15].

[1] `https://ibex-core.readthedocs.io/en/latest/03_reference/security.html#register-file-ecc`

✳ ✳ ✳

# 6 SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography

Shortly after the seminal work of Boneh et al. [BDL97] showed fault attacks on implementations of the RSA crypto system, it became clear that also symmetric schemes are susceptible to this type of active implementation attack. Starting with the differential fault attacks (DFA) of DES by Biham and Shamir [BS97], a rich field of research emerged that focuses on techniques to recover the secret key from faulty ciphertexts.

The effect of a fault induction on a cryptographic algorithm can be modelled as the change of an intermediate variable $x$ to a faulty intermediate variable $x'$. Such a change can occur due to a direct modification of $x$, but also due to instruction skips or addressing errors. Independent of the exact effect that leads from $x$ to $x'$, there are two approaches on how to prevent that this change is exploited by an attacker. The first approach is to perform computations in a redundant way. Then, one can determine the difference $\Delta = x - x'$ and suppress the output in case it is non-zero. Corresponding redundancy techniques range from simple temporal or spacial duplications to error detection codes. The second approach for managing $\Delta \neq 0$ are infection-based countermeasures. In this case, a cipher output is always provided but it is ensured that any non-zero $\Delta$ in the ciphertext is randomized such that it becomes useless for an attacker.

While most attack techniques and countermeasures focus on exploiting or preventing information leakage in case $\Delta \neq 0$, the question of whether an attacker can also learn information from ineffective faults has not been explored in depth so far. In this work, we consider a fault induction to be ineffective in case the fault induction (e.g., a voltage glitch) is performed, but a certain dependency on the concrete processed data values renders it ineffective. In this case, even though it holds that $\Delta = 0$, information leakage on the secret key can still occur since the occurrence of a ineffective fault itself is data-dependent.

**Our Contribution.** In this chapter, we generalize IFA attacks and introduce statistical ineffective fault attacks (SIFA). As we argue and show with practical evaluations, SIFA is typically not only applicable when SFA or IFA is applicable, but also in a broader range of scenarios – in particular in the presence of countermeasures. Our attack does not rely on a specific fault model. We simply require that there is some data-dependency between the occurrence of an ineffective fault induction and the targeted intermediate variable $x$. However, the attacker does not need to know any further details of this dependency. This means simply that the probability for changing an intermediate variable $x$ due to a fault induction is not the same for all values $x$. This bias of the probabilities for ineffective fault inductions is the sole requirement on the fault induction.

Like IFA, SIFA can be applied in settings where it is possible to perform fault inductions on encryption operations that process different data inputs using the same key and allow to observe whether the fault induction was ineffective. In the most simple case, we can consider a redundant block cipher implementation that duplicates the plaintext and only returns a ciphertext if the result of the redundant computations match. SIFA is applicable in this setting if an attacker can (1) query this implementation with a certain amount of different plaintexts (2) perform a fault induction at a certain point in time for one of the redundant computations and (3) observe whether or not a ciphertext is returned to determine the occurrence of a ineffective fault induction. While IFA typically requires strong fault models like stuck-at faults, the requirements for SIFA on the fault induction are minimal and corresponding faults can be induced easily in practice with a high frequency and without the need for sophisticated laboratory equipment.

To show this, we attack protected implementations that feature countermeasures against fault attacks like SFA or DFA. In particular, we target countermeasures based on detection and infection. In fact, countermeasures that are based on managing a fault effect $\Delta \neq 0$ are ideal targets for SIFA. These countermeasures allow the attacker to collect observations where the fault induction was ineffective. Our empirical study shows that these countermeasures can be easily bypassed in practice and that it is necessary to combine them with additional countermeasures to provide protection against SIFA attacks.

Our concrete attack results are as follows. First, we target a detection-based countermeasure for AES that uses simple time redundancy with subsequent comparison using a fault induction setup that can temporarily corrupt the clock signal at specific points in time. In order to show that SIFA is applicable on a variety of different hardware platforms, our evaluation is performed on 3 different AES implementations, attacking 8-bit (table-based) and 32-bit (bitsliced) software implementations as well as a hardware co-processor. In all cases, the number of needed faulty encryptions is entirely practical for a physical attack. SFA is not applicable here, since no exploitable faulty output is released. Although IFA is not prevented by simple time redundancy with subsequent comparison, it still relies on precise stuck-at faults in certain bytes, which are hard to achieve in practice. This is especially true in the case of the 32-bit software and the hardware co-processor implementations. In contrast, SIFA can exploit any case where ineffective faults lead to a biased distribution, even without knowledge about the distribution of these values.

We then target infective countermeasures, where typically neither SFA nor IFA are applicable. Here, we extend the software AES implementation from the AVRCryptoLib [Avr] and evaluate our attack for multiple security parameterizations. Again, simple clock glitches are used to induce the required faults, resulting in attacks that are rather easy to execute in practice and do not require any expensive laboratory equipment.

**Related Work.** SIFA extends and connects several other ideas that have previously been published in the literature. One keypoint of the presented attack is the fact that it exclusively exploits cases where a fault does not change the result of the computation. Therefore, our attack shares a common reference point with safe-error attacks [YJ00] and IFA [Cla07]. In a safe-error attack, the value of an intermediate variable is changed (fault effect $\Delta \neq 0$) and the knowledge whether the faulted value is used or not is exploited. Typically, safe-error attacks are used to attack asymmetric schemes. In contrast, ineffective fault attacks [Cla07] exploit specific cases where $\Delta = 0$ and the fault shows no effect. More concretely, IFA relies on strong and known fault models, like precise stuck-at-0 faults, in order to probe values of intermediate variables.

We extend this idea from stuck-at faults as already used by Biham and Shamir [BS97] to the case that ineffective faults lead to a non-uniform distribution of intermediate variables. As a result, we do not probe specific intermediate variables of a cryptographic; rather, we exploit their non-uniform distributions. Hence, we are naturally able to deal with noise (e.g., failure of fault induction, or faults induced at a wrong position). Our used methods to exploit non-uniform distributions are related to those in SFA [Fuh+13].

**Outline.** First, we give a short overview of infection countermeasures in Section 6.1. Then, we state the idea and show the working principle of the attack in Section 6.2. Section 6.3 contains the results of our practical attack while we discuss possible countermeasures in Section 6.4. We finally conclude in Section 6.5.

## 6.1 Background

The strategy of commonly known detection-based countermeasures is to detect that a fault changes an intermediate variable, e.g., by performing redundant operations. That is, once a fault is detected, the computation is aborted and no ciphertext is returned. In contrast, infection-based countermeasures always return a ciphertext, but attempt to process the ciphertext in such a way that the output becomes useless for an attacker in case of faults during the computation.

While a description of detection-based countermeasures is already provided in Section 5.2.2, we now give a detailed description of the infection countermeasure that we also target in this work.

**Infective Countermeasures.** While detection-based countermeasures detect a fault and then do not release a ciphertext, infective countermeasures always provide a ciphertext, but amplify a possibly induced fault in such a way that a faulty ciphertext becomes useless for an attacker. As an example for infection-based countermeasures, we consider the infective countermeasure presented by Tupsamudre et al. at CHES 2014 [TBM14] as an extension of an infective countermeasure presented by Gierlichs et al. [GST12]. Patranabis et al. [PCM15] give a formal proof for this countermeasure against DFA using a single fault induction under the assumption that the sequence of executed instructions is neither skipped, nor altered. In the proof, they evaluate the extent of mutual information between the differential and the key for a given fault model. If this mutual information is 0, the adversary gains no information about the key once the infection affects the entire ciphertext. The only attacks on this countermeasure so far are attacks that either skip or alter instructions [BG16]. The approach is summarized in Algorithm 6.1. We will now give the basic intention behind

**Algorithm 6.1** Infective countermeasure by Tupsamudre et al. (taken from [TBM14])

```
Require: $P$, $k^j$ for $j \in \{1, \dots, n\}$, $(\beta, k^0)$, $(n = 11)$ for AES-128
Ensure: $C = E_K(P)$, or infected state
 1: State $R_0 \leftarrow P$, Redundant state $R_1 \leftarrow P$, Dummy state $R_2 \leftarrow \beta$
 2: $i \leftarrow 1$, $q \leftarrow 1$
 3: $rstr \xleftarrow{\$} \{0,1\}^t$
 4: while $q \leq t$ do
 5:     $\lambda \leftarrow rstr[q]$
 6:     $\kappa \leftarrow (i \wedge \lambda) \oplus 2(\neg\lambda)$
 7:     $\zeta \leftarrow \lambda \cdot \lceil i/2 \rceil$
 8:     $R_\kappa \leftarrow RoundFunction(R_\kappa, k^\zeta)$
 9:     $\gamma \leftarrow \lambda(\neg(i \wedge 1)) \cdot BLFN(R_0 \oplus R_1)$
10:     $\delta \leftarrow (\neg\lambda) \cdot BLFN(R_2 \oplus \beta)$
11:     $R_0 \leftarrow (\neg(\gamma \vee \delta) \cdot R_0) \oplus ((\gamma \vee \delta) \cdot R_2)$
12:     $i \leftarrow i + \lambda$
13:     $q \leftarrow q + 1$
14: return $R_0$
```

Algorithm 6.1. For a more detailed description we refer to the original work of Tupsamudre et al. [TBM14]. Algorithm 6.1 works on three different states $R_0$, $R_1$ and $R_2$. State $R_0$ is initialized with the plaintext $P$ and is the state on which the primary AES computation is performed. State $R_1$ is also initialized with $P$ and serves as working state for the redundant AES computation. In the fault-free case, both states $R_0$ and $R_1$ should contain the ciphertext at the end

of the computation. The state $R_2$ is initialized with a random 128-bit value $\beta$ and serves as working state for the dummy round calculations. The key $k^0$ is chosen such that $RoundFunction(\beta, k^0) = \beta$.

Before the computation starts, a random string *rstr* of length $t$ is initialized randomly so that it contains 22 bits "1" and $t - 22$ bits "0". The algorithm iterates over *rstr* and executes for every "1" an AES round on $R_0$, or a redundant round on $R_1$ (22 rounds for 2 times 10 rounds AES plus 2 times the whitening key addition) in an alternating sequence, i.e., if a round on $R_0$ has been calculated, the next "1" executes a redundant round on $R_1$ so that after this calculation, the content of $R_0$ and $R_1$ should be the same in a fault-free case. For every "0", a dummy round is computed to update $R_2$. The security level with respect to the number of dummy rounds that are executed depends on the size of $t$ and can be chosen by the developer.

After every executed AES round, the algorithm checks if any of the values in registers $R_0$, $R_1$, or $R_2$ has been modified ($R_0 \neq R_1$ or $R_2 \neq \beta$). If this is the case, state $R_0$ is, from this point on, always overwritten with the content of $R_2$, which is then returned as ciphertext. Since the value stored in $R_2$ is random and has never been mixed with, nor depends in any other way on the value of the secret key, learning this value should be useless for the attacker.

In the following, we demonstrate that not only stuck-at faults can be exploited in IFA and introduce statistical ineffective fault attacks. On a high level, these attacks can be seen as an intersection of the principles exploited in the case of IFA [Cla07] and SFA [Fuh+13]. In Section 6.2, we explain the necessary conditions for our attack to work and demonstrate in Section 6.3 that they are usually fulfilled when attacking real devices with algorithmic countermeasures. In particular, the attacker does not need to assume any specific fault model and can successfully recover the key even if a "noisy" fault induction setup is used that may not always affect the targeted intermediate variable in the same way.

## 6.2 Statistical Ineffective Fault Attacks

In this section, we discuss the ideas behind the extension from ineffective fault attacks [Cla07] (IFA) to statistical ineffective fault attacks (SIFA). First, we review the effects of faults with the help of fault distribution tables to identify the necessary conditions for SIFA to work in Section 6.2.1. Then we introduce the working principle of SIFA in Section 6.2.2. Finally, we develop some theoretical background of our attacks in Section 6.2.3.

### 6.2.1 The Effects of Faults

The effects caused by faults during the execution of cryptographic primitives are manifold and depend on the method used to induce the fault (e.g., laser, clock glitches), the architecture and manufacturing technology of the attacked device, and various other parameters (e.g., targeting a register or arithmetic instruction). However, all faults have in common that they change the value of a

$b$-bit intermediate variable from a value $x$, which it would have for the correct execution, to a value $x'$ in the presence of a fault. Observing the probability of transitions from a certain value $x \rightarrow x'$ gives us a fault distribution table (see Section 6.2.3 for the exact definition).

With the help of such a fault distribution table, we are able to characterize the effects of a wide range of faults that can happen in practice. For example, this allows us to capture faults where the value of $x'$ is independent of the value $x$, like stuck-at faults, random faults. Besides that we can also capture biased faults that replace a value by another that is drawn from a non-uniform distribution or more complex relations where $x'$ depends in some sense on $x$, for instance by faulting the instruction that computes $x$. In Table 6.1, we show various examples of fault distribution tables for different faults on a 2-bit intermediate variable. Most fault countermeasures that work on an algorithmic level can only

**Table 6.1:** Fault distribution tables for several 2-bit fault models.

**(a)** Stuck-at-0

| $x \backslash x'$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | 1 | 0 | 0 | 0 |
| 01 | 1 | 0 | 0 | 0 |
| 10 | 1 | 0 | 0 | 0 |
| 11 | 1 | 0 | 0 | 0 |

**(b)** Random-And

| $x \backslash x'$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | 1 | 0 | 0 | 0 |
| 01 | $\frac{1}{2}$ | $\frac{1}{2}$ | 0 | 0 |
| 10 | $\frac{1}{2}$ | 0 | $\frac{1}{2}$ | 0 |
| 11 | $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ |

**(c)** Bit-flip

| $x \backslash x'$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 1 |
| 01 | 0 | 0 | 1 | 0 |
| 10 | 0 | 1 | 0 | 0 |
| 11 | 1 | 0 | 0 | 0 |

**(d)** Random fault

| $x \backslash x'$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ |
| 01 | $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ |
| 10 | $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ |
| 11 | $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ |

conceal cases where $x \neq x'$, because a fault induction that results in $x = x'$ is indistinguishable from a normal working condition. As a consequence, an attacker has access to ciphertexts where the attacked (faulted) intermediate variable follows a distribution determined by the diagonal (red values) in Table 6.1. The attacks presented in the following sections show that a non-uniform distribution in this diagonal can be exploited to recover the key. Therefore, for an implementation protected by such a fault countermeasure to be resistant against our attack, one of the two following conditions has to be fulfilled: Either the probability that an ineffective fault happens is negligible (as in Table 6.1c), or the distribution in the diagonal of the fault distribution table is uniform (as in Table 6.1d).

Despite the fact that our analysis of arguably somewhat abstract fault models indicate that the bit-flip and random fault models of Table 6.1c and Table 6.1d are not susceptible to SIFA, our practical experiments in Section 6.3 indicate that countermeasures cannot rely on the hope that only such "perfect" fault models occur in practice. For instance, consider the case where a bit-flip occurs probabilistically with the tendency to flip more often from 1 to 0 than from 0 to 1, as illustrated in Table 6.2. The resulting distribution has a biased diagonal. In the following section, we will explain how such distributions can be exploited. Since the fault distribution tables are typically not known by an attacker (unless the attacker is able to profile the device), our attack works without any knowledge of the fault distribution table. This is demonstrated by the practical attacks of

**Table 6.2:** Bit-flip from 1 to 0 with 75 % and from 0 to 1 with 50 %.

| | | $x'$ | | | |
|---|---|---|---|---|---|
| | | 00 | 01 | 10 | 11 |
| $x$ | 00 | $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ |
| | 01 | $\frac{3}{8}$ | $\frac{1}{8}$ | $\frac{3}{8}$ | $\frac{1}{8}$ |
| | 10 | $\frac{3}{8}$ | $\frac{3}{8}$ | $\frac{1}{8}$ | $\frac{1}{8}$ |
| | 11 | $\frac{9}{16}$ | $\frac{3}{16}$ | $\frac{3}{16}$ | $\frac{1}{16}$ |

Section 6.3, which we perform without any knowledge of the underlying fault model and fault distribution table.

### 6.2.2 Working Principle

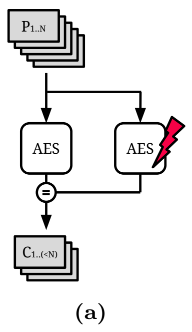
We now consider a protected implementation of AES (cf. Appendix A.1) as an example to show the working principle of SIFA. The attack can be split into 3 phases. The first phase is the actual fault attack and collection of suitable ciphertexts. In the second phase, parts of the last round key are guessed and the distribution of an intermediate state is evaluated. In the last phase of the attack, the partial key-guesses are ranked according a metric (e.g., the squared euclidean imbalance (SEI)). Once the correct candidate for each part of the last round key round key is identified, the main key can be simply derived by reversing the AES key-schedule.

**Collecting Ciphertexts.** Assume that we target one byte slightly before the last application of MixColumns, as illustrated in Figure 6.1. We request the ciphertexts for a number of plaintexts and fault each encryption. If the implementation is protected with a detection-based countermeasure, we only obtain those ciphertexts where the fault was ineffective; in case of an infective countermeasure, we need to filter for ineffective faults ourselves by comparing the obtained ciphertexts with a second, unfaulted encryption (or decryption).

**Key Guessing.** Following the fault model of Section 6.2.1, we obtain a set of filtered ciphertexts whose intermediate variable in one byte before the last MixColumns has a probability distribution given by the diagonal of the fault distribution table. The key recovery phase then works similar as previously described for SFA in Section 5.1.2. The main difference lies in the usage of correct instead of faulty ciphertexts as also illustrated in Figure 6.1. A closer insight in the number of ciphertexts required for key recovery is given in Section 6.2.3.

### 6.2.3 Statistical Model

In this section, we provide a more detailed statistical model for SIFA and justify the use of the SEI. Our aim is to investigate the effect of various parameters, such



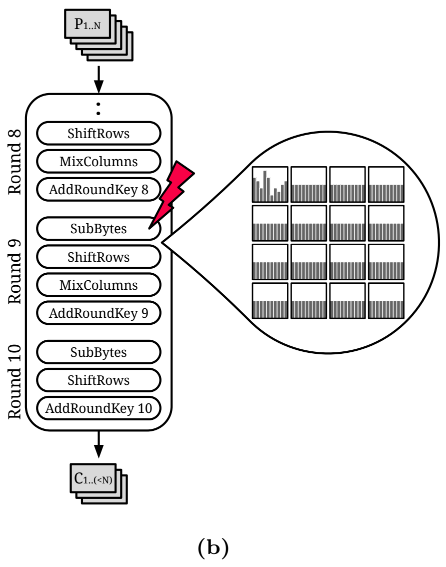


**Figure 6.1:** Simple illustration of SIFA attacks on AES implementations using a detection-based countermeasure. The attacker faults one of the redundant computations **(a)** before MixColumns in the penultimate round **(b)**.

as the fault distribution and the number of dummy rounds, on the necessary number of faulted ciphertexts to perform the attack with a certain success probability. We compare two scenarios: The practical scenario where the fault distribution is unknown to the attacker (CHI/SEI statistic), but also the theoretical scenario where the attacker happens to know the distribution (LLR statistic). The emphasis of our analysis is on the hardest case: An unknown fault distribution, close to uniform, with additional noise induced by countermeasures.

We consider the $b$-bit intermediate variable which contains the result of the operation targeted by the fault, and consider its distribution during the attack in more detail. From the attacker's point of view, the value of this variable on a particular input (in absence of faults) is a random variable $X$ which depends on the input and key. Additionally, the random variable $X'$ denotes the value of this variable on the same input, but where the attacker additionally attempted to fault the operation. We also refer to $X$ and $X'$ as "before" and "after" the fault, although this is not strictly accurate. Both $X$ and $X'$ take values $x \in \mathcal{X} = \{0, \ldots, 2^b - 1\}$. The action of the fault can be characterized by the transition probabilities

$$p_x(x') := \mathbb{P}[X' = x' | X = x].$$

In practice, this fault distribution table FDT $= (p_x(x'))_{x,x'}$ is usually not known, or can only be roughly estimated. To perform the proposed attack, it is not required to know the FDT. However, the success and efficiency of the attack depends on some of the table's properties. In the following, we will analyze the attack complexity and its dependency on the two relevant metrics: The fault's ineffectivity rate $\pi_=$, and the capacity $C(p)$ of the target distribution $p$.

### Direct Sampling: Detection Countermeasure

We first consider attacks on cipher implementations equipped with detection-based countermeasures. We can only take advantage of samples where $X = X'$, i.e., the fault is ineffective. We assume that $X$ is uniformly distributed since it models an intermediate variable in the penultimate round of a block cipher implementation that is queried with varying inputs. Hence, $\mathbb{P}[X = x] = 2^{-b}$. Then, the probabilities $\pi_=$ of an ineffective fault (ineffectivity rate) and $\pi_{\neq}$ of an effective fault are

$$\pi_= = \mathbb{P}[X' = X] = \sum_{x' \in \mathcal{X}} \frac{p_{x'}(x')}{2^b}, \qquad \pi_{\neq} = 1 - \pi_=.$$

We target the conditional distribution $p_=(x')$ of $X'$ in case of ineffective faults, i.e., the diagonal of the fault distribution table (see Section 6.2.1):

$$p_=(x') := \mathbb{P}[X' = x' | X' = X] = \frac{p_{x'}(x')}{2^b \cdot \pi_=}.$$

The attacker neither knows this distribution, nor can she directly observe $X'$. However, based on the observed cipher output and a key hypothesis for the $\kappa$-bit last-round key material as in Section 6.2.2, she obtains a hypothesis $\hat{X}'$ for the values of $X'$ and the set of filtered ciphertexts. She can then analyze the distribution $\hat{p}$ of $\hat{X}'$ for a fixed key guess across multiple samples and distinguish the correct key $k_0$ from the wrong keys $k_i$ with $1 \leq i < 2^\kappa$ as follows:

- For an incorrect key guess, we assume a distribution very close to uniform, which we denote by $\theta_i(x')$. Note that in practice, this is not necessarily the case, in particular for partially correct key guesses. For example, for a byte-stuck-at fault and a key guess that is only incorrect in one byte, the capacity is expected to drop from 255 to about 1, instead of 0. Nevertheless, we do later show that this assumption is valid for the practical evaluations that we perform in Section 6.3.
- For the correct key guess, we sample the unknown distribution $p(x') = p_=(x')$. If $p_=(x')$ differs significantly from uniform, we can distinguish these two cases. More concretely, we can identify the samples from $p_=(x')$ produced by the correct key $k_0$ among the collection of samples from the nearly uniform distributions $\theta_i(x') \approx \theta(x') = 2^{-b}$ produced by the wrong keys $k_i$.

To identify the correct key $k_0$ and its distribution $p = p_=$, we associate a score statistic $S(\hat{p})$ with each key candidate and the corresponding distribution $\hat{p}$, and rank the key candidates according to this statistic. This approach is closely related to statistical cryptanalysis, such as differential and linear cryptanalysis, and has been theoretically analyzed in those contexts. Under the assumption that $S(\hat{p})$ is independently normally distributed for samples from either $p$ or $\theta$, as done by Selçuk in [Sel08],

$$S(\hat{p}) \sim \begin{cases} \mathcal{N}(\mu_\mathrm{R}, \sigma_\mathrm{R}^2) & \text{if } \hat{p} \text{ was produced by } p\,, \\ \mathcal{N}(\mu_\mathrm{W}, \sigma_\mathrm{W}^2) & \text{if } \hat{p} \text{ was produced by } \theta\,, \end{cases} \tag{6.1}$$

Selçuk analyzed the success probability of ranking the correct key $k_0$ among the top $2^{\kappa-a}$ of $2^\kappa$ key candidates based on $N$ samples, where they call $a$ the advantage. Then, the difference $\Delta_a$ between the score of $k_0$ and the score of the wrong key with rank $2^{\kappa-a}$ (quantile $\alpha = 1 - 2^{-a}$) is normally distributed with parameters

$$\Delta_a \sim \mathcal{N}(\mu_\Delta, \sigma_\Delta^2), \qquad \begin{aligned} \mu_\Delta &= \mu_\mathrm{R} - \mu_\mathrm{W} - \sigma_\mathrm{W}\, \Phi_{0,1}^{-1}(\alpha)\,, \\ \sigma_\Delta^2 &\approx \sigma_\mathrm{R}^2 \qquad \text{for sufficiently large } 2^\kappa \text{ [BGN12a]}, \end{aligned}$$

and thus the success probability depending on $N$ and $a$ (or $\alpha$) can be estimated as in [Sel08]

$$\mathbb{P}[\Delta_a > 0] \approx \Phi_{0,1}\left(\frac{\mu_\mathrm{R} - \mu_\mathrm{W} - \sigma_\mathrm{W}\, \Phi_{0,1}^{-1}(\alpha)}{\sigma_\mathrm{R}}\right)\,. \tag{6.2}$$

If one is interested in the probability that the right key has the highest rank we can set $a = \kappa$:

$$\mathbb{P}[\Delta_\kappa > 0] \approx \Phi_{0,1}\left(\frac{\mu_\mathrm{R} - \mu_\mathrm{W} - \sigma_\mathrm{W}\Phi_{0,1}^{-1}(1 - 2^{-\kappa})}{\sigma_\mathrm{R}}\right)\,.$$

To obtain useful complexity estimates from Equation 6.2, we need a normally distributed statistic $S(\hat{p})$ parameterized by $\mu$ and $\sigma^2$ according to Equation 6.1. We first consider the (unusual) case [1] that we know the real distribution $p = p_=$. Then, the Neyman-Pearson lemma [NP33; CT06] states that the optimal statistic $S$ is the log-likelihood ratio

$$S(\hat{p}) = \mathrm{LLR}(\hat{p}) = \mathrm{LLR}(\hat{p}, p, \theta) := N \sum_{x\in\mathcal{X}} \hat{p}(x) \log_2 \frac{p(x)}{\theta(x)}\,.$$

For large $N$, $\mathrm{LLR}(\hat{p})$ tends towards a normal distribution as required in Equation 6.1 [CT06; BJV04]. The success probability in Equation 6.2 then depends on the Kullback-Leibler divergence $D(p\|\theta)$ and an auxiliary metric $D_\Delta(p\|\theta)$ that is defined in the extended version of [BGN12a] in [BGN12b] (Appendix A.2):

$$D(p\|\theta) := \sum_{\substack{x\in\mathcal{X}\\ p(x)\neq 0}} p(x) \log_2 \frac{p(x)}{\theta(x)}\,, \quad D_\Delta(p\|\theta) := \sum_{\substack{x\in\mathcal{X}\\ p(x)\neq 0}} p(x) \left[\log_2 \frac{p(x)}{\theta(x)}\right]^2 - D(p\|\theta)^2\,.$$

[1] Note that in this case, we could also target the last round with lower $a$ and $N$, but more repetitions to obtain the full key.

If $p$ is very close to uniform $\theta$, these can be approximated using the capacity $C(p,\theta)$ [BCQ04]:

$$C(p,\theta) := \sum_{x\in\mathcal{X}} \frac{(p(x)-\theta(x))^2}{\theta(x)} \approx 2\,D(p\|\theta) \approx D_\Delta(p\|\theta) \quad \text{(only if } p \text{ is close to } \theta.)$$

The resulting estimate for the necessary number of samples $N_{\text{LLR}}$ to achieve a success probability $P = \mathbb{P}[\Delta_a > 0]$ can be derived as [BJV04; BGN12a]:

$$N_{\text{LLR}} \approx \left[\frac{\Phi_{0,1}^{-1}(P)\sqrt{D_\Delta(p\|\theta)} + \Phi_{0,1}^{-1}(\alpha)\sqrt{D_\Delta(\theta\|p)}}{D(p\|\theta)+D(\theta\|p)}\right]^2 \approx \frac{2[\Phi_{0,1}^{-1}(P)+\Phi_{0,1}^{-1}(\alpha)]^2}{C(p,\theta)}\,.$$

When applied to the AES fault analysis scenario, knowing $p = p_=$ means both knowing the exact fault distribution of the ineffective faults and guessing 8 bits of the equivalent key to $K_9$ in the penultimate round, with a correspondingly increased advantage $a$. In the context of differential cryptanalysis, it has been demonstrated [BGN12a] that even small errors in the estimate of $p$ can significantly increase the necessary number of samples. Since such exact models of $p_=$ are usually not available for practical fault attacks, we can consider less optimal, but more robust statistics.

The classical test statistic for an unknown distribution $p$ is Pearson's $\chi^2$:

$$\text{CHI}(\hat{p}) := \chi^2(\hat{p},\theta) = N\sum_{x\in\mathcal{X}} \frac{(\hat{p}(x)-\theta(x))^2}{\theta(x)}\,,$$

or, for uniform $\theta$, the closely related squared euclidean imbalance (SEI):

$$\text{SEI}(\hat{p}) := \sum_{x\in\mathcal{X}} (\hat{p}(x)-\theta(x))^2 = (N\cdot 2^b)^{-1}\cdot \text{CHI}(\hat{p})\,.$$

The statistic $\text{CHI}(\hat{p})$ is distributed according to the (noncentral) chi-squared distribution with $k = |\mathcal{X}|-1 = 2^b-1$ degrees of freedom and noncentrality parameter $\lambda_{\text{R}} = N\,C(p,\theta)$ or $\lambda_{\text{W}} = 0$. For large $k$ and $N$, this tends towards a normal distribution with parameters [HCN09; Dro+89]

$$\text{CHI}(\hat{p}) \sim \begin{cases} \mathcal{N}(\mu_{\text{R}} = k + NC(p,\theta), & \sigma_{\text{R}}^2 = 2\,[k+2NC(p,\theta)])\,, \\ \mathcal{N}(\mu_{\text{W}} = k, & \sigma_{\text{W}}^2 = 2k)\,. \end{cases}$$

Based on these parameters, we can express the equation in Equation 6.2 as a degree-two polynomial as in [BGN12b] (Appendix A.2) and estimate the necessary number of samples as

$$\begin{aligned} N_{\text{CHI}} &\approx \frac{s+\sqrt{s^2-t}}{C(p,\theta)} && \left(s = \sqrt{2k}\,\Phi_{0,1}^{-1}(\alpha) + 2\Phi_{0,1}^{-2}(P), \quad t = 2k(\Phi_{0,1}^{-2}(\alpha) - \Phi_{0,1}^{-2}(P))\right) \\ &= \frac{\sqrt{2k}\,\Phi_{0,1}^{-1}(\alpha)}{C(p,\theta)} && \text{(for success probability } P = 0.5.) \end{aligned}$$

Summarizing, both statistics lead to an estimated number of samples that is proportional to $1/C(p,\theta)$, where the constant depends on the desired success probability $P$ and advantage $a$ (or quantile $\alpha = 1 - 2^{-a}$). However, these estimates are only useful if the resulting $N$ is reasonably large, that is, if $p$ is not extremely different from $\theta$.

### Noisy Sampling: Infective Countermeasure

So far, we assumed that for the correct key guess, the attacker makes the correct hypothesis $\hat{X}' = X'$, and thus directly samples the distribution $p_=(x')$. We will now show that the same approach also generalizes naturally to cases where the attacker only obtains noisy measurements.

As an example, consider the infective countermeasure discussed in Section 6.1 that performs $R = r + 11 + 11$ rounds of AES where $r$ denotes the number of dummy rounds. The remaining (non-dummy) rounds correspond to two redundant encryptions of the plaintext $P$ and always occur in alternating order.

To identify computations with ineffective faults, the attacker has to compare the faulted ciphertexts $C'$ with previously obtained correct ciphertexts $C$ for the same plaintexts $P$, and keeps only the samples where $C = C'$. Assuming the same fault model as before, she will keep a fraction of about $\pi_=$ samples. However, she does not know whether the ineffective fault really occurred in a penultimate non-dummy round, or elsewhere: in a dummy round or the wrong round. To calculate the impact of this countermeasure on key recovery, we can make the following considerations.

Counting from the end, we denote with $t$ the round that is targeted by the attacker, and with $s$ the index of a non-dummy round. Then, for a certain value of $t$, a fault induction hits a penultimate non-dummy round if the index $s$ of that round has the value 2 or 3. If we are now interested in cases where the attacker can target either of the two penultimate non-dummy rounds, the probability of these two cases can be summed up ($\sigma_+$). If we consider cases where the attacker has to choose one of the two in advance, the maximum of the two individual probabilities is relevant ($\sigma_{\max}$). Figure 6.2 illustrates the relation between $\sigma$ and different choices for $r$ and $t$ and how it is calcualted. For completeness, we also plot the values for the corresponding observed probability ($\sigma = \sigma_*$) from our practical evaluation in Section 6.3.2. We consider $\sigma_+$ to be most representative for a realistic fault attack setting.

$$\sigma = \begin{cases} \sigma_+ = \sigma_2 + \sigma_3, \\ \sigma_{\max} = \max\{\sigma_2, \sigma_3\}, \end{cases} \qquad \sigma_s = \frac{\binom{t}{s} \cdot \binom{R-t-1}{22-s-1}}{\binom{R}{22}} .$$

Depending on whether round $t$ was indeed relevant, the hypothesis $\hat{X}'$ for the correct key now samples one of two distributions: If $t$ was relevant, $\hat{X}' = X'$ and we sample $p_=(x')$; else, we sample a distribution close to uniform. Thus, we sample a noisy variable $X''$ with distribution $p_\approx(x'')$, where

$$p_\approx(x'') = \sigma\, p_=(x'') + (1-\sigma)\, 2^{-b} = \sigma\, (p_=(x'') - 2^{-b}) + 2^{-b}.$$

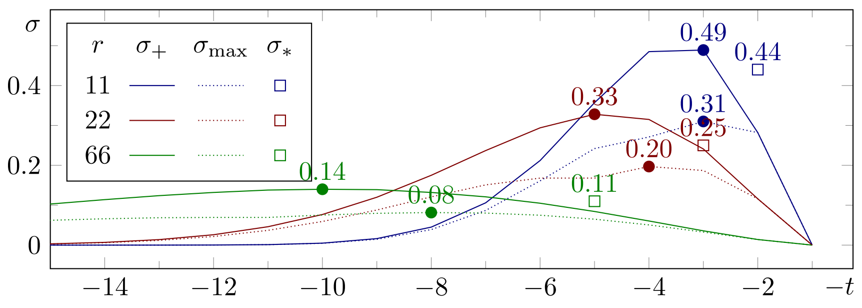


**Figure 6.2:** Probability $\sigma$ of successful sampling for $r \in \{11, 22, 66\}$ dummy rounds.

The capacity of this distribution is

$$C(p_{\approx}) = \sum_{x \in \mathcal{X}} \frac{(p_{\approx}(x) - 2^{-b})^2}{2^{-b}} = \sum_{x \in \mathcal{X}} \frac{\left(\sigma\,(p_{=}(x'') - 2^{-b})\right)^2}{2^{-b}} = \sigma^2\, C(p_{=}).$$

Thus, the expected data complexity for noisy sampling is $\sigma^{-2}$ times higher compared to direct sampling.

In summary, the expected number of faults the attacker has to induce to collect enough samples is inverse proportional to $\pi_{=} \cdot \sigma^2 \cdot C(p_{=})$, where the constant depends on the desired success probability $P$ and advantage $a$.

### 6.2.4 Examples and Simulations

To illustrate the statistical model in more detail, we consider a simulation of the attack with a random-and fault, i.e., each set bit of the target byte is flipped from 1 to 0 with probability $\frac{1}{2}$. The ineffectivity rate of this fault is $\pi_{=} = (3/4)^8 \approx 10\,\%$. We attack an AES implementation protected with the infective countermeasure (Section 6.1) with $r = 22$ dummy rounds and target round $R - t = 44 - 4 = 40$, obtaining a signal of $\sigma = \frac{1111}{3526} \approx 0.315$ among the ineffectively faulted samples. The expected target distribution $p(x)$ for the correct key is illustrated together with the uniform distribution $\theta$ in Figure 6.3 and depends on the Hamming weight $\mathrm{hw}(x)$:

$$p(x) = \sigma \cdot 2^{8-\mathrm{hw}(x)}/3^8 + (1-\sigma) \cdot 2^{-8}.$$

To compare the practically necessary number of samples $N$ (with or without knowledge of $p(x)$) with the predictions of Section 6.2.3, we evaluate the statistics $\mathrm{LLR}(\hat{p})$ and $\mathrm{CHI}(\hat{p})$. For runtime reasons, we limit our analysis to comparing the correct last-round key against $2^{24}$ wrong key candidates (out of $2^{32}$; we set one byte to the correct value).

For the $\mathrm{LLR}(\hat{p})$ statistic, we need to know the exact target distribution after addition of the penultimate round key, so we need to guess a byte $K'$ of the penultimate round key in addition to the 24-bit key guess $K$. For simplicity, we

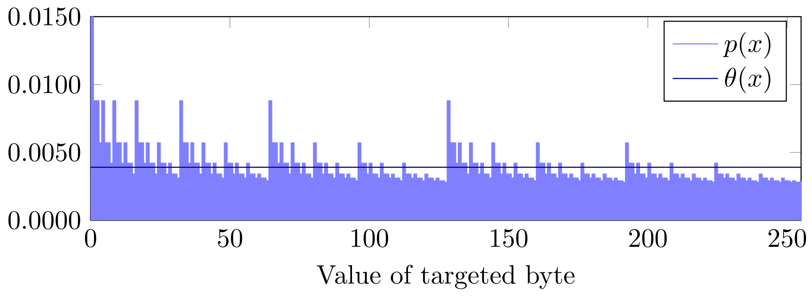


**Figure 6.3:** Distributions $p$ and $\theta$ for random-and fault and infective countermeasure.

evaluate each candidate $K$ based on the statistic

$$S(\hat{p}) = \max_{K'} \mathrm{LLR}(\hat{p}, p_{K'}, \theta) = \max_{K'} N \sum_{x \in \mathcal{X}} \hat{p}(x) \cdot \log_2 \frac{p(x \oplus K')}{\theta(x)} .$$

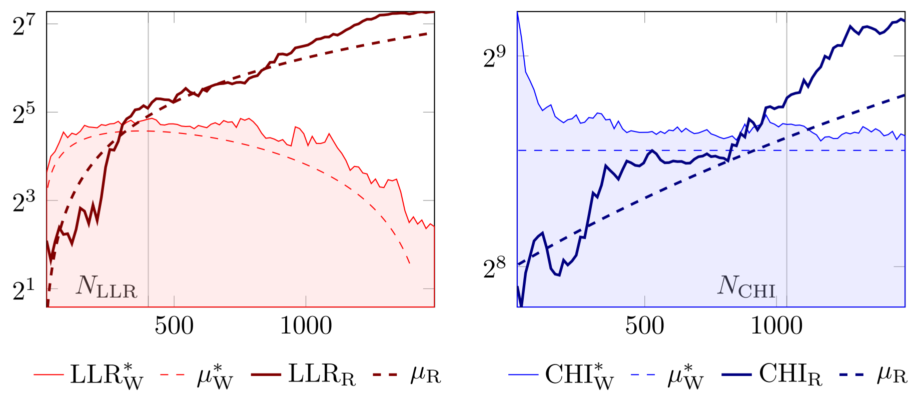
To reflect this in the model and evaluate the probability that the correct 24-bit $K$ is ranked highest, we set the advantage to $a = \kappa + 8 = 32$, so $\alpha = 1 - 2^{-32}$. Based on the model of Section 6.2.3, we expect the statistics $\mathrm{LLR}_\mathrm{R}$ of the right key, $\mathrm{LLR}_\mathrm{W}$ of any wrong key, and $\mathrm{LLR}^*_\mathrm{W}$ of the best wrong key to be normally distributed with the following parameters:

$$\begin{aligned} \mu_\mathrm{R} &= N D(p\|\theta) \approx 0.075N & \sigma^2_\mathrm{R} &= N D_\Delta(p\|\theta) \approx 0.252N \\ \mu_\mathrm{W} &= -N D(\theta\|p) \approx -0.064N & \sigma^2_\mathrm{W} &= N D_\Delta(\theta\|p) \approx 0.157N \\ \mu^*_\mathrm{W} &= \mu_\mathrm{W} + \Phi^{-1}_{0,1}(\alpha)\,\sigma_\mathrm{W} \approx -0.064N + 2.469\sqrt{N} & \sigma^{2*}_\mathrm{W} &\ll \sigma^2_\mathrm{W} . \end{aligned}$$

For the CHI statistic, we use $a = \kappa = 24$ and expect the statistics $\mathrm{CHI}_\mathrm{R}$, $\mathrm{CHI}_\mathrm{W}$, and $\mathrm{CHI}^*_\mathrm{W}$ to be normally distributed with the following parameters:

$$\begin{aligned} \mu_\mathrm{R} &= k + N\,C(p,\theta) \approx 255 + 0.131N & \sigma^2_\mathrm{R} &= 2k + 4N\,C(p,\theta) \approx 510 + 0.525N \\ \mu_\mathrm{W} &= k = 255 & \sigma^2_\mathrm{W} &= 2k = 510 \\ \mu^*_\mathrm{W} &= \mu_\mathrm{W} + \Phi^{-1}_{0,1}(\alpha)\,\sigma_\mathrm{W} \approx 375 & \sigma^{2*}_\mathrm{W} &\ll \sigma^2_\mathrm{W} . \end{aligned}$$

Figure 6.4 compares the resulting model (dashed: $\mu_\mathrm{R}, \mu^*_\mathrm{W}$) with the statistics obtained in a key recovery attack using simulated fault inductions (solid: $S(\hat{p}_\mathrm{R}), S(\hat{p}^*_\mathrm{W})$). For for both statistics, the correct key candidate is marked by a thick line, the best wrong key candidate is marked by a thin line while the remaining wrong candidates are contained in the (colored) area below. The predicted necessary number of samples $N$ for success probability $P = 0.8$ and with advantage $a = 24$ (for CHI) or $a = 32$ (for LLR) is marked as $N_\mathrm{CHI}$ and $N_\mathrm{LLR}$, respectively. This estimate quite accurately matches the practically necessary $N$. It is worth noting that for both statistics, the best wrong key candidate scores slightly better than predicted with $\mu_\mathrm{W}$. This can be partly explained with the not-entirely-uniform distribution of the target variable for partially correct key



**(a)** LLR($\hat{p}$) statistic

**(b)** CHI($\hat{p}$) statistic

**Figure 6.4:** Simulation results for infection countermeasure and random-and fault.

guesses, as discussed in Section 6.2.3. In case of CHI($\hat{p}$), both the right and wrong keys scored slightly higher than expected.

We repeated simulations for other fault models (bit-stuck-at-0, byte-stuck-at-0) and for both countermeasures (detection, infective). The model matches the practical results similarly well when the capacity $C(p, \theta)$ is not too large and thus $N$ is not too small to justify the normal approximation. In particular, for the byte-stuck-at-0 fault, $C(p, \theta) \gg 1$, and the model predicts fewer than the practically necessary $N \approx 4$ ($\sigma = 1$, detection) or $N \approx 15$ ($\sigma = 0.315$, infective) samples.

In summary, having insight in the concrete effect of a fault allows to model the scores of the correct and best wrong key and accurately predict the necessary number of samples for a successful attack. In such a case, the LLR outperforms the CHI (or SEI) statistic in terms of required samples. However, for LLR, even small errors in the estimate of $p$ can significantly increase the necessary number of samples [BGN12a]. Hence, in practice, the CHI (or SEI) statistic is preferable, since the attacker can reliably and efficiently recover the key in the presence of countermeasures without any knowledge of $p$ as demonstrated in Section 6.3.

## 6.3 Practical Evaluation

For the practical evaluation of SIFA we have performed multiple experiments implemented on various microcontrollers listed in Table 6.3. First, we show the practical applicability of SIFA against a time redundant 8-bit software AES, a time redundant hardware AES co-processor, and a time redundant 32-bit bitsliced AES. We then show practical attacks against the infective countermeasure by Tupsamudre et al. [TBM14] for several security parameterizations.

**Table 6.3:** Target microcontrollers of our attack evaluation.

| Name | ALU Size | Core | CPU Freq. |
|---|---|---|---|
| ATXmega 256A3 | 8-bit | Atmel AVR | 12 MHz |
| ATXmega 128D4 | 8-bit | Atmel AVR | 7 MHz |
| STM32 F3 | 32-bit | ARM Cortex-M4 | 7 MHz |

**Fault Setup.** In order to induce the faults we have used clock glitches. To be more precise, we insert an additional fast edge on the original clock signal that violates timing requirements of electronic signals on the device. This can then lead to undefined behaviour such as erroneous computation results or entirely skipped instructions. By additionally varying width and offset of the induced edge, it is possible fine-tune the concrete effect of fault induction on the currently executed instruction as well as its reproducibility. We have used an FPGA for generating both the original clock signal and the clock glitch for the device under test. For sake of simplicity, we determined our attack parameters with the help of an unprotected implementation in the case of the detection-based countermeasure. In case that an unprotected implementation is not accessible to an attacker, determining the fault parameters is much more time consuming, but still feasible. We want to point out that the demonstrated attacks do not require a profiling of the actual distribution of the induced fault. In fact, we performed the key recovery attacks without any knowledge about the distribution of the targeted intermediate variable.

All experiments are performed in a fully automated attack setup. By using this setup, we are able to perform about 20 faulted encryptions per second, or 72 000 per hour. The time required to collect enough correct ciphertexts for key recovery is somewhere between 1 minute and 2 hours.

**Possible Effects of Clock Glitches.** We first give some intuition explaining the possible scenarios that can occur when using clock glitches for fault induction on various platforms. Later, when discussing the individual attack results, we will refer to these scenarios to help explain our results. The following three scenarios can arise:

1. *Missed Fault.* Occurs if the parameterization of a clock glitch is incorrect and the targeted device is not affected in any way. This situation can for example occur if a glitch is inserted onto the clock signal close towards the end of an ordinary clock cycle when all signals have already stabilized.
2. *Successful and Effective Fault.* Occurs if the induced clock glitch influences the target instruction/register and the result of the current computation (e.g. encryption) is affected.

3. *Successful but Ineffective Fault.* Occurs if the induced clock glitch influences the target instruction/register but no effect on the outcome of the current computation (e.g. encryption) is observed.

Generally speaking, traditional fault attacks exploit *Successful and Effective Faults*, while IFA, and SIFA exploit *Successful but Ineffective Faults.* Depending on the attacked implementation and the quality of laboratory equipment, the occurrence of *Missed Faults* varies. For an attacker, it is usually not possible to distinguish between a *Missed Faults* and a *Successful but Ineffective Fault*, since both have no effect on ciphertexts. In fact even if we know the key in our attacks, we cannot reliably distinguish between those two cases. While distinguishing between both cases is not needed in the attack the amount of *Missed Faults* does impact the success probability.

**Properties of (Ineffective) Biased Faults.** Besides properties that are only mostly related to the fault induction setup alone we define two additional properties that are also related to the concrete type of computations that are targeted by the fault induction:

1. *Bias.* The resulting bias of a certain intermediate variable/byte defines its distance from a uniform random distribution, when observed over multiple encryptions. In the context of SIFA, we measure this distance via the squared euclidean imbalance (SEI). The bias that we observe and exploit in SIFA stems from the combination of *Missed Faults* and *Successful but Ineffective Faults.*
2. *Fault Ineffectivity Rate.* This rate determines how often there is no effect on the computational outcome after fault induction:

$$\frac{\#(\textit{Successful but Ineffective Faults}) + \#(\textit{Missed Faults})}{\#(\textit{Successful and Effective Faults})}$$

The performance of a fault attack is often measured in the number of required faulted encryptions. An ideal fault for SIFA would cause a strong bias, have a high *Fault Ineffectivity Rate*, and would never miss. Such ideal faults are difficult to achieve in practice, since the requirements are somewhat contradictory. When considering the common fault models the stuck-at fault on bit-level is a good candidate for SIFA, since it has a very high *Fault Ineffectivity Rate* of 50% while still causing a decent bias. Higher rates are possible if, e.g., *Missed Faults* occur, however, the observed bias would then be reduced.

In our practical evaluation of SIFA, we deal with faults that are not necessarily optimal for SIFA. On some platforms we observe strong biases but in combination with very low *Fault Ineffectivity Rates.* On other platforms we observe high *Fault Ineffectivity Rates* but weak biases. Nevertheless, we are able to perform practical attacks on various platforms with different countermeasures in place. This demonstrates the versatility of SIFA and biased faults in general. At the end of each practical experiment we shortly discuss our findings regarding fault scenarios in combination with the fault properties described above.

### 6.3.1 Attacks on Detection-based Countermeasures

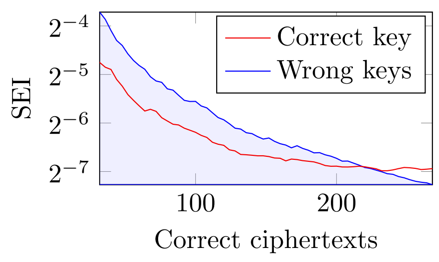
We first target a detection-based countermeasure that uses simple time redundancy with subsequent comparison (Algorithm 5.1). Here, the encryption is executed twice and only if the results of both encryptions are identical, the ciphertext is returned. Note that our attack is just as effective in case more than two redundant executions are performed. We evaluated our attack both for software AES implementations and AES co-processor implementations. The attacks against software AES were evaluated on the 8-bit ATXmega 256A3 computing SubBytes via table lookups, and a 32-bit STM32 F3 using a bitsliced implementaion. The attack against the hardware co-processor AES was performed on the ATXmega 256A3.

**8-bit Software AES.** We used the AES implementation from the AVR-CryptoLib [Avr] as a basis for our implementation featuring a detection-based countermeasure (cf. Section 5.2.2) and performed experiments on both an ATXmega 256A3 and an ATXmega 128D4. Our attacks target the output of the S-box calculation in round 9, and we only induce a fault in one of the two redundant AES encryptions as illustrated in Figure 6.1.

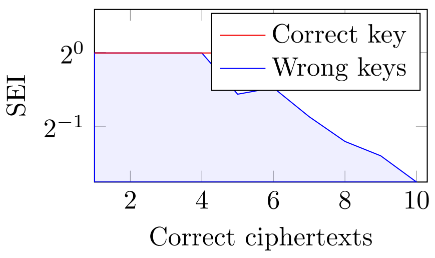
The results presented in Figure 6.5 show the number of correct ciphertexts ($\approx$ 220 and 5) needed until the correct 4-byte key candidate has the highest SEI and thus can be reliably distinguished. In both cases, roughly 1 000 faulted encryptions were necessary to collect the required amount of unaffected and correct ciphertexts. From these results we can see that our fault inductions had quite different effects on both types of microprocessors. On the ATXmega 128D4 platform we are able to induce reliable faults that affect single instructions/bytes. Here, the *Fault Ineffectivity Rate* of $1/256 = 0.39\%$ is very low but the induced bias is strong. We hence suspect that the fault induction set an intermediate (8-bit) variable to a certain value. The observed distribution in the case of an ineffective fault then only contains samples of one of all possible values. In contrast, the *Fault Ineffectivity Rate* of 34% is comparably high on the ATXmega 256A3 platform, possibly due to the occurrence of *Missed Faults*.

**32-bit Bitsliced Software AES on STM32 F3.** In order to evaluate our attack for bitsliced AES implementations, we have used the constant-time bitsliced implementation by Schwabe and Stoffelen [SS16]. The attack setup itself is similar to the one in the previous section.

About 22 000 correct ciphertexts were required to reliably recover 4 bytes of the AES key, as shown in Figure 6.6. In total we have performed 130 000 faulted AES encryptions and received about 26 000 correct ciphertexts. Hence, the *Fault Ineffectivity Rate* was 20 % in this setting. Again, we do expect a mixture of weak biases and *Missed Faults*. Please note that this result is meant as a proof of concept rather than a concrete performance estimation of SIFA against 32-bit platforms.



**(a)** ATXmega 256A3: 220 correct ciphertexts stemming from about **1 000** faulted encryptions are required.



**(b)** ATXmega 128D4: 5 correct ciphertexts stemming from about **1 300** faulted encryptions are required.

**Figure 6.5:** Attacks on 8-bit software AES, detection countermeasure. SEI of the correct key ($\mathrm{SEI}_\mathrm{R}$) vs. best SEI for a wrong key ($\mathrm{SEI}^*_\mathrm{W}$) for $N$ correct encryptions.

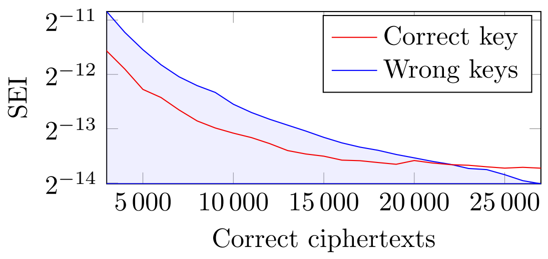


**Figure 6.6:** Attacks on 32-bit bitsliced SW AES, STM32 F3, detection countermeasure. SEI of the correct key ($\mathrm{SEI}_\mathrm{R}$) vs. best SEI for a wrong key ($\mathrm{SEI}^*_\mathrm{W}$) for $N$ correct encryptions. 22 000 correct ciphertexts are required, stemming from about **130 000** faulted encryptions.

**Hardware Co-Processor AES on ATXmega 256A3.** In our attack against the integrated AES co-processor on the ATXmega 256A3, approximately 550 correct ciphertexts were required for recovering 4 bytes of the AES key as shown in Figure 6.7. In total about 800 faulted encryptions were required, the *Fault Ineffectivity Rate* is hence about 69 %. As explained before, such a high rate is only possible if *Missed Faults* occur. We strongly expect that the former is the case, maybe in combination with a weak bias.

### 6.3.2 Attacks on Infective Countermeasures

We evaluated our attack on the infective countermeasure by Tupsamudre et al. [TBM14] from CHES 2014 (Algorithm 6.1). Since the hardware co-processor of the ATXmega 256A3 only computes one complete call of AES, we limit this attack evaluation to purely software-based implementations on the ATXmega 128D4.

We extended the AES implementation from the AVRCryptoLib [Avr] according to Algorithm 6.1. The implementation of the AES round functions itself was

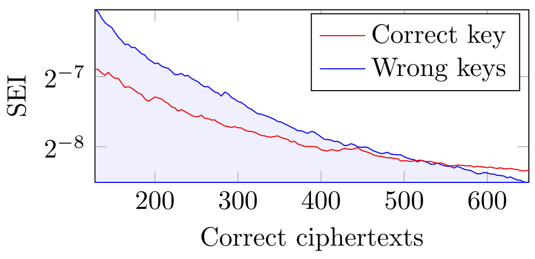


**Figure 6.7:** Attacks on HW AES co-processor, ATXmega 256A3, detection countermeasure. SEI of the correct key ($\text{SEI}_\text{R}$) vs. best SEI of a wrong key ($\text{SEI}^*_\text{W}$) for $N$ correct encryptions. 550 correct ciphertexts were required, stemming from about **800** faulted encryptions.

not modified. Since the authors in [TBM14] did not give any recommendations for $t$, we have evaluated our attack for $t \in [11, 22, 66]$, leading to AES encryptions that require 33, 44, and 88 AES round function calls, respectively.

We started with a simulation of multiple encryption runs in order to determine when a penultimate, non-dummy AES round is performed with highest probability. Clearly, the best time for the attack depends on $t$. According to the simulation results in Table 6.4 we hit a penultimate, non-dummy round with highest probability when targeting the 31st, 41st, and 83rd round, respectively. Once we know the best round for the attack, we can use a similar fault parameterization as in the other experiments with the ATXmega 128D4. In contrast to the detection-based scenario, we cannot detect ineffective faults by observing just one encryption when infection is used. Hence, we always perform one encryption twice, one with fault induction, and one without.

**Table 6.4:** Occurrence of dummy round hits for infective AES.

| Dummy Rounds $t$ | Total AES Rounds | Target Round | Correct Round Hit Probability |
|---|---|---|---|
| 11 | 33 | 31 | 44 % |
| 22 | 44 | 41 | 25 % |
| 66 | 88 | 83 | 11 % |

As mentioned earlier the efficiency of SIFA is, among others, determined by the bias of the induced fault, the fault's *Ineffectivity Rate*, and thus might vary between experiments. In order to allow for an easier comparison between the results for various $t$ we only show practical results where the *Ineffectivity Rate* and the strength of the bias are similar. Both properties can be roughly estimated by an attacker after successful key recovery.

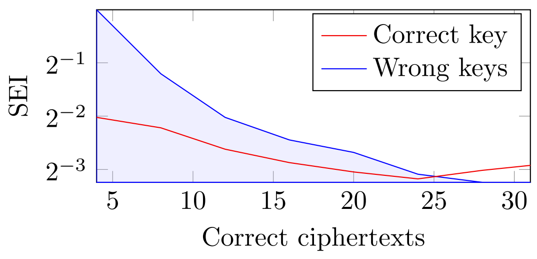
Our attack evaluation was performed for a variable number of dummy rounds with $t \in [11, 22, 66]$, the results are shown in Figure 6.8. In our practical evaluation on the ATXmega 128D4 platform, and depending on $t$, 6 500, 9 000, 46 000 faulted encryptions were necessary to gather the $25, 34, 180$ correct ciphertexts that allowed us to recover 4 key bytes. Here we conclude that the bias of the induced fault was strong, the *Ineffectivity Rate* was low, and the number of *Missed Faults* was also low.

Compared to the analysis in Section 6.2.3 and Table 6.4, the observed increase in the necessary number of ciphertexts roughly matches the predictions, in particular from $t = 22$ to $t = 66$ (increase roughly $\times 5.9$, predicted $\times 5.2$). The observed increase compared to $t = 11$ (roughly $\times 1.4$ to $t = 22$ and $\times 8.0$ to $t = 66$) is less than predicted using the the measured probabilities from Table 6.4 (predicted $\times 3.1$ and $\times 16.0$, respectively). This may indicate that the probabilities are closer to the theoretical estimates $\sigma_+$ from Figure 6.2 (predicted $\times 1.4$ and $\times 11.0$, respectively) and/or a relatively high number of ciphertexts required in the specific experiment for $t = 11$. The latter is very likely because the estimates and normal approximations of Section 6.2.3 assume a relatively low capacity and are not accurate for high capacities and small $N$, as we have for $t = 11$.

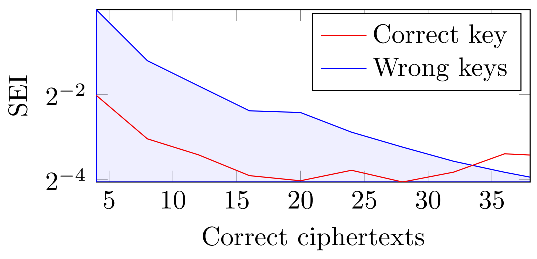
The small available number of samples is not sufficient to derive a detailed fault model (for the sake of comparing our results in more detail with the theoretical model; of course, we do not require the model to perform the attack). However, based on the available data for $t = 22$ and $t = 66$, we can make an educated guess at the effects of our fault setup. The fault ineffectivity rate is very close to $1/256$, as we would expect from a fault that affects a whole byte and a very small number of missed faults, as discussed above. The observed distribution among the ineffective faults during the key recovery phase also suggests a noisy stuck-at distribution, with a signal $\sigma$ less strong than expected for the infection countermeasure. This may indicate that if the stuck-at fault hits the correct round, it hits the correct byte (correct fault effect) in a fraction around half or three quarters of the cases. This model is also a good fit to explain the necessary number of correct ciphertexts for $t = 22$ and $t = 66$ (for $t = 11$, the number of samples is too low to make any useful statements): For example, for $t = 66$ based on $N = 237$ collected samples, assuming $\sigma \approx 0.065$ (see Figure 6.2, and close to $0.5 \cdot 0.11$ from Table 6.4 as assumed above), we would expect about 16.5 stuck-at hits (observed: 18). Using advantage $a = 32$ and target success probability $p = 0.8$, the model would predict that we need roughly 160 ciphertexts, which is only slightly less than we actually needed (Figure 6.8c). For the other cases, similar models also slightly underestimate the necessary number of ciphertexts.

## 6.4 Discussion of other Implementation Countermeasures

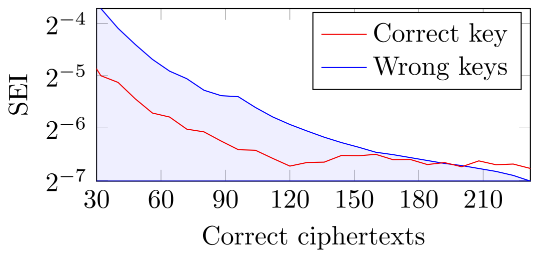
We have demonstrated the effectiveness of SIFA on two different countermeasures using various platforms in the previous section. Nevertheless, more countermeasures exist and we discuss the impact of some of them on SIFA in this section. Overall, SIFA seems to be a powerful attack vector and so far, the main point



**(a)** ATXmega 128D4, **t=11**. 25 correct ciphertexts were required, stemming from about **6 500** faulted encryptions.



**(b)** ATXmega 128D4, **t=22**. 34 correct ciphertexts were required, stemming from about **9 000** faulted encryptions.



**(c)** ATXmega 128D4, **t=66**. 180 correct ciphertexts were required, stemming from about **46 000** faulted encryptions.

**Figure 6.8:** Attacks on infective countermeasure. SEI of the correct key ($\mathrm{SEI}_{\mathrm{R}}$) vs. best SEI of a wrong key ($\mathrm{SEI}_{\mathrm{W}}^{*}$) for $N$ correct encryptions.

we can say regarding countermeasures is that more noise, e.g., by using dummy rounds increases the attack complexity.

### 6.4.1 Infection by Patranabis et al.

The infection countermeasure by Patranabis et al. [PCM15] can be seen as an extended version of the one described in Section 6.1. Hence, we limit our description solely to the actual differences between both designs. The extended countermeasure aims at tackling a shortcoming of the previous design that allowed successful attacks if an attacker is able to alter the control flow or force precise instruction skips. To mitigate this attack vector two adaptations were proposed.

First, an additional randomized string *cstr* is introduced that raises the uncertainty in the execution order of cipher state $R_0$ and redundant state $R_1$ in each round. *cstr* is of length $2t$ and is made up of $t$ 2- bit tuples $[x_i, y_i]$, each of which has either the value $[0, 1]$ or $[1, 0]$. While the execution order of $R_0$ and $R_1$ within one round was fixed in the previous design, *cstr* can now be used to additionally shuffle their execution.

Second, temporary masking is introduced that hides $R_0$ and $R_1$ at the end of an odd round and reveals them at the beginning of the corresponding even round. This has the effect that neither $R_0$ nor $R_1$ expose the output of the previous round after an odd round.

In SIFA, every time we receive a correct ciphertext stemming from an ineffective fault induction on an AES with correct key, we can reduce the number of key candidates. Since both $R_0$ and $R_1$ use the correct key, their execution order within one round is irrelevant for SIFA. The additional temporary masking of states does not affect SIFA either, since all round function calls still work with the original states. Hence, we expect SIFA to perform against the extended version of infection as well as against the version by Tupsamudre et al. [TBM14].

### 6.4.2 Fault Space Transformation

Fault Space Transformation (FST), is a novel fault countermeasure proposed by Patranabis et al. [Pat+17]. This countermeasure works with two redundant states $R_0$ and $R_1$, similar as a detection-based countermeasure that uses redundancy with subsequent comparison. So the encryption is performed on $R_0$ and $R_1$, but under a special linear encoding $R_1 = W(R_0)$ such that it is difficult to induce similar faults in both states.

While there are many possible choices for the linear encoding $W$, the authors propose the usage of the AES-MixColumns function. This choice of $W$ has the beneficial side effect that a one-byte fault in one state is mapped to a 4-byte fault in the other state and vice versa. This linear dependency between $R_0$ and $R_1$ increases the difficulty of inducing two equivalent faults or the exploitation of two biased faults up to a point where they can be considered infeasible in practice. The threat of both DFA as well as DFIA is hence prevented. For a more detailed description of FST we refer to the original paper [Pat+17].

However, SIFA solely relies on observing whether a fault induction in one AES state (either $R_0$ and $R_1$) is ineffective. Since the state $R_0$ is calculated without linear encoding, an attacker, who is able to only fault the branch calculating $R_0$ can expect the same attack complexities as on an ordinary detection-based countermeasures (Section 6.3.1). Faulting the encoded state would work too, the observed bias would be different but still be as strong as in the non-encoded state. Like mentioned before, in SIFA the existence of a bias is sufficient for key recovery. The exact distribution of the bias does not need to be known by the attacker. Hence, SIFA performs against FST as well as shown in the attacks against AES with detection-based countermeasure.

### 6.4.3 Majority Voting

SIFA relies on detecting whether an induced fault is ineffective. A complete mitigation of this attack vector would require a compensation of any fault induction such that every observed encryption is correct. One technique that attempts to do this is Majority Voting (MV) to select which ciphertext is returned. In MV the same computation is performed $n$ times, where $n$ is odd and $\geq 3$. The final output is then defined as the majority over all computed outputs. It depends on the implementations what happens if all ciphertexts differ. Here, we assume that in this case no ciphertext is returned. Such a scheme would prevent SIFA, using single faults per encryption, since one faulty ciphertext is always "overruled" by at least two correct ciphertexts.

In the case of SIFA against MV with $n = 3$ one can simply perform one ordinary targeted $9^{\text{th}}$ round biased fault induction in one computation and any random fault induction in one of the other computations. That way a correct ciphertext has the majority if the biased fault induction is ineffective. For this implementation of the countermeasure, we expect that an attack works with a similar complexity as for detection-based countermeasures, but requiring two faults per execution.

Note that different implementations of a majority voting are possible. For instance, a majority voting on bit level is possible. If $n = 3$, and at least two computations are erroneous, an attacker will get an erroneous ciphertext returned. Still, SIFA is possible as described before, however, an attacker now needs additional computations in order to identify erroneous ciphertexts.

### 6.4.4 Masking

Masking (cf. Section 2.2.1) is a widely-deployed countermeasure against side-channel attacks, where the secret intermediate variables are split into $d$-shares. On the first glance, masking prevents a direct application of SIFA. One way to apply SIFA on masking is to use multiple faults on all shares of a single intermediate variable. While using only single faults for their proposed attack, using faults on multiple locations is a strategy already proposed by Clavier [Cla07] to apply IFA on masked implementations. As a simple example consider an attacker, who is able to induce a fault that sets one variable more likely to 0. If this fault is applied on all shares of the same variable, also the native variable will likely be biased and hence, SIFA will work.

A straightforward application scenario for SIFA against masking are bitsliced implementations that work on all shares concurrently [JS17]. Here, one biased fault is likely to cause a joint non-uniform distribution over all shares that can be exploited by an attacker in the exact same way as described in this chapter. However, SIFA against masking is not restricted to these implementations, or multiple faults. In Chapter 8, we demonstrate that SIFA is applicable on masked implementations of cryptographic primitives with fault countermeasures using just a single fault induction.

## 6.5 Conclusion

In this chapter, we have provided an extensive insight on ineffective faults, where faults are being induced, but not showing an effect. The introduced statistical ineffective fault attacks (SIFA) can be seen as an intersection of the principles exploited by ineffective fault attacks (IFA) [Cla07] and by statistical fault attacks (SFA) [Fuh+13]. While previous work on IFA relies on strong models like stuck-at faults, we were able to relax these conditions up to a point were we only require that intermediate variables follow an unknown but non-uniform distribution in cases where fault inductions have been ineffective. Hence, no special fault profiling of a targeted device is necessary.

SIFA inherits the ability from IFA that it only exploits the output of valid computations, which makes the attack independent of the degree of redundancy used in a countermeasure. As a consequence, it is not harder to attack a detection-based countermeasure performing 16 redundant operations compared to a countermeasure just performing 2. On the other hand, like SFA, SIFA works with minimal assumptions on the effect of the faults. Thus, similar as it has been shown for SFA (e.g., in [Dob+16]), we are able to demonstrate the feasibility of SIFA on various platforms in practice. However, in contrast to SFA, the practical attacks with SIFA are possible even in the presence of countermeasures against fault attacks.

We have demonstrated the improvements of our work over IFA, amongst others by showing the applicability of SIFA on detection-based countermeasures utilizing 32-bit-bitsliced software AES implementations, or hardware co-processor AES implementations. In both cases the induction of precise stuck-at faults in certain bytes, as required by IFA, is considerably harder and was not possible in our fault setup.

Ultimately, we show that SIFA has new applications where neither SFA nor IFA are applicable by demonstrating attacks against an infective countermeasure presented at CHES 2014 [TBM14].

⁎ ⁎ ⁎

# 7 Fault Attacks on Nonce-based Authenticated Encryption: Application to Keyak and Ketje

In Chapter 6 we have shown the capabilities of statistical ineffective fault attacks (SIFA) in case of typical block cipher implementations that feature fault countermeasures such as double-execution or infection. In this chapter, we extend the scope of our analysis to authenticated encryption schemes which are becomming increasingly popular, a trend that is also reflected by NIST's currently ongoing standardization process of lightweight authenticated encryption schemes [Nat18; McK+17]. In particular, we show that SIFA can be applied to the initialization performed in nonce-based authenticated encryption schemes. By targeting the initialization performed during unwrap calls, authenticated encryption schemes provide the attacker with an oracle whether a fault was ineffective or not. This information is all the attacker needs to mount statistical ineffective fault attacks.

As observed by many publications [SKC14; SC15; SC16], the uniqueness of the nonce in authenticated encryption schemes prohibits the straightforward application of analysis techniques like differential fault attacks (DFA) [BS97] to the wrapping phase. In the case of unwrapping, the built-in validation of the authenticity of the processed data often provides an implicit detection of induced faults. Therefore, several attacks published so far assume scenarios, where the uniqueness of the nonce is not ensured [SKC14] or unverified plaintext is released [SC15]. Other works consider attack scenarios without misuse assumptions but require a precise faults inductions at multiple locations during one execution of the authenticated encryption scheme [SC16]. Recently, statistical fault attacks (SFA) that are applicable to a wide-range of AES-based authenticated encryption schemes including popular modes like GCM, CCM and OCB have been published [Dob+16]. However, the presented attacks face some limitations. In particular, they are only applicable to schemes where the secret key is processed right before the data is output. Thus, it is typically not applicable to stream ciphers or duplex-based authenticated encryption modes.

**Our Contribution.** In this chpater, we present the – to the best of our knowledge – first fault attacks targeting unwrap calls that are applicable to implementations of a broad range of nonce-based authenticated encryption schemes. In particular, the presented attacks are applicable whenever the nonce is mixed with the secret key during the initialization. This includes stream ciphers and duplex-based authenticated encryption schemes for which most of the existing fault attacks are not applicable.

We focus our analysis on Keyak and Ketje designed by Bertoni, Daemen, Peeters, Van Assche, and Van Keer [Ber+c; Ber+b]. Both designs are make use of the Keccak-$p$ family of permutations [Ber+11c], which also underlies Keccak/SHA-3 [Nat15]. Please note that the presented attacks do not exploit a weakness inherent in the design of Keyak and Ketje, these two primitives just serve as an example to show the applicability of fault attacks on implementations of duplex-based authenticated encryption schemes.

Our attacks are based on statistical ineffective fault attacks (SIFA) (cf. Chapter 6) and do not require an extensive profiling or characterization of the attacked device. Additionally, they are resilient against "noise" induced by miss-located faults, or in general fault inductions that do not behave as intended. As a consequence, they can be easily applied in practice as demonstrated by our attack targeting 8-bit software implementations of Keyak and Ketje running on an AVR Xmega 128D4. Our choice of using the Xmega 128D4 as an evaluation platform is motivated by the fact that our fault induction setup is most reliable on this platform and so we can more accurately show differences in attack performance between the two schemes. After inducing faults during unwrapping and filtering for the inputs of 24 unaffected computations, we can recover large parts of the secret keys. The remaining unknown key bits can then either be brute-forced or further reduced by repeating the attack and inducing the fault at a different point in time.

**Outline.** In Section 7.1, we give a short overview of authenticated encryption schemes and provide a more detailed description of Keyak and Ketje, the two authenticated encryption schemes that are the main target of our practical attack evaluation. In Section 7.2 we discuss the idea and working principle of the attack. Section 7.3 describes the practical evaluation of our fault attack on a real microprocessor. We conclude the chapter in Section 7.4.

## 7.1 Background

In this section, we first recall the concept of nonce-based authenticated encryption with associated data. We then give a description of the family of permutations Keccak-$p$ and the two duplex-based authenticated encryption schemes Keyak [Ber+c] and Ketje [Ber+b]. While Keyak and Ketje make use of instances of Keccak-$p$, their modes of operation are slightly different.

### 7.1.1 Authenticated Encryption

An authenticated encryption scheme provides confidentiality and authenticity of data and consists of two phases, the wrap call and the unwrap call. The wrap call is usually modeled as a function of four input parameters: a secret key $K$, unique nonce $N$, associated data $A$ and plaintext $P$ [Rog02]. The output of is usually a tuple that consists of a ciphertext $C$ and tag $T$:

$$\mathcal{E}(K, N, A, P) = (C, T)$$

The corresponding unwrap call takes the following five inputs: a secret key $K$, unique nonce $N$, associated data $A$, ciphertext $C$ and tag $T$. The authenticity of $A$ and $C$ is verified via a MAC function that outputs a temporary tag $T^\star$ that is then compared to the given $T$. If they are not authentic the computed plaintext $P$ is not released and the special error symbol $\perp$ is returned instead:

$$\mathcal{D}(K, N, A, C, T) \in \{P, \perp\}$$

The concrete implementation of authenticated encryption schemes can differ significantly. Currently, many of the popular modes like GCM [MV04], CCM [WHF03], EAX [BRW03], and OCB [Rog+01] are all based on block ciphers like AES [DR20]. However, since the announcement of CAESAR [CAE14], we can also see an increasing number of dublex-based authenticated encryption schemes. In the next section, we will present two such dublex-based designs: Keyak and Ketje, in more detail, since we will use them to describe the attack and for the practical evaluation.

### 7.1.2 Keccak-p

Keccak-$p$ is a family of permutations and one of their members, Keccak-$f$[1600], is used in the SHA-3 standard for hashing and extendable-output functions [Ber+11b; NIS15]. Keccak-$p[b,n_r]$ is parameterized by a $b$-bit state, organized in $5 \times 5$ lanes, and number of rounds $n_r$. The round function of Keccak-$p$ consists of the 5 operations: $\theta, \rho, \pi, \chi, \iota$ that are applied to the state in the presented order in every round. From these 5 operations $\chi$ is the only nonlinear transformation. The purpose of $\theta, \pi$ and $\rho$ is to cause diffusion while $\iota$ breaks some symmetries.

### 7.1.3 Keyak

Keyak is a family of authenticated encryption schemes using the Motorist mode of operation [Ber+c]. While there exist five instances of Keyak, named River, Lake, Sea, Ocean and LunarKeyak, covering two different state sizes and different degrees of supported parallelization, we limit our description to the (recommended) LakeKeyak instance. LakeKeyak makes use of the Keccak-$p$[1600,$p$] ermutation and performs authenticated encryption with 128 to 256 bits of secret key, up to 1200 bits of nonce, and 128-bit tags. In the following, we describe the Motorist mode of operation, as used in Keyak. Whenever we refer to Keyak we mean LakeKeyak.

**Motorist Mode.** The Motorist mode defines how incoming plaintexts are processed together with key, nonce, associated data in Keyak. It can be seen as a layer on top of a duplex construction, more exactly, full-state keyed duplex construction [Ber+11a], with the main difference being the size of the input blocks. While the duplex construction only allows input blocks as large as the outer part (rate $r$) of the underlying permutation, Motorist uses a full-state keyed duplex [MRV15] that can make use of the full width of the permutation during absorption as shown in Figure 7.1. While Motorist does provide support for sessions, we do not describe this functionality in more detail here as our implementation attack simply starts a new session for each call.

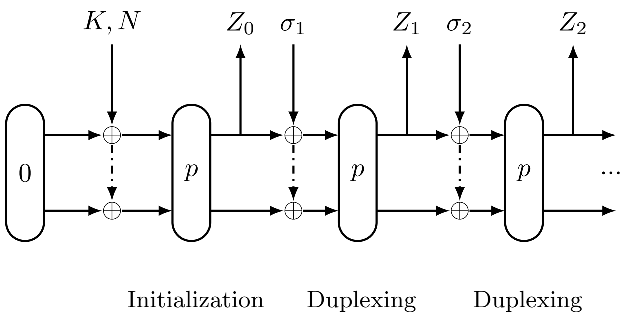


**Figure 7.1:** Simplified depiction of a full-state keyed duplex. $p$ denotes Keccak-$p$, $\sigma$ denotes the input string, and $Z$ denotes the key stream.

### 7.1.4 Ketje

Ketje is a family of authenticated encryption scheme consisting of the MonkeyWrap mode that makes use of different members of Keccak-$p$. While there exist four different instances of Ketje, named Ketje Jr, Ketje Sr, Ketje Minor and Ketje Major, making use of the four different permutations Keccak-$p$[200], Keccak-$p$[400], Keccak-$p$[800] and Keccak-$p$[1600], our practical evaluation is performed on Ketje Jr. Ketje Jr is, together with some of the other instances, intended for usage on constrained devices. It performs authenticated encryption with a 96-bit secret key and up to 86-bits of nonce. Different to KEYAK, in the specification of KETJE every call of the permutation is slightly twisted. The twisted permutation KECCAK-$p^*$ is an extended version of the standard permutation KECCAK-$p$ and always starts with an additional call of $\pi^{-1}$ and ends with an additional call to $\pi$. Note that in an actual implementation, these additional calls are only necessary during absorbing/squeezing but not between two consecutive permutation rounds. In the following we describe the MONKEYWRAP mode of operation, as used in KETJE. Whenever we refer to KETJE we mean KETJE JR.

**Monkey Wrap Mode.** The MONKEYWRAP mode defines how incoming plaintexts are processed together with key, nonce, associated data in KETJE. The initialization of MONKEYWRAP is called *Start* which is similar to the initialization of the MOTORIST mode. First, key $K$ and nonce $N$ are XOR-ed into the zero-initialized state. Then 12 rounds of twisted KECCAK-$p^*$ permutation are performed. The key stream generation *Step* is accomplished by performing duplexing calls, yet this time not the full width of the permutation is utilized, as illustrated in Figure 7.2. Since the rate $r$ of the permutation in KETJE is very small only a 1-round twisted KECCAK-$p^*$ permutation is needed in between *Step* calls. Before the extraction of the tag starts, a 6-round twisted KECCAK-$p^*$ permutation is performed. While MONKEYWRAP does provide support for sessions, we do not describe this functionality in more detail here as our implementation attack simply starts a new session for each call.

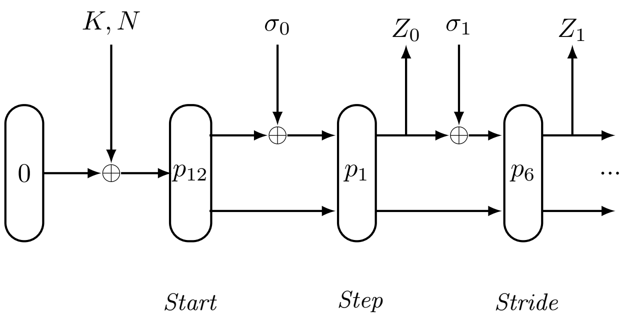


**Figure 7.2:** Simplified depiction of the MONKEYWRAP mode as used in KETJE JR. $p_{n_r}$ denotes the application of a $n_r$-round twisted KECCAK-$p^*[200]$ permutation, $\sigma$ denotes the input string, and $Z$ denotes the key stream.

## 7.2 Attack Strategy

In our attack, we target the unwrap of queries in implementations of LAKEKEYAK and KETJE JR. To be precise, we observe the behavior of the unwrap of valid messages $(N, A, C, T)$ in the presence of faults that are induced during the initialization phase. The valid messages stem from wrap queries to a corresponding authenticated encryption instance loaded with the secret key under attack. For both schemes, the initialization is the application of instances of KECCAK-$p$ to a state that is the concatenation of the secret key $K$ and a publicly known nonce $N$. If the fault induction changes the outcome of this computation, the later computed tag $T$ will also change with overwhelming probability. When compared to the transmitted tag $T$, the verification will then fail. If the induced fault does not change the outcome of the initialization, the verification will succeed and the unwrapping will return a plaintext. Please note that the actual plaintext is not needed for the attack, we solely assume that the attacker is able to distinguish a failed verification from a successful one.

As shown in Chapter 6, inducing faults in a specific intermediate variable of a (cryptographic) computation over different inputs, followed by a subsequent filtering for correct computations, lets us collect a set of input/output pairs for which the targeted intermediate variable shows a biased distribution. In our case, correct computations, and thus the occurrence of ineffective faults (or less desirable missed faults), can be deduced from the condition that the tag verification succeeds. Hence, we assume that the attacker is able to affect one or multiple bits of the internal state before the application of $\chi$ in the 2nd round of the initialization, so that the distribution of these bits is non-uniform for the filtered inputs $(N, A, C, T)$. More concretely, we assume that the attacker is able to collect several nonces $N$, which lead to one or multiple biased bits before the 2nd round $\chi$-layer of the initialization. From this knowledge, the attacker is able to extract information about the secret key. In the following section, we give a detailed description of how key recovery is achieved for KEYAK. A very similar approach can then be used to perform key recovery for KETJE.

### 7.2.1 SIFA Key Recovery Strategy for Keyak

We now describe how key recovery can be performed when performing SIFA attacks on implementations of KEYAK. Similar as previously shown for the AES block cipher in Chapter 6, in case of KEYAK, this requires identifying key bits that are combined with an attacker controlled input (nonce) bits after a non-linear and subsequent (linear) mixing layer. We consider the calculation of $A_{\chi_2}[x, y, z]$ as one such suitable choice and evaluate the this intermediate variable under every possible assignment of the key bits and for every previously collected value of the nonce $N$. For the right key guess, we expect to observe the highest bias in $A_{\chi_2}[x, y, z]$. But at first, we have to identify the involved bits.

To do so, we need to determine the bits at the input of the linear layer of the 2nd round, which are involved in the calculation of $A_{\chi_2}[x, y, z]$. The linear layer of one round of KECCAK-$p[1600, 12]$ consists of the application of the single round functions $\theta$, $\rho$, and $\pi$. The function $\pi$ just swaps the words, so that

$$A_{\chi_2}[x, y, z] = A_{\pi_2}[(x + 3y) \bmod 5, x, z] \ .$$

The function $\rho$ rotates each lane by a different offset $R[x, y]$. Hence,

$$A_{\chi_2}[x, y, z] = A_{\rho_2}[(x + 3y) \bmod 5, x, (z - R[(x + 3y) \bmod 5, x]) \bmod 64] \ .$$

Finally, $\theta$ computes its output by XOR-ing each bit with the parity of two columns in the array, thus, one bit $A_{\chi_2}[x, y, z]$ is the sum of 11 input bits to $\theta$.

$$\begin{aligned} A_{\chi_2}[x, y, z] =& A_{\theta_2}[(x + 3y) \bmod 5, x, (z - R[(x + 3y) \bmod 5, x]) \bmod 64] \\ & \oplus \bigoplus_{y'=0}^{4} A_{\theta_2}[(x + 3y - 1) \bmod 5, y', (z - R[(x + 3y) \bmod 5, x]) \bmod 64] \\ & \oplus \bigoplus_{y'=0}^{4} A_{\theta_2}[(x + 3y + 1) \bmod 5, y', (z - R[(x + 3y) \bmod 5, x] - 1) \bmod 64] \end{aligned}$$

Each of the 11 bits $A_{\theta_2}[x_i, y_i, z_i]$ can be calculated using three input bits to $\chi$. Therefore,

$$\begin{aligned} A_{\theta_2}[x_i, y_i, z_i] =& A_{\chi_1}[x_i, y_i, z_i] \oplus \\ & ((A_{\chi_1}[(x_i + 1) \bmod 5, y_i, z_i] \oplus 1) \cdot A_{\chi_1}[(x_i + 2) \bmod 5, y_i, z_i]) \ . \end{aligned}$$

Note that two bits at the input of $\theta$ in the $2^{\text{nd}}$ round needed in the calculation of $A_{\chi_2}[x, y, z]$ are adjacent bits of the same S-box, namely

$$\begin{aligned} & A_{\theta_2}[(x + 3y) \bmod 5, x, (z - R[(x + 3y) \bmod 5, x]) \bmod 64] \\ & A_{\theta_2}[(x - 3y - 1) \bmod 5, y, (z - R[(x + 3y) \bmod 5, x]) \bmod 64] \ . \end{aligned}$$

As a consequence, $A_{\chi_2}[x, y, z]$ depends on only 31 bits of $A_{\chi_1}[x_j, y_j, z_j]$. The bits at the input to the $1^{\text{st}}$ round that are needed to compute the 31 bits $A_{\chi_1}[x_j, y_j, z_j]$ can be determined in a similar manner as done for the second round. However, doing so for general values of $x$ and $y$ gets a bit clumsy, hence, we focus on the restricted case of calculating $A_{\chi_2}[0, 0, 0]$. Determining the necessary bits to calculate $A_{\chi_2}[x, y, z]$ by hand is quite time consuming and also error prone. Thus, we have used a search tool [DEM15], which has been developed to search for linear trails to identify the bits at the input of KECCAK-$p$ that are involved in the calculation of a certain $A_{\chi_2}[x, y, z]$. In Figure 7.3, we give the involved bits for calculating $A_{\chi_2}[0, 0, 0]$. The figure represents one lane as hexadecimal value, where bits that are set to 1 are needed in the calculation of $A_{\chi_2}[0, 0, 0]$. A corresponding figure for KETJE JR is given in Figure 7.4.

## 7.2.2 Recovered Bits

In this section, we will discuss how much information on the key bits can be recovered by exploiting a bias in $A_{\chi_2}[x, y, z]$. For the sake of simplicity, we will stick to the example of $A_{\chi_2}[0, 0, 0]$. Bits having a gray background in Figure 7.3 are bits that represent the 128 key bits. Hence, to compute $A_{\chi_2}[0, 0, 0]$, 25 bits

| Input to | Bit positions | | | | |
|---|---|---|---|---|---|
| | C-62-C1---9C---1 | 8E-12----C3----C | 6-1--45----E-3-4 | -384983--118---6 | 1---4228184--181 |
| | C-62-C1-189C---- | 8E-12----C38---C | 661--45----E-3-- | -384982--118-3-6 | 1---5A28184--181 |
| $\theta_1$ | D-62-C1---9C---- | 8E212----C3----C | 6-1--45---3E-3-- | -384986--118---6 | 1---4228194--181 |
| | C-62-C1---DC---- | 8E-12----C34---C | 6-11-45----E-3-- | -3849C2--118---6 | 1-8-4228184--181 |
| | C-624C1---9C---- | CE-12----C3----C | 6-1--45----E-3-8 | -384982--118-1-6 | 1--44228184--181 |
| | ---------------1 | 8--------------1 | 8--------------1 | 8--------------- | ---------------1 |
| | ---------------1 | 8--------------1 | 8--------------- | 8--------------- | ---------------1 |
| $\chi_1$ | ---------------1 | 8--------------1 | 8--------------- | 8--------------- | ---------------1 |
| | ---------------1 | 8--------------1 | 8--------------- | 8--------------- | ---------------1 |
| | ---------------1 | 8--------------1 | 8--------------- | 8--------------- | ---------------1 |
| | ---------------1 | 8--------------- | ---------------- | ---------------- | ---------------1 |
| | ---------------- | 8--------------- | ---------------- | ---------------- | ---------------1 |
| $\theta_2$ | ---------------- | 8--------------- | ---------------- | ---------------- | ---------------1 |
| | ---------------- | 8--------------- | ---------------- | ---------------- | ---------------1 |
| | ---------------- | 8--------------- | ---------------- | ---------------- | ---------------1 |
| | ---------------1 | ---------------- | ---------------- | ---------------- | ---------------- |
| | ---------------- | ---------------- | ---------------- | ---------------- | ---------------- |
| $\chi_2$ | ---------------- | ---------------- | ---------------- | ---------------- | ---------------- |
| | ---------------- | ---------------- | ---------------- | ---------------- | ---------------- |
| | ---------------- | ---------------- | ---------------- | ---------------- | ---------------- |

**Figure 7.3:** Bits involved in calculation of $A_{\chi_2}[0,0,0]$. The position of the 128-bit key is highlighted in gray. Zeros are replaced by - to improve readability.

of the key have to be guessed. However, only the 17 bits:

$$A_{\theta_1}[0,0,0]\ \ , A_{\theta_1}[0,0,18], A_{\theta_1}[0,0,20], A_{\theta_1}[0,0,23], A_{\theta_1}[0,0,36], A_{\theta_1}[0,0,43],$$
$$A_{\theta_1}[0,0,53], A_{\theta_1}[0,0,54], A_{\theta_1}[1,0,2]\ \ , A_{\theta_1}[1,0,20], A_{\theta_1}[1,0,21], A_{\theta_1}[1,0,27],$$
$$A_{\theta_1}[1,0,48], A_{\theta_1}[1,0,58], A_{\theta_1}[1,0,59], A_{\theta_1}[1,0,63], A_{\theta_1}[2,0,62]$$

can influence $A_{\chi_2}[0,0,0]$ in a nonlinear manner, while the 8 bits:

$$A_{\theta_1}[0,0,19], A_{\theta_1}[0,0,42], A_{\theta_1}[0,0,49], A_{\theta_1}[1,0,3]\ \ , A_{\theta_1}[1,0,26], A_{\theta_1}[1,0,45],$$
$$A_{\theta_1}[1,0,57], A_{\theta_1}[2,0,61]$$

only have a linear influence.

As a consequence, we can at most uniquely identify the 17 bits that influence $A_{\chi_2}[0,0,0]$ in a nonlinear way. For the 8-bits that influence $A_{\chi_2}[0,0,0]$ in a linear way, only their XOR-sum (parity) effects the value of $A_{\chi_2}[0,0,0]$. Since for 8 bits, half of the possible assignments have parity 0 and the other half has parity one, we get at least $2^7$ key candidates that always lead to the same result. Please note that this is a rather simplistic evaluation and does not consider the dependencies of the nonlinear bits and also the bits, which are used as nonce and constants. In fact, the key recovery depends on the value of these bits, since an unfortunate choices for the nonce can, for instance, lead to situations, where some S-boxes are linearized for some key bits, or some key bits are always blocked, so that they do not influence $A_{\chi_2}[0,0,0]$. For instance, let us have a look at the results of one of our concrete experiments given in Section 7.3. Instead of recovering 17 out of the 25 bits uniquely from $2^7$ key candidates scoring best, we are able to recover 15 of the 25 bits uniquely out of $2^9$ key candidates that score best.

| Input to | Bit positions | | | | |
|---|---|---|---|---|---|
| | ff | bf | 7f | bf | fb |
| | fe | bf | 7f | bf | fb |
| $\theta_1$ | fe | bf | 7f | ff | fb |
| | fe | bf | 7f | bf | fb |
| | fe | ff | 7f | bf | ff |
| | -1 | 81 | 81 | 8- | -1 |
| | -1 | 81 | 8- | 8- | -1 |
| $\chi_1$ | -1 | 81 | 8- | 8- | -1 |
| | -1 | 81 | 8- | 8- | -1 |
| | -1 | 81 | 8- | 8- | -1 |
| | -1 | 8- | -- | -- | -1 |
| | -- | 8- | -- | -- | -1 |
| $\theta_2$ | -- | 8- | -- | -- | -1 |
| | -- | 8- | -- | -- | -1 |
| | -- | 8- | -- | -- | -1 |
| | -1 | -- | -- | -- | -- |
| | -- | -- | -- | -- | -- |
| $\chi_2$ | -- | -- | -- | -- | -- |
| | -- | -- | -- | -- | -- |
| | -- | -- | -- | -- | -- |

**Figure 7.4:** Bits involved in calculation of $A_{\chi_2}[0,0,0]$. Zeros are replaced by - to improve readability.

## 7.3 Practical Evaluation

We now describe the practical evaluation of our attack on a microprocessor implementation. Although we have performed attacks on both Keyak and Ketje, we limit our description to (Lake)Keyak, since the attack procedure is similar for both schemes. We do, however, state the results for both schemes at the end of this section. We start this section by giving a quick overview of the attack procedure in Section 7.3.1. We then describe the hardware/software that we have used to perform our attack evaluation in Section 7.3.2. After that, we state requirements on a fault setup more generally in Section 7.3.3. Finally, we present the results of our fault attacks on LakeKeyak and Ketje Jr in Section 7.3.4.

### 7.3.1 Attack Procedure

As described in Section 7.2, our key recovery exploits the input of specific unwrap calls to Keyak. We are interested in unwrap calls that have a bias in one or multiple bits of the state before $\chi$ in the 2nd round. To achieve the required filtering of inputs we use statistical ineffective fault attacks (SIFA), as proposed in Chapter 6.

Before the attack we set the secret key of the microprocessor Keyak implementation to a constant and unknown value. During the attack we send valid messages, consisting of random nonce and tag, to the microprocessor, induce a clock glitch with constant offset during the computation and observe the behavior.

The valid messages stem from wrap queries to a corresponding authenticated encryption instance loaded with the secret key under attack. The tag verification is used to detect whether or not an induced fault was ineffective.

### 7.3.2 Attack Setup

The practical evaluation of our fault attack was done on an 8-bit Xmega 128D4 microprocessor. The attacked software implementation of Keyak consists of two parts. The first part is a C implementation of the Motorist mode of operation. The second part is a fast 8-bit AVR optimized assembler implementation of Keccak-$p$. Both implementations are taken from the eXtended Keccak Code Package [Ber+a] and therefore represent a good target software implementation for our practical evaluation. The clock signal of the microprocessor is generated by a Spartan-6 FPGA running at 12 MHz. We additionally use this FPGA for the insertion of glitches onto the clock signal. The insertion of clock glitches is achieved by adding an additional fast edge onto the clock signal at a specified point in time. If timed correctly, this violates normal operating conditions of the device and can result in temporary erroneous computations on the microprocessor.

In our practical evaluation we can force strong biases in virtually every state bit that is affected by $\chi$, however only in blocks of 8 bits at a time (which is not surprising on a 8-bit architecture). We suspect that our glitch does skip one of the Xor instructions in the bit-sliced $\chi$ implementation, but we cannot say for sure though.

### 7.3.3 Attack Setup - Requirements

As we use SIFA, the requirements we have on the locality and especially the effect of the fault are quite relaxed. Basically, we only need some sort of bias in any bit at the input of $\chi$ in the $2^{\text{nd}}$ round. This can be achieved, e.g., by faulting instructions in $\chi$, slightly before $\chi$, or by directly faulting registers using lasers. In the case of AES, such fault inductions have already been demonstrated for multiple microprocessors (cf. Chapter 6) and even for hardware co-processors [Dob+16]. One way to find a suitable glitch location in practice would be to estimate the clock cycles until the targeted operation is executed. Hence, in our scenario, one can estimate the time frame of the $2^{\text{nd}}$ round and try to induce a glitch in several different clock cycles towards the end of that round.

### 7.3.4 Results

**Keyak.** As already mentioned in Section 7.2.2, when getting a bias in the bit $A_{\chi_2}[0,0,0]$ located at the input of the $2^{\text{nd}}$ round $\chi$-layer, 25 bits of the key are involved in its calculation. In our attack, we guess these 25 bits and evaluate the bias in $A_{\chi_2}[0,0,0]$ for each key guess. Since some of the guessed key bits only influence $A_{\chi_2}[0,0,0]$ in a linear manner, we get several equivalent key candidates having the same bias. As a consequence, Figure 7.5 shows the advantage in bits the attacker gets from guessing key candidates down to a bias which also the correct key guess over just randomly guessing the key, which is $\log_2(\#\text{total keys}) - \log_2(\#\text{candidate keys})$. 24 inputs to such unwrap

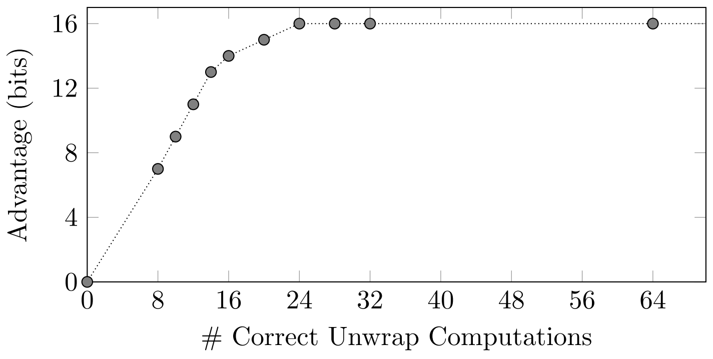


**Figure 7.5:** Attack on KEYAK. Advantage in bits when targeting $A_{\chi 2}[0,0,0]$ and guessing the associated 25 bits of the 128-bit key.

computations are necessary to get a maximum advantage of 16 bits. In our case, we get $2^9$ keys ranked top that have the same bias (not considering its sign). From those $2^9$ keys, the values of 15 key bits can be uniquely determined. Due to the architecture of the implementation, we do not only get a bias in one bit, but one byte. By combining this information, we can uniquely determine 82-bits of the key.

In our attack setup, we are able to perform about 20 faulted unwrap computations per second. According to the practical evaluation, in about 1 out of 250 of the cases the induced fault is ineffective. The total time it took us to gather the required amount of inputs is roughly 5 minutes.

**Ketje.** In the attack on implementations of KETJE we use the same fault location as in the case of KEYAK. This is however not strictly necessary. Even though both schemes use variants of KECCAK-$p$ during initialization, the influence of key bits on one of our biased bits before $\chi$ in the 2nd round is quite different, mainly due to the fact that the lane sizes are different. In contrast to LAKEKEYAK, in KETJE JR nearly all key bits influence each of our biased bits, most of the time in a linear way. Hence, for KETJE JR we instead guess the 200-bit equivalent key before $\chi$ in the 1st round (i.e., after the first linear layer). By doing so we can reduce the dependency on the equivalent key to 31 bit and guessing becomes feasible in practice.

In our attack setup we can recover about 19 bits of the equivalent key that correspond to one biased bit in about 10 hours using a single thread on an Intel Xeon CPU. Note that this time can be significantly improved, since we used for our evaluation purposes just the unoptimized reference implementation. Furthermore, the task of key guessing can be parallelized trivially. If we parallelize the computations for, e.g., the 8 bits that were affected by our fault induction we can recover 152 bits of the equivalent key in the same amount of time. The remaining bits can be determined either by brute-force or repeating key recovery for a different fault location.

In total, again 24 inputs of unaffected unwrap computations are necessary for key recovery as shown in Figure 7.6. The total time it took us to gather the required amount of inputs is below 5 minutes. Hence, the time complexity of entire attack is dominated by the key guessing and was performed in about 10 hours.

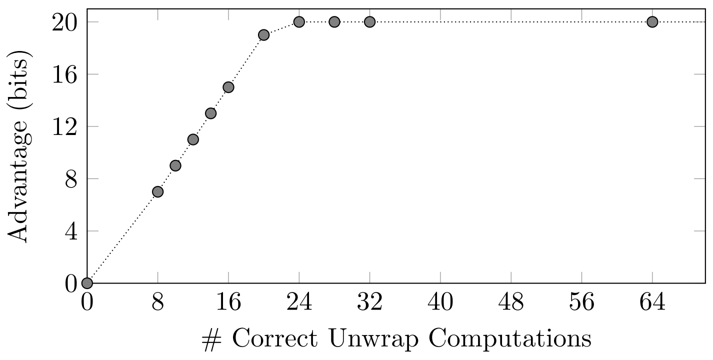


**Figure 7.6:** Attack on Ketje. Advantage in bits when targeting $A_{\chi 2}[0,0,0]$ and guessing the associated 31 bits of the 200-bit equivalent key.

## 7.4 Conclusion

In this chapter, we have presented the first fault attacks targeting a broad range of nonce-based authenticated encryption schemes. While fault attacks on authenticated encryption have already been shown before, they are mostly limited to schemes that additionally feature a final key addition, and thus not directly applicable to most duplex-based authenticated encryption modes. We close this gap and show attacks based on SIFA, which are in principle applicable to most nonce-based authenticated encryption schemes that perform some sort of initialization where the nonce (or an other publicly known input) is mixed with the secret key. Since we only need to know whether a fault induction was ineffective or not, attacking the unwrapping in authenticated encryption schemes gives us a perfect oracle. Our attack evaluation is focused on Keyak and Ketje, however, we conjecture that our attack can also be adopted to other schemes like the CAESAR finalists Acorn, Aegis, Ascon, Morus, etc. in a rather straightforward way.

SIFA is resistant to popular fault countermeasures like double-execution and infection-based countermeasures and even additional masking does not preclude this attack vector. The key recovery is capable of dealing with an arbitrary amount of noise (however requiring more faulted unwrap computations) that might arise due to possibly imperfect fault inductions. The effort required to perform our attack is rather low. We neither require perfectly timed faults nor precise knowledge about the effect of the induced fault. In our fault setup we are able to collect enough material for key recovery within 5 minutes. The actual key recovery for Keyak and Ketje is easily parallelizable and takes about 30 minutes and 10 hours, respectively. The hardware cost of the attack setup does not exceed 300$.

＊＊＊

# 8 Statistical Ineffective Fault Attacks on Masked AES with Fault Countermeasures

The previous chapters have shown the capabilities of statistical ineffective fault attacks (SIFA) against implementations of various kinds of symmetric cryptographic schemes, and in the presence of typical fault countermeasure techniques. In this chapter, we extend the scope of our analysis to implementations featuring additional countermeasures against passive side-channel attacks, like power [KJJ99] or EM analysis [QS01]. In the case of symmetric cryptography, one commonly used approach to protecting an implementation against these attacks is to use masking, essentially a secret-sharing scheme, in software or hardware [RP10; Rep+15a; GMK16].

When considering combined countermeasures against both active and passive attacks respectively, the standard reasoning is that the effects of masking and error-detection add up. For example, we can assume a block cipher implementation protected by a masking scheme operating on $d + 1$ shares that performs each masked block cipher computations $r + 1$ times and only releases an output if all redundant computations match. In this case, the implementation is typically assumed to be secured against up to $r$ fault inductions in the block cipher computations due to the detection countermeasure, as well as secured against side-channel attacks able to observe up to $d$ intermediate variables of choice due to the masking scheme.

This reasoning is valid for fault attacks that exploit faulty outputs of a cryptographic algorithm to reveal the key. The most prominent attacks of this type are differential fault attacks (DFA) [BS97] and statistical fault attacks (SFA) [Fuh+13]. However, some variants of fault attacks are based on a different approach. ineffective fault attacks (IFA) [Cla07] and statistical ineffective fault attacks (SIFA) (cf. Chapter 6), exploit those outputs of a cipher that are correct although a fault induction has been performed. While IFA requires exact knowledge about the location and effect of a fault, SIFA has much more relaxed requirements, thus allowing to exploit noisy faults whose exact effect is unknown to the attacker. The basic idea of SIFA is to repeatedly execute a cryptographic operation with a fixed key for different inputs and to apply a fault induction for each execution. The attacker then collects those outputs of the cryptographic operation where the fault induction has not changed any intermediate variable. Given that the implementation is protected by an error-detection scheme, such as a redundant execution of the cipher, this corresponds exactly to the valid outputs of the system. In fact, the error detection that is implemented against DFA provides exactly the filtering of the outputs which is needed to apply SIFA or IFA.

**Our Contribution.** Before, implementations combing masking and error-detection schemes have been typically thought to be secure against attacks exploiting single ineffective faults due to masking, as discussed for example by Clavier for IFA [Cla07]. It was typically assumed that all shares representing an intermediate variable would need to be faulted for exploiting ineffective faults and it was an open question whether this can be done efficiently in practice.

In this chapter, we show that SIFA attacks are much more powerful than expected before. Our central contribution is to show that the practical difficulty of performing SIFA attacks is not only independent of the degree of redundancy but also essentially independent of the number of shares in a typical masking scheme. This fact makes SIFA an especially promising attack strategy against cryptographic implementations featuring countermeasures against both power analysis and fault attacks. Additionally, and in some contrast to [Fuh+13] and our works in Chapter 6 and Chapter 7, we show that SIFA does not necessarily require faults whose direct effect on the targeted intermediate variable causes a bias. Instead, any type of fault effect, such as deterministic bitflips, while not considered suitable for mounting SIFA attacks before, can be suitable to mount SIFA attacks. In order to better explain why this is the case, we introduce a change of perspective that separates the location of a fault induction within a computation from the location of effected intermediate variables that an attacker can observe during key recovery. To back up our claims we provide a broad evaluation based on multiple (masked) S-boxes and two implementations of the entire AES block cipher.

More concretely, we demonstrate that faulting a single share during the computation of an S-box is often sufficient to induce a bias in an native intermediate variable, which can then be exploited with a statistical analysis based on SIFA. Unlike classical fault attacks, attackers cannot directly use this fault as a distinguisher for the key recovery attack: they cannot recover this intermediate variable from observing the ciphertext and guessing parts of the key, but can only recover the native output of the S-box (cf. Figure 8.1). We analyze the impact of the local fault on the native output for several different S-boxes (including

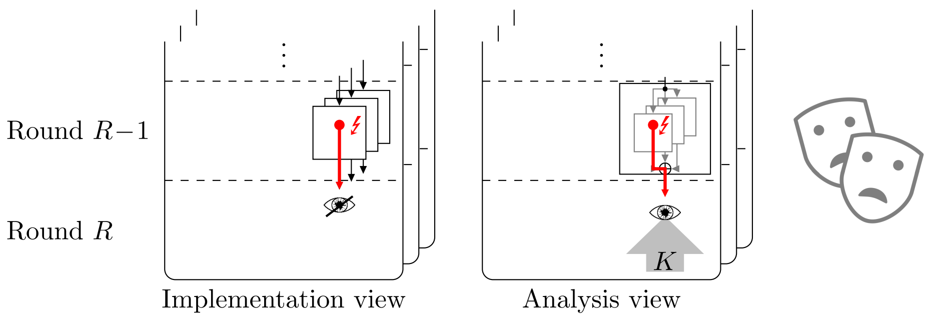


**Figure 8.1:** Biased fault attacks on masked, redundant implementations: High-level view.

the AES S-box), fault distributions, masking schemes, and protection orders, as well as other fault countermeasures like dummy rounds. We conclude that in all analyzed cases, a single fault attempt per encryption is sufficient to recover part of the key, given a suitable number of faulted encryptions. This number depends on the precision of the fault and the deployed countermeasures; for example, 1000 encryptions with a cheap clock glitching setup are sufficient for an 8-bit AES software implementation protected with 10th-order masking and arbitrary temporal redundancy on block cipher level running on a standard 8-bit microcontroller.

## 8.1 Faults on Masking

In this section, we study how single faults in masked cryptographic computations affect their native inputs and outputs. First, we discuss how faults influence the distribution of native outputs of masked AND gates. After that, we evaluate how single faults influence native inputs/outputs of entire masked S-boxes. Finally, we take a closer look at the root cause allowing SFA and SIFA in masked S-boxes and argue that these attacks are probably always applicable no matter the concrete S-box design.

For an easier understanding of why single faults on a single share can cause a bias in native variables, we consider very simple fault models such as stuck-at faults in the following exposition. However, it is important to note that the attack approach generalizes efficiently to noisy, unpredictable faults. For a discussion of how the attack complexity scales under the influence of noise, we refer to Section 8.3 and the SIFA analysis in Chapter 6).

### 8.1.1 Faulting Masked AND Gates

Masking is a secret sharing technique that splits security sensitive variables of cryptographic algorithms into multiple randomized shares to protect their implementations against certain power analysis attacks. We provide a more detailed description of masking in Section 2.2.1. While the application of masking can be easily performed for functions which are linear over GF($2^n$) – for example, the masked calculation of $x \oplus y$ can be performed share-wise ($x_i \oplus y_i$) –, the secure implementation of nonlinear functions usually requires the introduction of fresh randomness. We now consider the generic algorithm for masked multiplication (or AND in GF(2)) by Ishai et al. [ISW03] as an example for such a nonlinear function (cf. Algorithm 2.1). In order to securely calculate $q = x \cdot y$, each of the $d+1$ shares of $x$ is multiplied with each of the shares of $y$, resulting in $(d+1)^2$ multiplication terms. Subsequently, the multiplication terms are summed up together with fresh random variables denoted $r_{i,j}$, and distributed to the output shares $q_i$. In general, the joint distribution of any $d$ shares of $q$ in the masked multiplication algorithm is uniform, or in other words, any $d$ shares are independently and identically (uniformly) distributed. It thus appears as if in order to insert a bias in the underlying native variable, an attacker would need to insert a biased fault in either each share of $x$ or $y$, or to insert a bias in each of the component functions in the calculation of $q$. However, in the following, we show that this intuition is not right.

We first note that the calculation of the AND $q = x \cdot y$ itself has a probability of 25% for $q$ to be 1. An attacker therefore successfully biases the masked AND gate if the probability of $q$ to be 1 is more or less likely than 25%. As an example, we consider an attacker who can skip any AND calculation in Section 2.2.1, for instance the first AND calculating $x_0y_0$ in $q_0$. The shared function then effectively calculates $q$ (i.e. $q_0 \oplus q_1$) to be $x_1y_1 \oplus x_0y_1 \oplus x_1y_0$, which has a probability of 37.5% to be 1. If the attacker instead introduced a fault that skips the addition of the uniformly random bit $r_{0,1}$ in $q_0$, then the distribution of $q$ would again be biased, since the probability of observing a 1 changes from 25% to 50%.

We observe the same biases when looking at single faults for other masked AND gates, like the one used in the CMS scheme of Reparaz et al. [Rep+15a] or in the domain-oriented masking (DOM) scheme by Gross et al. [GMK16]. This same bias behavior results from the fact that these masked ANDs calculate the same terms $x_iy_j$. The masked ANDs only differ in the arrangement of $x_iy_j$ in $q_0$ and $q_1$, and the amount of used fresh randomness. Since $q$ is equal to $q_0 \oplus q_1$, the arrangement of the terms has no influence on the bias behavior of $q$, and a fault of an addition of a single random $r$ bit has the same impact on all masked ANDs.

Another prominent protection mechanism falling in the category of masking schemes are threshold implementations [NRR06]. When compared to masking schemes like CMS and DOM, threshold implementations require an increased number of shares to achieve first-order side-channel resistance but can do so without requiring fresh randomness. In order to explore the impact on threshold implementations, we look at a four-share realization of a first-order masked AND gate by Nikova et al. [NRR06]:

$$\begin{aligned}
q_0 &= (x_2 \oplus x_3)(y_1 \oplus y_2) \oplus y_1 \oplus y_2 \oplus y_3 \oplus x_1 \oplus x_2 \oplus x_3 \\
q_1 &= (x_0 \oplus x_2)(y_0 \oplus y_3) \oplus y_0 \oplus y_2 \oplus y_3 \oplus x_0 \oplus x_2 \oplus x_3 \\
q_2 &= (x_1 \oplus x_3)(y_0 \oplus y_3) \oplus y_1 \oplus x_1 \\
q_3 &= (x_0 \oplus x_1)(y_1 \oplus y_2) \oplus y_0 \oplus x_0
\end{aligned} \tag{8.1}$$

For this shared And gate, we perform two experiments. In the first experiment, we have a look at the distribution of the output $q = q_0 \oplus q_1 \oplus q_2 \oplus q_3$ assuming an instruction skip. In the second, we fix one input share $x_0$ to zero and look what happens.

For the instruction skip, we assume that in $q_0$ one instruction is skipped and so $q_0 = (x_2)(y_1 \oplus y_2) \oplus y_1 \oplus y_2 \oplus y_3 \oplus x_1 \oplus x_2 \oplus x_3$ is calculated, the other shares are processed correctly. In this case, we observe that for all 256 possible values of the shared input, the native output is 160 times (62.5%) 0 and 96 (37.5%) times 1, a clear deviation of the value a correctly computed And should have.

Next, we fix $x_0$ to zero and perform the computations according to Equation 8.1 for all 256 possible values the shared input can take. If we now look at $q$, we see that 192 times a 0 (75%) appears and 64 times a 1 (25%), which corresponds to the distribution of a correct And gate. However, if we only consider correct computations of $q = x \cdot y$, we observe that only 192 out of 256 computations are performed correctly. For those correct computations $q$ is 160 times a 0 (83.3%) and 32 times a 1 (16.6%). This "filtered" distribution is the one an attacker can potentially exploit in the case of SIFA. In the next section, we will discuss the consequences of our observations with respect to S-boxes.

### 8.1.2 Faulting Masked S-boxes

In this section, we discuss how single faults influence the behavior of S-boxes. It is worth mentioning that our selection of masked S-boxes is arbitrary and does not imply that those S-boxes are weaker or more susceptible to SFA and SIFA than others. We have selected those S-boxes, because they have a simple and compact description. We will start with a compact 4-bit S-box called Sbox13 [Ull+11] shown in Figure 8.2. We have implemented a masked implementation of this S-box in software by using a four-shared threshold implementation of And (see Equation 8.1), Or and Xor. We target exclusively the And labeled with $q$, $x$, and $y$ in Figure 8.2.

For the first experiment, we assume an instruction skip that alters the execution of the first And of the S-box, changing the calculation of one share $q_0$ to $q_0 = (x_2)(y_1 \oplus y_2) \oplus y_1 \oplus y_2 \oplus y_3 \oplus x_1 \oplus x_2 \oplus x_3$. In Figure 8.3a, we record the distribution of each native input variable and each native output variable for each of the $2^{4 \cdot 4} = 65536$ shared input combinations, which can be exploited by an SFA [Fuh+13], as well as the "filtered" distribution for ineffectively faulted S-box transitions, which can be exploited by SIFA (cf. Chapter 6). This "filtered" distribution stems from the subset of transitions, for which the induction of the fault has no influence on the output of the S-box.

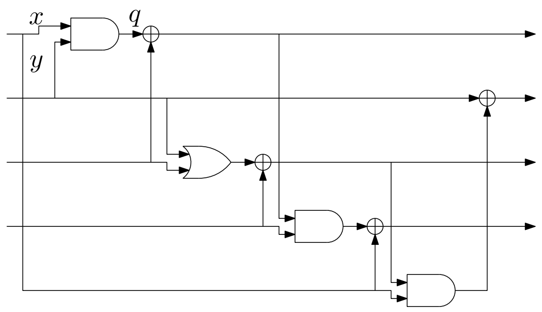
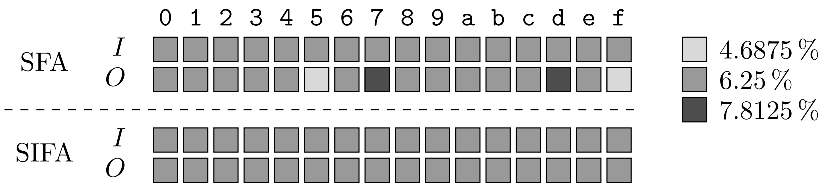
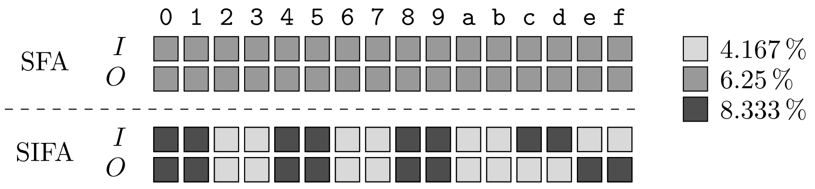


**Figure 8.2:** Schematic of the $4 \times 4$ S-box: Sbox13 [Ull+11].



**(a)** Fault example 1: Skip first XOR instruction in share $q_0$ in Equation 8.1.



**(b)** Fault example 2: Set input share $x_0$ of the first AND to zero in Equation 8.1.

**Figure 8.3:** Distribution of input $I$ and output $O$ for faulted $4 \times 4$ Sbox13.

As we can see in Figure 8.3a, in the unfiltered case, we see a clearly non-uniform distribution, which can be possibly exploited by an SFA. However, we observe a uniform distribution in the SIFA case. So does this mean this S-box is secure against SIFA?

Let us consider the distributions we obtain when setting one share $x_0$ at the input of the first AND permanently to 0. The corresponding distributions we get in this experiment are shown in Figure 8.3b. As we can see in Figure 8.3b, the situation for this fault changes, so that now, the ineffective faults can be exploited, whereas the distribution without filtering cannot be exploited. Until now, we have exploited the sequential sharing of instructions, especially that we can change the distribution of an AND gate. So one might wonder what happens if an S-box is directly shared, so that the output shares are uniformly distributed.

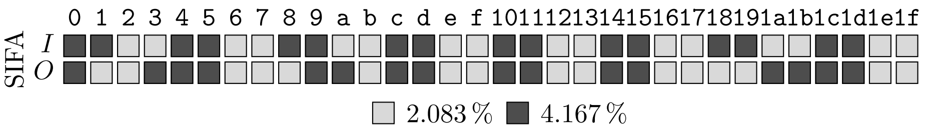


**Figure 8.4:** Distribution of $I, O$ for faulted 4-shared KECCAK S-box.

To explore the case of directly shared S-boxes, let us take a closer look at the uniform 4-share threshold implementation of the KECCAK S-box proposed by Bilgin et al. [Bil+13]. Here, $A[i]$, $B[i]$, $C[i]$, and $D[i]$ represent the 4 shares of bit $i$ with $i = 0, \dots, 4$. While the bit $i = 3$ is calculated by:

$$\begin{aligned}
A'[3] &\leftarrow B[3] \oplus B[0] \oplus C[0] \oplus D[0] \oplus ((B[4] \oplus C[4] \oplus D[4])(B[0] \oplus C[0] \oplus D[0])) \\
B'[3] &\leftarrow C[3] \oplus A[0] \oplus (A[4](C[0] \oplus D[0]) \oplus A[0](C[4] \oplus D[4]) \oplus A[0]A[4]) \\
C'[3] &\leftarrow D[3] \oplus (A[4]B[0] \oplus A[0]B[4]) \\
D'[3] &\leftarrow A[3]
\end{aligned}$$

the other bits $i = 0, 1, 2, 4$ are calculated by:

$$\begin{aligned}
A'[i] \leftarrow & B[i] \oplus B[i{+}2] \oplus \\
& ((B[i{+}1] \oplus C[i{+}1] \oplus D[i{+}1])(B[i{+}2] \oplus C[i{+}2] \oplus D[i{+}2])) \\
B'[i] \leftarrow & C_i \oplus C[i{+}2] \oplus \\
& (A[i{+}1](C[i{+}2] \oplus D[i{+}2]) \oplus A[i{+}2](C[i{+}1] \oplus D[i{+}1]) \oplus A[i{+}1]A[i{+}2]) \\
C'[i] \leftarrow & D[i] \oplus D[i{+}2] \oplus (A[i{+}1]B[i{+}2] \oplus A[i{+}2]B[i{+}1]) \\
D'[i] \leftarrow & A[i] \oplus A[i{+}2]
\end{aligned}$$

Now, in our simple experiment, let us consider that bits 0 to 3 are calculated correctly and an attacker changes the value of one input share $A[0]$ always to 0 before the calculation of the 4 shares for output bit $i = 4$. Then an attacker is able to mount SIFA as indicated by the distributions of Figure 8.4. This leads again to an exploitable bias of the distribution of native variables at the output of the S-box and the attacker can mount SIFA.

The aim of this section was to give reproducible, easy-to-follow examples of inducing a bias in the native variable of masked S-boxes by just faulting one share of the S-box. We want to mention that the given ways and locations of introducing the faults are not exhaustive and that there are many more locations and various types of faults that make an attack successful. In the next section, we give a closer view on the problem of protecting an S-box against these attacks and get more insight into the effect allowing statistical attacks with the help of a 3-bit S-box as an example.

### 8.1.3 A Closer Look

In general, fault attacks exploit knowledge about intermediate variables of cryptographic primitives, which is gained by disturbing the computation or intermediate variables directly. In the case of DFA [BS97], this knowledge is that in certain intermediate bits or bytes a difference is induced, while others remain fault-free. In the case of SFA and SIFA, this knowledge is that the distribution of certain intermediate variables is changed from a uniform to a non-uniform distribution. This allows an attacker to guess parts of the round key and calculate backwards to these influenced intermediate variables from collected ciphertexts. If a key guess is wrong, an attacker expects to see a distribution of intermediate variables, which is closer to uniform compared to the guess of the right key.

Getting such a non-uniform distribution of intermediate variables can be achieved in many ways. For ciphers following the SPN structure, where every S-box and the linear layer is a bijective function (permutation), non-uniform distribution of intermediate variable can be achieved, for instance, by disturbing the computation of a single S-box, so that this S-box does not act as a permutation anymore. While such a behavior can be expected from an native S-box in the case of a fault induction, this seems quite counterintuitive for masked implementations at first glance. Thus, we will first discuss the native case to get more insight in which cases SFA and SIFA will work. Then we will take a closer look on masked S-box implementations.

### Influencing Native S-boxes

First, we will explore the case of SFA. We consider a bijective S-box, as illustrated in Figure 8.5. For the sake of simplicity, let us assume that the S-box is implemented in a bit-sliced manner (as a sequence of instructions), or as a Boolean circuit in hardware. Let us further assume that an attacker influences the correct computation via fault inductions. By faulting this computation, it is very likely that the faulted S-box does not behave as a bijection, but rather as a non-surjective function. This leads to a non-uniform distribution of intermediate variables, which can be exploited with SFA. In the case of SIFA, fault inductions that affect the S-box output are filtered out and hence not considered in the analysis. This filtering can happen directly by a detection-based countermeasure that does not release faulty outputs or by an attacker that repeats each query, once without fault induction, and comparing the obtained results.

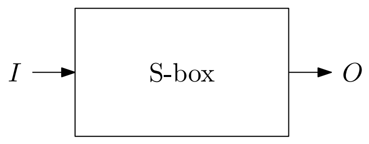
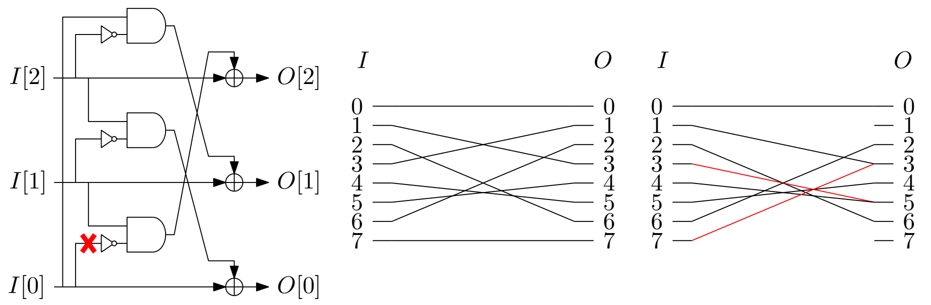


**Figure 8.5:** An native S-box.

To get more insight into the behavior of a faulted S-box, we will use the $3 \times 3$ S-box $\chi$ based on Daemen's $\chi$-layer [DGV94; Dae95] as an illustrative example. Figure 8.6a shows the 3-bit S-box $\chi$, where the red cross represents a fault that sets the input of the subsequent inversion to zero and hence the input of the

AND gate to 1. Please note that this is only one example of many how to fault an S-box to apply an SFA or SIFA.



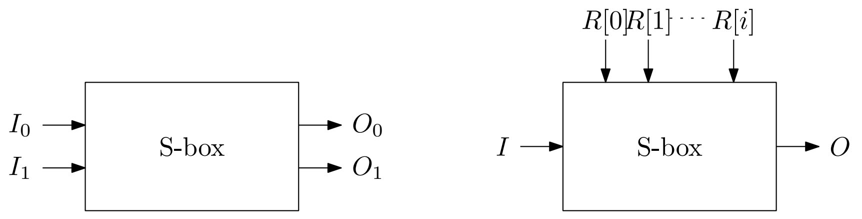
**(a)** Circuit with single fault **(b)** Unfaulted mapping **(c)** Faulted mapping

**Figure 8.6:** 3-bit S-box $\chi$ with a single fault.

The fault depicted in Figure 8.6a changes the behavior of the $\chi$ S-box, which is not bijective anymore. Figure 8.6c shows the transition graph for the faulted S-box. The fault just changes the transitions depicted in red.

In our case, the transitions mapping from $3 \to 1$ and $7 \to 7$ in the unfaulted S-box (Figure 8.6b), now map from $3 \to 5$ and $7 \to 3$ (considering $I[0]$ and $O[0]$ being the LSB). This means that the output values 1 and 7 never appear, but 3 and 5 appear twice, leading to a non-surjective behavior. In the case of SIFA, the red edges represent fault inductions that show an effect on the output. When applying SIFA, those edges are filtered out and just the black edges remain. Hence, when performing SIFA, the correct transitions $3 \to 1$ and $7 \to 7$ do not appear and an attacker can observe an exploitable non-uniform distribution of intermediates.

### Influencing Masked S-boxes



**(a)** Implementation with 2 shares **(b)** Functional equivalent view

**Figure 8.7:** A masked S-box.

Now let us have a look at a shared bijective S-box. For instance, consider the masked S-box shown in Figure 8.7a that takes two shares as input and returns two output shares. Here, the goal for an attacker is also to influence the distribution or transitions of the native variables $I = I_0 \oplus I_1$ and $O = O_0 \oplus O_1$, but does not care about the concrete values of the shares. For this reason, it is easier to work with the functional equivalent model shown in Figure 8.7b. In this model, we see a masked S-box as a function, which takes an native input $I$ and some randomness $R[i]$ and produces an native output $O$. Here, some of the random bits symbolize all values a shared input can take in a real implementation, e.g., $I_0 = I \oplus R[0]$, and $I_1 = R[0]$, while others represent randomness used in the masked implementation. Now, we can see masking as a very special and complicated function, which takes the inputs $I$, $R[0]$, $R[1]$,... $R[i]$ and produces an output $O$, so that the same $I$ always leads to the same $O$ for all possible choices of $R[0]$, $R[1]$,... $R[i]$. It seems very unlikely that a shared S-box behaves in the same manner in the presence of faults.

To apply SFA successfully, we need that not all values for $O$ (iterating over all values of $I$) appear the exact same number of times when counting for all possible assignments of $R[0]$, $R[1]$, and $R[2]$. This prerequisite is very likely to hold considering an attacker that can tamper with the intermediate calculation performed, even when restricting the attacker to just manipulate one share used in an intermediate calculation. Similarly, to apply SIFA successfully, we need a fault such that among the ineffectively faulted computations, not all values for $I$ or $O$ appear the exact same number of times over all values of $R[0]$, $R[1]$,... $R[i]$. This condition is similarly very likely to happen in practice when introducing just single faults, as we will show with our practical experiments in Section 8.2.

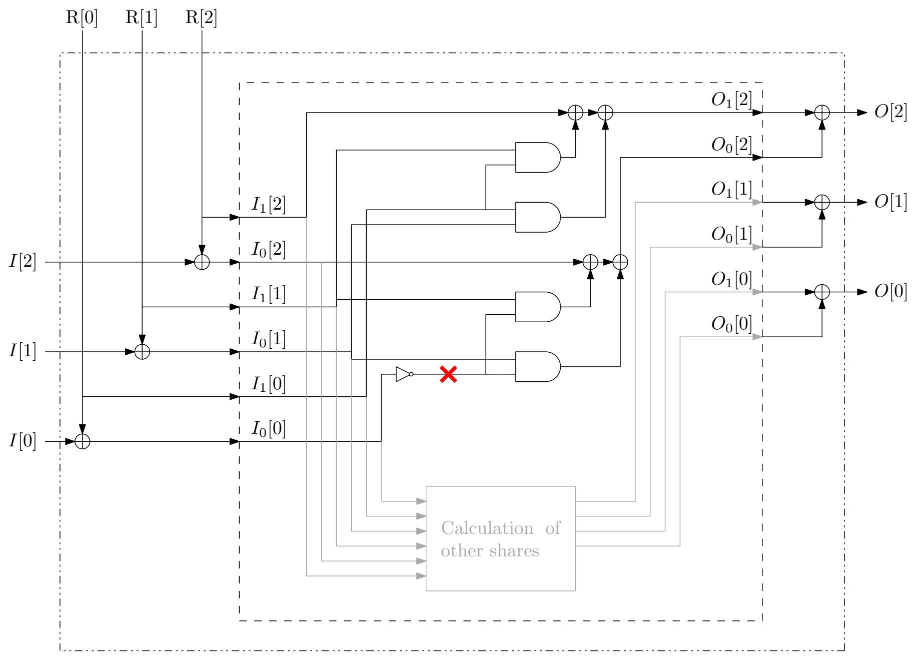


**Figure 8.8:** Single fault on masked 3-bit $\chi$ S-box.

As an example, consider again 3-bit $\chi$ S-box, now with the following masked implementation:

$$O_0[i] \leftarrow (I_0[i+1] \oplus 1)I_0[i+2] \oplus ((I_0[i+1] \oplus 1)I_1[i+2] \oplus I_0[i])$$
$$O_1[i] \leftarrow I_1[i+1]I_0[i+2] \oplus (I_1[i+1]I_1[i+2] \oplus I_1[i])$$

This masked S-box just serves us as an illustrative example of the effect of faults on an S-box, hence, we do not care about potential positioning of registers or additional randomness at the output for re-sharing. Figure 8.8 shows the equivalent circuit of the S-box, where again we just set a single value to 0. The result of this fault is that the value of $O_0[2]$ equals $I_0[2]$. Everything else is calculated correctly. For our example depicted in Figure 8.8, we list all possible assignments of $I[0]$, $I[1]$, $I[2]$, $R[0]$, $R[1]$, and $R[2]$ in Table 8.1. The entries marked in red in Table 8.1 are entries where the fault depicted in Figure 8.8 has an effect. Due to the more complex calculations that happen for masked S-boxes, we get a more complex relation between masks and actual values of bits. For instance, the transition $2 \rightarrow 6$ is only valid if $R[0] = 1$ and wrong ($2 \rightarrow 2$) if $R[0] = 0$.

**Table 8.1:** Transitions of faulted masked 3-bit S-box $\chi$.

| $I[2]$ | $R[2]$ | $I[1]$ | $R[1]$ | $I[0]$ | $R[0]$ | $O[2]$ | $O[1]$ | $O[0]$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

Again, we can represent all possible transitions from inputs $I$, to the outputs $O$ in a graph shown in Figure 8.9 (in a similar way as in Figure 8.6c). However, due to the $2^3$ possible ways of masking our input variables, each transition from input to output will happen 8 times for an *unfaulted* masked 3-bit S-box $\chi$. In the faulted case, this condition does not hold anymore as shown in Table 8.1. Hence, we have additional transitions shown in red in Figure 8.9. These "wrong" transitions also reduce the number of times the "correct" transition happens.

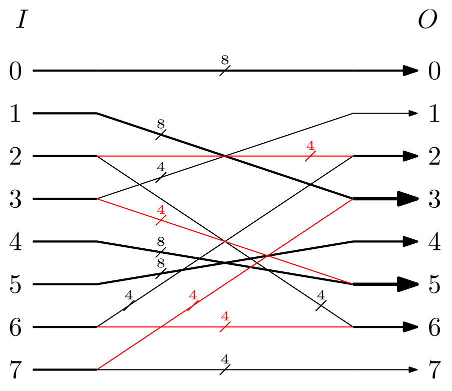


**Figure 8.9:** Transition graph of faulted masked 3-bit S-box $\chi$.

If we now count the number of transitions in Figure 8.9 that lead to a certain value $O$, we see that 12 transitions lead to values 3 and 5, whereas only 4 lead to 1 and 7. This means that an attacker faulting a device can apply an SFA, since the attacker can expect a non-uniform distribution of the variable after the S-box for correct key guesses. If we apply SIFA, the transitions marked in red in Figure 8.9, will be filtered. As an effect, the transitions $2 \rightarrow 6$, $3 \rightarrow 1$, $5 \rightarrow 2$, and $7 \rightarrow 7$ appear with reduced frequency for uniformly distributed $R[0]$, $R[1]$, and $R[2]$. Again, this can be exploited in a key recovery attack.

## 8.2 Attack Evaluation

In this section, we demonstrate the applicability of statistically ineffective fault attacks (SIFA) for two very different publicly available masked AES implementations. First, we perform a practical attack evaluation for the provable secure, higher-order masked AES implementation from Rivain et al. [RP10] on a standard 8-bit microcontroller (ATXmega 128D4). We then present a comprehensive evaluation of simulated faults for the 32-bit, bitsliced, first-order masked AES implementation of Schwabe and Stoffelen [SS16]. Since both implementations do not originally have additional fault countermeasures in place, we added temporal redundancy, meaning that the block cipher is executed multiple times and the ciphertexts are compared. The number of redundant computations was set to two, since more redundancy does not affect the effectiveness of SIFA.

Our experiments require multiple faulted encryptions, but only one fault induction per encryption. The precise location as well as the actual effect of the induced fault does not need to be known by the attacker. Indeed, inducing a fault anywhere in the shared S-box in round 9 likely leads to a situation that is similar as described in Section 8.1.3. The resulting joint non-uniform (biased) distribution over all shares of an intermediate variable can then be used to distinguish correct and wrong key candidates (cf. Section 7.1).

### 8.2.1 Practical Attack on AES from Rivain et al.

The higher-order masked AES from Rivain et al. [RP10] consists of a generic $d$th-order masked S-box that is combined with a linear layer for the $d+1$ shares of the AES state. The target of our fault induction is the shared S-box implementation in round 9. First, we briefly describe the implementation of the masked S-box. Then we present the attack setup that was used for fault induction. The results of this practical evaluation are stated at the end of this section.

#### Generic Higher-Order Masked S-box

The algebraic description of the 8-bit AES S-box consists of determining the multiplicative inverse of a number in $\mathbb{F}_{2^8} = \mathbb{F}_2[x]/(x^8+x^4+x^3+x+1)$, followed by an affine transformation over $\mathbb{F}_2^8$. While masking the affine transformation is trivial, since it can be calculated separately for each share, the calculation of the masked multiplicative inverse requires more work. In the design of Rivain et al. the inversion is calculated via the power function $x \to x^{254}$ over $\mathbb{F}_{2^8}$ which is in return calculated via the square-and-multiply algorithm. The squaring operation in $\mathbb{F}_{2^8}$ is a linear function which leaves the masking of the field multiplication as the only non-trivial task. The used algorithm for $d$th-order masked field multiplication is based on the ISW scheme (cf. Algorithm 2.1). For a more detailed description we refer to the original paper [RP10].

#### Attack Setup

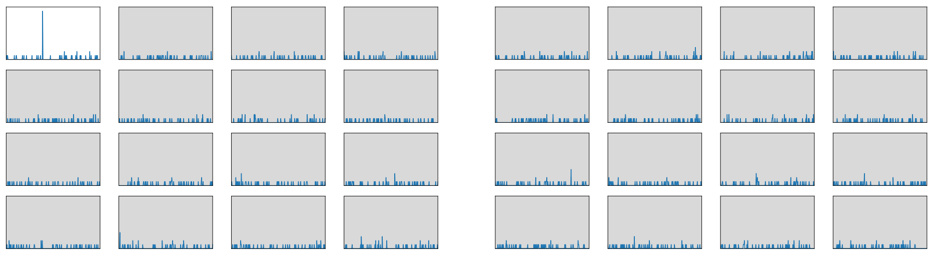
In the practical evaluation, we perform fault inductions on an ATXmega 128D4 via clock glitching. More precisely, we insert an additional fast clock cycle in between two ordinary clock cycles during the execution of one of the redundant encryptions. The width of the induced clock cycle is chosen such that it is recognized by the microprocessor but too short to allow a correct execution of the current instruction. The target of our fault is one of the higher-order masked field multiplication operations (`SecMult`) that occur multiple times (1st-order: 64 times, 10th-order: 2880 times) during the computation of the masked S-box in round 9. We neither require to fault one specific `SecMult` invocation nor to fault one specific instruction within any of the `SecMult` invocations. In fact, any fault that causes a joint non-uniform distribution over all shares (cf. Section 8.1.2) is sufficient for our attack. For this reason, finding a suitable fault location might actually be easier for higher-order masked implementations since the runtime of masked S-boxes grows faster than the runtime of the masked linear layer for increasing masking orders.

The implementation of the higher-order masked AES by Coron et al. [Cor17] has a configurable masking order $d$. Our experiments were performed up to 10th-order masking, with temporal redundancy as an additional fault countermeasure. In order to attack such a protected implementation we exploit statistically ineffective faults. The restriction to ineffective faults is required, since we want to circumvent the redundancy countermeasure and the subsequent statistical fault analysis is required, since the effect of a fault is almost impossible to predict, in particular if masked and/or unknown implementations are attacked.

If the induced fault results in a faulty computation we do not observe any ciphertext, because of the redundancy countermeasure. However, we do observe correct ciphertexts stemming from faulted encryptions where the induced fault was ineffective. This filtered set of correct ciphertexts can then again be used to perform key recovery. Since our attacks work comparably well for any masking order $d$ we only state concrete results for $d = 10$ in this section.

### Results

In Figure 8.10, we show the distribution of AES state bytes in round 9 from the 48 collected correct ciphertexts. For a correct guess of round key 10, which has to be done in 32-bit chunks in an actual attack, we can observe a strong biased distribution in one of the state bytes in round 9 which enables key recovery. Note that in the attack we only exploit correct ciphertexts and thus successfully circumvent the redundancy countermeasure. Already 20 (of the 48) collected correct ciphertexts stemming from about 1 000 faulted encryptions the SEI of the observed distributions is the highest for the correct partial key guess. The time required for collecting the correct ciphertexts was about 3 minutes, and key recovery was performed in about 2 minutes with negligible memory requirements. Repetitions of the experiment lead to very similar results.



**(a)** Correct key guess. **(b)** An incorrect key guess.

**Figure 8.10: 10th-order masked AES with temporal redundancy.** One clock glitch was performed during the calculation of the masked Sbox in round 9. Distribution of AES state bytes after S-box in round 9 after collecting a sufficient amount of correct ciphertexts. 32 key bits can be recovered.

### 8.2.2 Simulated Attacks on AES Implementations from Schwabe and Stoffelen

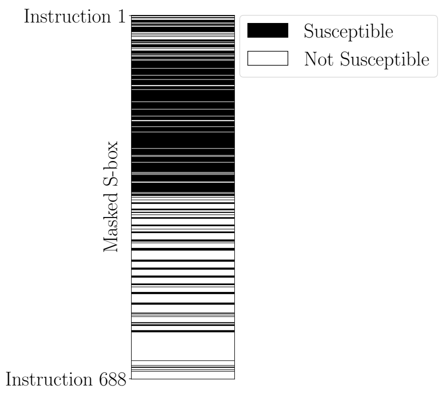
In this section, we present a comprehensive analysis of simulated faults for an assembler-optimized, masked AES implementation for the 32-bit STM32 F3 platform with high practical relevance of Schwabe and Stoffelen [SS16]. This implementation can encrypt two 128-bit inputs per block cipher call in CTR mode and is fully unrolled. For our purposes, we add temporal redundancy but only encrypt one 128-bit input per block cipher call in ECB mode to have the same scenario as in the previous section. We briefly describe the masked AES S-box implementation of Schwabe and Stoffelen and then we discuss the results of our evaluation that is split into three parts:

First, we analyze how many instructions in the masked S-box in round 9 are "*susceptible*" to faults considering two common fault models, i.e., would allow an attacker to mount SIFA. Then we pick one susceptible instruction and discuss the required effort of mounting SIFA considering 7 common fault models. For each fault model we give the required amount of faulted encryptions and the amount of recoverable key bits. Finally, we present a figure that illustrates the effect of faulting the masked Sbox on the native AES state bytes (that are observed during key recovery).

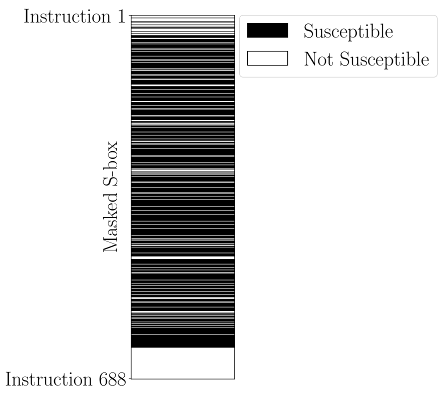
**Bitsliced 1st-Order Masked S-box.** One shared S-box computation consists of 688 instructions and is executed in parallel for the entire AES state. The implementation is based on the efficient bitsliced, S-box implementation from Boyar et al. [BP12]. Masking via Trichina gates [Tri03] as well as efficient platform-specific scheduling was added by Schwabe and Stoffelen [SS16].

**Susceptible Instructions.** In this section, we demonstrate that SIFA is neither restricted to a specific fault model nor requiring precise information of the attacked implementation. We have performed experiments using two common fault models (single bitflip and byte-stuck-at-0) and simulated fault inductions that cause an erroneous value in the result of the targeted instruction of an S-box computation in round 9. This simulation is performed separately for each of the 688 instructions in the masked S-box. For each instruction and both types of simulated faults we performed 2 000 encryptions and collected correct ciphertexts from unaffected encryptions (if there were any). Then we performed key recovery, i.e., for each targeted instruction, both types of simulated faults, and the corresponding set of collected ciphertexts we guessed 32 bits of the last round key, calculated back to round 9 and checked if some bytes follow a non-uniform distribution (using the SEI distinguisher). To reduce the runtime of the evaluation we took one shortcut by always fixing 16-bit of the 32-bit partial key guess to the correct value. While this significantly reduced the runtime of our exhaustive analysis, it does not affect the results.

The results of our analysis are presented in Figure 8.11. Figure 8.11a shows for each of the 688 instructions within the masked S-box whether or not single bitflips in this specific instruction allow an attacker to mount SIFA, i.e., recover bits of the key. Note that black lines represent susceptible instructions while white lines represent the other instructions. An instruction is not susceptible, e.g., if a bitflip is never ineffective, always ineffective, or does not lead to a non-uniform distribution that is distinguishable from uniform after observing 2 000 faulted encryptions. In total, 359 out of 688 (52%) of the instructions are susceptible to single bitflips. In Figure 8.11b, we show the same analysis using byte-stuck-at-0 faults instead. Here, 483 (70%) of the instructions are susceptible. If we oppose these results with the fact that the masked linear layer in round 9 only consists of 290 instructions, it is fairly safe to say that finding a suitable fault location should be easy in practice.



**(a)** Bitflips: 359 (52%) of the S-box instructions are susceptible.



**(b)** Byte-stuck-at-0: 483 (70%) of the S-box instructions are susceptible.

**Figure 8.11:** Instructions in the masked S-box in round 9 that are susceptible to faults and allow to mount SIFA.

**Attack Performance for Various Fault Models.** After determining that large parts of the masked S-box implementation are susceptible to single fault inductions in the previous section, we now discuss the effort of key recovery when targeting one of these susceptible instructions. This time we consider 7 different fault models and the results are presented in Table 8.2. For each of the 7 fault models we give the number of faulted encryptions, the number of resulting ineffective fault inductions (i.e., correct ciphertexts), and the number of key bits that can be recovered from those correct ciphertexts. Each experiment was repeated 3 to 5 times, and the averaged values are presented. The relative position of the affected bytes/bits within the targeted 32-bit register is not important for the analysis.

From the results we can see that faults with fine granularity (only affecting single bits) allow an attacker to recover key bits faster (= using less faulted encryptions). However, only 32 key bits can be recovered in these scenarios since only a small portion of the AES state is affected by those fault inductions.

On the other side, faults that affect whole bytes/registers require more faulted encryptions but cause a non-uniformity in bigger portions of the AES state. Consequently, an attacker can recover more key bits.

**Table 8.2:** 32-bit STM32 F3: Attack evaluation when targeting one of the instructions in the masked S-box in round 9. Each experiment was repeated 3 to 5 times, the resulting numbers were averaged.

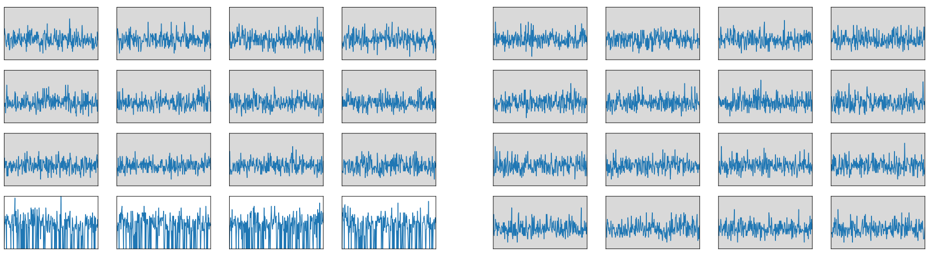
| **Fault Effect** | **# Ineffective Faults** | **# Faulted Encryptions** | **# Recoverable Key Bits** |
|---|---|---|---|
| Flip one bit | 194 | 386 | 32 |
| Set one bit to zero | 214 | 428 | 32 |
| Randomize one bit | 574 | 763 | 32 |
| Flip one byte | 192 | 2 940 | 128 |
| Set one byte to zero | 192 | 3 129 | 128 |
| Randomize one byte | 602 | 1 808 | 128 |
| Instruction skip | 400 | 45 527 | 128 |

**Non-Uniformity of AES State Bytes.** Finally, we present more arguments why the statistical nature of SIFA is crucial to mount the manifold attacks presented in this work. If we take a look at Figure 8.12, we can see the distribution of AES state bytes in round 9 using one byte-stuck-at-0 fault per encryption. For this result we assumed a correct guess of round key 10, in an actual attack, the key recovery has to be done by guessing 32-bit chunks of round key 10 at a time. Even though our simulated faults are noise-free, i.e., they have the same effect on each faulted instruction, there is no easy way (other than having precise knowledge of the attacked implementation and the induced fault) for the attacker to predict the resulting non-uniformity in the state bytes which would allow for faster key recovery than in SIFA (or SFA). In fact, the observed distributions have some relations with the pen-and-paper examples given in Section Section 8.1.2 and can be expected to vary significantly depending on the attacked implementation, the fault location, and the actual fault effect. This motivates our choice of simply using a metric of non-uniformity to distinguish the key candidates.

## 8.3 Discussion

### 8.3.1 On the Nature and Number of Faults

In Section 8.1, we explored the behavior of masked building blocks using deterministic stuck-at-0 faults or instruction skips. The reason for this is to make the processes leading to a bias easier to understand. However, it is easy to see that making the fault probabilistic, e.g., assuming a more realistic setup, where an instruction skip does not work all the time, or that a bit is only set to 0 with a certain probability, just affects the bias an attacker observes and hence,



**(a)** Correct key guess. **(b)** An incorrect key guess.

**Figure 8.12: 1st-order masked, bitsliced AES with temporal redundancy.** Byte-stuck-at-0 fault model. Distribution of AES state bytes after S-box in round 9 after collecting a sufficient amount of correct ciphertexts. The whole 128-bit key can be recovered.

the amount of ciphertext the attacker has to collect, but the attack still works (see Chapter 6 for more details).

Furthermore, clock glitches and setting values to zero are not an exhaustive list of effects that a fault could have in order to make the attack work. For instance, in Section 8.2.2 we show that attacks are possible even for random faults and bitflips. All in all, the only requirement we have on the fault is that it leads to a biased distribution of the native variable at a suitable place in the primitive among the filtered encryptions. In general, a fault can have a more complex nature than only the cases discussed in Section 8.1 or Section 8.2.2. For instance, in software implementations of masked ciphers, a large number of instructions are LOAD instructions from memory since all shares might not fit in the registers. We have observed in experiments that skipping a LOAD can also lead to biased S-box distributions, but having a quite complex effect depending on previous calculations. However, the big benefit of SIFA is the fact that an attacker does not have to know or model the effect of a fault.

This fact also comes into play when dealing with the location of a fault. The examples for the location of the fault given in Section 8.1 just show one out of many different locations, where a fault targeting the S-box leads to a biased distribution that can be exploited in SFA or SIFA. In Section 8.2.2, we evaluated the number of instructions that can be faulted and in turn, can be exploited in an attack for one particular implementation. However, how many such locations exist crucially depends on how the S-box is implemented. In a similar manner as for the effect of a fault, an attacker does not have to know, or to aim for just one specific instruction or location to fault. The only requirement for the attack to work in practice is that the faulted location leads to a bias. All these points make the attack to be executed in practice quite easily, even with a rather cheap setup using clock glitches as demonstrated in Section 8.2.

The last point, we want to discuss regarding faults is the number of faults per execution. We have opted for a single fault per execution, since inserting multiple faults per execution is usually considered to be harder. This is probably related to the prominence of fault attack techniques where attacks requiring multiple faults per execution have high requirements of the exact location and effects of the induced faults. However, at least in the case of SFA, we are not interested in the number of faults, since there are no strong requirements regarding effect and location. In fact, injecting multiple consecutive faults usually just leads to a lower number of necessary ciphertexts as the bias increases. For SIFA, the situation is slightly different; however, multiple faults injected in the computation of a single S-box might to reduce the required number of faulted encryptions, while being not necessarily harder to conduct in practice.

### 8.3.2 Countermeasures

In the following, we discuss the effectivity and practicability of well-known countermeasures against our attacks. Since most fault countermeasures prevent the SFA variants of our attack using a single fault per execution, we focus on the SIFA variants and show that it is not easy to prevent these attacks. In fact, countermeasures such as detection even facilitate certain aspects of SIFA in practice.

**Self-Destruction.** The most radical approach of destroying the device as soon as a fault is detected is a valid countermeasure against any fault attack. However, this technique has a few downsides and limitations, including false positives and additional effort to reliably destroy a circuit.

A lot of cryptographic devices deployed in the field like smart cards and RFID tags have to function and operate under rather tough conditions. They typically have to deal with abrupt loss of power, for instance if a smart card is withdrawn from the terminal while working. Furthermore, they have to handle power spikes from electrostatic discharges or electromagnetic fields. Hence, deciding between an active fault attack and interference due to normal usage is not a trivial task and would potentially lead to detection of a huge amount of false positives that render such an approach useless for a wide range of applications.

One way to compensate some false positives is to destruct a device only once a certain amount of faults was detected. Such a fault counter could be considered an effective countermeasure, yet is still not used by a large portion of embedded devices, since it is challenging to implement appropriately tamper-resistant, especially in hardware.

**Correction.** A different approach to make use of redundancy is to correct the effect of a fault, for instance using error-correcting codes or simple majority voting. However, correction-based countermeasures usually can be reduced to the detection-based case using additional faults. How hard this is and which requirements this might have on the precision of the fault crucially depends on the implementation of the countermeasure. As an example let us assume a simple majority voting between the result of 3 block cipher calls. To do this, the 3 block cipher calls take the same inputs and hence, perform redundant computations.

An attacker can now use an additional fault to just ensure that the computations performed on one redundant block cipher call will always be incorrect. This usually does not require a precise fault. This reduces the majority voting of 3 block cipher calls to a construction which essentially behaves as a detection-based countermeasure (or infection-like countermeasure if the majority voting happens at bit-level of the ciphertext). Then, an attacker can proceed with the same attacks as before, using a second targeted fault.

**Infection.** In Chapter 6, the application of SIFA on an infective countermeasure [TBM14] has been demonstrated. The employed dummy rounds in this countermeasure increase the needed number of faulted encryptions until the key can be recovered. However, when aiming to prevent SIFA, dummy rounds that do not infect the state in the case of a fault should provide even more protection. Hence, we explore this countermeasure next.

**Hiding.** The goal of hiding countermeasures is to reduce the attacker's knowledge of what is currently computed, and thus effectively decrease his precision when placing the fault. Examples include adding dummy rounds randomly between the relevant rounds, or shuffling the order of execution, for example the order in which the 16 AES S-boxes per round are executed. In the following, we analyze the case of dummy AES rounds in more detail, and show that the noise introduced this way quadratically increases the necessary number of faulty encryptions for the analysis. We consider a protected AES implementation and make the following assumptions for our model:

- The attacker needs to fault round 9 out of 10 (identical) AES rounds.
- The protected implementation executes 10 real AES rounds and $(k-1)\cdot 10$ ineffective dummy rounds in a uniformly random ordering, labeled $1,\dots,R$ with $R = 10k$.
- The attacker targets round $R-t$. Three outcomes are possible:
    1. Hit: It is the real round 9 with probability $\sigma$, resulting in a distribution with ineffectivity rate $\pi_{\mathsf{fault}}$ and ineffective distribution $p_{\mathsf{fault}}$.
    2. Miss: It is a dummy round with ineffectivity rate $\pi_{\mathsf{dummy}} = 1$ and uniform ineffective distribution $\theta$
    3. Miss: It is a real round, but not round 9, with ineffectivity rate $\pi_{\mathsf{fault}}$ and uniform ineffective distribution $\theta$. For simplicity, we assume an ineffectivity rate of $\pi_{\mathsf{dummy}} = 1$, so this case can be merged with Item 2.

With these assumptions, the success probability that round $R-t$ of $R$ is a hit (signal) is

$$\sigma_{\mathsf{fault}}[R,t] = \mathbb{P}\big[\text{Hit in round } R-t\big] = \frac{t\cdot\binom{R-t-1}{8}}{\binom{R}{10}} = \frac{90\cdot t}{R(R-1)}\prod_{s=2}^{9}\left[1-\frac{t-1}{R-s}\right].$$

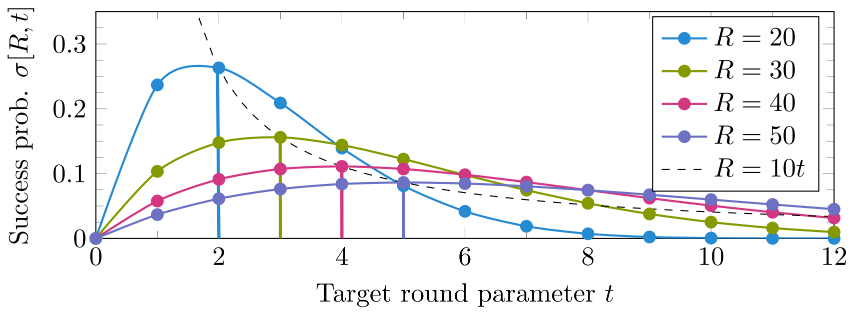


**Figure 8.13:** Success probability $\sigma[R,t]$ when targeting round $R-t$, for different $R = 10k$. For the choice $t = k$ (dashed line), $\sigma[10k,k] \approx \frac{0.387}{k}$ for $k \gg 2$.

This parametrized function is plotted in Figure 8.13 and attains its maximum near $t = k = \frac{R}{10}$. The resulting function $\sigma_{\mathsf{fault}}[10k,k]$ for the optimized success probability is also plotted in Figure 8.13 (dashed, with $x$-axis $t = k$), and can be approximated as

$$\sigma_{\mathsf{fault}}[10k,k] = \mathbb{P}\big[\text{Hit in round } 9k \text{ of } 10k\big] = \frac{90k}{10k\cdot(10k-1)}\cdot\prod_{s=2}^{9}\left[1-\frac{k-1}{10k-s}\right]$$
$$= \frac{1}{k}\cdot\prod_{s=1}^{9}\frac{9k-s+1}{10k-s} \xrightarrow{k\to\infty} \frac{1}{k}\cdot\left(\frac{9}{10}\right)^{9} \approx \frac{1}{k}\cdot 0.387 \qquad \text{for large } k\,.$$

A SIFA attacker samples the resulting distribution $p_{\mathsf{total}}$ among the ineffective faults, with a total ineffectivity rate of $\pi_{\mathsf{total}}$ and a signal of $\sigma_{\mathsf{total}}$:

$$\begin{aligned}\pi_{\mathsf{total}} &= \sigma_{\mathsf{fault}}\cdot\pi_{\mathsf{fault}} + (1-\sigma_{\mathsf{fault}})\cdot 1 = 1-(1-\pi_{\mathsf{fault}})\cdot\sigma_{\mathsf{fault}}\\ \sigma_{\mathsf{total}} &= \frac{\sigma_{\mathsf{fault}}\cdot\pi_{\mathsf{fault}}}{\pi_{\mathsf{total}}}\\ p_{\mathsf{total}}(x) &= \sigma_{\mathsf{total}}\cdot p_{\mathsf{fault}}(x) + (1-\sigma_{\mathsf{total}})\cdot\theta(x)\end{aligned}$$

The necessary sample size to distinguish $p_{\mathsf{total}}$ is inverse proportional to the capacity

$$C(p_{\mathsf{total}}) = \sigma_{\mathsf{total}}^2\cdot C(p_{\mathsf{fault}})\,,$$

which corresponds to a data complexity proportional to $(\pi_{\mathsf{total}}\cdot\sigma_{\mathsf{total}}^2\cdot C(p_{\mathsf{fault}}))^{-1}$. Thus, for a fixed fault setup with $p_{\mathsf{fault}}, \pi_{\mathsf{fault}}$, increasing the dummy factor $k$ increases the data complexity of the attack quadratically (Figure 8.14):

$$(\pi_{\mathsf{total}}\cdot\sigma_{\mathsf{total}}^2)^{-1} = \frac{1-(1-\pi_{\mathsf{fault}})\cdot\sigma_{\mathsf{fault}}}{\sigma_{\mathsf{fault}}^2\cdot\pi_{\mathsf{fault}}^2} \approx k^2\frac{1}{(0.387\cdot\pi_{\mathsf{fault}})^2} - k\,\frac{1-\pi_{\mathsf{fault}}}{0.387\cdot\pi_{\mathsf{fault}}^2}\,.$$

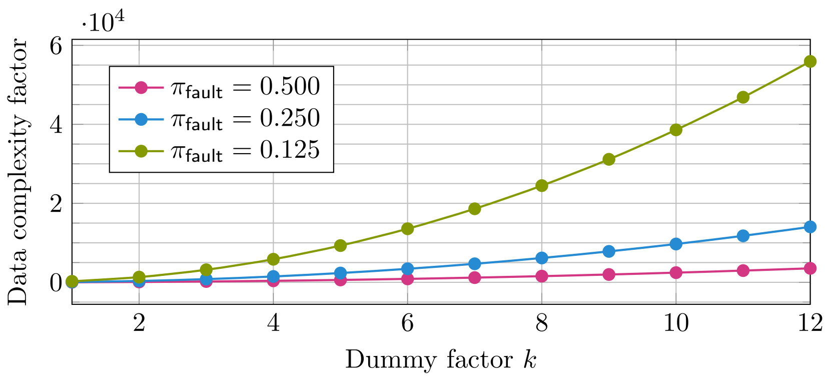


**Figure 8.14:** Increasing data complexity with dummy factor $k$, for different $\pi_{\mathsf{fault}}$.

**Shuffling.** Similar to dummy rounds, shuffling operations reduces the attacker's precision and success probability in hitting the right S-box and thus induces noise in the distribution. However, in the case of SIFA, there is an important difference due to the ineffectivity rate in case of misses, which we assume is the same as in case of hits. For this reason, the data complexity will also grow quadratically in the number of shuffled operations in the relevant scope (e.g., 16 S-boxes), but only linearly instead of quadratically in the inverse ineffectivity rate $\pi_{\mathsf{fault}}^{-1}$.

**Limiting the Data Complexity.** For our attacks to work, we usually need several faulted encryptions per key to retrieve it. Therefore, methods that restrict the usage of the key and hence, put a limit on the data complexity can be a viable strategy for providing protection against this type of attack. Existing re-keying strategies can be roughly split into two groups, one where the used key is derived via a re-keying function from a static master key [Med+10; Dob+17; Dob+15; Ber+17a] and the other group being methods where a secret internal state is maintained and constantly updated. In the first group, the problem of protection against the attack is basically shifted to the re-keying function and has to be solved there.

### 8.3.3 Choice of the Target and Attack Setup

It is important to note that we have not chosen the S-boxes in Section 8.1 or AES in Section 8.2 as targets of our attacks because we have found them to be weaker than others. In fact, we chose them, because many masked implementations for them are publicly available.

Furthermore, we performed the practical experiments using clock glitches because the equipment is cheap and we do not have easy access to other, more sophisticated equipment at the moment. Obviously, the attack is not limited to a specific fault induction method; quite on the contrary, we expect other methods of inserting faults, such as lasers, needle probes, etc., to be far superior compared to our cheap setup using clock glitches (cf. Chapter 7).

### 8.3.4 Further Applications

For the sake of simplicity, we have put the main focus of this chapter on the application of statistical fault attacks for (masked) design strategies using bijective S-boxes, where we want to distinguish a uniform from a non-uniform distribution. However, this does not mean that the attack in only applicable on primitives using bijective S-boxes. As discussed in Section 8.1.3, a fault attack may influence only some transitions in the transition graph, while leaving others intact. In the case of SIFA, an attacker can observe and exploit the “filtered” graph, where most likely only the intact transitions remain. Note that in the masked case, there is more than one transition from one input to one output value, due to masks. Hence, an attacker can potentially exploit all cases where this “filtered” transition graph shows a differently distributed occurrence of input and output transitions.

As another narrative restriction, we have restricted our focus on block ciphers. One potential countermeasure one could come up with against our attack is the use of a PRF like the AES-PRF [MN17] instead of a block cipher. Such a PRF prevents an attacker from observing ciphertexts and decrypting backwards under guessing the key. However, SIFA remains possible by targeting the input of the AES-PRF since here, a known input like a nonce is usually processed. In general, the presented attacks are almost always applicable whenever some known input is mixed with a secret, which covers most stateless symmetric cryptographic primitives. However, it is an interesting future research topic to evaluate how well such attacks will work.

## 8.4 Conclusion

In this chapter we have demonstrated that SIFA is a very powerful attack. We show that state-of-the-art countermeasures against implementation attacks, redundancy against faults and masking against side-channels, are not as effective against SIFA as expected. In particular, SIFA is still possible using just a single fault per execution, contradicting the common folklore that masking plus a detection-based countermeasure provides sufficient protection against fault attacks.

We have presented a comprehensive analysis of simulated faults for an assembler-optimized, masked AES implementation for the 32-bit STM32 F3 platform that might be of high practical relevance. We show that most of the instructions of the masked S-box implementation are “susceptible” to faults and can be exploited in SIFA using any of the common fault models. Moreover, the practical feasibility of the attack was shown by attacking a $10^{\text{th}}$-order masked AES software implementation with arbitrary temporal redundancy on block cipher level on a standard 8-bit microcontroller without specific security features using a cheap clock glitch setup. Even with such a cheap setup, we are able to recover 32 bits of the key after collecting 20 ciphertexts where the fault is ineffective, needing approximately a total of 1000 encryptions where a single fault induction is performed.

# 9 Protecting against Statistical Ineffective Fault Attacks

Masking and redundant computation are amongst the most popular algorithmic countermeasures for cryptographic implementations against active and passive attacks respectively. In masking one splits input and intermediate variables of cryptographic computations into $d+1$ random shares such that the observation of up to $d$ shares does not reveal any information about their corresponding native value. Redundant computation, on the other hand, is used to detect malicious or environmental influences that could lead to faulty cipher outputs. Up until recently, implementations combining masking with some kind of redundancy were typically assumed to offer protection against both power analysis and fault attacks. However, in Chapter 8, we have shown that statistical ineffective fault attacks (SIFA) are applicable to ciphers protected with masking as well as fault detection or infection. SIFA succeeds in doing this by exploiting the dependence of faults propagating to the cipher output on the value of intermediate variables. The mere presence of a fault and the non-faulty outputs of the cipher computations provide sufficient information to retrieve the value of these intermediate variables. Consequently, the exploration of countermeasures against SIFA that do not rely on protocols or physical protection mechanisms is of great interest. In this chapter, we explore different countermeasure strategies against SIFA.

**Our Contribution.** The contributions presented in this chapter are two-fold. First, we analyze the general root causes that lead to successful SIFA attacks in more detail. To study these causes and also describe resistance against SIFA and Differential Power Analysis (DPA) [KJJ99], we introduce an abstraction layer between the algorithmic specification of a cipher and its implementation in hardware or software. In this layer, we express the cipher as a *circuit*, taking as input an array of variables and returning as output an array of variables. We split this circuit into sub-circuits where each sub-circuit takes as input variables that are either the cipher's input or the other sub-circuit's outputs. This is done recursively until we get to the level of circuits that we no longer split into sub-circuits and that we call basic circuits. In the implementation of these basic circuits it is then essential to *keep the computations and internal variables of the different basic circuits* separated. Thus, basic circuits are the natural place to define the concept of faults and their effectiveness in an unambiguous way, and it is also insightful to describe DPA resistance similar to the probing model [ISW03].

Second, we present two different approaches that, starting from the algorithmic description, allow to specify circuits that mitigate SIFA and DPA. The first approach relies on finding descriptions of ciphers that restrict basic circuits to permutations which only operate on an incomplete set of shares. In particular, those permutations are either linear or variants of the Toffoli gate [Tof80], the simplest invertible nonlinear function. This strategy allows for fault detection at the end of a cipher, e.g., by means of redundant computation and comparison of the outputs. We then show that masked 3-bit, 4-bit, and 5-bit S-boxes can be built using Toffoli gates as their only nonlinear component, thus offering protection against SIFA. A similar strategy can be applied to the AES S-box; however, for AES, we require fault detection at the output of the AES S-box. We verified the correctness of the masking of our Toffoli-based circuits for S-boxes of Keccak and AES using `maskVerif`, a tool for formal verification of masking schemes [Bar+15].

The second approach does not restrict basic circuits to be permutations and can thus be applied more broadly to add SIFA-protection to arbitrary circuits. It works by adapting the error-detection circuit for more fine-grained detection and can be generalized to cover multi-fault SIFA, albeit at a higher implementation cost.

**Related Work.** Potential mitigations against SIFA are already discussed in Chapter 8. One strategy considered there is to move from error detection to error correction, such as a majority voting. This has the effect that more than one fault is needed, since a single fault can always be corrected. This rough concept is developed further and brought into practice by several papers [Bre+20; Sah+19; SRM20].

The paper of Breier et al. [Bre+20] proposes a countermeasure based on error-correcting codes that can be implemented with error-correcting hardware gates. Shahmirzadi et al. [SRM20] investigate how error-correcting codes can be correctly embedded in a cryptographic implementation and show that their construction provides practical advantages over simple majority voting. Contrary to the two papers mentioned before, the Transform-and-Encode framework by Saha et al. [Sah+19] proposes an instantiation that combines masking with majority voting as an error-correcting code. Their strategy has the benefit that the error correction is only applied at the output of the S-boxes.

In contrast to the above-mentioned countermeasures using error correction, our strategy only requires error detection. We propose to either use masking schemes based on the Toffoli gate or transform implementations of arbitrary masking schemes. By doing this in combination with suitable error detection, we ensure that the effect of a fault never depends on a native (unmasked) value. In most cases, the detection of the error can be done once on the primitive level (e.g., after one block cipher execution, or execution of a cryptographic permutation). Hence, it even can be implemented in a time-redundant manner, or as an encrypt-decrypt strategy. Therefore, we get an overhead of roughly two, due to the additional redundancy in either space or time, compared to implementation featuring only protection against passive side-channel attacks. In contrast, the computational overhead of majority voting is at least three, with more overhead for more sophisticated error correction strategies, as shown by Shahmirzadi et al. [SRM20].

Ramezanpour et al. [RAD20] proposed a different way of achieving independence of a fault effect from the native value. As an example, they give an implementation based on a threshold implementation [NRR06; NRS11] of Canright's AES S-box [Can05], where they introduce additional shares and additional computations. This results in an overhead of a factor of 2 compared to a standard threshold implementation in their FPGA implementation [RAD20]. Moreover, if protection against differential fault attacks is also needed, then this cost will increase further. In contrast to this, our scheme also works with masking based on two shares and has a low overhead, allowing much more efficient implementations in practice.

Another strategy that may protect against fault attacks, including SIFA, are approaches based on actively secure multi-party computation protocols like CAPA [Rep+18], as long as a fault does not affect all parties. However, such methods are quite expensive. As pointed out by the authors of CAPA [Rep+18], their implementations are to be considered as a proof-of-concept and are too costly to be used in practice. For example, an implementation of the AES S-box using three shares and one MAC key requires 156 bytes of randomness per S-box evaluation.

**Outline.** We start with a description of our abstraction layer and discuss how we model fault and side-channel attacks on it in Section 9.1. Then, we show how to describe some ciphers only relying on incomplete permutations as basic circuits in Section 9.2. Afterwards, we describe a circuit for the AES that can withstand single-fault SIFA and first-order DPA in Section 9.3. Section 9.4 deals with the protection of arbitrary circuits. In Section 9.5, we verify the correctness of the circuits for the S-boxes of AES and KECCAK using `maskVerif` and discuss considerations when implementing the circuits in software or hardware.

## 9.1 The Circuit Abstraction Layer and Fault Model

In this section, and based on our description of SIFA (cf. Section 6.2.2) and masking (cf. Section 2.2.1), we define an abstract circuit model and the corresponding fault model that we use as a tool for building hardware or software cipher implementations that offer resistance against SIFA and side-channel attacks. One major purpose of the circuit abstraction layer is that it allows us to define our fault model in a clean and tidy way. In the following sections, we will make use of the term *cipher*. By the term *cipher*, we mean block ciphers, tweakable block ciphers, or cryptographic permutations.

### 9.1.1 Definition of the Circuit Abstraction Layer

We propose an abstraction layer between the algorithmic description of a (masked) cipher or permutation and the hardware or software implementation and present our formalism here. At the circuit abstraction level, we break up the deterministic algorithmic description into a number of interconnected *circuits*.

By a circuit, we mean a fully specified deterministic function taking as input an array of input variables and returning as output an array of output variables. Trivially, the algorithm of a (masked) cipher itself defines a circuit. More interestingly, we can break up that circuit into a number of interconnected *sub-circuits* that take care of all the processing. The composite circuit that a sub-circuit is part of is called its *super-circuit*. The variables of the super-circuit and its sub-circuits are related in the following ways:

- Each input variable of a sub-circuit is either an input variable of its super-circuit or the output variable of another sub-circuit.
- Variables in a super-circuit that are neither super-circuit input nor output variables are called *intermediate* variables.
- We consider duplication a form of processing and hence, any super-circuit input or intermediate variable propagates to at most one sub-circuit input or to the super-circuit output.

This can be applied recursively, where every sub-circuit can at the same time be a super-circuit with its own sub-circuits.

Let us illustrate this with an example: non-masked AES-128. Its circuit takes as input a 128-bit key and a 128-bit plaintext and returns as output a 128-bit ciphertext. Two obvious sub-circuits are the key schedule and the datapath. The former has the 128-bit key as input and a 1408-bit expanded key as output. The latter has the 128-bit plaintext and the expanded key as input and the 128-bit ciphertext as output. The datapath circuit can be split in 11 sub-circuits: one for each round and an initial round key addition circuit. The input of each round sub-circuit is the output of another circuit and a round key taken from the expanded key. The circuits for the first 9 rounds are identical, we say they are 9 instances of the same *circuit class*. The round circuit class can be split into

4 sub-circuits: SubBytes, ShiftRows, MixColumns and AddRoundKey. One may merge ShiftRows with SubBytes or MixColumns when targeting implementations where this step does not represent processing. A SubBytes circuit splits naturally into 16 sub-circuits of the same class, representing the S-box.

In this chapter, we will use the following formalism to specify circuits. We specify the input variables and their type, the output variables and their type and how to compute the output variables from the input variables. Unless stated otherwise, variables are binary. Circuits can be defined from scratch with simple operations such as addition and multiplication of variables, or in terms of sub-circuits. We call the former a basic circuit and the latter a composite circuit. Circuit 9.1.a provides an example of a basic circuit with three binary input variables and two binary output variables. A circuit can be used as a sub-circuit in the specification of a composite circuit. An example of a composite circuit with four binary input variables and two binary output variables is given in Circuit 9.1.b. The composite circuit is specified by two sub-circuits where

**(9.1.a)** Basic circuit

Name: ExampleCircuit1
Input: $(a, b, c)$
$a \leftarrow b \odot c$
$b \leftarrow c \oplus a$
Output: $(a, b)$

**(9.1.b)** Composite circuit

Name: ExampleCircuit2
Input: $(a, b, c, d)$
$(a, b) \leftarrow \text{ExampleCircuit1}(a, b, c)$
$(a, b) \leftarrow \text{ExampleCircuit1}(d, a, b)$
Output: $(a, b)$

the processing on the input variables is based on their location in the input array. When naming intermediate variables, one can make use of the fact that any variable shall be used exactly once. This implies that once a variable has occurred as the input of a sub-circuit, its name becomes available for another intermediate variable. In case we want to use a variable twice or more, we can put a cloning circuit. This is a circuit cloning variable $a$ to $b$: $(a, b) \leftarrow \text{Clone}(a)$, or to $a, b$ and $c$: $(a, b, c) \leftarrow \text{Clone}(a)$. In our convention, a composite circuit must use all output variables of its sub-circuits as outputs. If one wishes to omit a variable, this can be specified explicitly with a sinkhole circuit: $\text{Sinkhole}(a)$ is a simple circuit taking one variable and returning no variables.

We refer to circuits that have as many input variables as output variables (and of the same type) as *transformative* circuits. For these circuits we have a simplified convention. We specify them operating on a *state* which is used for both input and output. When they are used in the specification of a composite circuit, we omit the arrow and output variable.

At the circuit abstraction level, we see computation as the application of a particular input at the circuit's input and the observation of the result at its output. Furthermore, such circuits can be injective, surjective, both or neither.

**Definition 1.** *A circuit (class) is injective if for every input it returns a different output. It is surjective if for all outputs there is at least one input. We call a circuit (class) that is both injective and surjective a permutation circuit (class).*

Clearly, a permutation circuit can be modeled as a transformative circuit.

### 9.1.2 Fault Model

We model faults at the level of a circuit during computation. In the absence of faults, the output of a circuit of a given class is fully determined by its input. A circuit fault is simply any deviation from this.

**Definition 2.** *A circuit fault during a computation is a deviation of the circuit instance from its circuit class. Namely, the fault modifies the circuit in such a way that it could return for at least one input an output that does not correspond with the one prescribed by the circuit class.*

This definition covers a wide range of faults, including bit flips, or set-to-0 or set-to-1 faults of input, output, or intermediate variables, but also modifications of entries in lookup tables. Circuit faults are abstract and the mapping to physical faults that occur in actual implementations is often non-trivial. The hardware implementation of a cipher circuit that has many sub-circuit instances of the same class may re-use the same combinatorial hardware for all those instances. A permanent fault in such a hardware would correspond to a circuit fault in all circuit instances that it is used to implement. Similarly, a faulted entry in the AES S-box lookup table implies circuit faults in all S-box circuits of the AES circuit. Although faults often occur in implementations of circuit classes, at circuit abstraction level we see circuit faults in circuit instances. This is more general as faults may be induced for single executions of a program sequence or combinatorial circuit, or different faults may be induced for different instances.

A circuit fault does not necessarily imply a faulty circuit output. For example, a single faulted entry in an AES lookup table only leads to a faulty circuit output if the circuit input hits that entry. A stuck-at-0 fault affecting an input variable is not visible at the output if the variable is 0 anyway. For this reason, we define the concept of fault effectiveness.

**Definition 3.** *A circuit fault is effective during a computation if it leads to a faulty circuit output.*

When protection against faults is a concern, one typically performs redundant computations. At circuit level, this can be done by feeding it with variables that satisfy some conditions. In duplication, this can be done by two circuits of the same class that operate on input variable arrays set to the same value. In the absence of faults, this will also be the case for the outputs. Another possibility is a single circuit where the input variables satisfy some linear relation and in the absence of a fault, the output variables will satisfy the same. These circuits propagate a kind of *redundancy condition* that, if not satisfied, implies a fault must have occurred. Detection of faults can be done with a circuit as well. We call this a fault detection circuit. It (typically) simply propagates the input variables unchanged to the output but has an additional binary output, called *fault alert*, which is false when the redundancy condition is satisfied and true otherwise. This fault detection circuit may opt to use only the output variables of the cipher circuit, or it may use duplicates of intermediate variables as well.

### 9.1.3 Masking in Circuits

We speak of a *masked* circuit when it corresponds to a masked cipher. It operates on share variables and preserves the property that learning $d$ shares does not give information on native variables. In a similar way as the probing model [ISW03], we model side-channel attacks by allowing an attacker to probe all associated variables and to observe all computations of certain sub-circuits. The observation of a single sub-circuit does not give information about native variables only if the sub-circuit is *incomplete*.

**Definition 4.** *A sub-circuit in a masked circuit is incomplete if the input variables do not include all shares of a single native variable.*

For linear functions, such a partition into incomplete sub-circuits can be done quite easily as shown in Circuit 9.2.a. However, if we just consider one Boolean

**(9.2.a)**

```
Name: SharedXOR
State: $(a_0, a_1, b_0, b_1)$
XORFirst$(a_0, b_0)$
XORFirst$(a_1, b_1)$
```

**(9.2.b)**

```
Name: XORFirst
State: $(a, b)$
$a \leftarrow a \oplus b$
```

AND, such a partition into incomplete sub-circuits is more complex. Hence, many papers dealing with masking aim to find efficient masked implementations for the Boolean AND [ISW03; Bel+17; Rep+15a; Cnu+16; GIB18; GM17; GMK16; Bar+17].

If we just focus on the sharing of an AND, $c = a \odot b$, using 2 shares, such a sharing requires the addition of a resharing variable R. This is needed to ensure that the shares $c_0$ and $c_1$ are each independent of the native value of $c$. The resharing variable R is a circuit input. It may be derived from a dedicated random number generator or from another unrelated calculation, e.g., as shown in Changing of the Guards [Dae17]. A possible partition of masked AND into incomplete sub-circuits is then given in Circuit 9.3.a. This definition requires a lot of cloning. We can alternatively use sinkholes as in Circuit 9.3.b.

### 9.1.4 SIFA on Masked Circuits

Implementing a masked cipher based on a circuit with incomplete sub-circuits and with fault countermeasures such as duplication at cipher level with a fault detection circuit at the end of the cipher circuit is not sufficient to prevent SIFA. In this section we explain why. We assume a SIFA attacker that can make many computations but is limited to a circuit fault in a single sub-circuit (including fault detection circuits) during each computation. The success of this attack

**(9.3.a)** With cloning

```
Name: SharedAnd
Input: $(a_0, a_1, b_0, b_1, \mathrm{R})$
$(\mathrm{R}, \mathrm{R}') \leftarrow$ Clone($\mathrm{R}$)
$(a_0, a_0') \leftarrow$ Clone($a_0$)
$(a_1, a_1') \leftarrow$ Clone($a_1$)
$(b_0, b_0') \leftarrow$ Clone($b_0$)
$(b_1, b_1') \leftarrow$ Clone($b_1$)
$(c_0) \leftarrow$ AndXor$(a_0, b_1, \mathrm{R})$
$(c_0) \leftarrow$ AndXor$(a_0', b_0, c_0)$
$(c_1) \leftarrow$ AndXor$(a_1, b_0', \mathrm{R}')$
$(c_1) \leftarrow$ AndXor$(a_1', b_1', c_1)$
Output: $(c_0, c_1)$
```

```
Name: AndXor
Input: $(a, b, c)$
$d \leftarrow a \odot b$
$c \leftarrow d \oplus c$
Output: $(c)$
```

**(9.3.b)** With sinkholes

```
Name: SharedAnd
Input: $(a_0, a_1, b_0, b_1, \mathrm{R})$
$(c_0, \mathrm{R}) \leftarrow$ Clone($\mathrm{R}$)
AndXor1$(a_0, b_1, c_0)$
AndXor1$(a_0, b_0, c_0)$
$(c_1, \mathrm{R}) \leftarrow$ Clone($\mathrm{R}$)
AndXor1$(a_1, b_0, c_1)$
AndXor1$(a_1, b_1, c_1)$
Sinkhole$(\mathrm{R}, a_0, a_1, b_0, b_1)$
Output: $(c_0, c_1)$
```

```
Name: AndXor1
State: $(a, b, c)$
$d \leftarrow a \odot b$
$c \leftarrow d \oplus c$
```

relies on whether the behavior of the fault alert of a detection circuit depends on native variables.

To see that this is still possible in the presence of only incomplete sub-circuits, we give an example. Consider the single masked AND gate with 2 shares (cf. Circuit 9.3.b). We see that every input share is an input to two ANDXOR circuits and is combined with share 0 of a native variable in one of them and share 1 in the other. For instance, faulting $a_0$ to $a_0 \oplus 1$ at the input of ANDXOR1$(a_0, b_1, c_0)$ propagates to $a_0$ in ANDXOR1$(a_0, b_0, c_0)$. It will flip $c_0$ in ANDXOR1$(a_0, b_1, c_0)$ iff $b_1 = 1$ and $c_0$ in ANDXOR1$(a_0, b_0, c_0)$ iff $b_0 = 1$. The result is that it will flip $c_0$ an odd number of times iff $b_0 \oplus b_1 = b = 1$. Hence it will propagate to $c_0$ and hence also the native variable $c$ if $b = 1$ and not if $b = 0$. Resistance against SIFA requires us to construct circuits that ensure that the propagation of circuit faults in sub-circuits to the cipher circuit output is independent of native variables.

One condition that we use to achieve this is that each sub-circuit is incomplete. If a circuit is not incomplete, faulting such a sub-circuit might have fault effects that depend on native values. Second, we have to ensure that the circuit is built in such a way that the propagation of the fault effect does not lead to ineffective faults depending on native values. We have essentially two options to achieve this:

1. Build circuits of incomplete sub-circuits where an effective fault at the output of a single sub-circuit can never become ineffective at the output of the cipher circuit (Section 9.2).
2. Build a fault detection circuit that catches effective faults at the output of sub-circuits before they can become ineffective (Section 9.4).

Although a cipher circuit can be built from sub-circuits in a recursive way with multiple layers, in the remainder we will consider only a single level of sub-circuits. We will call these sub-circuits *basic circuits*. We then consider a single fault per execution of a cipher as a circuit fault in a single basic circuit instance. Furthermore, in a first-order side-channel attack (e.g., first-order DPA), we allow an attacker to observe a single basic circuit instance. This means that an attacker has knowledge about all variables associated to this basic circuit and the computation done within it.

### 9.1.5 On Detection Circuits

We consider detection circuits to be part of our circuit and hence, they can also be a target in fault attacks. Thus, in general, detection circuits are also incomplete, meaning that they have to operate on an incomplete set of shares. A simple solution to ensure this is to duplicate shares at the input of the cipher and to do the redundant computations on each set of shares. Then, detection circuits can check the consistency directly on duplicated shares.

In the next sections, we also present strategies protecting against single fault SIFA that only require fault detection on native values. This means that redundant computations do not have to be performed on duplicated sets of shares. Instead, only the native input values to a cipher can be duplicated and different randomness can be used to share the duplicated native values. However, when checking native values for faults, care has to be taken. If detection circuits operate on native values, faults on these circuits may leak parts of the native values. This is not a problem if these native values are not secret, e.g., when detection circuits operate on the ciphertext output of a cipher. However, if the compared native values have to be kept secret, care has to be taken that the shares representing the native value are never combined within this circuit. In addition, the detection circuits have to be placed in a manner so that faults on them do not reveal the native values.

## 9.2 Ciphers from Incomplete Permutation Circuits

In this section, we investigate how we can implement ciphers so that they are protected against single-fault SIFA. The heart of our strategy is to split a cipher into basic circuits that are permutations and that are incomplete. To do this, we use constructions common in the field of reversible computing [Lan61; Ben73; Tof80]. In particular, we use the Toffoli gate [Tof80] and related constructions as essential basic circuits.

### 9.2.1 The High-level Strategy

In this section, we aim to implement strategy 1 of Section 9.1.4. We do this by building a cipher circuit out of incomplete permutation basic circuits.

In this way, any single circuit fault in a basic circuit that is effective for that basic circuit is also effective for the cipher circuit. This follows from the following lemma and corollary.

**Lemma 1.** *Any composite circuit built from permutation sub-circuits is itself a permutation circuit.*

*Proof.* The circuit has at least one sub-circuit $A$ with all its input variables also input variables of its super-circuit. Now write the super-circuit as the serial composition of two circuits:

- The first circuit applies $A$ to the input variables and returns the corresponding output variables. The remaining variables are just copied from input to output.
- The second circuit is the super-circuit with the circuit $A$ replaced by the identity.

The first circuit is a permutation as $A$ is a permutation. We can iteratively apply this trick to the second circuit until it only contains a single sub-circuit. In this way we write the super-circuit as a series of invertible circuits. □

**Corollary 1.** *In a composite circuit built from permutation sub-circuits, any fault at the output of a sub-circuit will propagate to the output of the super-circuit.*

*Proof.* We can use the decomposition in circuits in the proof of Lemma 1 to split the super-circuit in two permutation circuits where the faulty output variables of a sub-circuit are input variables to one of the two permutation circuits. Therefore, the fault will propagate to the output. □

Thanks to Corollary 1, we can limit fault detection to the cipher's output.

**Corollary 2.** *Assume we have a composite circuit built from redundant incomplete permutation basic circuits. Further, assume that redundant circuits are implemented so that they do not influence each other. In addition, the redundant output shares are checked with the help of incomplete detection circuits. Then, the resulting composite circuit withstands single fault SIFA.*

A single fault can always only target a single incomplete basic circuit. This fault can be effective at the output of the targeted incomplete basic circuit or not. However, since the basic circuit is incomplete, the effectiveness of a fault never depends on a native value. In the case that the fault shows an effect, Corollary 1 ensures that the fault propagates through the circuit and is detected at the output by incomplete detection circuits. Also from this event, an attacker cannot infer any information, since it is also independent of native values. Thus, the effectiveness of a fault is independent of native values and hence, cannot be exploited by SIFA.

If we want to protect (tweakable) block ciphers, also the (tweak) key-schedule has to be a permutation (which it typically is), that is split in incomplete permutation basic circuits. In this case, we consider the last round-key and last tweak together with the ciphertext as output of the cipher that has to be checked for faults.

Typically, a cipher consists of a sequence of rounds and the round function has a linear and a nonlinear layer. We consider only ciphers where both layers are permutations.

For the linear layer, a split in incomplete permutation basic circuits is straightforward. In particular, a linear function $y = f(x)$ can be split in $d+1$ incomplete basic circuits that each operate on a single share of the native state variables. If $f$ is a permutation, then so are the basic circuits computing $y_i = f(x_i)$. Furthermore, a single fault always causes a change in the native value of $y$.

For the nonlinear layer sub-circuit, a split in incomplete permutation sub-circuits is less trivial. Typically, the nonlinear layer consists of the parallel application of a nonlinear S-box to subsets of the state variables. Hence the challenge is to build a circuit for the masked S-box in terms of incomplete permutation basic circuits.

We do this by constructing masked S-box circuits that are permutations using basic circuits of the inherently bijective Toffoli gate (cf. Section 9.2.2) and variants. We follow a two-stage approach: first express the (unmasked) S-box in terms of Toffoli gates and then build a circuit of the masked Toffoli gate using incomplete Toffoli-gate basic circuits.

As a consequence of our design strategy, we end up with a round function circuit where each basic circuit is incomplete and a permutation on the shared state. This implies that it preserves uniformity of the sharing and hence, no fresh randomness is required during the rounds for realizing first-order DPA secure circuits.

### 9.2.2 Incomplete Permutation Basic Circuits

In the following, we present our permutation basic circuits that we will use to realize circuits for S-boxes. In essence, we need three different basic circuits. The first one is the Toffoli gate [Tof80], a nonlinear 3-bit permutation. We denote it by $p_T(a, b, c)$ and define it in Circuit 9.4.a. For brevity in our S-box constructions, we also define a permutation $p_\chi(a, b, c)$ in Circuit 9.4.b that is a close variant of it. In addition, we need the basic circuit XorFirst$(a, b)$ from Circuit 9.2.b to realize some S-boxes. In the first step, we build circuits out

**(9.4.a)** Toffoli gate

Name: $p_T$

State: $\{a, b, c\}$

$\text{T} \leftarrow b \odot c$

$a \leftarrow a \oplus \text{T}$

**(9.4.b)**

Name: $p_\chi$

State: $\{a, b, c\}$

$\text{T} \leftarrow \bar{b} \odot c$

$a \leftarrow a \oplus \text{T}$

of these basic circuits. Those circuits are masked versions of the basic circuits and will be used as building blocks for the S-boxes. First, let us have a look at a circuit for the 2-share masked Toffoli gate shown in Circuit 9.5.a, that we will refer to as $p_{TS}(a_0, a_1, b_0, b_1, c_0, c_1)$. As can be seen in Circuit 9.5.a, all $p_T$ sub-circuit instances are incomplete. Thanks to the fact that the basic circuits are

| **(9.5.a)** Masked Toffoli gate | **(9.5.b)** |
|---|---|
| Name: $p_{TS}$ | Name: $p_{\chi S}$ |
| State: $\{a_0, a_1, b_0, b_1, c_0, c_1\}$ | State: $\{a_0, a_1, b_0, b_1, c_0, c_1\}$ |
| $p_T(a_0, b_0, c_1)$ | $p_\chi(a_0, b_0, c_1)$ |
| $p_T(a_0, b_0, c_0)$ | $p_\chi(a_0, b_0, c_0)$ |
| $p_T(a_1, b_1, c_1)$ | $p_T(a_1, b_1, c_1)$ |
| $p_T(a_1, b_1, c_0)$ | $p_T(a_1, b_1, c_0)$ |

permutations on the state, any circuit fault in a single $p_T$ sub-circuit instance that is effective at its output will also be effective at the output of the super-circuit. Moreover, any effective fault due to a single sub-circuit fault can at most affect a single share per variable, and will hence result in a faulty native variable at that point. Thanks to the correctness of sharing, this fault will propagate to the super-circuit output. These observations are also true for the masked version $p_{\chi S}$ of $p_\chi$ (Circuit 9.5.b).

Next, we will show how to build circuits of two-share masked S-boxes using the basic circuits introduced here. For the sake of completeness, we note that the same properties can be achieved in a similar form for three-share threshold implementation as shown in Circuit 9.6.a. The algorithmic representation fulfills the three requirements for threshold implementations (TI):

1. *Correctness*, since the gate correctly implements the equations $a = a \oplus b \odot c$ which can be checked be adding all output shares of $a$ (the equations $b = b$ and $c = c$ are trivial).
2. *Uniformity*, which follows from the fact that for each output share a single share of $a$ appears in additive form.
3. *Non-completeness*, because for each calculation of one output share, one share index never appears (e.g., the calculation of the output share $a_0$ does not use any shares with the index 1 like $b_1$ or $c_1$).

The threshold implementation of $p_{\chi T}$ follows analogously.

### 9.2.3 3-bit S-boxes

Recently, 3-bit S-boxes have become more prominent with their usage in PRINT-cipher [Knu+10], LowMC [Alb+15], or Xoodoo [Dae+18]. As a representative of these S-boxes, we focus on the protection of the 3-bit $\chi$-layer [Dae95; Dae+18]. The mapping $\chi$ operates on circular arrays of bits and it complements all bits that have the pattern 01 in the bits at their right. $\chi$ is bijective if and only if the length of the circular array is odd. The $\chi$ mapping in the round function of ciphers typically operates on a large set of short odd-length sub-arrays of the state in parallel. We will refer to $n$-bit $\chi$ as $\chi_n$.

Daemen et al. [Dae+18] pointed out that it is possible to compute $\chi_3$ in-place in its registers as a sequence of three Toffoli gates. This immediately yields a

**(9.6.a)** Algorithmic representation of masked Toffoli gate ($p_{TT}$) using 3 shares

```
Name: $p_{TT}$
State: $\{a_0, a_1, a_2, b_0, b_1, b_2, c_0, c_1, c_2\}$
$p_T(a_0, b_0, c_0)$
$p_T(a_0, b_0, c_2)$
$p_T(a_0, b_2, c_0)$

$p_T(a_1, b_1, c_1)$
$p_T(a_1, b_1, c_0)$
$p_T(a_1, b_0, c_1)$

$p_T(a_2, b_2, c_2)$
$p_T(a_2, b_2, c_1)$
$p_T(a_2, b_1, c_2)$
```

circuit for two-share masked $\chi_3$ in terms of permutation sub-circuits:

```
Name: Masked_chi3
State: $\{a_0, a_1, b_0, b_1, c_0, c_1\}$
$p_{\chi S}(a_0, a_1, b_0, b_1, c_0, c_1)$
$p_{\chi S}(b_0, b_1, c_0, c_1, a_0, a_1)$
$p_{\chi S}(c_0, c_1, a_0, a_1, b_0, b_1)$
```

Recall from Section 9.2.2 that $p_{\chi S}$ is just a composite circuit and that its basic circuits are $p_T$ or $p_\chi$. Still, a fault effect stemming from a single basic circuit shows an effect in the native values at the S-box output.

### 9.2.4 4-bit S-boxes

The construction and design of 4-bit S-boxes has been intensively studied in literature.

Using affine equivalence, De Cannière [De 07] partitions all 4-bit bijective S-boxes in 302 equivalence classes, where 1 class contains all affine functions, 6 classes contain quadratic functions, and 295 classes represent the cubic functions [Bil+15].

As shown by Bilgin et al. [Bil+15], 144 cubic classes can be constructed by iterating the S-boxes of the quadratic classes separated by affine layers up to 3 times. This covers many prominent S-boxes, e.g., the S-boxes used in Noekeon [Dae+00] and Present [Bog+07], but also several of the 16 S-boxes observed to be “optimal” by Leander and Poschmann [LP07]. We focus on the 6 classes of quadratic functions. The variables $a$, $b$, $c$, and $d$ indicate the input and output bits of the S-box, where $a$ is the most significant bit. The operations needed to compute the 6 quadratic classes are summarized in Table 9.1.

**Table 9.1:** The 6 classes of quadratic 4-bit S-boxes [Bil+15] expressed in terms of $p_T$ and $p_\chi$.

| 0123456789ABDCFE | 0123456789CDEFAB | 0123457689CDEFBA |
|---|---|---|
| Name: $Q^4_4$<br>State: $\{a, b, c, d\}$<br>$p_T(d, a, b)$ | Name: $Q^4_{12}$<br>State: $\{a, b, c, d\}$<br>$p_T(b, a, c)$<br>$p_T(c, a, b)$ | Name: $Q^4_{293}$<br>State: $\{a, b, c, d\}$<br>$p_T(d, b, c)$<br>$p_T(b, a, c)$<br>$p_T(c, a, b)$ |

| 0123456789BAEFDC | 012345678ACEB9FD | 0123458967CDEFAB |
|---|---|---|
| Name: $Q^4_{294}$<br>State: $\{a, b, c, d\}$<br>$p_T(c, a, b)$<br>$p_T(d, a, b)$<br>$p_T(d, a, c)$ | Name: $Q^4_{299}$<br>State: $\{a, b, c, d\}$<br>$p_T(b, a, c)$<br>$p_T(c, a, b)$<br>$p_T(b, a, c)$<br>$p_T(c, a, d)$<br>$p_T(d, a, c)$ | Name: $Q^4_{300}$<br>State: $\{a, b, c, d\}$<br>XorFirst$(b, a)$<br>XorFirst$(c, a)$<br>$p_T(a, b, c)$<br>$p_T(b, a, c)$<br>$p_\chi(c, b, a)$ |

Using Table 9.1, Circuit 9.5.a, and Circuit 9.5.b, it is straightforward to build circuits for two-share masked versions for 144 out of the 295 cubic classes of S-boxes [Bil+15] from incomplete permutation basic circuits. For S-boxes which are not in these classes, we refer to results regarding the implementation of 4-bit permutations using reversible components. For instance, Golubitsky and Maslov [GM12] give optimal implementations (with respect to a certain set of reversible gates) for all 4-bit permutations using at most 15 reversible gates. However, note that the set of reversible gates used may differ from the basic circuits $p_T$ and $p_\chi$ used in this section and hence, we consider the exploration of this as future work.

### 9.2.5 5-bit S-boxes

Shende et. al [She+03] show that every permutation (S-box) with an odd number of inputs can be implemented using reversible gates by using at most one additional variable. However, as we will see next, the need for this additional variable forces us to deviate from the strategy that each basic circuit is a permutation. In particular, we will make use of Sinkhole$(r)$ and $(r_1, r_0) \leftarrow$ Clone$(r_0)$ introduced in Section 9.1.1.

In this work, we only focus on the 5-bit S-box $\chi_5$, which has several prominent uses. For instance, it is used in the Keccak-$p$ permutations inside Ketje [Ber+b], Keyak [Ber+c], Kravatte [Ber+17b], and Keccak [Ber+11c]

(SHA-3). Moreover, $\chi_5$ is also the core of ASCON's S-box [Dob+]. We base our circuit for two-share masked $\chi_5$ on an implementation of $\chi_5$ [Dob+] with input bits $a$, $b$, $c$, $d$, and $e$ and an intermediate variable $r$, as shown in Circuit 9.7.a. To provide an implementation of $\chi_5$ that withstands single-fault SIFA, we again

**(9.7.a)** $\chi_5$

```
Name: $\chi_5$
State: $\{a, b, c, d, e\}$
$(r) \leftarrow$ AndNot$(a, e)$
$p_\chi(a, b, c)$
$p_\chi(c, d, e)$
$p_\chi(e, a, b)$
$p_\chi(b, c, d)$
XorFirst$(d, r)$
Sinkhole$(r)$
```

```
Name: AndNot
Input: $(b, c)$
$a \leftarrow b \odot \bar{c}$
Output: $(a)$
```

rely on $p_{\chi S}(a_0, a_1, b_0, b_1, c_0, c_1)$ as a building block. We introduce additional input variables $r_0$ and $r_1$, which have to be initialized with random values such that $r_0 \oplus r_1 = 0$. This allows us to argue the security of the following scheme in Circuit 9.8.a. We end up with a construction which is the repeated applica-

**(9.8.a)** Masked $\chi_5$ with constraints

```
Name: Masked_chi5_v1
Input: $\{a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1, r_0, r_1\}$
$p_{\chi S}(r_0, r_1, e_0, e_1, a_0, a_1)$
$p_{\chi S}(a_0, a_1, b_0, b_1, c_0, c_1)$
$p_{\chi S}(c_0, c_1, d_0, d_1, e_0, e_1)$
$p_{\chi S}(e_0, e_1, a_0, a_1, b_0, b_1)$
$p_{\chi S}(b_0, b_1, c_0, c_1, d_0, d_1)$
XorFirst$(d_0, r_0)$
XorFirst$(d_1, r_1)$
Sinkhole$(r_0, r_1)$
Output: $\{a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1\}$
```

**(9.8.b)** Masked $\chi_5$ with cloning

```
Name: Masked_chi5_v2
State: $\{a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1, r_0\}$
$(r_1, r_0) \leftarrow$ Clone$(r_0)$
$p_{\chi S}(r_0, r_1, e_0, e_1, a_0, a_1)$
$p_{\chi S}(a_0, a_1, b_0, b_1, c_0, c_1)$
$p_{\chi S}(c_0, c_1, d_0, d_1, e_0, e_1)$
$p_{\chi S}(e_0, e_1, a_0, a_1, b_0, b_1)$
$p_{\chi S}(b_0, b_1, c_0, c_1, d_0, d_1)$
XorFirst$(d_0, r_0)$
XorFirst$(d_1, r_1)$
Sinkhole$(r_1)$
```

tion of permutation $p_{\chi S}$ on 12 bits of the state $a_0$ to $r_1$. Due to this iterative construction, a fault that has an effect on any native output variable of one $p_{\chi S}$ would have an effect on the native output variables of the whole circuit if $r_0$ and $r_1$ would be part of the output. However, $r_0$ and $r_1$ end in Sinkhole$(r_0, r_1)$. Hence, we have to show that this never leads to an effect of a fault disappearing.

As can be seen in Circuit 9.8.a, $d_0$ and $d_1$ are only written in the basic circuits XorFirst$(d_0, r_0)$ and XorFirst$(d_1, r_1)$. Furthermore, the calculation of $r_0$ and $r_1$ is independent of $d_0$ or $d_1$. As a consequence, a fault in a single basic circuit that happens before the execution of $\textsc{XorFirst}(d_0, r_0)$ and $\textsc{XorFirst}(d_1, r_1)$ can never have an effect on the shares of $d$ and $r$ at the same time. Hence, the basic circuits $\textsc{XorFirst}(d_0, r_0)$ and $\textsc{XorFirst}(d_1, r_1)$ never cancel the effect of a fault on a single basic circuit, and effects of faults on the native value of $r$ carry over to $d$.

In a similar spirit as Sugawara for AES [Sug19], it is possible to use one share $r_0$ of the output of one S-box layer as input to the next layer of S-boxes. Hence, it is possible to implement ciphers which use the sharing shown in Circuit 9.8.b without the need for additional randomness, except the one needed for the initial sharing and for the first S-box layer. We have verified exhaustively that Circuit 9.8.b is a permutation on the bits $a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1$, and $r_0$ and that the masking is indeed correct using `maskVerif` (cf. Section 9.5.1).

## 9.3 AES S-box from Incomplete Permutation Basic Circuits

So far, the main focus has been on S-boxes that have a rather simple and compact description over $\mathbb{F}_2$. However, there exist S-boxes with complex descriptions over $\mathbb{F}_2$, but more concise descriptions over larger binary fields, i.e., $\mathbb{F}_{2^n}$. Hence, we will apply our method to the representation of the S-box over $\mathbb{F}_{2^n}$. The most prominent example that falls into this category is the S-box of AES [DR02]. Building on Canright's description [Can05], we can derive a description that is better suited for our proposed countermeasure.

Figure 9.1 shows Canright's description of the AES S-box just using reversible computations, basically transforming the idea of Sugawara [Sug19, Figure 8] from 3-shared to 2-shared masking. This can be done by replacing all $\mathbb{F}_{2^n}$ multiplications in Canright's description by Toffoli gates operating in $\mathbb{F}_{2^n}$ using an additional input that is set to 0. To distinguish it from the binary Toffoli gate defined in Section 9.2.2, we denote a Toffoli gate over $\mathbb{F}_{2^n}$ by $p_T^n(a, b, c)$ with $a, b, c \in \mathbb{F}_{2^n}$. In the following, we denote multiplication and addition over $\mathbb{F}_{2^n}$ by $\cdot$ and $+$, respectively:

Name: $p_T^n$
State: $\{a, b, c\}$
$a \leftarrow a + b \cdot c$

We can also define a masked version of the Toffoli gate over $\mathbb{F}_{2^n}$ in a similar way as in Circuit 9.5.a. Here, $a_0, b_0, c_0, a_1, b_1, c_1 \in \mathbb{F}_{2^n}$ denote the shares of $a, b, c$ and

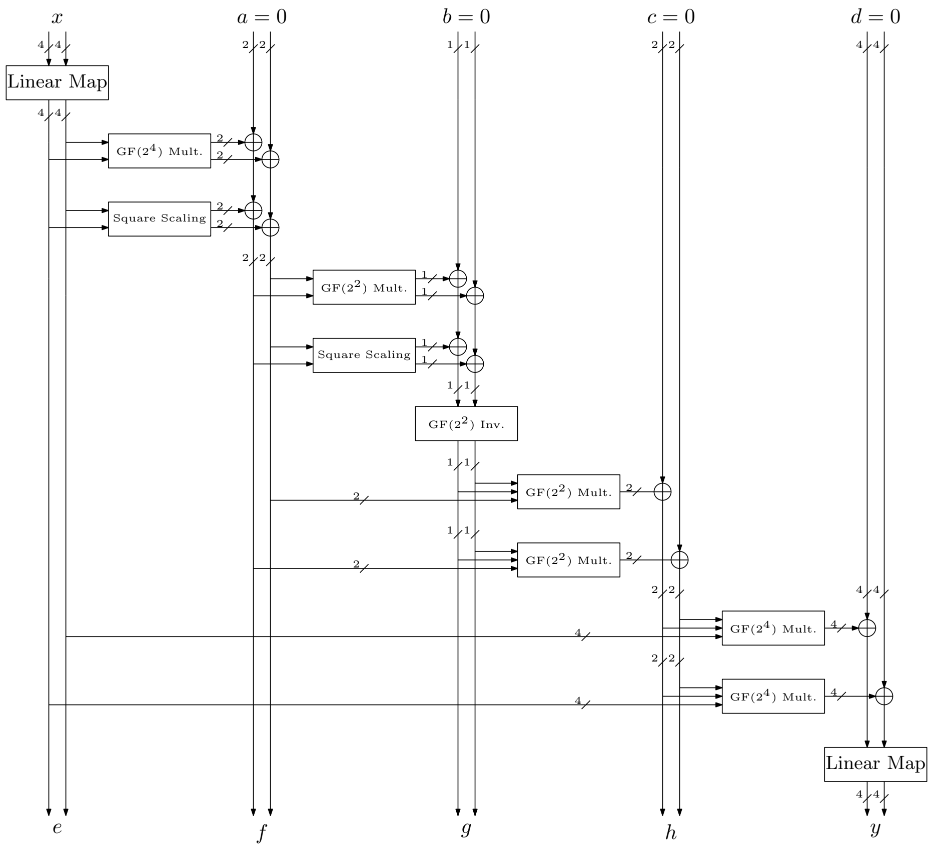


**Figure 9.1:** Description of the AES S-box ($y = S(x)$) relying on invertible computations.

the shared version of $p_T^n$ is denoted by $p_{TS}^n(a_0, a_1, b_0, b_1, c_0, c_1)$:

Name: $p_{TS}^n$
State: $\{a_0, a_1, b_0, b_1, c_0, c_1\}$
$p_T^n(a_0, b_0, c_1)$
$p_T^n(a_0, b_0, c_0)$
$p_T^n(a_1, b_1, c_1)$
$p_T^n(a_1, b_1, c_0)$

The arguments for the security and fault propagation of $p_{TS}$ are analogous to Section 9.2.2. Again, each $p_T^n$ is incomplete and hence, the effect stemming from faulting a single instance can never depend on a native value. Furthermore, each fault effect caused by a fault on a single instance of $p_T^n$ is then visible in the native value. Since only the shares of $a$ are updated dependent on the shares of $b$ and the shares of $c$, a change in a native value at any point caused by a fault on a single instance of $p_T^n$ can never become ineffective and hence, is visible at the output of $p_{TS}^n$.

We will now use $p_{TS}^n$ to build a circuit for the 2-share masked AES S-box. If we take a look at the building block given in Figure 9.1, the multiplications in $\mathbb{F}_{2^n}$ (later encapsulated in $p_T^n$) are the only nonlinear components of the S-box. The square scaling over $\mathbb{F}_{2^n}$ ($s_{sc}^n(a,b,c)$) is a linear reversible operation, the linear maps ($\Gamma(a)$ and $\Xi(a)$) are linear permutations, the addition of the constant (AddConstant$(a)$) is an affine operation, and the inversion (Inv$(a)$) over $\mathbb{F}_{2^2}$ corresponds to a simple bit-swap. We will construct the S-box with the help of these basic circuits: $\Gamma$, AddConstant, Inv, $p_T^n$, $s_{sc}^n$, and $\Xi$ with the following description. With superscripts $H$ and $L$, we denote the higher half of coefficients and the lower half of the coefficients, respectively:

Name: $\Gamma$
State: $\{x\}$

$$\begin{bmatrix} x_7\\ x_6\\ x_5\\ x_4\\ x_3\\ x_2\\ x_1\\ x_0 \end{bmatrix} \leftarrow \begin{bmatrix} 1&1&1&0&0&1&1&1\\ 0&1&1&1&0&0&0&1\\ 0&1&1&0&0&0&1&1\\ 1&1&1&0&0&0&0&1\\ 1&0&0&1&1&0&1&1\\ 0&0&0&0&0&0&0&1\\ 0&1&1&0&0&0&0&1\\ 0&1&0&0&1&1&1&1 \end{bmatrix} \cdot \begin{bmatrix} x_7\\ x_6\\ x_5\\ x_4\\ x_3\\ x_2\\ x_1\\ x_0 \end{bmatrix}$$

Name: $\Xi$
State: $\{x\}$

$$\begin{bmatrix} x_7\\ x_6\\ x_5\\ x_4\\ x_3\\ x_2\\ x_1\\ x_0 \end{bmatrix} \leftarrow \begin{bmatrix} 0&0&1&0&1&0&0&0\\ 1&0&0&0&1&0&0&0\\ 0&1&0&0&0&0&0&1\\ 1&0&1&0&1&0&0&0\\ 1&1&1&1&1&0&0&0\\ 0&1&1&0&1&1&0&1\\ 0&0&1&1&0&0&1&0\\ 0&1&0&1&0&0&1&0 \end{bmatrix} \cdot \begin{bmatrix} x_7\\ x_6\\ x_5\\ x_4\\ x_3\\ x_2\\ x_1\\ x_0 \end{bmatrix}$$

Name: $s_{sc}^4$
State: $\{a,b,c\}$
$\mathrm{T}_0 \leftarrow b \oplus c$
$a \leftarrow a \oplus ((\mathrm{T}_0^0 \oplus \mathrm{T}_0^2)\|(\mathrm{T}_0^3 \oplus \mathrm{T}_0^1)\|(\mathrm{T}_0^0 \oplus \mathrm{T}_0^1)\|\mathrm{T}_0^0)$

Name: $s_{sc}^2$
State: $\{a,b,c\}$
$\mathrm{T}_0 \leftarrow b \oplus c$
$a \leftarrow a \oplus (\mathrm{T}_0^1\|(\mathrm{T}_0^0 \oplus \mathrm{T}_0^1))$

Name: Inv
State: $\{a\}$
$a \leftarrow a^0\|a^1$

Name: AddConstant
State: $\{a\}$
$a \leftarrow a \oplus \texttt{0x63}$

Multiplication $c \leftarrow a \cdot b$ in $GF(2^2)$
$c^1 \leftarrow ((a^1 \oplus a^0) \odot (b^1 \oplus b^0)) \oplus (a^1 \odot b^1)$
$c^0 \leftarrow ((a^1 \oplus a^0) \odot (b^1 \oplus b^0)) \oplus (a^0 \odot b^0)$

Multiplication $c \leftarrow a \cdot b$ in $GF(2^4)$
$\mathrm{T}_0 \leftarrow (a^H \oplus a^L) \cdot (b^H \oplus b^L)$
$\mathrm{T}_1 \leftarrow \mathrm{T}_0^0\|(\mathrm{T}_0^0 \oplus \mathrm{T}_0^1)$
$c^H \leftarrow \mathrm{T}_1 \oplus (a^H \cdot b^H)$
$c^L \leftarrow \mathrm{T}_1 \oplus (a^L \cdot b^L)$

Having discussed all necessary building blocks, we are ready to give our shared implementation of the AES S-box. In the description (Circuit 9.9.a) of the AES S-box, we use variables in different fields: $x_0, x_1, d_0, d_1, e_0, e_1, y_0, y_1 \in \mathbb{F}_{2^8}$, $a_0, a_1, c_0, c_1, f_0, f_1, h_0, h_1 \in \mathbb{F}_{2^4}$, and $b_0, b_1, g_0, g_1 \in \mathbb{F}_{2^2}$. Furthermore, we require that $a_0 + a_1 = 0$, $b_0 + b_1 = 0$, $c_0 + c_1 = 0$, $d_0 + d_1 = 0$ to correctly compute the AES S-box $y_0 + y_1 = S(x_0 + x_1)$. As can be seen in Circuit 9.9.a, each of the basic circuits is incomplete and hence, the effect stemming from faulting a single instance is independent of native values. Next, we have to show that the effect of

**(9.9.a)** Masked AES S-box with constraints $a_0 + a_1 = b_0 + b_1 = c_0 + c_1 = d_0 + d_1 = 0$

```
Name: Masked_AES _v1
Input: $\{x_0, x_1, a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1\}$
$\Gamma(x_0)$
$\Gamma(x_1)$
$p^4_{TS}(a_0, a_1, x^H_0, x^H_1, x^L_0, x^L_1)$
$s^4_{sc}(a_0, x^H_0, x^L_0)$
$s^4_{sc}(a_1, x^H_1, x^L_1)$
$p^2_{TS}(b_0, b_1, a^H_0, a^H_1, a^L_0, a^L_1)$
$s^2_{sc}(b_0, a^H_0, a^L_0)$
$s^2_{sc}(b_1, a^H_1, a^L_1)$
$\mathrm{Inv}(b_0)$
$\mathrm{Inv}(b_1)$
$p^2_{TS}(c^H_0, c^H_1, a^L_0, a^L_1, b_0, b_1)$
$p^2_{TS}(c^L_0, c^L_1, a^H_0, a^H_1, b_0, b_1)$
$p^4_{TS}(d^H_0, d^H_1, x^L_0, x^L_1, c_0, c_1)$
$p^4_{TS}(d^L_0, d^L_1, x^H_0, x^H_1, c_0, c_1)$
$\Xi(d_0)$
$\Xi(d_1)$
$\mathrm{AddConstant}(d_0)$
$e_0 \leftarrow x_0, e_1 \leftarrow x_1, f_0 \leftarrow a_0, f_1 \leftarrow a_1$
$g_0 \leftarrow b_0, g_1 \leftarrow b_1, h_0 \leftarrow c_0, h_1 \leftarrow c_1$
$y_0 \leftarrow d_0, y_1 \leftarrow d_1$
Output: $\{e_0, e_1, f_0, f_1, g_0, g_1, h_0, h_1, y_0, y_1\}$
```

**(9.9.b)** Masked AES S-box with cloning

```
Name: Masked_AES _v2
Input: $\{x_0, x_1, a_0, b_0, c_0, d_0\}$
$(a_1, a_0) \leftarrow \mathrm{Clone}(a_0)$
$(b_1, b_0) \leftarrow \mathrm{Clone}(b_0)$
$(c_1, c_0) \leftarrow \mathrm{Clone}(c_0)$
$(d_1, d_0) \leftarrow \mathrm{Clone}(d_0)$
$\Gamma(x_0)$
$\Gamma(x_1)$
$p^4_{TS}(a_0, a_1, x^H_0, x^H_1, x^L_0, x^L_1)$
$s^4_{sc}(a_0, x^H_0, x^L_0)$
$s^4_{sc}(a_1, x^H_1, x^L_1)$
$p^2_{TS}(b_0, b_1, a^H_0, a^H_1, a^L_0, a^L_1)$
$s^2_{sc}(b_0, a^H_0, a^L_0)$
$s^2_{sc}(b_1, a^H_1, a^L_1)$
$\mathrm{Inv}(b_0)$
$\mathrm{Inv}(b_1)$
$p^2_{TS}(c^H_0, c^H_1, a^L_0, a^L_1, b_0, b_1)$
$p^2_{TS}(c^L_0, c^L_1, a^H_0, a^H_1, b_0, b_1)$
$p^4_{TS}(d^H_0, d^H_1, x^L_0, x^L_1, c_0, c_1)$
$p^4_{TS}(d^L_0, d^L_1, x^H_0, x^H_1, c_0, c_1)$
$\Xi(d_0)$
$\Xi(d_1)$
$\mathrm{AddConstant}(d_0)$
$e_0 \leftarrow x_0, e_1 \leftarrow x_1, f_0 \leftarrow a_0, f_1 \leftarrow a_1$
$g_0 \leftarrow b_0, g_1 \leftarrow b_1, h_0 \leftarrow c_0, h_1 \leftarrow c_1$
$y_0 \leftarrow d_0, y_1 \leftarrow d_1$
Sinkhole$(e_1, f_1, g_1, h_1)$
Output: $\{y_0, y_1, f_0, g_0, h_0, e_0\}$
```

a fault on a single instance is always present in the native values of our circuit. If a single fault targets $\Gamma$, an effect will be visible in the native value of $e$. An effect caused by a fault on a $p^n_T$ or $s^n_{sc}$ processing shares of $x$ and $a$ will show an effect on the native values of $x$ ($e$) and $a$ ($f$), since the value of the shares of $a$ only depend on the shares of $x$ and $p^n_T$ and $s^n_{sc}$ are a permutation on the shares. The same argument is valid for $p^n_T$ or $s^n_{sc}$, processing shares of $a$ and $b$, since $\mathrm{Inv}(b)$ is just a bit-swap. Next, we have a series of $p^n_T$ processing shares of $a$, $b$, and $c$. Since the shares of $c$ only depend on the shares of $a$ and $b$, again a change in a native value is always visible in the output. The same is true for the last set of $p_T^n$ processing shares of $x$, $c$, and $d$, followed by a share-wise permutation $\Xi$.

However, in the context of using this S-box within the AES, we cannot further use, e.g., $e_0, e_1$ as input for $d_0, d_1$ for another S-box, since $e_0 + e_1$ is not 0 in general. For this reason, we remove shares from the input and the output of our S-box in a similar spirit to Sugawara [Sug19] in Circuit 9.9.b, which we have also formally verified using `maskVerif` (cf. Section 9.5.1).

The S-box in Circuit 9.9.b is still a permutation on its inputs as we show in Appendix Figure 9.2. Since we do not have any restrictions on the inputs of the S-box, we are free to reuse $e_0, f_0, g_0, h_0$ as inputs for another S-box calculation, and hence, do not have to always generate fresh sharings of 0. In particular, this allows for first-order side-channel secured implementations of AES without the need for additional randomness in the masked S-boxes.

However, since we discard $e_1, f_1, g_1, h_1$ at the output of the S-box, we hinder faults from propagating and thus, have to employ fault countermeasures on S-box level for these values (respectively for their native values $e$, $f$, $g$, and $h$). While this results only in a rather small overhead for implementations that use duplication to protect against fault attacks, this might become quite expensive for implementations that use time redundancy since one might have to store all the values that need to be checked in the time redundant computation. However, this cost can be significantly reduced by computing and storing a checksum or fingerprint of these values instead. For instance, one might only store one set of $e_0, e_1, f_0, f_1, g_0, g_1, h_0, h_1$ all initialized to 0 and always update those shares by a linear checksum with the output $e_0, e_1, f_0, f_1, g_0, g_1, h_0, h_1$ of the S-box before truncation. Note that by using a linear checksum this can be done for $e_0, f_0, g_0, h_0$ and $e_1, f_1, g_1, h_1$ independently and thus secured against first-order side-channel attacks. By computing this checksum for the original and redundant computations, a fault will be detected by checking the output of the redundant AES computations and the checksum value.

## 9.4 Protecting Arbitrary Circuits

Our approach from Section 9.2 works by constructing the masked circuit for a cipher in a particular way with particular basic circuits. The cost of building masked circuits this way varies depending on the specific S-box. When optimizing with respect to other metrics, such as latency, other approaches may be more suitable than the one introduced in Section 9.2, which is essentially serial. The AES example in Section 9.3 also showed that additional error checks of intermediate values are helpful in cases when rewriting the entire cipher is hard. In this section, we generalize this approach and explore how a general masked circuit can be protected against SIFA by defining a suitable error detection circuit. The goal is to take an existing masked circuit, whose basic circuits are for example individual Boolean gates, and identify the relevant intermediate values to check for errors in addition to the cipher output.

$$
\begin{aligned}
&x_0 = \Gamma^{-1}(e_0)\\
&x_1 = x_0 + S^{-1}(y_0 + y_1)\\
&e_1 = \Gamma(x_1)\\
&\mathrm{T}_0 = 0\\
&s^4_{sc}(\mathrm{T}_0, e^H_0, e^L_0)\\
&a_0 = f_0 + \mathrm{T}_0 + e^H_0 \cdot (e^L_0 + e^L_1)\\
&a_1 = a_0\\
&\mathrm{T}_1 = 0\\
&s^4_{sc}(\mathrm{T}_1, e^H_1, e^L_1)\\
&f_1 = a_1 + \mathrm{T}_1 + e^H_1 \cdot (e^L_0 + e^L_1)\\
&\mathrm{T}_0 = 0\\
&s^2_{sc}(\mathrm{T}_0, f^H_0, f^L_0)\\
&b_0 = g_0^{-1} + \mathrm{T}_0 + f^H_0 \cdot (f^L_0 + f^L_1)\\
&b_1 = b_0\\
&\mathrm{T}_1 = 0\\
&s^2_{sc}(\mathrm{T}_1, f^H_1, f^L_1)\\
&g_1 = (b_1 + \mathrm{T}_1 + f^H_1 \cdot (f^L_0 + f^L_1))^{-1}\\
&c^H_0 = h^H_0 + f^L_0 \cdot (g_0 + g_1)\\
&c^H_1 = c^H_0\\
&h^H_1 = c^H_1 + f^L_1 \cdot (g_0 + g_1)\\
&c^L_0 = h^L_0 + f^H_0 \cdot (g_0 + g_1)\\
&c^L_1 = c^L_0\\
&h^L_1 = c^L_1 + f^H_1 \cdot (g_0 + g_1)\\
&d^H_0 = \Xi^{-1}(y^H_0 + \texttt{0x6}) + e^L_0 \cdot (c_0 + c_1)\\
&d^H_1 = \Xi^{-1}(y^H_1) + e^L_1 \cdot (c_0 + c_1)\\
&d^L_0 = \Xi^{-1}(y^L_0 + \texttt{0x3}) + e^H_0 \cdot (c_0 + c_1)\\
&d^L_1 = \Xi^{-1}(y^L_1) + e^H_1 \cdot (c_0 + c_1)
\end{aligned}
$$

**Figure 9.2:** Inverse computation to show that Circuit 9.9.b is a permutation.

We first recall the computation and fault model in order to introduce the general criterion for single-fault SIFA-resistance in Section 9.4.1. In Section 9.4.2, we show how to satisfy this criterion by extending a general masked implementation with local error detection checks. Then, in Section 9.4.3, we identify necessary steps and conditions such that global checks are sufficient. Finally, we discuss how to extend this approach to higher-order attacks, where the adversary applies multiple faults in each execution, in Section 9.4.4.

### 9.4.1 A General Criterion for Resistance against Single-Fault SIFA

We consider the directed acyclic graph (DAG) induced by a masked cipher circuit composed of basic circuits (in the sense of Section 9.1.3 that basic circuits are incomplete). This *computation graph* consists of nodes that represent the basic circuits $f \in \mathcal{F}$ and that are connected by edges that represent the intermediate variables $v \in \mathcal{V}$. We identify a node $f$ with $n$ input edges $\textsc{in}(f) = (x_1, \ldots, x_n)$ and $m$ output edges $\textsc{out}(f) = (y_1, \ldots, y_m)$ with the corresponding vectorial Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$, $(x_1, \ldots, x_n) \mapsto (y_1, \ldots, y_m)$ of the basic circuit. We distinguish linear nodes, whose function is affine linear over $\mathbb{F}_2$, and nonlinear nodes.

As defined in Section 9.1, we consider a powerful single-fault attacker who may replace any node $(y_1, \ldots, y_m) = f(x_1, \ldots, x_n)$ by an arbitrary faulted node $(y_1^*, \ldots, y_m^*) = f^*(x_1, \ldots, x_n)$. We denote the difference between the values of an edge $v$ in the correct execution and $v^*$ in the faulted execution by $\delta v = v \oplus v^*$, similar to differential cryptanalysis, and write the resulting deviation in the output variables of a node as a function $\delta f$ of the input value:

$$(y_1^*, \ldots, y_m^*) = f^*(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \oplus \delta f(x_1, \ldots, x_n)\,.$$

The fault $\delta v$ may propagate to other nodes, and we call a node $f \in \mathcal{F}$ *active* in a faulted execution if either $\delta v = 1$ for any input edge $v \in \textsc{in}(f)$ or $f$ is the faulted gate modified by the attacker.

We denote the fault alert by $\Delta$ and the set of variables it checks by $\mathcal{V}_\Delta$, i.e., $\Delta := \bigvee_{v \in \mathcal{V}_\Delta} \delta v = \bigvee_{v \in \mathcal{V}_\Delta} (v \oplus v^*)$. The SIFA attacker collects plaintext-ciphertext samples with $\Delta = 0$, as they receive no output if $\Delta = 1$, and uses this condition to derive information about the value of edges near the faulted node $f^*$.

**Example.** As an example throughout this section, Figure 9.4 lists the operations of a masked implementation of the 3-bit S-box $\chi_3$ together with its computation graph similar to [GSM17]. In the graph, Clone(·) nodes are represented by small bullets. In the circuit on the left-hand side, for compactness, we omit calls $(v, v') = \text{Clone}(v)$ (i.e., variables named $v'$ are always clones) and list up to two nodes per line. When combined with the error detector $\Delta$ that checks the output variables of the circuit, this implementation is susceptible to single-fault SIFA with several possible fault locations. One of these is illustrated in Figure 9.4: If a bitflip is induced as indicated (ϟ) in the input $a_0$ of the

Clone($a_0$) node, then the condition $\Delta = 0$ implies $b_0 \oplus b_1 = b = 0$. We want to protect this implementation against single-fault SIFA by modifying the detector (or the structure of the DAG).

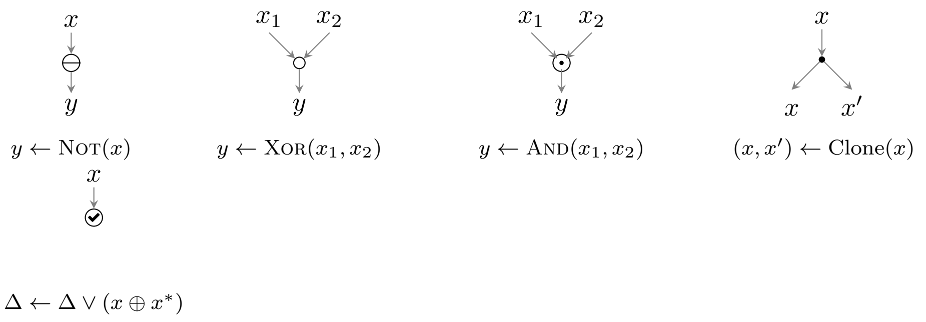

$x$
$y$
$y \leftarrow \textsc{Not}(x)$
$x_1$ $x_2$
$y$
$y \leftarrow \textsc{Xor}(x_1, x_2)$
$x_1$ $x_2$
$y$
$y \leftarrow \textsc{And}(x_1, x_2)$
$x$
$x$ $x'$
$(x, x') \leftarrow \text{Clone}(x)$
$x$
$\Delta \leftarrow \Delta \vee (x \oplus x^*)$

**Figure 9.3:** Nodes for basic circuits in the computation graph examples.

Input: $(a_0, a_1, b_0, b_1, c_0, c_1)$

$$\begin{array}{lll}
\mathrm{T}_0 \leftarrow \overline{b'_0} \odot c'_1 & ; & \mathrm{T}_2 \leftarrow a'_1 \odot b'_1 \\
\mathrm{T}_1 \leftarrow \overline{b'_0} \odot c'_0 & ; & \mathrm{T}_3 \leftarrow a'_1 \odot b'_0 \\
\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus a'_0 & ; & \mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus c'_1 \\
r_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1 & ; & t_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3
\end{array}$$

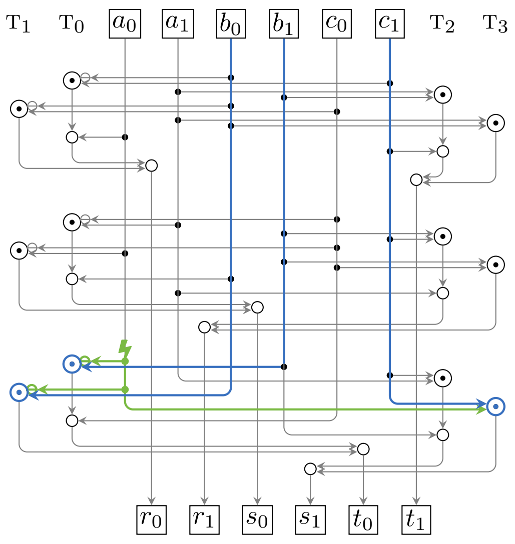
$$\begin{array}{lll}
\mathrm{T}_0 \leftarrow \overline{c'_0} \odot a'_1 & ; & \mathrm{T}_2 \leftarrow b'_1 \odot c'_1 \\
\mathrm{T}_1 \leftarrow \overline{c'_0} \odot a'_0 & ; & \mathrm{T}_3 \leftarrow b'_1 \odot c'_0 \\
\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus b'_0 & ; & \mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus a'_1 \\
s_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1 & ; & r_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3
\end{array}$$

$\lightning a_0$

$$\begin{array}{lll}
\mathrm{T}_0 \leftarrow \overline{a'_0} \odot b'_1 & ; & \mathrm{T}_2 \leftarrow c'_1 \odot a_1 \\
\mathrm{T}_1 \leftarrow \overline{a'_0} \odot b_0 & ; & \mathrm{T}_3 \leftarrow c_1 \odot a_0 \\
\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus c_0 & ; & \mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus b_1 \\
t_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1 & ; & s_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3
\end{array}$$

Output: $(r_0, r_1, s_0, s_1, t_0, t_1)$

**(a)** Circuit, $(v, v') = \text{Clone}(v)$ omitted



**(b)** Computation graph

**Figure 9.4:** Bitflip in masked $\chi_3$ using 2 shares (resharing at the output omitted).

**Criterion for Resistance against Single-Fault SIFA.** Consider a masked implementation with a detection-based countermeasure defined by an error detector $\Delta$ that only returns the result of the computation if $\Delta = 0$. We call the implementation *single-fault SIFA-resistant* if each possible single fault is either detected by $\Delta$ or activates at most one nonlinear node.

To see why this criterion is sufficient, consider a fault $f^*$. The attacker collects plaintext-ciphertext samples with $\Delta = 0$, as they receive no output if $\Delta = 1$. The samples satisfy one of the following two conditions:

- $\delta y = 0$, i.e., no bitflip happened because $\delta f(x_1, \dots, x_n) = 0$. The attacker learns at most these values $x_1, \dots, x_n$. Since the implementation is masked, this information is independent of the native input and output values and thus does not allow the attacker to derive any information on the processed data or keys.
- $\delta y \neq 0$, but the resulting bitflip(s) did not propagate to $\Delta$. The criterion implies that there is at most one active nonlinear node, i.e., either $f$ or another nonlinear node $f'$ with some changed input $v^* = v \oplus \delta v$. The attacker may exploit this differential information to learn the inputs of this active nonlinear node, which are however independent of the native inputs, and will not learn anything from the other, trivial differentials (of nonlinear nodes with zero input difference or of linear nodes).

### 9.4.2 Protection against SIFA using Fine-Grained Detection

We now explore how a masked implementation can be extended with a suitable detector $\Delta$ in order to achieve a single-fault SIFA-resistant implementation.

**Basic Idea.** A straightforward, albeit not very efficient approach to satisfy the single-fault SIFA-resistance criterion follows directly from its definition: We can add local checks for inputs of nonlinear nodes. Assume for instance that we duplicate the implementation and feed the same inputs to both instances. For each nonlinear node $f$ and each of its input edges $v \in \textsc{in}(f)$, we add a check to update the detector $\Delta \leftarrow \Delta \vee (v \oplus v^*)$. We alternatively represent this as a single checking node ✓ in the DAG of a single instance of the implementation. Then, a fault may activate a single nonlinear node $f$ without detection by $\Delta$ (if the attacker faults either the nonlinear node itself or the preceding check), but it cannot activate two nodes, since there are no paths without a check either from any node to two nonlinear nodes or from one nonlinear node to another. Thus, any single-bit fault in one of the two redundant computations or in the additional circuitry for the detector $\Delta$ are either detected or do not leak information to the attacker.

**Reducing Checks.** It is, however, not necessary to check the inputs to all nonlinear nodes separately. For example, in many circuits, most inputs to nonlinear nodes in the DAG are directly cloned from the shares of the inputs, i.e., the node input checks would check the same variable many times. Instead, we want to check only once. In the DAG, this corresponds to a binary subtree rooted in an input variable whose inner nodes are Clone($\cdot$) nodes and whose leaves are other nodes. We refer to edges ending in leaves as twigs and to the other, inner edges as stems. The basic approach checks each twig ending in a nonlinear node and thus precludes a fault that activates this twig in addition to another parent or sibling edge in the DAG. Instead, it is sufficient to check only those twigs whose sibling edge is also a twig (rather than a stem), and to check only one of the two twigs. In other words, we check a variable that serves as input to multiple nodes only once, right before its last use. We call this last check the *sink* of (this part of) the tree, and it will be activated if more than one twig in (this part of) the tree is active. Additionally, we also consider the circuit output variables as sinks, since they will either be checked in the next nonlinear layer, or propagate faults deterministically to the cipher output. As a result, every edge $v$ in the DAG is a *safe* edge that has a sink $s$ such that there is exactly one directed path $v \to s$ and it contains at most one nonlinear node. This implies the single-fault SIFA-resistance criterion of Section 9.4.1.

In the $\chi_3$ example, by checking only such variables and only after they are used for the last time, we can reduce the number of checks to 6 (instead of 24 in the naive approach), i.e., once for each input variable. The result is illustrated in Figure 9.5.

### 9.4.3 Ensuring Fault Propagation

In this section, we discuss under which conditions the fine-grained, local detection of Section 9.4.2 can be replaced by global checks, similar to Section 9.2. We will again use the concept of sink nodes as in Section 9.4.2, in the sense of nodes whose activation will be detected by $\Delta$. However, instead of implementing actual local checks in the sink nodes, these sinks are virtual nodes whose effect on $\Delta$ follows from properties of the cipher or masking approach.

First consider a uniform direct sharing of an invertible S-box. Since the sharing is uniform, the masked circuit is also invertible. As a consequence, for fixed resharing inputs, if any of the intermediate masked S-box output bits are activated by a fault, this will activate at least one bit in the masked cipher output. Thus, if the detection variables $\mathcal{V}_\Delta$ include all masked cipher output variables, then the S-box output variables can serve as sinks – what remains to be done is to ensure that each edge is a safe edge with respect to these sinks, and ideally, to get rid of the requirement to perform redundant computations for the same values of the shares. We first address the latter question. For simplicity, we assume that all nodes except cloning nodes have a single output bit.

Instead of the individual shares of the S-box output bits, we can use the native S-box output as sinks and add corresponding virtual nodes that compute these as sums of the masked S-box outputs to the circuit. Any fault in this native S-box output would activate at least one bit in the native cipher output, so the detection variables $\mathcal{V}_\Delta$ can be reduced to the unmasked values and evaluated for arbitrary resharing inputs.

Now, we still need to ensure that any edge $v$ in the S-box circuit is a safe edge with respect to one of these sinks $s$, i.e., that there is exactly one directed path $v \to s$ and it contains at most one nonlinear node. This may be violated due to cloning nodes (or, generally, nodes with multiple outputs) and due to composition

Input: $(a_0, a_1, b_0, b_1, c_0, c_1)$

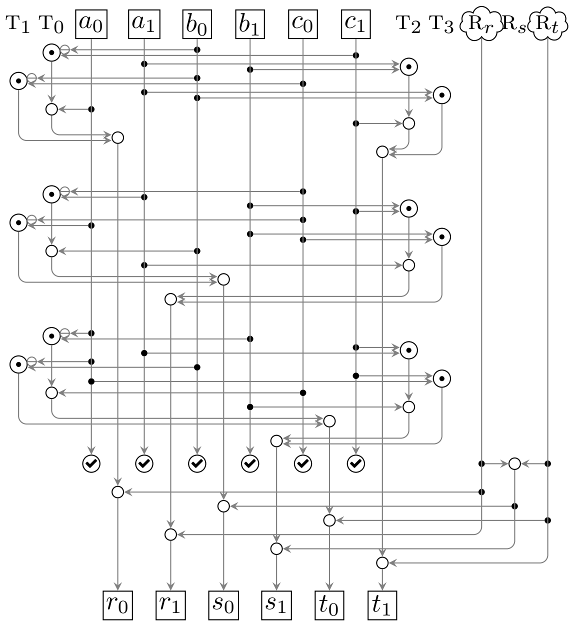
$$\begin{array}{lll}
\mathrm{T}_0 \leftarrow \overline{b'_0} \odot c'_1 & ; & \mathrm{T}_2 \leftarrow a'_1 \odot b'_1 \\
\mathrm{T}_1 \leftarrow \overline{b'_0} \odot c'_0 & ; & \mathrm{T}_3 \leftarrow a'_1 \odot b'_0 \\
\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus a'_0 & ; & \mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus c'_1 \\
r_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1 & ; & t_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3
\end{array}$$

$$\begin{array}{lll}
\mathrm{T}_0 \leftarrow \overline{c'_0} \odot a'_1 & ; & \mathrm{T}_2 \leftarrow b'_1 \odot c'_1 \\
\mathrm{T}_1 \leftarrow \overline{c'_0} \odot a'_0 & ; & \mathrm{T}_3 \leftarrow b'_1 \odot c'_0 \\
\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus b'_0 & ; & \mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus a'_1 \\
s_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1 & ; & r_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3
\end{array}$$

$$\begin{array}{lll}
\mathrm{T}_0 \leftarrow \overline{a'_0} \odot b'_1 & ; & \mathrm{T}_2 \leftarrow c'_1 \odot a'_1 \\
\mathrm{T}_1 \leftarrow \overline{a'_0} \odot b'_0 & ; & \mathrm{T}_3 \leftarrow c'_1 \odot a'_0 \\
\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus c'_0 & ; & \mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus b'_1 \\
t_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1 & ; & s_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3 \\
\mathrm{R}_s \leftarrow \mathrm{R}'_r \oplus \mathrm{R}'_t & ; & \textsc{Checks} \\
r_0 \leftarrow r_0 \oplus \mathrm{R}'_r & ; & s_0 \leftarrow s_0 \oplus \mathrm{R}'_s \\
t_0 \leftarrow t_0 \oplus \mathrm{R}'_t & ; & r_1 \leftarrow r_1 \oplus \mathrm{R}_r \\
s_1 \leftarrow s_1 \oplus \mathrm{R}_s & ; & t_1 \leftarrow t_1 \oplus \mathrm{R}_t
\end{array}$$

Output: $(r_0, r_1, s_0, s_1, t_0, t_1)$

**(a)** Circuit, $(v, v') = \text{Clone}(v)$ omitted



**(b)** Computation graph

**Figure 9.5:** Single-fault SIFA-resistant $\chi_3$ using 2 shares, with local checks. Checks is short for the error detecting sub-circuits $\Delta \leftarrow \Delta \vee (v \oplus v^*)$ for $v \in \{a_0, a_1, b_0, b_1, c_0, c_1\}$.

of nonlinear nodes within an S-box. If the circuit contains such a composition of nonlinear nodes, it needs to be decomposed into smaller, bijective circuits with nonlinear depth 1 first, similar to Section 9.2. For cloning, we consider the cloning subtree as in Section 9.4.2. We need to ensure that whenever two twigs in this tree activate, a sink $s$ activates. In particular, this implies that for every cloning node $b$, there must be a sink $s$ such that there is a unique path $b \to s$, and this path contains only linear nodes. This may require restructuring the tree such that at least one of the last two uses of a variable (the tree root) is in a linear node, taking care that the modifications do not invalidate the security of the masked implementation.

The approach is easy to apply to the $\chi_3$ example. We perform the following modifications to the circuit from Figure 9.4 so that each edge is now a safe edge:

1. Delay $r_0 \leftarrow r_0 \oplus a_0$ and $t_1 \leftarrow t_1 \oplus c_1$ until the very end,
2. Delay $r_1 \leftarrow r_1 \oplus a_1$, $s_0 \leftarrow s_0 \oplus b_0$, $s_0 \leftarrow s_1 \oplus b_1$, and $t_0 \leftarrow t_0 \oplus c_0$ (optional),
3. Move the resharing to preserve security of the masking.

The resulting circuit in Figure 9.6 shows similarities with the Toffoli-based implementation of $\chi_3$ in Section 9.2, but there are still significant differences; most notably, the necessity for resharing variables $\mathrm{R}_r, \mathrm{R}_t$ and the lower depth of the circuit in Figure 9.6.

### 9.4.4 Towards Protection against Multiple Faults

So far, we focused only on single-fault SIFA attackers and corresponding countermeasures. Both the attack approach and the countermeasure with local checks can be generalized to a multi-fault attacker who faults up to $d$ basic circuits (nodes).

Consider a circuit protected by $d$th-order masking with $d+1$ or more shares, i.e., an attacker who learns up to $d$ shares of any variable or observes up to $d$ basic circuits still does not gain any information on any native value. Let the circuit be implemented with at least $d+1$ redundant computations and an error detector $\Delta$ of at least $d$ bits. For simplicity, assume that each of the attacker's $d$ faults is a bitflip fault on one of the intermediate variables. We call the implementation *$d$-fault SIFA-resistant* if each possible $d$-bit fault is either detected by $\Delta$ or activates at most $d$ nonlinear nodes in total.

This criterion can, for instance, be satisfied by checking all inputs to nonlinear nodes with the following construction. We use $d+1$ redundant computations and an $n_\Delta$-bit error detector $\Delta = (\Delta_1, \dots, \Delta_{n_\Delta})$, where $n_\Delta = d$ for odd $d$ and $n_\Delta = d+1$ for even $d$. For each relevant input edge, we clone $d$ times to update $d$ different error detector bits $\Delta_i$ with the differences to all $d$ other computations. In other words, we compute all $\binom{d+1}{2}$ differences in this bit between any two redundant computations and ensure that for each computation, each of the $d$ comparisons activates a different detector $\Delta_i$. Distributing the $\binom{d+1}{2}$ differences to the various $\Delta_i$ corresponds to an edge coloring problem with $n_\Delta$ colors in the complete graph with $d+1$ vertices, which is easy to solve. Then, activating $k$ nodes in one computation without detection by $\Delta$ requires at least $\min(k, d+1)$ faults: each node can either be activated without triggering $\Delta$ by placing a fault between the check (with its cloning) and the nonlinear node; or it can be activated while triggering $d$ error detector bits $\Delta_i$, each of which requires either a fault in the corresponding computation or faulting $\Delta_i$ directly to eliminate. Thus, in summary, at least $d + 1$ faults would be required in order to activate $d + 1$ or more nonlinear nodes and thus learn $d + 1$ shares of any variable to deduce information on its native value.

Clearly, without further optimizations, this approach can only be practical for very small protection order $d$. Since the size of each masked implementation grows quadratically in $d$, and the checking cost per nonlinear node in this implementation also grows quadratically in $d$, the construction is only of theoretic interest for larger $d$.

## 9.5 Implementation

In this section, we describe our experimental results on correctness and performance and discuss how well our circuit model matches the reality of hardware and software implementations.

### 9.5.1 Formally Verifying the Masking of Toffoli-based Circuits

To gain additional trust in the soundness of our masking, we verified the circuits using a tool-assisted approach. More specifically, we make use of `maskVerif`, a tool by Barthe et al. [Bar+15] for formally verifying masking schemes. `maskVerif` takes as input a (masked) circuit description that mainly consists of simple logical operations such as AND, XOR, or NOT. The interface of the circuit can consist of (shared) variables, as well as additional randomness. Given such a masked circuit, `maskVerif` can verify if the implemented masking is indeed correct in a specified leakage model and with a specified protection order. For more details about `maskVerif` we refer to the original publication [Bar+15] and the tool's website [GBB].

We verified the correctness of the masking of our Toffoli-based circuits for S-boxes from AES and KECCAK using `maskVerif`. Therefore, we converted the circuits from Circuit 9.8.b (5-bit $\chi$) and Circuit 9.9.b (AES S-box) into a `maskVerif`-compatible format and successfully verified their first-order security in the probing model and in the presence of (propagation delay) glitches[1]. Note that, for implementations in hardware and software, additional design considerations are necessary in practice, which we discuss in part in Section 9.5.3 and Section 9.5.4.

### 9.5.2 Benchmarks and Practical Evaluation

In this section, we give a preliminary evaluation of the costs of our proposed countermeasure. We start with a theoretical estimation of the costs and then

[1]The used code is available at https://github.com/sifa-aux/countermeasures

we demonstrate the effectiveness and low overhead of our countermeasure by implementing KECCAK-$f$[200] on a low-end 8-bit AVR XMEGA128D4 microprocessor as an example. We consider both our Toffoli-based masked S-box (Circuit 9.8.b) and a traditionally masked S-box using domain-oriented masking (DOM) [GSM17]. We then use these implementations for comparing their runtime and code size and for verifying the SIFA-protection of Circuit 9.8.b in a practical evaluation.

**Cost Estimation.** First, let us discuss protection against single-fault SIFA. The costs of our countermeasures can be roughly split into two parts: the cost of masking and the cost of redundancy. Let us first start with the cost of redundacy. Since we perform plain double execution, we get roughly a factor two of overhead. This overhead is in terms of space due to roughly the doubled number of registers needed to store the state and might be in time, e.g., in software implementations where everything has to be executed twice, or in hardware if designers decide to not duplicate the whole circuit, but perform time redundancy instead. So, if we consider that masking is needed anyway and our masking strategies are not worse than common strategies, we get the same overhead that would be needed for fault protection by duplication.

Now, let us take a look at the efficiency of our masking proposal. Let us start with the way of masking that we introduce in Section 9.2 and Section 9.3. This way of masking relies on describing a cipher in terms of incomplete permutation circuits, and hence, mainly relies on masked versions of the Toffoli gate and related constructions. As it can be seen in Figure 9.1, such a description comes even quite natural in the case of AES and does not incur a prohibitive increase in the number of operations needed to compute an S-box. In addition, those masked implementations are secure without the need of online randomness. As an example, let us have a look at KECCAK's S-box and compare a DOM implementation [GSM17] with masking the Toffoli-based description given in Circuit 9.8.b.

Let us start with the DOM implementation of KECCAK's S-box. To mask this with two shares, we need 5 DOM ANDNOTS plus 10 XORS. Hence, in total, we roughly need 20 2-bit ANDS/ANDNOTS, 30 2-bit XORS, and 5 random bits for calculating KECCAK's S-box. In contrast, in Circuit 9.8.b, we need 5 calls to $p_{\chi S}$ plus two XORS. Thus, we have a total of 20 2-bit ANDS/ANDNOTS, 22 2-bit XORS, and 0 random bits for calculating KECCAK's S-box. Hence, in terms of operations, the Toffoli-based version has a slight advantage. However, we need storage for one more share at the input. We will see how this compares in a practical implementation in Section 9.5.2.

The overhead incurred by following the strategy of Section 9.4 depends on a number of factors. We focus again on the DOM implementation of KECCAK's S-box (with 20 ANDS/ANDNOTS, 30 XORS, and 5 resharing bits), both as a point of comparison and as the target circuit for the countermeasure. First consider the approach with local checks from Section 9.4.2, Figure 9.5. On top of the $2 \times 20 + 2 \times 30 = 100$ gates for duplicate execution of this S-box, our countermeasure adds 10 Xors and 10 Ors for the checking circuit, plus an additional state of 1 bit globally. It is necessary to either execute both duplicate instances in parallel or invest additional space to keep track of the checked intermediate values. Other metrics, such as the circuit depth, remain essentially unchanged. With the improvements for global checks from Section 9.4.3, Figure 9.6, the countermeasure comes with zero overhead compared to the basic DOM implementation with simple redundancy.

When considering the ideas from Section 9.4.4 for higher-order protection, the overheads are more substantial. When considering a straightforward application without any optimizations, this can be estimated as follows when focusing only on nonlinear gates (which are responsible for the overhead). For protection against $d$-fault SIFA, we need a masked implementation of $d$th order with (at least) $d+1$ shares, as well as $d+1$-fold redundant execution with an error detector of $n_\Delta \in \{d, d+1\}$ bits. With the higher-order DOM approach [GSM17], each And-gate of an unprotected S-box circuit corresponds to $(d+1)^2$ Ands plus $(d+1)^2$ Xors in a masked circuit and requires up to $d \cdot (d+1)/2$ resharing bits. These gates are duplicated $d+1$ times for redundancy, resulting in a total of $2 \times (d+1)^3$ gates (Ands and Xors). Our countermeasure adds a check (1 Xor plus 1 Or) for each of the inputs of these Ands, for a total of $(d+1)^2 \times 2 \times \frac{(d+1)\cdot d}{2} \times 2 = 2d \times (d+1)^3$ gates. This corresponds to an overhead of a factor of $d$ in the number of gates compared to DOM masking with redundancy (only for the nonlinear gates, the linear gates add no overhead). We expect that for concrete circuits, significant optimizations similar to Section 9.4 are possible. Still, we consider this approach to be primarily of theoretical interest.

**Practical Benchmarks.** To keep the comparison as fair as possible, we opted to take the compact C implementation of Keccak-$f$[200] from the eXtended Keccak Code Package [Ber+a] as the basis for our implementations. We then simply duplicated all linear operations and replaced the S-box by Toffoli/DOM masked assembly versions[2]. The resulting performance numbers are hence not necessarily representative for the maximum performance on 8-bit platforms but very representative for a direct comparison of the two S-box implementations. The resulting numbers for our comparison are shown in Table 9.2. We measured the runtime without the runtime cost of a PRNG and discuss the needed amount of random bits as a separate metric. From the presented numbers, it is easy to see that our Toffoli-based S-box is on par with the DOM variant in terms of runtime, and binary size. Please note that the AVR XMEGA128D4 has not been designed for cryptographic purposes, and hence, might allow for side-channel attacks due to violations of the seperation of the shares. However, what we show in the next section is that our proof-of-concept implementation still provides protection against single fault SIFA.

**Evaluation of SIFA Resistance.** We also evaluated the SIFA resistance of our designs by means of simulated fault inductions and a practical evaluation

[2]The used code is available at `https://github.com/sifa-aux/countermeasures`

**Table 9.2:** Comparison of computing 18 rounds of KECCAK-$f$[200] using different implementations. Numbers do not include generation of randomness: The DOM approach requires 5 random bits per S-box ($200+200\times18 = 3800$ bits in total), which can be reduced with techniques such as Changing of the Guards. The Toffoli approach requires just $200 + 40$ in total when implemented as proposed for Circuit 9.8.b.

| Implementations | Runtime w/o PRNG (Clock Cyles) | Binary Size (Bytes) |
|---|---|---|
| Empty main.c `-Os` | 0 | 5 385 |
| Compact C with ASM S-box `-Os` | 49 371 | 6 939 |
| Masked with ASM DOM S-box `-Os` | 107 617 | 9 648 |
| Masked with ASM Toffoli S-box `-Os` | 109 753 | 9 632 |
| Compact C with ASM S-box `-O3` | 38 455 | 9 811 |
| Masked with ASM DOM S-box `-O3` | 88 332 | 12 434 |
| Masked with ASM Toffoli S-box `-O3` | 87 426 | 12 385 |

on an AVR XMEGA128D4 microprocessor using clock glitches. The evaluation methodology is the same as the one that was used in Section 8.2.2. We take our Toffoli masked assembly S-box implementation (the same that is used in the benchmarks), target one instruction with a fault induction, call the S-box with every possible input, and check whether the correct unmasked S-box outputs follow a uniform distribution or not. This procedure is then repeated for every instruction within the S-box.

According to our practical evaluation, where clock glitches cause effects like memory corruption or instruction skips, no instruction within our S-box implementation is susceptible to SIFA. This result is backed up by our simulated fault induction experiments where we simulate the effect of stuck-at faults and bitflips for which we do not own a set-up to reproduce them in practice.

### 9.5.3 From Abstract Models to Software Implementations

In Section 9.1, we have introduced an abstraction model and explicitly defined what faults are in this model. When considering the implementation of our circuits in software, it would seem that even in the most trivial implementations, it is ensured that basic circuits are nicely separated and hence, fault attacks and also side-channel attacks cannot be a threat. However, the reality is more subtly nuanced. Hence, we want to discuss what has to be considered when implementing our circuits in real software implementations and which faults on software implementations are covered by considering faults on basic circuits.

As mentioned in Section 9.1, we consider circuit faults in a single basic circuit instance. This directly corresponds to faults in software that directly manipulate values of variables stored in registers of a CPU [Sel+18], change a variable in memory before it is loaded, or even target the load of a variable from memory [Col+19]. Furthermore, it also covers cases where a fault, like a clock glitch, changes the outcome of a computation. However, what is notably only partially covered is the case of an instruction skip, meaning that an operation is not performed and the register values are kept untouched. This can lead to cases where the boundaries between basic circuits are violated. This is especially a threat if an implementation of a basic circuit uses registers in addition to the registers storing the shares in order to store results of intermediate computations. However, potential negative effects of a clock glitch can be mitigated by always initializing an additional register to 0 before use, or by performing instructions on the shares in place (e.g., $a_0 = a_0 \oplus b_0$) whenever possible.

What is not covered by our considerations are faults that change the execution flow of a program to a greater extent than skipping the single instructions, like manipulating the program counter. Furthermore, we do not consider the use of loops and conditional statements apart from their usage in detecting faults. What is also not considered are faults that change the operands used in operations. All these faults have in common that they may totally change the program that is executed to a point where the key is just put out in plain. Such faults have to be prevented by other means.

Furthermore, our model considers a single permanent fault, e.g, permanently faulting a lookup table, as multiple faults. However, we advise not to use implementations with lookup tables.

A notable case that is not considered in our abstraction are LOAD and STORE instructions from memory to registers. In the simplest case, there are enough registers to store all necessary variables so that during a cryptographic computation, no LOADs and STOREs are needed. However, if this is not the case and a variable has to be reloaded, this might cause problems. For instance, let us consider the circuit shown in Circuit 9.5.b. Here, $b_0$ is just read and never written. So if we do not consider fault protection, it can be assumed that the register $b_0$ can just be overwritten, since the value can be reloaded from memory anyway. If we consider our fault protection mechanisms, this means that a faulted value in register $b_0$ might vanish, which in turn would allow SIFA again. To prevent this in general, we have to assume that values are changed and write them back to memory if their use is later required.

Furthermore, registers have to be properly initialized before usage. The problem with uninitialized registers is that shares can be combined, which leads to exploitable leakage, or, in the case of a clock glitch, to an unmasked use of a variable. Typically, the problem with uninitialized registers can be easily solved by always writing 0 to them before the result of a computation is stored.

Finally, we want to note that modern ciphers can usually be implemented in a bit-sliced manner, meaning that for a system using $x$-bit registers, a single computation, and thus, a single fault like a clock glitch leads to a single fault in up to $x$ S-boxes. For ciphers that consist of layers applying many small bijective S-boxes in parallel to the state, we can define basic circuits to work on bit-vectors instead of single bits. This implies that injecting a single fault in several of these parallel S-boxes in a single layer causes no issues with respect to our strategy of Section 9.2, since these faults will correspond to a single circuit fault of an incomplete circuit.

### 9.5.4 From Abstract Models to Hardware Implementations

In general, our abstraction as circuit lends itself quite naturally to dedicated hardware implementations, but requires additional considerations. In particular, one needs to take into account the effects of glitches. Glitches are the result of the behavior of the physical layout and are thus unavoidable. Since signals do not propagate evenly through a hardware-circuit (due to differences in the capacitance of wires, different wire lengths, manufacturing imperfections, et cetera.) the output of gates could change (glitch) several times before reaching a stable logic state. In the context of a fault induction, also faults can "glitch". As a result, in each clock cycle there is sequence of transitional states in the physical circuit that depend on combinations of variables that differ from the ones that the circuit should finally compute.

These effects imply that the behavior of a hardware-circuit cannot be controlled by just using combinatorial logic gates. Using registers limits these transitional effects in the sense that it puts barriers between combinatorial blocks. Registers stabilize a signal before entering the next logic gates through a separation in different clock cycles. The cost for the gained control over the signals is not only the increased gate count, but also the evaluation of the hardware-circuit requires more clock cycles and thus, the latency increases.

Hence, when instantiating our abstract circuits in hardware, registers are required at several places to separate the basic circuits and ensure resistance to glitching effects. This is no different for other masking methods. For instance, TI implementations [NRR06; NRS11] use registers after each uniformly shared function and the DOM scheme [GMK16] uses a register stage in each shared nonlinear gate to hinder security-critical glitches from propagating into the next shared function which could violate the security requirements.

Figure 9.7 shows a masked Toffoli gate in hardware which already includes the required registers (FF) for a glitch resistant first-order side-channel protection. Furthermore, this variant also resists single-fault SIFA attacks. The upper two registers are required to hinder the propagation of glitches that could violate the side-channel resistance of the implementation. The lower four registers are the relevant ones for protection against SIFA. Again, no share can be used twice in two different basic circuits within the same clock cycle. This would be the case when switching the order of the multiplication of $b_1$ and $c_1$ with $b_1$ and $c_0$, for instance, because a single fault of the input $c_0$ would affect both multiplications with the two shares of $b$.

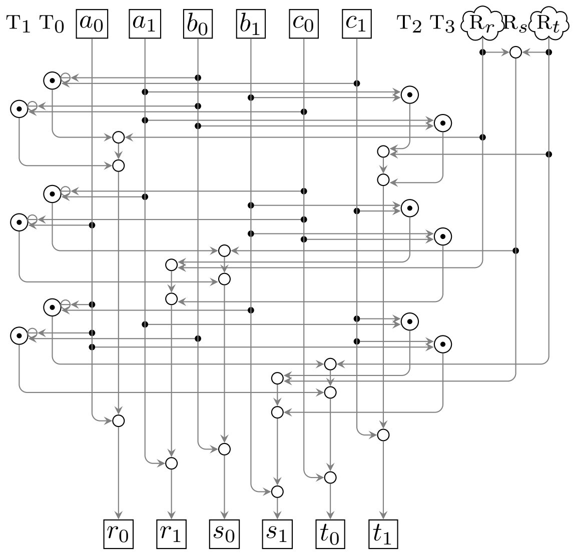
A secure hardware variant of the threshold Toffoli gate is shown in Figure 9.8. The registers ensure that a single fault cannot influence all shares of variables that are fed into nonlinear AND gates without detecting it at the output.

## 9.6 Conclusion

In this chapter, we have proposed two different approaches to counteract SIFA on an algorithmic level. First, we show that by relying on Toffoli gates for the nonlinear operations in the implementation of masked ciphers, we can construct circuits where a single fault in the computation of the cipher is either (1) not exploitable by SIFA or (2) detectable via redundant computations that are typically implemented to cope with other fault attacks like DFA. This approach can be implemented efficiently, and its applicability was shown for 3-bit, 4-bit, and 5-bit S-boxes. Additionally, we show how this approach could be extended to the AES S-box using the Toffoli gate for bigger fields and fine-grained detection on S-box level that can be implemented efficiently for implementations using duplication and is also quite efficient for implementations using time-redundancy when using a linear checksum. We verified the correctness of the masking of our Toffoli-based circuits for S-boxes from AES and KECCAK using `maskVerif`. Finally, we show how this approach based on fine-grained detection can be generalized to protect arbitrary masked circuits, and how it can be extended to cope with multi-fault SIFA, albeit at a higher implementation cost.

```
Input: $(a_0, a_1, b_0, b_1, c_0, c_1)$
$\mathrm{R}_s \leftarrow \mathrm{R}'_r \oplus \mathrm{R}'_t$
$\mathrm{T}_0 \leftarrow \overline{b'_0} \odot c'_1$ ;  $\mathrm{T}_2 \leftarrow a'_1 \odot b'_1$
$\mathrm{T}_1 \leftarrow \overline{b'_0} \odot c'_0$ ;  $\mathrm{T}_3 \leftarrow a'_1 \odot b'_0$
$r_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{R}'_r$ ;  $t_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{R}'_t$
$r_0 \leftarrow r_0 \oplus \mathrm{T}_1$ ;  $t_1 \leftarrow t_1 \oplus \mathrm{T}_3$
$\mathrm{T}_0 \leftarrow \overline{c'_0} \odot a'_1$ ;  $\mathrm{T}_2 \leftarrow b'_1 \odot c'_1$
$\mathrm{T}_1 \leftarrow \overline{c'_0} \odot a'_0$ ;  $\mathrm{T}_3 \leftarrow b'_1 \odot c'_0$
$s_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{R}'_s$ ;  $r_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{R}_r$
$s_0 \leftarrow s_0 \oplus \mathrm{T}_1$ ;  $r_1 \leftarrow r_1 \oplus \mathrm{T}_3$
$\mathrm{T}_0 \leftarrow \overline{a'_0} \odot b'_1$ ;  $\mathrm{T}_2 \leftarrow c'_1 \odot a'_1$
$\mathrm{T}_1 \leftarrow \overline{a'_0} \odot b'_0$ ;  $\mathrm{T}_3 \leftarrow c'_1 \odot a'_0$
$t_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{R}_t$ ;  $s_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{R}_s$
$t_0 \leftarrow t_0 \oplus \mathrm{T}_1$ ;  $s_1 \leftarrow s_1 \oplus \mathrm{T}_3$
$r_0 \leftarrow r_0 \oplus a_0$ ;  $t_1 \leftarrow t_1 \oplus c_1$
$s_0 \leftarrow s_0 \oplus b_0$ ;  $r_1 \leftarrow r_1 \oplus a_1$
$t_0 \leftarrow t_0 \oplus c_0$ ;  $s_1 \leftarrow s_1 \oplus b_1$
Output: $(r_0, r_1, s_0, s_1, t_0, t_1)$
```

**(a)** Circuit, $(v, v') = \mathrm{Clone}(v)$ omitted



**(b)** Computation graph

**Figure 9.6:** Single-fault SIFA-resistant $\chi_3$ using 2 shares, with global checks.

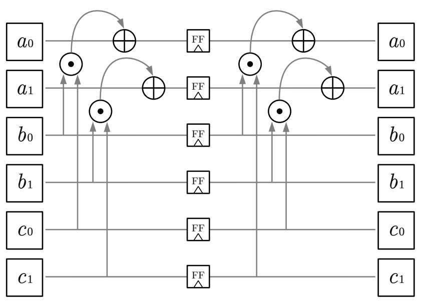


**Figure 9.7:** Masked and single-fault SIFA-protected Toffoli gate in hardware.

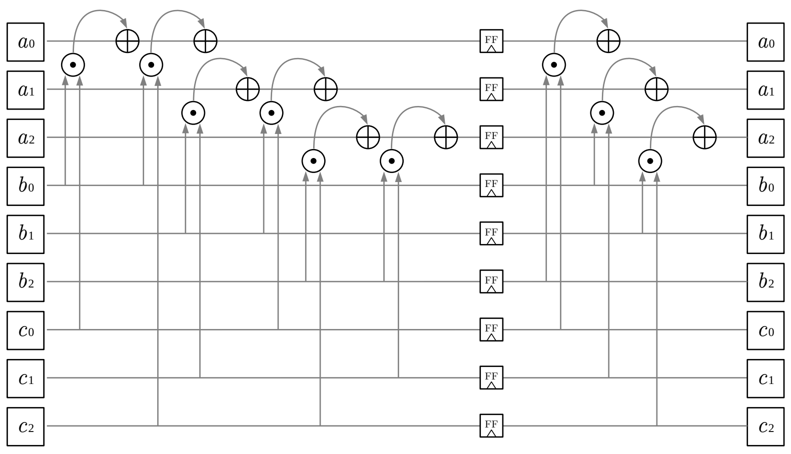


**Figure 9.8:** 3-share TI and single-fault SIFA-protected Toffoli gate in hardware.

# 10 Conclusion and Outlook

Don't wish it were easier, wish you were better.

*Chief - Animal Crossing*

In this thesis, we have analyzed and advanced the state-of-the-art of implementations security by presenting new attack and defense techniques that are relevant, e.g., to cryptographic algorithms currently competing in NIST's standardization processes for lightweight or post-quantum cryptography.

In the first part, we have focused on passive implementation attacks on recent proposals for quantum computer secure encryption schemes. We have presented a power analysis attack capable of recovering the complete private key of a (higher-order masked) implementation of a lattice-based encryption scheme from a single power trace. Our attack clearly shows that, even though lattice-based cryptography is vastly different from established RSA and ECC constructions, profiled attacks should not be neglected, and masking countermeasures alone are not necessarily sufficient. In the subsequent chapter, we have presented an improved version of this attack that requires a significantly less powerful attacker. The practicality of this work is demonstrated by performing it on an off-the-shelf microprocessor.

In the second part of this thesis, we have shifted our focus to active implementation attacks and present statistical ineffective fault attacks (SIFA), a fault attack technique capable of circumventing typical algorithmic countermeasures that were previously believed to offer sufficient protection against fault attacks. A particularly interesting property of SIFA is the fact that, from an attacker's perspective, only very limited knowledge about the attacked device is required.

We have also presented algorithmic defense mechanisms that can prevent SIFA with fairly low overheads and, coincidentally, also allow the construction of particularly efficient masking schemes to counteract certain types of power analysis attacks.

Even though implementation security already has a long history that reaches back about two decades, substantial progress has been and is still being made. In the following, we want to share our perspective on how this research field might progress over the next few years and what questions remain unanswered after this thesis.

**Performance and Cost.** From an industry perspective, one of the main challenges with implementation security is the accompanied overhead, in terms of area/code size and runtime, that can increase by several orders of magnitude compared to plain implementations [Bel+20]. The importance of efficiency is also reflected by the NIST's currently ongoing standardization process for lightweight cryptography [NIS18]. Here, the goal is to select future standards for authenticated encryption that should not only outperform current AES-based schemes but also allow the addition of countermeasures against implementation attacks at low cost. One way to approach this problem is to use cryptographic schemes based on lightweight building blocks that are comparably cheap to protect against implementation attacks. Nearly all candidates in the final round of the standardization process follow this approach, e.g., by using cryptographic building blocks with low-degree nonlinear layers that keep the overhead of masking comparably low. Another approach to improve efficiency involves the usage of cryptographic modes that can either reduce the attack surface of certain implementation attacks or prevent them entirely. The attack surface of DPA attacks can, for example, be reduced by using cryptographic modes allowing leveled implementations that restrict the need for algorithmic countermeasures to only certain parts of a cryptographic computation [PSV15; Dob+21]. Entire mode-level protection against certain attacks can be achieved, e.g., by using GGM tree constructions [GGM86; Dob+20] that essentially restrict the attacker to only observing the processing of two different inputs under the same key, which prevents attacks like DPA or SIFA. It will be interesting to see how much further efficiency can be improved, possibly using entirely new ideas that have not yet been explored by the research community.

**Open Hardware.** One quite recent trend in the semiconductor industry is a push towards open hardware, as shown by projects like OpenTitan [Ope] or Caliptra [Cal] that are backed up by big companies such as AMD, Google, Microsoft, and Nvidia. The goal of these projects is the design of (mostly) open-source *root-of-trust* chips that can be used, e.g., to realize smart cards, trusted execution environments, or hardware security modules. While today's cryptography already follows Kerckhoff's principle – a cryptosystem should be secure even if everything about the system, except the key, is public knowledge – the same cannot be said about their concrete implementations in the context of implementation attacks. Here, concrete knowledge about the design of a cryptographic device can certainly help to either improve the performance or significantly reduce the practical effort of implementation attacks, especially in the case of profiled power analysis and fault attacks. To compensate for that, a higher baseline of defense will be needed for open devices if they want to provide the same security as their closed-source counterparts. Nevertheless, one advantage of open devices in this context is the ability of, e.g., research communities to study them in a level of detail that was not easy to do before. On top of that, an accompanying (mostly) open ecosystem for the design/manufacturing of such devices may also give new research opportunities or increase trust in a product from a consumer perspective. It will be interesting to see how "open" such devices can be and what security guarantees they can provide in a couple of years when they are projected to first enter the market.

**Formal Verification.** The topic of formal verification in the context of implementation security is still a relatively young research field. One particularly interesting application of formal verification methods is for checking the correct separation of shares during a masked cryptographic computation on hardware circuits. Pure algorithmic correctness of a masked cipher description is generally not sufficient for practical security due to physical defaults of hardware circuits like signal transitions/glitches that can temporarily leak information about multiple shares. Formal verification methods can help identify such issues already at a pre-silicon stage, e.g., by analyzing the netlist of a masked hardware circuit [Blo+18; Bar+19; Gig+21; KSM20]. One current limitation of such tools is scalability, as they are often based on model-counting techniques that generally do not scale well with circuit size. Nevertheless, they can already be used to make strong arguments about the security for one/few rounds of a masked cipher, which generally gives a strong indication of the security of the entire masked cipher design. Alternatively, if masking-related security statements about an entire masked cipher design are desired, one can either check somewhat weaker properties [MM22], or make use of dedicated composable masking schemes that are efficient to verify (also regarding physical defaults) but come at the cost of increased area/code-size, energy, or runtime requirements [Bar+16; Fau+18; CS20]. It will be interesting to see how much further the performance of generic masking verification tools can be improved and, in contrast, how much the overhead of composable masking schemes can be reduced.

✳ ✳ ✳

# List of Contributions

## Publications

In the following, we list the author's scientific publications that are part of this thesis.

[Dae+20a] Joan Daemen, Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Florian Mendel, and Robert Primas. "Protecting against Statistical Ineffective Fault Attacks." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.3 (2020), pp. 508–543.

[Dob+18a] Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Stefan Mangard, Florian Mendel, and Robert Primas. "Statistical Ineffective Fault Attacks on Masked AES with Fault Countermeasures." In: *ASIACRYPT (2)*. Vol. 11273. Lecture Notes in Computer Science. Springer, 2018, pp. 315–342.

[Dob+18b] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. "SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 547–572.

[Dob+18c] Christoph Dobraunig, Stefan Mangard, Florian Mendel, and Robert Primas. "Fault Attacks on Nonce-Based Authenticated Encryption: Application to Keyak and Ketje." In: *SAC*. Vol. 11349. Lecture Notes in Computer Science. Springer, 2018, pp. 257–277.

[PP19] Peter Pessl and Robert Primas. "More Practical Single-Trace Attacks on the Number Theoretic Transform." In: *LATINCRYPT*. Vol. 11774. Lecture Notes in Computer Science. Springer, 2019, pp. 130–149.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. "Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption." In: *CHES*. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 513–533.

## Further Collaborations

In the following, we list further collaborations that led to scientific publications which are *not* part of this thesis.

[Blo+22] Roderick Bloem, Barbara Gigerl, Marc Gourjon, Vedad Hadzic, Stefan Mangard, and Robert Primas. “Power Contracts: Provably Complete Power Leakage Models for Processors.” In: *CCS*. ACM, 2022, pp. 381–395.

[DMP22] Christoph Dobraunig, Bart Mennink, and Robert Primas. “Leakage and Tamper Resilient Permutation-Based Cryptography.” In: *CCS*. ACM, 2022, pp. 859–873.

[Dob+20] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, Bart Mennink, Robert Primas, and Thomas Unterluggauer. “Isap v2.0.” In: *IACR Trans. Symmetric Cryptol.* 2020.S1 (2020), pp. 390–416.

[ENP19] Maria Eichlseder, Marcel Nageler, and Robert Primas. “Analyzing the Linear Keystream Biases in AEGIS.” In: *IACR Trans. Symmetric Cryptol.* 2019.4 (2019), pp. 348–368.

[Gig+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: *USENIX Security Symposium*. USENIX Association, 2021, pp. 1469–1468.

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. “Secure and Efficient Software Masking on Superscalar Pipelined Processors.” In: *ASIACRYPT (2)*. Vol. 13091. Lecture Notes in Computer Science. Springer, 2021, pp. 3–32.

[GPM23] Barbara Gigerl, Robert Primas, and Stefan Mangard. “Formal Verification of Arithmetic Masking in Hardware and Software.” In: *ACNS* (2023).

[Ham+21] Mike Hamburg, Julius Hermelink, Robert Primas, Simona Samardjiska, Thomas Schamberger, Silvan Streit, Emanuele Strieder, and Christine van Vredendaal. “Chosen Ciphertext k-Trace Attacks on Masked CCA2 Secure Kyber.” In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.4 (2021), pp. 88–113.

[HPB21a] Vedad Hadzic, Robert Primas, and Roderick Bloem. “Proving SIFA Protection of Masked Redundant Circuits.” In: *ATVA*. Vol. 12971. Lecture Notes in Computer Science. Springer, 2021, pp. 249–265.

[HPB21b] Vedad Hadzic, Robert Primas, and Roderick Bloem. “Proving SIFA Protection of Masked Redundant Circuits (Extended Version).” In: *CoRR* abs/2107.01917 (2021).

[KPP20] Matthias J. Kannwischer, Peter Pessl, and Robert Primas. “Single-Trace Attacks on Keccak.” In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.3 (2020), pp. 243–268.

[Nag+22] Rishub Nagpal, Barbara Gigerl, Robert Primas, and Stefan Mangard. “Riding the Waves Towards Generic Single-Cycle Masking in Hardware.” In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.4 (2022).

[SP20] Stefan Steinegger and Robert Primas. “A Fast and Compact RISC-V Accelerator for Ascon and Friends.” In: *CARDIS*. Vol. 12609. Lecture Notes in Computer Science. Springer, 2020, pp. 53–67.

[Vaf+22] Navid Vafaei, Sara Zarei, Nasour Bagheri, Maria Eichlseder, Robert Primas, and Hadi Soleimany. “Statistical Effective Fault Attacks: The Other Side of the Coin.” In: *IEEE Trans. Inf. Forensics Secur.* 17 (2022), pp. 1855–1867.

✳ ✳ ✳

# A
# Descriptions of Selected Cryptographic Algorithms

We now give descriptions of cryptographic algorithms that are frequently used as discussion examples in Part I and Part II respectively. In Appendix A.1 we describe the symmetric block cipher AES. In Appendix A.2 we describe a asymmetric lattice-based encryption scheme that CRYSTALS-KYBER [Bos+18a] is based on.

## A.1 The Advanced Encryption Standard

The Advanced Encryption Standard (AES), designed by Joan Daemen and Vincent Rijmen [DR20], is an iterated block cipher that consists of the repeated applications of round transformations on a state. AES has a fixed state size of 128 bits, represented as 16 bytes $x_0, x_1, \ldots, x_{15}$ that are arranged in a $4 \times 4$ array as follows:

$$\begin{bmatrix} x_0 & x_4 & x_8 & x_{12} \\ x_1 & x_5 & x_9 & x_{13} \\ x_2 & x_6 & x_{10} & x_{14} \\ x_3 & x_7 & x_{11} & x_{15} \end{bmatrix}$$

While AES supports key sizes of 128, 192, or 256 bits, we restrict our description to the AES-128 variant using a 128-bit key. The description itself is based on the one given in [TMA11b].

For basic block cipher operation the AES state is first initialized with 16 bytes of plaintext that is then processed together with the key to produce 16 bytes of corresponding ciphertext. This processing involves 10 rounds of repeated application of four round functions in the following order:

1. **SubBytes** - The SubBytes function is a nonlinear permutation consisting of an S-box applied to the bytes of the state. Each byte of the state matrix is replaced by its multiplicative inverse, followed by an affine mapping. Thus, the input byte $x$ is related to the output $y$ of the S-box by the relation, $y = Ax^{-1} + B$, where $A$ and $B$ are constant matrices.
2. **ShiftRows** - The ShiftRows function is a byte-wise permutation of the state that rotates rows with indices $\{0, 1, 2, 3\}$ by $\{0, 1, 2, 3\}$ elements to the left respectively.
3. **MixColumns** - The MixColumn function is a linear permutation operating on the state column by column. Each column of the state matrix is considered as a 4-dimensional vector where each element belongs to $\mathrm{F}(2^8)$. A $4 \times 4$ matrix $M$ whose elements are also in $\mathrm{F}(2^8)$ is then used to map this column into a new vector. This operation is applied on all the 4 columns of the state matrix. Here, $M$ and its inverse $M^{-1}$ are defined as:

$$M = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \quad M^{-1} = \begin{pmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{pmatrix}$$

4. **AddRoundKey** - Each byte of the state is Xor-ed with a byte from a corresponding array of round subkeys.

Besides that, AES features an additional initial round consisting only of AddRoundKey, and skips the MixColumns operation in round 10. The round keys used within AddRoundKey are derived from the main key using the so-called AES KeySchedule that performs SubBytes and Xor operations to generate the next round key from the current one. In case of decryption operation, one can essentially perform all previously described transformations in reverse to derive a 16-byte plaintext from a 16-byte ciphertext and the key.

## A.2 Lattice-Based Public-Key Encryption

We now give a simplified description of the RLWE-based public-key encryption scheme proposed by Lyubashevsky, Peikert, and Regev [LPR10]. It operates with polynomials over the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ and is parameterized by the tuple $(n, q, \sigma)$. $n$ denotes the dimension of the polynomials, $q$ is the modulus for the base field $\mathbb{Z}_q$, and $\sigma$ is the standard deviation for a discrete Gaussian distribution $D_\sigma$. We use boldface lowercase letters to interchangeably denote polynomials in $\mathcal{R}_q$ as well as their respective coefficient vectors.

**Key generation.** For key-pair generation, two polynomials $\mathbf{r}_1$ and $\mathbf{r}_2$ are sampled from the discrete Gaussian distribution $D_\sigma$. Next, the public key $\mathbf{p}$ is computed as $\mathbf{p} = \mathbf{r}_1 - \mathbf{a}\mathbf{r}_2$. The uniformly-random polynomial $\mathbf{a}$ is either a global domain parameter or is also included in the public key. $\mathbf{r}_2$ is the private key, $\mathbf{r}_1$ is simply discarded.

**Encryption.** First, the plaintext $\mathbf{m}$ is encoded as $\overline{\mathbf{m}} \in \mathcal{R}_q$. In a simple variant of encoding, the bits of $\mathbf{m}$ are simply multiplied by $q/2$. Then, three error polynomials $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3 \in D_\sigma$ are sampled. The ciphertext is a pair of polynomials $(\mathbf{c}_1, \mathbf{c}_2)$ with $\mathbf{c}_1 = \mathbf{a}\mathbf{e}_1 + \mathbf{e}_2$ and $\mathbf{c}_2 = \mathbf{p}\mathbf{e}_1 + \mathbf{e}_3 + \overline{\mathbf{m}}$.

**Decryption.** The private key $\mathbf{r}_2$ is used to compute $\mathbf{m}^\star = \mathbf{c}_1\mathbf{r}_2 + \mathbf{c}_2$. The original message $\mathbf{m}$ is then retrieved by feeding $\mathbf{m}^\star$ to a decoder. There, one computes the distance of each coefficient in $\mathbf{m}^\star$ to $q/2$. If this distance is $< q/4$, then the decoder outputs 1, otherwise 0.

＊＊＊

# Bibliography

[AFG13] Martin R. Albrecht, Robert Fitzpatrick, and Florian Göpfert. "On the Efficacy of Solving LWE by Reduction to Unique-SVP." In: *ICISC 2013*. Ed. by Hyang-Sook Lee and Dong-Guk Han. Vol. 8565. LNCS. Springer, 2013, pp. 293–310.

[AJS16] Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. "NewHope on ARM Cortex-M." In: *SPACE*. Vol. 10076. LNCS. Springer, 2016, pp. 332–349.

[Alb+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. "Ciphers for MPC and FHE." In: *EUROCRYPT (1)*. Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 430–454.

[Alb+17] Martin R. Albrecht, Florian Göpfert, Fernando Virdia, and Thomas Wunderer. "Revisiting the Expected Cost of Solving uSVP and Applications to LWE." In: *ASIACRYPT (1)*. Vol. 10624. LNCS. Springer, 2017, pp. 297–322.

[Alk+17a] Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, and Douglas Stebila. *NewHope*. Submission to [NIS]. `https://newhopecrypto.org/`. 2017.

[Alk+17b] Erdem Alkim et al. *FrodoKEM*. Submission to [NIS]. `https://frodokem.org/`. 2017.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. "On the concrete hardness of Learning with Errors." In: *J. Math. Cryptol.* 9.3 (2015), pp. 169–203.

[AS18] Jacob Alperin-Sheriff. *Programmable Hardware, Microcontrollers and Vector Instructions*. Post on the NIST pqc-forum, `https://groups.google.com/a/list.nist.gov/d/msg/pqc-forum/_0mDoyry1Ao/Tt7yHpjSDgAJ`. 2018.

[Ava+17] Roberto Avanzi et al. *CRYSTALS-Kyber*. Submission to [NIS]. `https://pq-crystals.org/kyber`. 2017.

[Avr] *AVR Crypto Lib*. `https://git.cryptolib.org/?p=avr-crypto-lib.git`.

[Ays+18] Aydin Aysu, Youssef Tobah, Mohit Tiwari, Andreas Gerstlauer, and Michael Orshansky. "Horizontal side-channel vulnerabilities of post-quantum key exchange protocols." In: *HOST*. IEEE Computer Society, 2018, pp. 81–88.

[Bao+21] Zhenzhen Bao, Avik Chakraborti, Nilanjan Datta, Jian Guo, Mridul Nandi, Thomas Peyrin, and Kan Yasuda. *PHOTON-Beetle Authenticated Encryption and Hash Family.* https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/photon-beetle-spec-final.pdf. 2021.

[Bar+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. "The Sorcerer's Apprentice Guide to Fault Attacks." In: *Proc. IEEE* 94.2 (2006), pp. 370–382.

[Bar+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. "Verified Proofs of Higher-Order Masking." In: *EUROCRYPT (1)*. Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 457–485.

[Bar+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: *CCS*. ACM, 2016, pp. 116–129.

[Bar+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. "Parallel Implementations of Masking Schemes and the Bounded Moment Leakage Model." In: *EUROCRYPT (1)*. Vol. 10210. Lecture Notes in Computer Science. 2017, pp. 535–566.

[Bar+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. "Masking the GLP Lattice-Based Signature Scheme at Any Order." In: *EUROCRYPT (2)*. Vol. 10821. LNCS. Springer, 2018, pp. 354–384.

[Bar+19] 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: *ESORICS (1)*. Vol. 11735. Lecture Notes in Computer Science. Springer, 2019, pp. 300–318.

[BCQ04] Alex Biryukov, Christophe De Cannière, and Michaël Quisquater. "On Multiple Linear Approximations." In: *CRYPTO*. Vol. 3152. Lecture Notes in Computer Science. Springer, 2004, pp. 1–22.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)." In: *EUROCRYPT*. Vol. 1233. Lecture Notes in Computer Science. Springer, 1997, pp. 37–51.

[Bei+20] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. "Lightweight AEAD and Hashing using the Sparkle Permutation Family." In: *IACR Trans. Symmetric Cryptol.* 2020.S1 (2020), pp. 208–261.

[Bel+17] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. "Private Multiplication over Finite Fields." In: *CRYPTO (3)*. Vol. 10403. Lecture Notes in Computer Science. Springer, 2017, pp. 397–426.

[Bel+20] Davide Bellizia, Olivier Bronchain, Gaëtan Cassiers, Vincent Grosso, Chun Guo, Charles Momin, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. “Mode-Level vs. Implementation-Level Physical Security in Symmetric Cryptography - A Practical Guide Through the Leakage-Resistance Jungle.” In: *CRYPTO (1)*. Vol. 12170. Lecture Notes in Computer Science. Springer, 2020, pp. 369–400.

[Ben73] Charles H. Bennett. “Logical Reversibility of Computation.” In: *IBM Journal of Research and Development* 17.6 (1973), pp. 525–532.

[Ber+a] Guido Bertoni, Joan Daemen, Seth Hoffert, Michael Peeters, Gilles Van Assche, and Ronny Van Keer. *eXtended Keccak Code Package*. https://github.com/XKCP/XKCP.

[Ber+b] Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, and Ronny Van Keer. *CAESAR submission: Ketje v2*. https://keccak.team/files/Ketjev2-doc2.0.pdf.

[Ber+c] Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, and Ronny Van Keer. *CAESAR submission: Keyak v2*. https://keccak.team/files/Keyakv2-doc2.2.pdf.

[Ber+11a] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. “Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications.” In: *SAC*. Vol. 7118. Lecture Notes in Computer Science. Springer, 2011, pp. 320–337.

[Ber+11b] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *The Keccak reference*. https://keccak.team/papers.html. 2011.

[Ber+11c] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. *The Keccak SHA-3 submission (Version 3.0)*. http://keccak.noekeon.org/Keccak-submission-3.pdf. 2011.

[Ber+16] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. *NTRU Prime*. 2016.

[Ber+17a] Francesco Berti, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. “On Leakage-Resilient Authenticated Encryption with Decryption Leakages.” In: *IACR Trans. Symmetric Cryptol.* 2017.3 (2017), pp. 271–293.

[Ber+17b] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. “Farfalle: parallel permutation-based cryptography.” In: *IACR Trans. Symmetric Cryptol.* 2017.4 (2017), pp. 1–38.

[Ber+19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. “The SPHINCS$^+$ Signature Framework.” In: *CCS*. ACM, 2019, pp. 2129–2146.

[Bey+20] Tim Beyne, Yu Long Chen, Christoph Dobraunig, and Bart Mennink. “Dumbo, Jumbo, and Delirium: Parallel Authenticated Encryption for the Lightweight Circus.” In: *IACR Trans. Symmetric Cryptol.* 2020.S1 (2020), pp. 5–30.

[BG16] Alberto Battistello and Christophe Giraud. “A Note on the Security of CHES 2014 Symmetric Infective Countermeasure.” In: *COSADE*. Vol. 9689. Lecture Notes in Computer Science. Springer, 2016, pp. 144–159.

[BGN12a] Céline Blondeau, Benoît Gérard, and Kaisa Nyberg. “Multiple Differential Cryptanalysis Using LLR and $\chi$ 2 Statistics.” In: *SCN*. Vol. 7485. Lecture Notes in Computer Science. Springer, 2012, pp. 343–360.

[BGN12b] Céline Blondeau, Benoît Gérard, and Kaisa Nyberg. *Multiple Differential Cryptanalysis using* LLR*and* $\chi^2$ *Statistics*. Cryptology ePrint Archive, Paper 2012/360. https://eprint.iacr.org/2012/360. 2012.

[Bil+13] Begül Bilgin, Joan Daemen, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Gilles Van Assche. “Efficient and First-Order DPA Resistant Implementations of Keccak.” In: *CARDIS*. Vol. 8419. Lecture Notes in Computer Science. Springer, 2013, pp. 187–199.

[Bil+15] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, Natalia N. Tokareva, and Valeriya Vitkup. “Threshold implementations of small S-boxes.” In: *Cryptogr. Commun.* 7.1 (2015), pp. 3–33.

[Bin+17] Nina Bindel et al. *qTESLA*. Submission to [NIS]. https://qtesla.org. 2017.

[BJV04] Thomas Baignères, Pascal Junod, and Serge Vaudenay. “How Far Can We Go Beyond Linear Cryptanalysis?” In: *ASIACRYPT*. Vol. 3329. Lecture Notes in Computer Science. Springer, 2004, pp. 432–450.

[Blo+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: *EUROCRYPT (2)*. Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 321–353.

[Blo+22] Roderick Bloem, Barbara Gigerl, Marc Gourjon, Vedad Hadzic, Stefan Mangard, and Robert Primas. “Power Contracts: Provably Complete Power Leakage Models for Processors.” In: *CCS*. ACM, 2022, pp. 381–395.

[Bog+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: *CHES*. Vol. 4727. Lecture Notes in Computer Science. Springer, 2007, pp. 450–466.

[Bos+16] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. “Frodo: Take off the Ring! Practical, Quantum-Secure Key Exchange from LWE.” In: *CCS*. ACM, 2016, pp. 1006–1018.

[Bos+18a] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. “CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM.” In: *EuroS&P*. IEEE, 2018, pp. 353–367.

[Bos+18b] Joppe W. Bos, Simon Friedberger, Marco Martinoli, Elisabeth Oswald, and Martijn Stam. “Assessing the Feasibility of Single Trace Power Analysis of Frodo.” In: *SAC*. Vol. 11349. Lecture Notes in Computer Science. Springer, 2018, pp. 216–234.

[BP12] Joan Boyar and René Peralta. "A Small Depth-16 Circuit for the AES S-Box." In: *SEC*. Vol. 376. IFIP Advances in Information and Communication Technology. Springer, 2012, pp. 287–298.

[Bre+20] Jakub Breier, Mustafa Khairallah, Xiaolu Hou, and Yang Liu. "A Countermeasure Against Statistical Ineffective Fault Analysis." In: *IEEE Trans. Circuits Syst.* 67-II.12 (2020), pp. 3322–3326.

[BRW03] Mihir Bellare, Phillip Rogaway, and David A. Wagner. *EAX: A Conventional Authenticated-Encryption Mode*. 2003.

[BS90] Eli Biham and Adi Shamir. "Differential Cryptanalysis of DES-like Cryptosystems." In: *CRYPTO*. Vol. 537. Lecture Notes in Computer Science. Springer, 1990, pp. 2–21.

[BS97] Eli Biham and Adi Shamir. "Differential Fault Analysis of Secret Key Cryptosystems." In: *CRYPTO*. Vol. 1294. Lecture Notes in Computer Science. Springer, 1997, pp. 513–525.

[CAE14] CAESAR committee. *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*. 2014. URL: http://competitions.cr.yp.to/caesar.html.

[Cal] *Caliptra*. https://www.opencompute.org/blog/cloud-security-integrating-trust-into-every-chip. 2022.

[Can05] David Canright. "A Very Compact S-Box for AES." In: *CHES*. Vol. 3659. Lecture Notes in Computer Science. Springer, 2005, pp. 441–455.

[Cha+99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. "Towards Sound Approaches to Counteract Power-Analysis Attacks." In: *CRYPTO*. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 398–412.

[Cla07] Christophe Clavier. "Secret External Encodings Do Not Prevent Transient Fault Analysis." In: *CHES*. Vol. 4727. Lecture Notes in Computer Science. Springer, 2007, pp. 181–194.

[Cle+15] Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. "Efficient software implementation of ring-LWE encryption." In: *DATE*. ACM, 2015, pp. 339–344.

[Cnu+16] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "Masking AES with d+1 Shares in Hardware." In: *CHES*. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 194–212.

[Col+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: *HOST*. IEEE, 2019, pp. 1–10.

[Com] *Security IC Platform Protection Profile with Augmentation Packages*. https://www.commoncriteriaportal.org/files/ppfiles/pp0084b_pdf.pdf. 2014.

[Cor17] Jean-Sébastien Coron. *Higher Order Countermeasures for AES and DES*. 2017. URL: https://github.com/coron/htable#higher-order-countermeasures-for-aes-and-des.

[CRR02a] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. "Template Attacks." In: *CHES*. Vol. 2523. Lecture Notes in Computer Science. Springer, 2002, pp. 13–28.

[CRR02b] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. "Template Attacks." In: *CHES*. Vol. 2523. Lecture Notes in Computer Science. Springer, 2002, pp. 13–28.

[CS03] Ronald Cramer and Victor Shoup. "Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack." In: *SIAM Journal on Computing* 33.1 (2003), pp. 167–226. DOI: 10.1137/S0097539702403773. URL: https://doi.org/10.1137/S0097539702403773.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. "Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference." In: *IEEE Trans. Inf. Forensics Secur.* 15 (2020), pp. 2542–2555.

[CT06] Thomas M. Cover and Joy A. Thomas. *Elements of information theory (2. ed.)* Wiley, 2006.

[Dae+00] Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. *Nessie Proposal: the block cipher* NOEKEON. Nessie submission. http://gro.noekeon.org/. 2000.

[Dae17] Joan Daemen. "Changing of the Guards: A Simple and Efficient Method for Achieving Uniformity in Threshold Sharing." In: *CHES*. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 137–153.

[Dae+18] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. "The design of Xoodoo and Xoofff." In: *IACR Trans. Symmetric Cryptol.* 2018.4 (2018), pp. 1–38.

[Dae+20a] Joan Daemen, Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Florian Mendel, and Robert Primas. "Protecting against Statistical Ineffective Fault Attacks." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.3 (2020), pp. 508–543.

[Dae+20b] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. "Xoodyak, a lightweight cryptographic scheme." In: *IACR Trans. Symmetric Cryptol.* 2020.S1 (2020), pp. 60–87.

[Dae95] Joan Daemen. "Cipher and hash function design, strategies based on linear and differential cryptanalysis." PhD thesis. KU Leuven, 1995. URL: http://jda.noekeon.org/.

[De 07] Christophe De Cannière. "Analysis and design of symmetric encryption algorithms." PhD thesis. KU Leuven, 2007.

[DEM15] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. "Heuristic Tool for Linear Cryptanalysis with Applications to CAESAR Candidates." In: *ASIACRYPT (2)*. Vol. 9453. Lecture Notes in Computer Science. Springer, 2015, pp. 490–509.

[DGV94] Joan Daemen, René Govaerts, and Joos Vandewalle. "An Efficient Nonlinear Shift-Invariant Transformation." In: *Information Theory in the Benelux*. Ed. by B. Macq. Werkgemeenschap voor Informatie- en Communicatietheorie, 1994, pp. 108–115.

[DMP22] Christoph Dobraunig, Bart Mennink, and Robert Primas. “Leakage and Tamper Resilient Permutation-Based Cryptography.” In: *CCS*. ACM, 2022, pp. 859–873.

[Dob+] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *Ascon v1.2 Submission to the CAESAR Competition*. https://ascon.iaik.tugraz.at/files/asconv12.pdf.

[Dob+15] Christoph Dobraunig, François Koeune, Stefan Mangard, Florian Mendel, and François-Xavier Standaert. “Towards Fresh and Hybrid Re-Keying Schemes with Beyond Birthday Security.” In: *CARDIS*. Vol. 9514. Lecture Notes in Computer Science. Springer, 2015, pp. 225–241.

[Dob+16] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Victor Lomné, and Florian Mendel. “Statistical Fault Attacks on Nonce-Based Authenticated Encryption Schemes.” In: *ASIACRYPT (1)*. Vol. 10031. Lecture Notes in Computer Science. 2016, pp. 369–395.

[Dob+17] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, and Thomas Unterluggauer. “ISAP - Towards Side-Channel Secure Authenticated Encryption.” In: *IACR Trans. Symmetric Cryptol.* 2017.1 (2017), pp. 80–105.

[Dob+18a] Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Stefan Mangard, Florian Mendel, and Robert Primas. “Statistical Ineffective Fault Attacks on Masked AES with Fault Countermeasures.” In: *ASIACRYPT (2)*. Vol. 11273. Lecture Notes in Computer Science. Springer, 2018, pp. 315–342.

[Dob+18b] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. “SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography.” In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 547–572.

[Dob+18c] Christoph Dobraunig, Stefan Mangard, Florian Mendel, and Robert Primas. “Fault Attacks on Nonce-Based Authenticated Encryption: Application to Keyak and Ketje.” In: *SAC*. Vol. 11349. Lecture Notes in Computer Science. Springer, 2018, pp. 257–277.

[Dob+20] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, Bart Mennink, Robert Primas, and Thomas Unterluggauer. “Isap v2.0.” In: *IACR Trans. Symmetric Cryptol.* 2020.S1 (2020), pp. 390–416.

[Dob+21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. “Ascon v1.2: Lightweight Authenticated Encryption and Hashing.” In: *J. Cryptol.* 34.3 (2021), p. 33.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[DR20] Joan Daemen and Vincent Rijmen. *The Design of Rijndael - The Advanced Encryption Standard (AES), Second Edition*. Information Security and Cryptography. Springer, 2020.

[Dro+89] Feike C. Drost, Wilbert C. M. Kallenberg, D. S. Moore, and J. Oosterhoff. “Power Approximations to Multinomial Tests of Fit.” In: *Journal of the American Statistical Association* 84.405 (1989), pp. 130–141.

[Duc+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. "CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.1 (2018), pp. 238–268.

[ENP19] Maria Eichlseder, Marcel Nageler, and Robert Primas. "Analyzing the Linear Keystream Biases in AEGIS." In: *IACR Trans. Symmetric Cryptol.* 2019.4 (2019), pp. 348–368.

[Fau+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." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 89–120.

[Flu16] Scott R. Fluhrer. *Cryptanalysis of ring-LWE based key exchange with key share reuse.* 2016.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes." In: *CRYPTO*. Vol. 1666. LNCS. Springer, 1999, pp. 537–554.

[For07] Internet Research Task Force. *Using AES-CCM and AES-GCM Authenticated Encryption in the Cryptographic Message Syntax (CMS).* 2007. URL: https://tools.ietf.org/html/rfc5084.

[For15] Internet Research Task Force. *ChaCha20 and Poly1305 for IETF Protocols.* 2015. URL: https://tools.ietf.org/html/rfc7539.

[Fou+] Pierre-Alain Fouque et al. *Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU.* https://falcon-sign.info/falcon.pdf.

[Fuh+13] Thomas Fuhr, Éliane Jaulmes, Victor Lomné, and Adrian Thillard. "Fault Attacks on AES with Faulty Ciphertexts Only." In: *FDTC*. IEEE Computer Society, 2013, pp. 108–118.

[GBB] Benjamin Grégoire, Gilles Barthe, and Sonia Belaïd. *maskVerif: Automatic tool for the verification of side-channel countermeasures.* https://cryptoexperts.com/maskverif/.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. "How to construct random functions." In: *J. ACM* 33.4 (1986), pp. 792–807.

[GIB18] Hannes Groß, Rinat Iusupov, and Roderick Bloem. "Generic Low-Latency Masking in Hardware." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 1–21.

[Gig+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: *USENIX Security Symposium*. USENIX Association, 2021, pp. 1469–1468.

[GLP06] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. "Templates vs. Stochastic Methods." In: *CHES*. Vol. 4249. Lecture Notes in Computer Science. Springer, 2006, pp. 15–29.

[GM12] Oleg Golubitsky and Dmitri Maslov. "A Study of Optimal 4-Bit Reversible Toffoli Circuits and Their Synthesis." In: *IEEE Trans. Computers* 61.9 (2012), pp. 1341–1353.

[GM17] Hannes Groß and Stefan Mangard. "Reconciling d+1 Masking in Hardware and Software." In: *CHES*. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 115–136.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order." In: *TIS@CCS*. ACM, 2016, p. 3.

[Göt+12] Norman Göttert, Thomas Feller, Michael Schneider, Johannes Buchmann, and Sorin A. Huss. "On the Design of Hardware Building Blocks for Modern Lattice-Based Encryption Schemes." In: *CHES*. Vol. 7428. Lecture Notes in Computer Science. Springer, 2012, pp. 512–529.

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. "Secure and Efficient Software Masking on Superscalar Pipelined Processors." In: *ASIACRYPT (2)*. Vol. 13091. Lecture Notes in Computer Science. Springer, 2021, pp. 3–32.

[GPM23] Barbara Gigerl, Robert Primas, and Stefan Mangard. "Formal Verification of Arithmetic Masking in Hardware and Software." In: *ACNS* (2023).

[GS15] Vincent Grosso and François-Xavier Standaert. "ASCA, SASCA and DPA with Enumeration: Which One Beats the Other and When?" In: *ASIACRYPT (2)*. Vol. 9453. Lecture Notes in Computer Science. Springer, 2015, pp. 291–312.

[GS18] Vincent Grosso and François-Xavier Standaert. "Masking Proofs Are Tight and How to Exploit it in Security Evaluations." In: *EUROCRYPT (2)*. Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 385–412.

[GSM17] Hannes Groß, David Schaffenrath, and Stefan Mangard. "Higher-Order Side-Channel Protected Implementations of KECCAK." In: *DSD*. IEEE Computer Society, 2017, pp. 205–212.

[GST12] Benedikt Gierlichs, Jörn-Marc Schmidt, and Michael Tunstall. "Infective Computation and Dummy Rounds: Fault Protection for Block Ciphers without Check-before-Output." In: *LATINCRYPT*. Vol. 7533. Lecture Notes in Computer Science. Springer, 2012, pp. 305–321.

[GVW17] Florian Göpfert, Christine van Vredendaal, and Thomas Wunderer. *A Quantum Attack on LWE with Arbitrary Error Distribution*. 2017.

[Ham+21] Mike Hamburg, Julius Hermelink, Robert Primas, Simona Samardjiska, Thomas Schamberger, Silvan Streit, Emanuele Strieder, and Christine van Vredendaal. "Chosen Ciphertext k-Trace Attacks on Masked CCA2 Secure Kyber." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.4 (2021), pp. 88–113.

[HCN09] Miia Hermelin, Joo Yeon Cho, and Kaisa Nyberg. "Multidimensional Extension of Matsui's Algorithm 2." In: *FSE*. Vol. 5665. Lecture Notes in Computer Science. Springer, 2009, pp. 209–227.

[HPB21a] Vedad Hadzic, Robert Primas, and Roderick Bloem. "Proving SIFA Protection of Masked Redundant Circuits." In: *ATVA*. Vol. 12971. Lecture Notes in Computer Science. Springer, 2021, pp. 249–265.

[HPB21b] Vedad Hadzic, Robert Primas, and Roderick Bloem. "Proving SIFA Protection of Masked Redundant Circuits (Extended Version)." In: *CoRR* abs/2107.01917 (2021).

[HS13] Michael Hutter and Jörn-Marc Schmidt. "The Temperature Side Channel and Heating Fault Attacks." In: *CARDIS*. Vol. 8419. Lecture Notes in Computer Science. Springer, 2013, pp. 219–235.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. "Private Circuits: Securing Hardware against Probing Attacks." In: *CRYPTO*. Vol. 2729. Lecture Notes in Computer Science. Springer, 2003, pp. 463–481.

[JMV01] Don Johnson, Alfred Menezes, and Scott A. Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)." In: *Int. J. Inf. Sec.* 1.1 (2001), pp. 36–63.

[JS17] Anthony Journault and François-Xavier Standaert. "Very High Order Masking: Efficient Implementation and Security Evaluation." In: *CHES*. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 623–643.

[Kan+] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. *PQM4: Post-quantum crypto library for the ARM Cortex-M4*. https://github.com/mupq/pqm4.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis." In: *CRYPTO*. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397.

[Knu+10] Lars R. Knudsen, Gregor Leander, Axel Poschmann, and Matthew J. B. Robshaw. "PRINTcipher: A Block Cipher for IC-Printing." In: *CHES*. Vol. 6225. Lecture Notes in Computer Science. Springer, 2010, pp. 16–32.

[KPP20] Matthias J. Kannwischer, Peter Pessl, and Robert Primas. "Single-Trace Attacks on Keccak." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.3 (2020), pp. 243–268.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. "SILVER - Statistical Independence and Leakage Verification." In: *ASIACRYPT (1)*. Vol. 12491. Lecture Notes in Computer Science. Springer, 2020, pp. 787–816.

[Lan61] Rolf Landauer. "Irreversibility and Heat Generation in the Computing Process." In: *IBM Journal of Research and Development* 5.3 (1961), pp. 183–191.

[Liu+15] Zhe Liu, Hwajeong Seo, Sujoy Sinha Roy, Johann Großschädl, Howon Kim, and Ingrid Verbauwhede. "Efficient Ring-LWE Encryption on 8-Bit AVR Processors." In: *CHES*. Vol. 9293. Lecture Notes in Computer Science. Springer, 2015, pp. 663–682.

[LP07] Gregor Leander and Axel Poschmann. "On the Classification of 4 Bit S-Boxes." In: *WAIFI*. Vol. 4547. Lecture Notes in Computer Science. Springer, 2007, pp. 159–176.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors over Rings." In: *EUROCRYPT*. Vol. 6110. Lecture Notes in Computer Science. Springer, 2010, pp. 1–23.

[LS15] Adeline Langlois and Damien Stehlé. "Worst-case to average-case reductions for module lattices." In: *Des. Codes Cryptography* 75.3 (2015), pp. 565–599.

[Lyu+17] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, and Damien Stehlé. *CRYSTALS-Dilithium*. Submission to [NIS]. https://pq-crystals.org/dilithium. 2017.

[Mac03] David J. C. MacKay. *Information theory, inference, and learning algorithms*. Cambridge University Press, 2003.

[Mar14] Matteo Mariantoni. *Building a superconducting quantum computer*. Invited Talk at PQCrypto 2014. https://www.youtube.com/watch?v=wWHAs--HA1c. 2014.

[Mat93] Mitsuru Matsui. "Linear Cryptanalysis Method for DES Cipher." In: *EUROCRYPT*. Vol. 765. Lecture Notes in Computer Science. Springer, 1993, pp. 386–397.

[McK+17] Kerry A. McKay, Larry Bassham, Meltem Sönmez Turan, and Nicky Mouha. *NISTIR 8114: Report on Lightweight Cryptography*. https://doi.org/10.6028/NIST.IR.8114. 2017.

[Med+10] Marcel Medwed, François-Xavier Standaert, Johann Großschädl, and Francesco Regazzoni. "Fresh Re-keying: Security against Side-Channel and Fault Attacks for Low-Cost Devices." In: *AFRICACRYPT*. Vol. 6055. Lecture Notes in Computer Science. Springer, 2010, pp. 279–296.

[MM22] Nicolai Müller and Amir Moradi. "PROLEAD A Probing-Based Hardware Leakage Detection Tool." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.4 (2022), pp. 311–348.

[MN17] Bart Mennink and Samuel Neves. "Optimal PRFs from Blockcipher Designs." In: *IACR Trans. Symmetric Cryptol.* 2017.3 (2017), pp. 228–252.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[MRV15] Bart Mennink, Reza Reyhanitabar, and Damian Vizár. "Security of Full-State Keyed Sponge and Duplex: Applications to Authenticated Encryption." In: *ASIACRYPT (2)*. Vol. 9453. Lecture Notes in Computer Science. Springer, 2015, pp. 465–489.

[MV04] David A. McGrew and John Viega. "The Security and Performance of the Galois/Counter Mode (GCM) of Operation." In: *INDOCRYPT*. Vol. 3348. Lecture Notes in Computer Science. Springer, 2004, pp. 343–355.

[Nag+22] Rishub Nagpal, Barbara Gigerl, Robert Primas, and Stefan Mangard. "Riding the Waves Towards Generic Single-Cycle Masking in Hardware." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.4 (2022).

[Nat15] National Institute of Standards and Technology. *FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Federal Information Processing Standards Publication 202, U.S. Department of Commerce. 2015. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[Nat18] National Institute of Standards and Technology. *DRAFT Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process*. https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/Draft-LWC-Submission-Requirements-April2018.pdf. 2018.

[New] NewAE. *CW308T-STM32F*. https://wiki.newae.com/CW308T-STM32F.

[NIS] NIST. *Post-Quantum Cryptography Standardization*. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization.

[NIS15] NIST. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. 2015. URL: https://doi.org/10.6028/NIST.FIPS.202.

[NIS16a] NIST. *Post-Quantum crypto standardization*. http://csrc.nist.gov/groups/ST/post-quantum-crypto/call-for-proposals-2016.html. 2016.

[NIS16b] NIST. *Post-Quantum Cryptography*. https://csrc.nist.gov/projects/post-quantum-cryptography. 2016.

[NIS18] NIST. *Lightweight Cryptography*. https://csrc.nist.gov/Projects/lightweight-cryptography. 2018.

[NIS94] NIST. *FIPS Publication 186: Digital signature standard*. 1994. URL: http://csrc.nist.gov/encryption/.

[NP33] Jerzy Neyman and Egon S. Pearson. "On the problem of the most efficient tests of statistical hypotheses." In: *Philosophical Transactions of the Royal Society of London* (1933), pp. 289–337.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. "Threshold Implementations Against Side-Channel Attacks and Glitches." In: *ICICS*. Vol. 4307. Lecture Notes in Computer Science. Springer, 2006, pp. 529–545.

[NRS11] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. "Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches." In: *Journal of Cryptology* 24.2 (2011), pp. 292–321. DOI: 10.1007/s00145-010-9085-7.

[NSA16] NSA/IAD. *CNSA Suite and Quantum Computing FAQ*. 2016. URL: https://www.iad.gov/iad/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/cnsa-suite-and-quantum-computing-faq.cfm.

[Ode+18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. "Practical CCA2-Secure and Masked Ring-LWE Implementation." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.1 (2018), pp. 142–174.

[Ope] *OpenTitan*. https://opentitan.org/. 2019.

[Pat+17] Sikhar Patranabis, Abhishek Chakraborty, Debdeep Mukhopadhyay, and Partha Pratim Chakrabarti. "Fault Space Transformation: A Generic Approach to Counter Differential Fault Analysis and Differential Fault Intensity Analysis on AES-Like Block Ciphers." In: *IEEE Trans. Inf. Forensics Secur.* 12.5 (2017), pp. 1092–1102.

[PCM15] Sikhar Patranabis, Abhishek Chakraborty, and Debdeep Mukhopadhyay. “Fault Tolerant Infective Countermeasure for AES.” In: *SPACE*. Vol. 9354. Lecture Notes in Computer Science. Springer, 2015, pp. 190–209.

[PG13] Thomas Pöppelmann and Tim Güneysu. “Towards Practical Lattice-Based Public-Key Encryption on Reconfigurable Hardware.” In: *SAC*. Vol. 8282. Lecture Notes in Computer Science. Springer, 2013, pp. 68–85.

[POG15] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. “High-Performance Ideal Lattice-Based Cryptography on 8-Bit ATxmega Microcontrollers.” In: *LATINCRYPT*. Vol. 9230. Lecture Notes in Computer Science. Springer, 2015, pp. 346–365.

[PP19] Peter Pessl and Robert Primas. “More Practical Single-Trace Attacks on the Number Theoretic Transform.” In: *LATINCRYPT*. Vol. 11774. Lecture Notes in Computer Science. Springer, 2019, pp. 130–149.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. “Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption.” In: *CHES*. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 513–533.

[PSV15] Olivier Pereira, François-Xavier Standaert, and Srinivas Vivek. “Leakage-Resilient Authentication and Encryption from Symmetric Cryptographic Primitives.” In: *CCS*. ACM, 2015, pp. 96–108.

[QS01] Jean-Jacques Quisquater and David Samyde. “ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards.” In: *E-smart*. Vol. 2140. Lecture Notes in Computer Science. Springer, 2001, pp. 200–210.

[RAD20] Keyvan Ramezanpour, Paul Ampadu, and William Diehl. “RS-Mask: Random Space Masking as an Integrated Countermeasure against Power and Fault Analysis.” In: *HOST*. IEEE, 2020, pp. 176–187.

[Reg05] Oded Regev. “On lattices, learning with errors, random linear codes, and cryptography.” In: *STOC*. ACM, 2005, pp. 84–93.

[Rep+15a] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. “Consolidating Masking Schemes.” In: *CRYPTO (1)*. Vol. 9215. Lecture Notes in Computer Science. Springer, 2015, pp. 764–783.

[Rep+15b] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. “A Masked Ring-LWE Implementation.” In: *CHES*. Vol. 9293. Lecture Notes in Computer Science. Springer, 2015, pp. 683–702.

[Rep+16a] Oscar Reparaz, Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. “Additively Homomorphic Ring-LWE Masking.” In: *PQCrypto*. Vol. 9606. Lecture Notes in Computer Science. Springer, 2016, pp. 233–244.

[Rep+16b] Oscar Reparaz, Sujoy Sinha Roy, Ruan de Clercq, Frederik Vercauteren, and Ingrid Verbauwhede. “Masking ring-LWE.” In: *J. Cryptogr. Eng.* 6.2 (2016), pp. 139–153.

[Rep+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: *CRYPTO (1)*. Vol. 10991. Lecture Notes in Computer Science. Springer, 2018, pp. 121–151.

[Rog+01] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. "OCB: a block-cipher mode of operation for efficient authenticated encryption." In: *CCS*. ACM, 2001, pp. 196–205.

[Rog02] Phillip Rogaway. "Authenticated-encryption with associated-data." In: *CCS*. ACM, 2002, pp. 98–107.

[Roy+14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. "Compact Ring-LWE Cryptoprocessor." In: *CHES*. Vol. 8731. Lecture Notes in Computer Science. Springer, 2014, pp. 371–391.

[RP10] Matthieu Rivain and Emmanuel Prouff. "Provably Secure Higher-Order Masking of AES." In: *CHES*. Vol. 6225. Lecture Notes in Computer Science. Springer, 2010, pp. 413–427.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Commun. ACM* 21.2 (1978), pp. 120–126.

[Saa18] Markku-Juhani O. Saarinen. "Arithmetic coding and blinding countermeasures for lattice signatures - Engineering a side-channel resistant post-quantum signature scheme with compact signatures." In: *J. Cryptogr. Eng.* 8.1 (2018), pp. 71–84.

[Sah+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*. 2019.

[SC15] Dhiman Saha and Dipanwita Roy Chowdhury. "Scope: On the Side Channel Vulnerability of Releasing Unverified Plaintexts." In: *SAC*. Vol. 9566. Lecture Notes in Computer Science. Springer, 2015, pp. 417–438.

[SC16] Dhiman Saha and Dipanwita Roy Chowdhury. "EnCounter: On Breaking the Nonce Barrier in Differential Fault Analysis with a Case-Study on PAEQ." In: *CHES*. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 581–601.

[Sel08] Ali Aydin Selçuk. "On Probability of Success in Linear and Differential Cryptanalysis." In: *J. Cryptol.* 21.1 (2008), pp. 131–147.

[Sel+15] Bodo Selmke, Stefan Brummer, Johann Heyszl, and Georg Sigl. "Precise Laser Fault Injections into 90 nm and 45 nm SRAM-cells." In: *CARDIS*. Vol. 9514. Lecture Notes in Computer Science. Springer, 2015, pp. 193–205.

[Sel+18] Bodo Selmke, Kilian Zinnecker, Philipp Koppermann, Katja Miller, Johann Heyszl, and Georg Sigl. "Locked out by Latch-up? An Empirical Study on Laser Fault Injection into Arm Cortex-M Processors." In: *FDTC*. IEEE Computer Society, 2018, pp. 7–14.

[Sha16] Adi Shamir. *Financial Cryptography: Past, Present, and Future*. Invited Talk at Financial Cryptography 2016. 2016. URL: https://www.lightbluetouchpaper.org/2016/02/22/financial-cryptography-2016.

[She+03] Vivek V. Shende, Aditya K. Prasad, Igor L. Markov, and John P. Hayes. "Synthesis of reversible logic circuits." In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 22.6 (2003), pp. 710–722.

[Sho] Victor Shoup. *NTL: A Library for doing Number Theory*. http://www.shoup.net/ntl/.

[Sho94] P.W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring." In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[Sho99] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." In: *SIAM Rev.* 41.2 (1999), pp. 303–332.

[SKC14] Dhiman Saha, Sukhendu Kuila, and Dipanwita Roy Chowdhury. "EscApe: Diagonal Fault Analysis of APE." In: *INDOCRYPT*. Vol. 8885. Lecture Notes in Computer Science. Springer, 2014, pp. 197–216.

[SMY09] François-Xavier Standaert, Tal Malkin, and Moti Yung. "A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks." In: *EUROCRYPT*. Vol. 5479. Lecture Notes in Computer Science. Springer, 2009, pp. 443–461.

[SP20] Stefan Steinegger and Robert Primas. "A Fast and Compact RISC-V Accelerator for Ascon and Friends." In: *CARDIS*. Vol. 12609. Lecture Notes in Computer Science. Springer, 2020, pp. 53–67.

[SRM20] Aein Rezaei Shahmirzadi, Shahram Rasoolzadeh, and Amir Moradi. "Impeccable Circuits II." In: *DAC*. IEEE, 2020, pp. 1–6.

[SS16] Peter Schwabe and Ko Stoffelen. "All the AES You Need on Cortex-M3 and M4." In: *SAC*. Vol. 10532. Lecture Notes in Computer Science. Springer, 2016, pp. 180–194.

[Sto03] Amos J. Storkey. "Generalised Propagation for Fast Fourier Transforms with Partial or Missing Data." In: *NIPS*. MIT Press, 2003, pp. 433–440.

[Sug19] Takeshi Sugawara. "3-Share Threshold Implementation of AES S-box without Fresh Randomness." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.1 (2019), pp. 123–145.

[TBM14] Harshal Tupsamudre, Shikha Bisht, and Debdeep Mukhopadhyay. "Destroying Fault Invariant with Randomization - A Countermeasure for AES Against Differential Fault Attacks." In: *CHES*. Vol. 8731. Lecture Notes in Computer Science. Springer, 2014, pp. 93–111.

[TMA11a] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. "Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault." In: *WISTP*. Vol. 6633. Lecture Notes in Computer Science. Springer, 2011, pp. 224–233.

[TMA11b] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. "Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault." In: *WISTP*. Vol. 6633. Lecture Notes in Computer Science. Springer, 2011, pp. 224–233.

[Tof80] Tommaso Toffoli. “Reversible Computing.” In: *ICALP*. Vol. 85. Lecture Notes in Computer Science. Springer, 1980, pp. 632–644.

[Tri03] Elena Trichina. *Combinational Logic Design for AES SubByte Transformation on Masked Data*. 2003.

[Ull+11] Markus Ullrich, Christophe De Cannière, Sebastiaan Indesteege, Özgül Küçük, Nicky Mouha, and Bart Preneel. “Finding optimal bitsliced implementations of $4 \times 4$-bit S-boxes.” In: *ECRYPT Symmetric Key Encryption Workshop – SKEW 2011*. 2011, pp. 16–17.

[Vaf+22] Navid Vafaei, Sara Zarei, Nasour Bagheri, Maria Eichlseder, Robert Primas, and Hadi Soleimany. “Statistical Effective Fault Attacks: The Other Side of the Coin.” In: *IEEE Trans. Inf. Forensics Secur.* 17 (2022), pp. 1855–1867.

[VGS14] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. “Soft Analytical Side-Channel Attacks.” In: *ASIACRYPT (1)*. Vol. 8873. Lecture Notes in Computer Science. Springer, 2014, pp. 282–296.

[WHF03] D. Whiting, R. Housley, and N. Ferguson. *Counter with CBC-MAC (CCM)*. United States, 2003.

[WWM15] Mario Werner, Erich Wenger, and Stefan Mangard. “Protecting the Control Flow of Embedded Processors against Fault Attacks.” In: *CARDIS*. Vol. 9514. Lecture Notes in Computer Science. Springer, 2015, pp. 161–176.

[Yed03] Jonathan S. Yedidia. “Sparse factor graph representations of Reed-Solomon and related codes.” In: *Algebraic Coding Theory and Information Theory*. Vol. 68. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, 2003, pp. 91–98.

[YFW00] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. “Generalized Belief Propagation.” In: *NIPS*. MIT Press, 2000, pp. 689–695.

[YJ00] Sung-Ming Yen and Marc Joye. “Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis.” In: *IEEE Trans. Computers* 49.9 (2000), pp. 967–970.

## AFFIDAVIT

＊＊＊