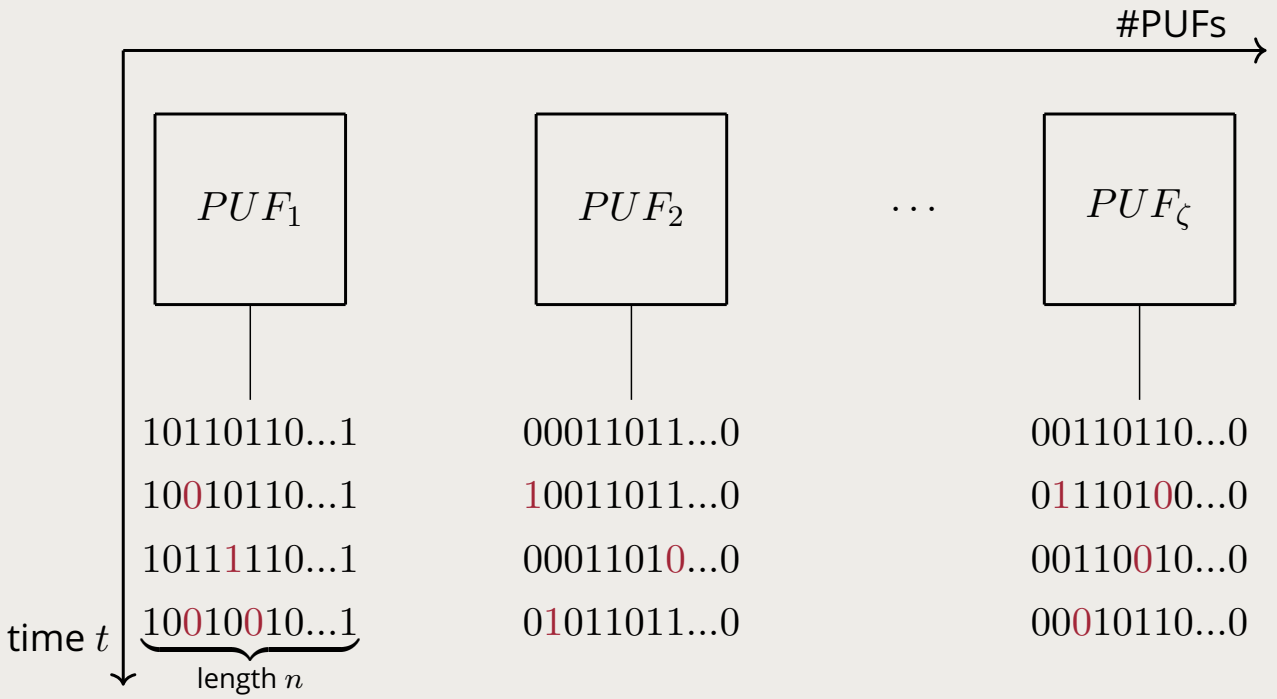
ulm university universität
uulm

Sven Müelich

Channel Coding for Hardware-Intrinsic Security



Band 2
Schriftenreihe des Instituts für Nachrichtentechnik

Sven Müelich

Channel Coding for Hardware-Intrinsic Security

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Unveränderte Neuauflage der Dissertation

Channel Coding for Hardware-Intrinsic Security, Sven Müelich, Fakultät für Ingenieurwissenschaften, Informatik und Psychologie der Universität Ulm, 2019

https://doi.org/10.18725/OPARU-21208

Impressum

Universität Ulm
Institut für Nachrichtentechnik
Prof. Dr.-Ing. Robert Fischer
Albert-Einstein-Allee 43
89081 Ulm

Eine Übersicht über alle Bände der Schriftenreihe finden Sie unter
https://www.uni-ulm.de/ntschriften

Institut für Nachrichtentechnik der Universität Ulm, 2020

ISBN 978-3-948303-10-5

ulm university universität
uulm

Channel Coding for Hardware-Intrinsic Security

A Dissertation by

Sven Müelich

born in Heidenheim

Presented to the Faculty of
Engineering, Computer Science and Psychology
of Ulm University in Germany

and the Institute of
Communications Engineering
supervised by Prof. Dr.-Ing. Martin Bossert

In Partial Fulfillment of the Requirements for the Degree of
Dr. rer. nat.

2019

Acting Dean:	Prof. Dr.-Ing. Maurits Ortmanns Institute of Microelectronics Faculty of Engineering, Computer Science and Psychology Ulm University, Germany
First Reviewer:	Prof. Dr.-Ing. Martin Bossert Institute of Communications Engineering Faculty of Engineering, Computer Science and Psychology Ulm University, Germany
Second Reviewer:	Prof. Dr.-Ing. Georg Sigl Chair of Security in Information Technology TUM Department of Electrical and Computer Engineering Technical University of Munich, Germany

Day of Conferral of Doctorate: October 11, 2019

Acknowledgments

This dissertation contains parts of my work in the field of Physical Unclonable Functions as a research assistant at the Institute of Communications Engineering at Ulm University in Germany. Many people are responsible that I had a great time during the last five years, while enjoying the ups and struggling with the downs caused by science.

First and foremost, I thank my supervisor and the director of the Institute of Communications Engineering, Prof. Dr.-Ing. Martin Bossert, for making this dissertation possible from several perspectives. After being with Martin for more than five years, I have to conclude that his relaxed leadership that boosts motivation and the working environment as well as his blind confidence in his doctoral students is something very special that cannot be expected everywhere else. Also, his continuous financial support enabled many enjoyable journeys to interesting scientific conferences located in, when forgetting about Hamburg–Harburg, beautiful and sunny places.

Furthermore, I would like to express my gratitude to Prof. Dr.-Ing. Georg Sigl from Technical University of Munich for taking the time to review this dissertation. I would also like to thank Prof. Dr. Uwe Schöning, Prof. Dr.-Ing. Frank Slomka and Prof. Dr. rer. nat. Frank Kargl for being part of the colloquium.

Many colleagues made my dissertation time to an, in many aspects, unforgettable experience. I want to thank Sven Puchinger, not only for being a faithful co-author throughout the years, also for being with me on a countless number of trips, with his always enthusiastic never mind personality, even when being tired due to very weird flight routes and schedules, or when being desperate due to getting stuck when using the german railway system for transportation. Mostafa Hosni Mohamed for being a fantastic travel mate, establishing the tradition that all our common trips ended with being sick for both of us. George Yammine for his great support when entertaining "children" at in total 13 events like for example "Schüler Ingenieurs Akademie" or "Studieninfotag", when representing a part of our faculty and presenting our institute's great "cryptodemo".

I am also grateful to those institute members, who are here in order to make my life easier, although when they actually are employed for other reasons: Our system administrators Werner Hack, Werner Birkle and Günther Haas, who always assisted with all kinds of technical issues. Heike Schewe for always providing extensive travel recommendations in the planning phases of all kinds of journeys, as well as Prof. Dr. Dr.-Ing. Wolfgang Minker for enthusiastically planning the cultural program for Sven Puchinger's and my trip to Saint Petersburg as well as for his extremely helpful visa recommendations. Ilse Walter and Michaela Baumann for assistence with bureaucratic issues, most often concerning business travels. Dr. Werner Teich for dealing with all organizational and bureaucratic questions that nobody else was able or willing to deal with.

I want to thank my co-authors (in alphabetical order) for fruitful discussions and nice collaborations: Joachim Becker, Sebastian Bitzer, Martin Bossert, Robert F. H. Fischer, Andreas Herkle, Matthias Hiller, Karim Ishak, Ludwig Kürzinger, Holger Mandry, David Mödinger, Maurits Ortmanns, Sven Puchinger, Johan Rosenkilde né Nielsen, Georg Sigl, Chirag Sudarshan, Antonia Wachter-Zeh, Norbert Wehn and Christian Weis.

I would also like to thank also all my colleagues that have not been mentioned so far for the common time at our institute (in alphabetical order): Zaid Dhannoon, Felix Frey, David Kracht, Dejan Lazich, Cornelia Ott, Johannes Pfeiffer, Michael Schelling, Carmen Sippel, Susanne Sparrer, Sebastian Stern and Henning Zörlein.

During the time of this dissertation, I advised and co-advised several talented students with their bachelor's and master's theses. It was a pleasure to work with (in chronological order) David Mödinger, Karim Ishak, Yonatan Marin, Alexander Tsaregorodtsev, Rebekka Schulz, Veniamin Stukalov, Liming Fan, Jhoiss Balois, Sushmita Raj, Sebastian Bitzer and Musab Ahmed, who all contributed directly or indirectly to this dissertation.

Furthermore, I am grateful to Sven Puchinger, Dejan Lazich, Werner Teich, Carmen Sippel, Cornelia Ott and Günther Haas for proof-reading parts of this dissertation, and also to Fe Hägele for giving valuable linguistic hints on improving the quality of the English text. Finally, I want to thank family and friends for their continuous support.

Ulm, April 2019 Sven Müelich

Abstract

HARDWARE-intrinsic security studies cryptographic methods, whose implementations are assisted by some specific physical properties of the hardware on which they are executed. *Physical Unclonable Functions (PUFs)* are a predominant part of that field and currently an active research area. The most investigated type of PUF is the so-called *silicon PUF*, representing an electronic device, which is embedded in an *integrated circuit (IC)* with some cryptographic functions. PUFs are used to generate a highly secret, time-invariant, true random bit sequence, referred to as *PUF response*. This randomly generated PUF response is unique for each individual PUF device and can easily be reproduced on request inside the IC over its entire lifetime. The PUF response is derived from the inherent randomness of some physical properties occurring from variations in the IC manufacturing process. These variations cannot be controlled with todays technologies. For example, the propagation delay of logic gates or the initialization state of memory cells can be used in order to generate a PUF response. Since such behaviors cannot be controlled, it is extremely unlikely to produce two PUFs with the same response. This is the reason why PUFs are called *unclonable*. Even the IC manufacturer cannot predict the individual response of an embedded PUF without performing a readout after IC manufacturing. If the IC manufacturer prevents the possibility to readout a PUF response in any way, not even by using any kind of IC tampering, the PUF response becomes secret to everyone.

Since PUFs can be interpreted as highly secret, true random bit sources, they are predestined for a variety of cryptographic applications such as, for example, secret key generation and storage, identification and authentication of various entities. A PUF response exists in its binary form only for a very short active time period during execution of the cryptographic function in which it is involved. Otherwise, in predominantly inactive periods, it is hidden in its analog form, consisting of unclonable analog physical parameter values of the PUF device. Every attempt to manipulate these parameter values uncontrollably changes the binary PUF response. Consequently, the PUF response is inseparably merged with the IC hardware and it is not possible to reconstruct its binary value during inactive periods. In the very short active periods, when the PUF response exists in its binary form, its secret can be protected by additional methods. Due to external influences like changes of temperature, supply voltage or IC aging, many PUF variants cannot reproduce their binary responses error-free. For such error-prone PUFs, methods from the field of error-correcting codes have to be applied to reliably reproduce binary PUF responses.

In current applications, however, all PUF types are only equipped with classical error-correcting codes, which are not tailored to the specific properties of individual PUF types. Consequently, the possibilities of reliability improvements of error-prone PUFs are not completely exhausted.

This dissertation considers several aspects of PUFs from the perspective of coding theory. Traditionally, for error correction in PUFs, a worst-case bit error probability is used in order to model the binary symmetric channel. As existing results in the literature indicate, this is a very conservative and sometimes even pessimistic assumption. In the theory of error-correcting codes, knowing characteristics of a channel is always beneficial in order to design codes that lead to an improvement of the error-correction performance. We derive

channel models for two different PUF variants, namely *Ring Oscillator PUFs (ROPUFs)* and *Dynamic Random Access Memory (DRAM) PUFs*. Using DRAM to construct PUFs is a comparatively new approach proposed in the literature. In contrast to the established variants, PUF responses extracted from DRAM are heavily biased towards either "0" or "1", and hence, debiasing methods have to be applied in addition to error correction. We propose methods that can be applied to solve the debiasing problem.

When dealing with noisy responses, secure sketches are a widely used concept. When reproducing a PUF response based on an erroneous re-extracted response, so-called helper data which are calculated and stored during initialization have to be used to map responses to codewords, such that decoding algorithms can be applied. We propose and analyze a new secure sketch that only uses an error-correcting code, but no further helper data. Also, we use our channel model, which we derived for ROPUFs, to construct new secure sketches.

Furthermore, we propose specific code constructions that can be used for error correction in the context of PUFs. Block codes and convolutional codes are considered for that purpose and we explain how to improve existing results from literature by using code classes (Reed–Muller codes, Reed–Solomon codes), decoding techniques (generalized minimum-distance decoding, power decoding, list decoding, using soft information at the input of the decoder, sequential decoding) or coding techniques (generalized concatenated codes), that have not been applied to PUFs before. Our code constructions result in a smaller block error probability, decoding complexity or codeword length in comparison to existing implementations.

The final part of this dissertation deals with security aspects. In particular, we consider timing attacks on the decoding algorithm, as a representative of the huge family of side-channel attacks. We study two techniques to prevent such attacks, namely a masking technique, as well as a modified decoding algorithm with a runtime that is constant and independent of the received word.

Contents

1

Introduction

Kerckhoffs's principle, formulated in 1883, is a very well-known and basic concept in cryptology. It indicates, that the security of a cryptosystem has to depend solely on the secrecy of the key, while the cryptographic algorithm itself is publicly known. The public knowledge of the algorithm allows, that a large community is able to analyze the system concerning its weaknesses. This concept significantly contributes to the improvement of cryptographic algorithms. If no efficient attack is found after a certain amount of time, the system is assumed to be secure with high probability. On the other hand, defining the key to be the only secret is the reason, why generation and protection of the secret key needs special attention.

A classical solution to the problem of key generation is to produce a key by using a *Pseudo Random Number Generator (PRNG)*. A PRNG is a deterministic algorithm, which based on an initial value that is called seed, produces a number that cannot be distinguished from a true random number by applying statistical methods [BRS+10]. While pseudo random numbers produced in such a way are good enough for many applications like, for example, numerical simulations or software tests, their quality is not sufficient for cryptographic purposes. Moreover, an attacker who gets knowledge about the seed for any reasons, is able to reproduce the generated numbers. To increase the quality of random numbers, they can be produced based on the state of complex physical systems, that cannot be influenced by todays technologies.

The classical solution to the problem of secure key storage is to store a key in a non-volatile memory, that is protected against attacks. A memory can be protected by applying physical techniques, which can be implemented by cryptographic co-processors that, for example, notice when probes are used to conduct measurements, or when the power supply is changed. As soon as any manipulation is detected, all secret data will be destroyed. Such a physical protection is expensive and complex. In addition, there exist various attacks that can be performed in order to compromise a secret that is hidden in a non-volatile memory, even when protection mechanisms are used. Information about a famous cryptographic co-processor can, for example, be found in [DLP+01].

As an alternative to the classical solutions, a true random number can be generated directly on an integrated circuit (IC) by exploiting its physical state. Hence, an *embedded Random Number Generator (embedded RNG)* can be designed. For this purpose, many requirements from embedded systems are inherited: ICs are required to be fabricable in mass production with a minimum amount of cost. Additionally, they are expected to consume as little chip area as possible and to efficiently produce random numbers in a short amount of time with high reliability. It also has to be considered, that the environmental conditions are able to

change during operation. According to [Wue08], we distinguish two types of such embedded RNGs: For both types, we consider a set of ICs that is produced with identical lithographic masks and hence, each IC fulfills the same functionality. Each IC implements an embedded RNG that can be invoked by a function call.

1. Type 1: Each time when an IC is invoked by a function call, it generates a new binary random number that is unpredictable and fulfills the security requirements of cryptographic applications. The random numbers that are created by the ICs are independent of each other.

2. Type 2: During initialization, each IC produces a different unpredictable binary random number. In contrast to embedded RNGs of Type 1, further function calls do not generate new random numbers. Instead, with each function call to an IC, the random number which it produced during initialization is regenerated. This behavior also persists in cases where the power supply was interrupted.

The embedded RNGs of type 1 specify *True Random Number Generators (TRNGs)*, while *Physical Unclonable Functions (PUFs)* are used to implement embedded RNGs of type 2. ICs that contain a PUF are used to generate cryptographic keys and simultaneously to protect them from manipulation by providing unclonability. The randomness is extracted from the physical characteristics of the IC. For example, delay characteristics of components or the behavior of memory cells can be exploited. A secret is unique for each IC, since the physical characteristics that are used depend on random variations within the manufacturing processes of physical objects, which cannot be controlled by the manufacturers due to physical and technical reasons. The secret generated in this way can be used in order to replace a PRNG for the secure generation of a cryptographic key.

Since the characteristics that are used in order to extract the secrets are robust over time, random numbers can be regenerated on demand and hence, a non-volatile memory that is used as key storage can be omitted. A secure key storage is implemented implicitly by the reproduction of the secret, when it is required by the cryptosystem. In contrast to a cryptographic co-processor, a PUF does not need energy in order to keep its secret. Hence, the secret is available in the system only for a very limited time. The binary random number that is produced by a PUF is usually called *response* and can be interpreted as the identifier of the IC, similar to a fingerprint that serves as identifier for a human being. In many cases, environmental conditions like temperature or supply voltage are able to induce some errors in the regenerated responses. This behavior makes PUFs interesting for people from the field of coding theory, since error-correcting codes can be used to circumvent this problem.

1.1 History of PUFs

This section briefly summarizes the emerge of PUFs from a historical point of view. One of the most prominent early examples of using randomness in order to identify physical objects is the identification and authentication of missiles during the cold war, as proposed by Bauder in the 1980s [GM89]. Particles are randomly distributed on the surface of missiles. In an initialization phase, pictures are taken while the missiles are illuminated from different angles. Later, for identification, new pictures are compared to the initial ones. More early examples of

identifying objects by using randomness induced through physical properties are summarized in [RDK12].

Usually, the birth of PUFs is accredited to Pappu [Pap01, PRTG02]. Since the idea of using randomness in physical systems for identification purposes was given in the context of one-way functions that are used by various cryptologic concepts, the technology was called *Physical One-Way Functions*. In contrast to PUF constructions which are used nowadays, the functions suggested by Pappu are optical-based instead of silicon-based. The functionality of these so-called *optical PUFs* is similar to the identification of missiles described above. Initially, some particles are randomly distributed on the surface of a transparent token. A laser is used to generate a two-dimensional speckle pattern, depending on the token and the configuration of the laser. Next, a camera is applied to capture that speckle pattern. Finally, the image is transformed to a binary sequence by using a hash function. As can be seen in a comparison of different constructions, the optical PUF can be interpreted as benchmark relating to its properties and security aspects, cf. [Mae13, Table 3.1]. Nevertheless, the optical PUF is rather useless in modern applications, since its operation requires a large experimental setup including all the mentioned components[1].

This problem was tackled from 2002 with the proposal of silicon PUFs, which can directly be implemented on an IC in addition to the ICs functionality [GCVDD02, Gas03]. Randomness is derived from statistical variations that occur during the manufacturing process and affects delays of wires or other components within an IC. Many constructions based on that concept were proposed in the following years. The constructions that are most often used in todays applications, and also within this dissertation, will be discussed in Chapter 2.1.1.

1.2 Related Fields

PUFs constitute an essential part of *hardware intrinsic security*, a field that deals with "security and cryptographic mechanisms embedded in hardware" [SN10]. Besides PUFs, hardware intrinsic security deals with hardware related attacks and countermeasures. This section highlights the overlaps of PUFs and other scientific fields.

Closely related to PUFs is the field of *biometrics*. Both, biometrics and PUFs, are used to identify something based on unique features which are supposed to be unclonable. Biometrics identifies persons, while PUFs are used to identify physical objects. Often, the response of a PUF is called the object's fingerprint. Both fingerprint and responses, identify an object uniquely, are inherently present in the object from the moment of its creation, and are reproducible over time. Also, both fields are dealing with noisy data that have to be used in cryptologic applications, since fingerprints as well as PUF responses are not perfectly reproducible due to physical reasons, such as noise that is present when measuring the state of a physical system. Many methods used for post-processing of noisy PUF measurements, like secure sketches and fuzzy extractors, originate from the field of biometrics. In both fields, physical measurements are translated into bit sequences, which are then used in cryptologic protocols. In contrast to PUFs, some biometric features like fingerprints can be extracted without having access to the physical feature, e.g., fingerprints left on some objects can be restored by using forensic approaches. The secrets of PUFs, however, cannot be revealed without access to the corresponding PUF. Literature, e.g. [TSK07], often explains the common

[1] A miniaturized construction of an optical PUF developed for experimental reasons can be found in [SSO+07].

concepts simultaneously in the context of biometrics and PUFs.

Quantum physics can also be used in order to make objects physically unclonable. According to the no-cloning theorem, it is not possible that an unknown quantum state can be copied. Hence, adding a quantum state, that is only known to an issuer, to a physical object, prevents attackers from cloning the object. Presently, approaches from the field of quantum physics are only of theoretical interest, since they require to maintain quantum states over long periods. Also, implementing quantum physical solutions today contradicts to the aim of providing low-cost solutions.

Since PUF responses are usually noisy, methods from the field of error correction are required. However, PUFs are not the only area within cryptology where error-correcting codes are applied. Cryptosystems based on error-correcting codes are studied in the context of *post-quantum cryptology*, which aims for the design of cryptologic algorithms which, in contrast to the widely used methods based on number theory, are resistant against attacks by quantum computers. An overview about cryptologic protocols based on error-correcting codes can be found in [BBD09].

1.3 Outline

The structure of this dissertation follows the different topics that can be considered in the field of PUFs. First, **Chapter 2** provides an introduction to both, Physical Unclonable Functions and coding theory, in order to connect these to fields as well as their, usually disjoint, communities. In Section 2.1, PUFs are defined and their properties are summarized. The most often used PUF constructions are given as examples of specific implementations, namely the Arbiter PUF, Ring Oscillator PUF (ROPUF) and SRAM PUF. Also, examples for applications of PUFs are provided. Section 2.2 deals with the fundamentals of coding theory and is divided according to the common model used in communications engineering, that consists of transmitter, channel, and receiver. Section 2.3 completes the preliminaries by linking PUFs and coding theory.

In order to simplify the access to the field of PUFs, a design decision was made: The extent of the preliminaries is reduced to the required amount of information that is necessary to study any other chapter of the dissertation. Previous knowledge, which is only needed within a specific chapter, is summarized at the beginning of the corresponding chapter. For example, Reed–Solomon codes, that are used for error correction in Chapter 5, will be introduced at the beginning of that chapter. With this decision in mind, the reader is encouraged to jump to the chapter in which he or she is most interested in, without the torture of going through theories that are not needed there. Chapters 3–6 contain new results. Most of them are already published and hence, the references are given at the end of each chapter's introduction.

As we will learn while studying the preliminaries, error-correcting codes are an essential component when constructing reliable security schemes based on PUFs. When implementing error correction for PUFs, channel and error models have rarely been considered in the literature so far. Instead, most error-correcting schemes for that application are based on very general channel models, like the binary symmetric channel. In **Chapter 3**, a channel and error model for ROPUFs is developed in Section 3.2, inspired by a similar channel model for SRAM PUFs that exists in the literature and is revisited in Section 3.1. Developing PUFs based on DRAM is a comparatively new approach when constructing PUFs. Section 3.3 intro-

duces to the established concepts used to extract responses from DRAM and derives channel models. In general, channel models can be used for improving error correction by selecting coding techniques that are particularly suited for these models. In the field of PUFs, channel models can in addition serve as the basis of so-called secure sketches.

Chapter 4 deals with such secure sketches, which are applied in order to reliably regenerate PUF responses from noisy and thus erroneous measurements. Well-known schemes from literature are reviewed in Section 4.1. New algorithms are constructed in the remainder of the chapter. In Section 4.2, a scheme that avoids extra helper data for regenerating responses is introduced and analyzed. Section 4.3 introduces new soft-decision secure sketches developed for ROPUFs. These algorithms depend on one of the channel models derived in Chapter 3.

Error correction is an important and indispensable component of a secure sketch. In **Chapter 5**, constructions of error-correcting codes that can be applied to improve existing constructions from literature are proposed. The chapter is sub-divided into constructions based on block codes and constructions based on convolutional codes, a classification which represents the differentiation usually done in the channel coding literature. Section 5.1 proposes ordinary concatenated codes based on Reed–Muller and Reed–Solomon codes that improve the results of a reference implementation from literature. Also, for the first time generalized concatenated codes are applied to PUFs. All the proposed code constructions based on concatenated codes outperform a well-known reference implementation. In Section 5.2, techniques from the field of convolutional codes that have not yet been applied to PUFs are discussed, implemented, evaluated and compared to a reference implementation from literature.

Chapter 6 touches the huge field of side-channel attacks. After giving a summary on methods that can be applied to attack PUFs, the contributions of the chapter are approaches to prohibit some types of side-channel attacks. Therefore, the use of a masking scheme as well as the design of a decoding algorithm with constant runtime is proposed.

Finally, **Chapter 7** summarizes and concludes the contents discussed in this dissertation, and provides an outlook to future research in the field of hardware intrinsic security.

2

Preliminaries

This chapter provides general fundamentals about both Physical Unclonable Functions (PUFs) and coding theory. The discussion of preliminaries that are only needed within individual chapters, for example, specific code classes, is postponed until required in favor of an improved readability. In addition, the following chapters aim for being readable widely independent from each other, by knowing the general fundamentals presented in this chapter. Section 2.1 introduces PUFs. Definitions are given and illustrated by examples. Also, some applications are stated in order to emphasize the practical relevance of the topic. Section 2.2 deals with the basic concepts of coding theory. Section 2.3 connects coding theory and PUFs. Hence, the contribution of this chapter is not only to provide a summary of preliminaries that are required to understand the rest of the dissertation, but also an approach to connect the communities from the fields of hardware intrinsic security and coding theory.

2.1 Physical Unclonable Functions

The origin of *Physical Unclonable Functions (PUFs)* is usually accredited to Pappu [Pap01]. This and other early works often use the terms *Physical Random Functions* or *Physical One-Way Functions*. All three terms can be used interchangeably. In this work we use the term Physical Unclonable Function, since that one is nowadays most often be used[1]. We start by defining what PUFs are in Section 2.1.1, state their properties in Section 2.1.2, and provide examples how PUFs actually can be constructed in Section 2.1.3. Finally, Section 2.1.4 gives examples of possible applications for which PUFs can be used. More extensive details about the subjects of Section 2.1 can be found in common literature about PUFs, e.g., [Mae13, BH12, WS14, SN10].

2.1.1 Definitions

Intuitively, a PUF can be understood as a physical object, from which a random bit sequence can be extracted, based on intrinsically present randomness that exists inside this object, due to uncontrollable and thus random variations in manufacturing processes of physical objects. Until today, the term PUF has been used for a variety of constructions. Often, provided definitions are ambiguous and vague. According to the multitude of publications concerning PUFs, the essential properties of PUFs can be summarized as done in Definition 2.1 (a).

[1] Also the term Physically Unclonable Function is widely used. [Mae13, Chapter 2.3.1] provides an extensive discussion about the linguistic differences between Physical and Physically Unclonable Functions.

Additionally, Definitions 2.1 (b) and (c) define subclasses, which are often distinguished in literature.

Definition 2.1. (a) A *Physical Unclonable Function (PUF)* is an unclonable physical object from which a unique and reproducible random bit sequence of finite length n, a so-called *response*, can be extracted.

(b) A PUF is called *silicon PUF*, when an integrated circuit (IC) that can be used as PUF is embedded on a silicon chip (e.g. [GCVDD02]).

(c) A PUF is called *intrinsic PUF*, if it is present on the device without the need of adding additional steps during the manufacturing process.

Definition 2.1 (a) includes the most important properties which a PUF needs to possess: uniqueness, reproducibility, and unclonability. Figure 2.1 visualizes uniqueness and reproducibility. The horizontal axis of the figure visualizes the *uniqueness* property: Let PUF_1, ..., PUF_ζ denote ζ PUFs from the same manufacturing line, i.e., ζ physical objects with the exact same functionality. Although having the same functionality, these objects differ in physical characteristics like, for example, the delay behavior in their components. Such characteristics cannot be influenced by the manufacturer during the manufacturing process due to physical and technical reasons. It is important to note, that these random properties are not added in an extra step during the manufacturing process, but they do intrinsically exist. Hence, they are often denoted as *intrinsic randomness* and are widely known as variabilities in the manufacturing process. They do not actively influence the object's functionality, because these effects are too insignificant for that purpose. However, they can be used in order to extract a bit sequence of finite length n, which depends only on these random effects. The goal is to attain uniqueness, i.e., to use the intrinsic randomness in such a way that no two PUFs produce the same response. In an ideal world, responses from different PUFs differ in 50% of the positions.

Since the characteristics of the objects which are used to produce the PUF responses cannot be controlled by the manufacturer, *unclonability* it attained. Two types of unclonability are distinguished. Physical unclonability, which means the impossibility to produce a physical clone, i.e., it is impossible, even for the manufacturer, to fabricate a physical copy of a given PUF such that the copy produces the exact same response as the original. On the contrary, mathematical unclonability deals with the development of a mathematical clone. This can be a software, based on algorithms from the field of machine learning, which mimics the behavior of the corresponding physical object. Often, a PUF is called unclonable, if both physical and mathematical unclonability is provided.

The vertical axis of Figure 2.1 visualizes *reproducibility*. Since the aforementioned intrinsic randomness is static over the lifetime of a PUF, responses can be re-extracted at different moments in time. Since PUFs behave sensitive to environmental conditions like changes in temperature, supply voltage or chip aging, responses in general are not perfectly reproducible[2]. Hence, responses extracted from the same PUF are expected to differ in some positions. This drawback can be handled by error correction, which is the central topic in this dissertation.

[2] In [LWK15] a perfectly reproducible PUF is proposed. The construction uses an arbiter with feedback, in order to expand the distance of the race of the two signals and thus to have a clear result.

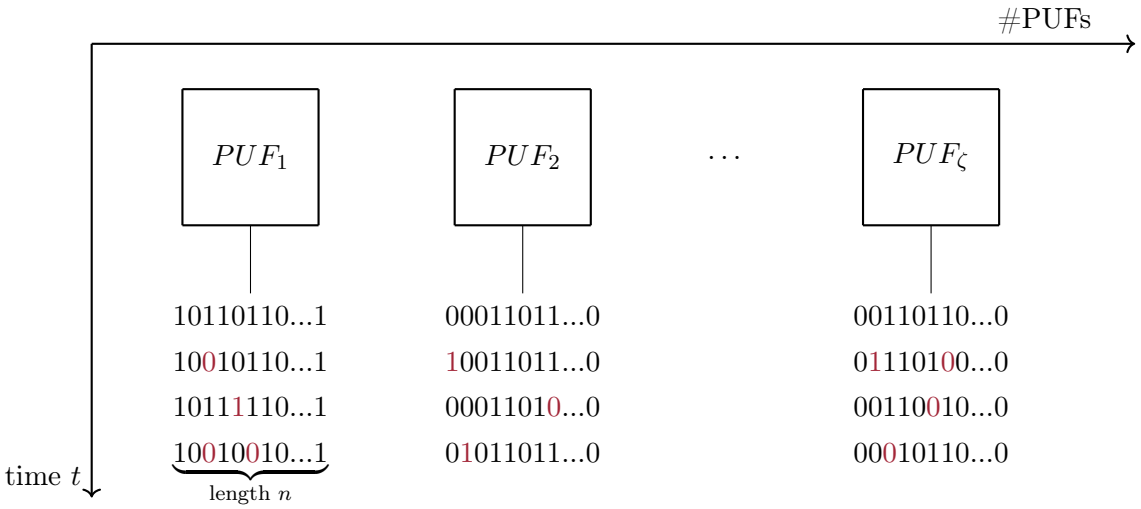


Figure 2.1: Uniqueness and reproducibility are the core properties of PUFs. Uniqueness, represented by the horizontal direction, ensures that responses extracted from different PUFs vary with a sufficiently large distance. Reproducibility, shown in the vertical direction, guarantees that a response of a PUF can be reproduced when needed, by evaluating the same PUF again. Since PUFs behave sensitive to environmental conditions, some bits of the regenerated responses are erroneous, as indicated by the red colored bits.

In addition to the properties uniqueness, reproducibility, and unclonability, Definition 2.1 emphasizes that a PUF is a physical object. Although it is named function, it is important to stress, that a PUF is not a deterministic function in a mathematical sense. As explained in the context of reproducibility, the function's output is not always the same, like we would expect when using a mathematical function. Even when sometimes considered as a deterministic function, it is beneficial to think of it as a probabilistic function or a physical object, like an IC, a chip card, or an FPGA.

A second type of PUF does not only have an output as depicted for the PUFs in Figure 2.1, but also an input. While the output of a PUF is called response, the input is called *challenge*. A pair consisting of a challenge and the corresponding response is called *challenge-response pair (CRP)*. Using a PUF with such a challenge-response behavior, responses depend on the used challenge in addition to the random effects within the device. Depending on the application, either a PUF with or without challenge-response behavior is the proper choice.

Often in the literature, *weak* and *strong* PUFs are distinguished. A PUF is called strong, when the number of possible CRPs is large. "Large" is usually not precisely specified, but means a number large enough, such that an adversary is not able to apply a learning algorithm to produce a mathematical clone. A PUF that is not strong is called a weak PUF. Weak PUFs, in contrast, have a small number of possible CRPs. The extreme case is a PUF from which only one possible response, often called its fingerprint, can be extracted. Such a PUF is also called *Physical Obfuscated Key (POK)*. Common literature, as for example [HYKD14], often classifies specific PUF constructions either into the category strong or weak. However, such a classification can be misleading in some cases, since most constructions can act as

strong or weak PUF, depending on the specific implementation[3].

2.1.2 Quality Measures

Usually, the measures of *average inter-response distance* and *average intra-response distance* are used in order to evaluate the quality of uniqueness and reproducibility, respectively. Most often the Hamming distance

$$\mathrm{dist}_{\mathrm{H}}(\boldsymbol{x}, \boldsymbol{y}) = |\{i : x_i \neq y_i\}|,$$

(where $\boldsymbol{x}, \boldsymbol{y}$ are any vectors of the same length) is used as distance measure when comparing PUF responses. In parts of this dissertation, additional measures are used according to [MCMS10] in favor of comparability of results. Both, the standard as well as the additional quality measures are introduced in Definition 2.2.

Definition 2.2. a) The *average inter-response distance* is a measure used to evaluate the uniqueness of a PUF construction by studying the distance between responses from different devices. The responses of all pairs of PUFs contribute to that calculation, i.e.,

$$\frac{2}{\zeta(\zeta-1)} \cdot \left(\sum_{u=1}^{\zeta-1} \sum_{v=u+1}^{\zeta} \frac{\mathrm{dist}_{\mathrm{H}}(\boldsymbol{r}_u, \boldsymbol{r}_v)}{n} \right), \tag{2.1}$$

where ζ is the number of PUFs, n is the response length, and $\boldsymbol{r}_u$ and $\boldsymbol{r}_v$ are the reference responses of devices u and v, respectively. The optimal value of the average inter-response distance in order to maximize the entropy is 0.5.

b) In a true random bit sequence, we expect half of the bits to be "1" in order to provide a bias-less random number. *The relative Hamming weight* of a PUF response gives us the relative number of ones in that response, i.e., the relative Hamming weight of response $\boldsymbol{r} = (r_1, \ldots, r_n)$ is

$$\frac{1}{n} \cdot \sum_{i=1}^{n} r_i. \tag{2.2}$$

To maximize the entropy, the optimal value of the relative Hamming weight is 0.5.

c) When comparing bit i ($i = 1, \ldots, n$) in responses extracted from different PUFs, the desired aim is that bit position i has a "1" in half of the considered responses. This measure is called *bit-aliasing*. The bit-aliasing value for position i can be calculated by

$$\frac{1}{\zeta} \cdot \sum_{j=1}^{\zeta} r_{i,j}, \tag{2.3}$$

where ζ is the number of PUFs and $r_{i,j}$ is response-bit i extracted from device j. When optimizing the entropy, the desired value of bit-aliasing at position i is 0.5.

[3] This becomes obvious when reviewing the response generation for *Ring Oscillator PUFs (ROPUFs)* and *Static Random Access Memory (SRAM) PUFs* in Examples 2.4 and 2.7, respectively.

d) The *average intra-response distance* is a measure which studies the difference between responses extracted from the same device, in order to evaluate reliability. This is done by calculating the distance between reference response and re-extracted responses. The average intra-response distance of device j is calculated by

$$\frac{1}{\ell} \cdot \sum_{k=1}^{\ell} \frac{\mathrm{dist_H}(\boldsymbol{r}_j, \boldsymbol{r}_{j,k})}{n}, \tag{2.4}$$

where ℓ is the number of re-extracted responses from PUF j, $\boldsymbol{r}_j$ is the reference response of PUF j, and $\boldsymbol{r}_{j,k}$ is the k^{th} re-extracted response from PUF j. Perfect reliability is guaranteed with an average intra-response distance of 0.0. The aim is to obtain an average intra-response distance close to zero.

e) The total number of distinct response bits that altered at least once, considering all measurements from one PUF should be close to zero.

Examples of these measures obtained with real-world PUF data can be found in Appendix A, Figures A.1–A.8.

2.1.3 Examples

In the literature, a multitude of different PUF constructions were proposed. We focus on silicon PUFs, according to Definition 2.1 (b), since this type is usually used for security purposes. Silicon PUFs can be classified into *delay-based PUFs* and *memory-based PUFs*. The intrinsic randomness of delay-based PUFs occurs from delays in the PUFs' components, while memory-based PUFs extract randomness based on random effects concerning memory cells. First, we give two popular examples from the family of delay-based PUFs. Example 2.3 explains Arbiter PUFs as introduced in [LLG+04], while Example 2.4 explains Ring Oscillator PUFs (ROPUFs) as proposed in [GCVDD02].

Example 2.3. Arbiter PUFs were introduced in [LLG+04]. As visualized in Figure 2.2, an Arbiter PUF consists of two, symmetrically designed, paths on an IC. These two paths are simultaneously stimulated with a signal. On the path through the IC, ν switching devices with two inputs in_i^0, in_i^1 and two outputs out_i^0, out_i^1 each, are passed ($i = 1, \ldots, \nu$). Depending on a challenge bit x_i, switch i is configured. If $x_i = 0$, the input in_i^0 of switch i is straightly connected with output out_i^0, and the input in_i^1 is straightly connected with output out_i^1. If $x_i = 1$, the inputs and outputs of switch i are connected crosswise, i.e., input in_i^0 is connected with output out_i^1, and input in_i^1 is connected with output out_i^0. If this construction is used as PUF, the bits $x_1, x_2, \ldots, x_\nu$ can be set by a binary challenge of length ν. Depending on which path is faster when transmitting the signal, the arbiter outputs either "0" or "1", which can be interpreted as response bit.

In order to generate responses of length n, several constructions exist. For example, the circuit can be evaluated n times using n different challenges. Alternatively, n of the circuits visualized in Figure 2.2 can be used to be evaluated with the same challenge. This latter approach only needs ν challenge bits, however much more chip area is required. In order to complicate model building attacks based on machine learning algorithms, an Arbiter PUF construction that uses intermediate results in order to configure some of the switching devices was suggested in [Lim04].

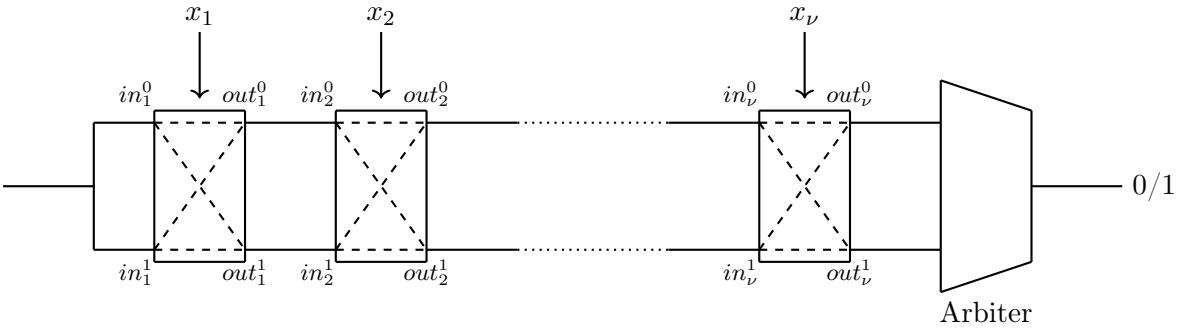


Figure 2.2: Arbiter PUF as proposed in [LLG+04, Lim04]. A signal simultaneously propagates on two different paths, which are configured by a binary challenge. Depending on which path transmits the signal faster, an arbiter generates either "0" or "1".

Example 2.4 explains ROPUFs according to [SD07]. There was an earlier proposal by [GCVDD02, Gas03], however literature most often cites [SD07] as the earliest proposal. Our results in Sections 3.2, 4.3 and 5.2.4 are based on ROPUFs.

Example 2.4. A *ring oscillator* is a circuit that consists of a NAND gate and inverters. If a signal propagates through, the circuit oscillates with a certain frequency, which is determined by the delays in the circuit's components. Since these delays are defined by uncontrollable variations during the manufacturing process, it is impossible to manufacture an ring oscillator with an exact predetermined frequency or two ring oscillators with the exact same frequency. A *Ring Oscillator Physical Unclonable Function (ROPUF)* consists of ξ ring oscillators. The output of one bit is produced by comparing the frequencies of two selected ring oscillators. In order to produce a length-n response, n pairs of ring oscillators have to be selected and compared. Response bit i is produced by comparing the frequencies of the ring oscillators in pair number i. The structure of an ROPUF is shown in Figure 2.3.

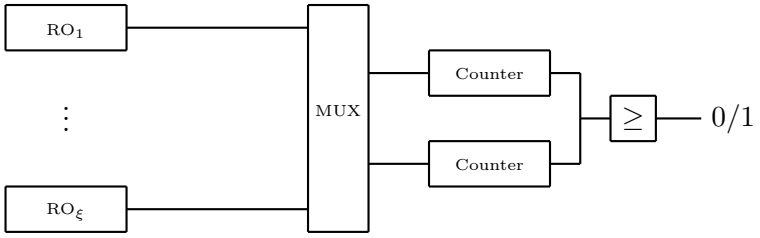


Figure 2.3: Ring Oscillator Physical Unclonable Function (ROPUF). A response bit is produced by comparing the frequencies of two ring oscillators.

a) To realize a weak PUF, a fixed sequence of ring oscillators has to be chosen for comparison. In order to maximize entropy, the response bits have to be generated independently. For fulfilling that requirement, the ring oscillator pairs have to be chosen such that no correlations exist. For example, choosing the $\binom{\xi}{2}$ possible pairs for producing a response of length $\binom{\xi}{2}$ reduces entropy, since we know the bit produced by comparing the ring

oscillators of pair (RO_1, RO_3), when we already know the bits produced by comparison of the ring oscillators in the pairs (RO_1, RO_2) and (RO_2, RO_3). To circumvent this effect, disjoint pairs of ring oscillators have to be chosen.

b) Implementing a strong PUF requires the possibility to produce a huge challenge-response set. Two possible solutions are proposed in [SD07]. Similar to the Arbiter PUF, ring oscillators can be extended, such that the delay path can be configured by challenge bits. When implementing the ROPUF on re-programmable logic like an FPGA, the challenge bits can be used in order to configure the ring oscillators. For example, the number of inverters can be chosen.

ROPUFs are a popular choice for PUF implementations on FPGAs. Since ROPUFs are comparatively slow, large and consume more power than some other constructions, they are not suited for resource-constrained implementations like, for example, RFID [SD07].

Remark 2.5. A variety of improvements has been suggested after ROPUFs were proposed. In [MS09] and [MS11], uniqueness suffering from correlated process variations is improved by suggesting a new design methodology. Also, a configurable ring oscillator design is studied, in order to enhance reliability, which is influenced by environmental conditions. [YQ10] proposes a sequential pairing algorithm to extract a random response from an ROPUF. This approach aims for an improved hardware utilization. [MKS12] suggests a method that can be used to expand the set of CRPs of an ROPUF in an area-efficient way.

Remark 2.6. A dataset, consisting of frequency measurements extracted from ROPUFs, is available in [MCMS10]. Responses generated from the measurements in that dataset are used within this dissertation as a source of real-world ROPUF data. The available frequency data were originally gathered by using 125 devices with 100 extracted responses per device. Later, the set of devices was extended to 193. The original dataset was analyzed by the creators of the dataset in [MCMS10]. Analysis of the extended dataset can be found in Appendix A. Even the amount of data is small from a statistical point of view, Table A.2 in Appendix A shows that in the field of PUFs, that amount of data is much more than the amount of data from which conclusions are usually drawn in the literature.

Besides delay-based PUFs, memory-based PUFs are very popular in applications, since memory cells are intrinsically present in many devices. Most memory-based PUFs extract randomness from the initial values of memory cells. A memory cell is a digital circuit with at least two possible stable states. Usually, when a memory cell is moved to an unstable state, it converges to one of the stable states, influenced by the random physical properties of its components. Example 2.7 explains SRAM PUFs, as introduced independently in [GKST07] and [HBF07]. They are one of the most often used memory-based PUFs. Our results in Chapter 5.2 are based on SRAM PUFs.

Example 2.7. SRAM PUFs are based on the initialization behavior of static random access memory (SRAM) cells. An SRAM cell is a circuit constructed based on two cross-coupled inverters. These memory cells have two possible stable states and, hence, can store two possible values, “0” and “1”. Usually, devices are equipped with huge sets of memory cells and thus are able to store huge amounts of bits. When the device is turned on, each memory cell starts in a so-called unstable state, before directly converging to one of its two possible

stable states. The stable state, which is chosen by a cell, is determined by a static component and by a dynamic component. The static component represents the physical mismatch of the cells' inverters, which is determined once per cell by uncontrollable physical effects during the manufacturing process. The dynamic component represents the noise, which is different for all cells and varies over time. In order to extract a response, a range of memory cells is read after initialization. To implement a strong PUF, a challenge can be used to select the memory cells, which are considered for extracting the response.

Based on experiments with SRAM cells, [GKST07] detected that the stable state, to which a cell converges, is random for each cell. Hence, the uniqueness property is fulfilled. It was also observed, that most of the cells always initialize to the same state with very high probability, and hence, responses can be reproduced. However, a comparably small amount of SRAM cells does not have a preferred initialization behavior. This situation occurs, when the noise dominates over the mismatch of the inverters. This observation explains why responses are not perfectly reproducible. In [MTV09a], a mathematical model for the distribution of SRAM responses was derived. This model is briefly revisited in Chapter 3.1 in the context of channel models and was used to generate responses in the results presented in Chapter 5.2.

Remark 2.8. Responses extracted from SRAM PUFs were published and analyzed in [Wil17]. The published data were gathered by using 144 devices with 101 extracted responses per device, each in 2015 and 2016.

Remark 2.9. Other types of memory-based PUFs have been proposed in the literature. Most of them are based on the same principles as SRAM PUFs, but differ in the components used to construct the memory cells. *Butterfly PUFs*, as proposed in [KGM$^+$08], use two cross-coupled latches instead of inverters, in order to mimic the behavior of an SRAM cell. Using that construction, two problems posed by SRAM PUFs are circumvented: The main drawback of SRAM PUFs is, that bits are extracted after the initialization of the memory cells, leading to a reboot of the device, which is necessary for extracting a response. Second, using re-programmable devices like FPGAs, often zeros are forced to be written into the cells directly after initialization, and hence, the random values cannot be accessed. The memory cells of a Butterfly PUF, however, can be excited by high voltage in order to transfer the cell in an unstable state during the operation of the device. If the voltage is lowered again, the cell will converge to one of the two possible stable states.

A similar construction which uses two cross-coupled NOR-gates, in order to implement a memory cell, was proposed in [SHO07] and can be found under the name *Latch PUF* in established PUF literature. Both, Butterfly and Latch PUFs are not fully intrinsic. Their components are available on the platform, however, they require dedicated circuits which have to be placed carefully in a special manner to guarantee the desired PUF behavior.

Flip-flop PUFs, as proposed in [MTV08], use the powerup values of flip-flops as response bits. In contrast to Butterfly and Latch PUFs, flip-flop PUFs are fully intrinsic, like SRAM PUFs. Like in Butterfly and Latch PUFs, the initial values can be easily extracted without a reboot, even when FPGA implementations are used. The main drawback of flip-flop PUFs is an extensive post-processing of the extracted bits due to a weak response inter-distance.

A comparatively new direction that occurs in the literature, is to use DRAM for the extraction of PUF responses. PUFs based on DRAM additionally use other principles to extract responses than the memory-based constructions discussed so far. Since this dissertation also

includes results concerning the generation of PUF responses from DRAM, a discussion is postponed to Chapter 3.3.

2.1.4 Applications

Most often, PUFs are applied in order to implement secure key generation and secure key storage. However, literature lists many additional applications from the field of cryptology, where PUFs can be applied as an alternative to classical methods. For example, PUFs can be used for identification [HBF07], authentication [SD07], intellectual property protection [GKST07], and counterfeit prevention [TB06]. Since there exists a huge variety of applications, this list does not claim to be complete. Rather, it aims at giving an intuition, that the topic considered in this dissertation is far away from pure theoretical interest and is relevant for lots of practical applications. It also has to be mentioned, that there are already companies, which focus on PUFs in their core business. For example, *Intrinsic ID* applies the SRAM technology in products which can for example be used for authentication purposes [Inc18]. Arbiter and ring oscillator technology can be found in products for authentication and key generation developed by *Verayo* [Ver18]. Also, FPGAs often include PUFs, cf. Altera [LKA15] and Xilinx [Pet18].

Generation of Cryptographic Keys

Cryptographic keys have to be random, unique and unpredictable. In order to provide randomness, *Pseudo Random Number Generators (PRNGs)* are often used. A PRNG is a deterministic algorithm that, based on a short bit sequence (called seed), generates a much longer bit sequence that cannot be distinguished from a true random bit sequence (i.e., a bit sequence generated by a memoryless binary symmetric source) by the use of a polynomial time algorithm (for example statistical tests, cf. [BRS$^+$10]). Two independent studies in 2012 revealed, that many keys for public-key cryptosystems like RSA contain common factors or repeat [HDWH12, LHA$^+$12]. As one of the main reasons for this result, the weak seeding of PRNGs is stated. For a proper operation, seeds need to be truly random. Thus, PUFs can be used in order to provide true randomness for seeding PRNGs.

Example 2.10. When using symmetric cryptosystems, as for example, the *Advanced Encryption Standard (AES)*, the (corrected) PUF response is hashed down to the desired key length and directly used as key for AES. Since AES is a symmetric cryptosystem and the same key is used for encryption and decryption, a possible use case is a scenario where transmitter and receiver are the same instance, which occurs for example in encryption and decryption of storage media. In a communication scenario, a public key cryptosystem can be used to encrypt the AES key generated by the PUF, in order to facilitate a secure key exchange between transmitter and receiver.

Key storage

Often, cryptographic keys are stored in a protected non-volatile memory [DLP$^+$01]. This approach has two essential drawbacks. First, non-volatile memory is comparably expensive and often needs more chip area than available on small devices. Second, there exist physical attacks which can be performed by adversaries in order to get access to a stored key. These

attacks have shown to be possible, even when protected memories are used, cf. for example [TJ09, Tar10]. Using PUFs, a secret key is derived from the extracted response. Since the key can be reproduced when needed, no physical key storage is required. Additionally, the key is only present in the device while it is processed by the cryptosystem and is deleted directly afterwards. This significantly limits the periods where the key is vulnerable.

Example 2.11. Cryptographic modules that are implemented by ICs often need a large number of sensitive security data in order to process their tasks. These sensitive data can be encrypted with a master key that is generated by a PUF and thus is called *IC-Eigenkey* [LW08]. Since the IC-Eigenkey can be regenerated on demand, it does not require a storage. Moreover, all sensitive security parameters that are needed by the system are only stored in an encrypted version, hence protection mechanisms for those data can be omitted.

Identification

RFID (radio frequency identifier) tags are widely applied for identification purposes. For example, RFID chips can be implemented on chip cards that can be used for access control. Also, products can be equipped with RFID tags in order to implement tracking techniques (e.g., electronic product code). In a classical system, identifiers are manually generated and stored in a non-volatile memory during an initialization phase. Using a PUF, the identifier can be derived from the system's intrinsic randomness and can be regenerated in the reproduction phase. Hence, additional chip area and process cost of implementing a non-volatile memory can be omitted. In [HBF07], an SRAM PUF is used to implement identification. Identification using PUFs can, for example, also be used for intellectual property protection and counterfeit prevention, as shown in the following example.

Example 2.12. In a classical, i.e., non-PUF solution, the manufacturer generates a unique identifier and stores it in a non-volatile memory on the chip. At the same time, the manufacturer stores the identifier in a database which contains the identifiers of all products. This database is assumed to be a trusted instance. In order to verify that the product is genuine, i.e., not a counterfeit, the user sends the identifier which is stored on a silicon chip to the manufacturer. The manufacturer checks, whether or not the identifier is contained in the database, and if yes, confirms that the product is genuine. There exist methods which an attacker can apply in order to extract the secret identifier from the chip's memory. As soon as an identifier is known, counterfeited products can be produced by storing the stolen identifier in their memories. In order to circumvent such attacks, PUFs can be used. Since the secrets are generated on demand when needed, there is no need to store them in a non-volatile memory. Solutions for counterfeit prevention using PUFs were studied for example in [TB06]. PUFs for counterfeit prevention are for example used by Canon [Can15]. The package of a camera is equipped with an RFID tag containing a PUF. To verify the product, the user can use his smartphone in order to initiate the PUF to produce the identifier, which is then directly transmitted to the manufacturers database. For example, the company *Toppan Printing* integrates PUFs into RFID tags for purposes in the context of counterfeit prevention [KHS12] .

2.2 Coding Theory

Error-correcting codes are used to protect information from alterations caused by noise during transmission. Linear error-correcting codes are a direct application of linear algebra, since they are defined to be vector subspaces. If not stated otherwise, the statements about linear algebra in this chapter can be found in any standard linear algebra textbook, e.g., [Mey00, Axl15, Sin13, Hog17]. This chapter deals with the basics of error correction and briefly summarizes the concepts from the field of coding theory which will be used in the remainder of this dissertation. The typical scenario of information transmission is outlined and the corresponding components, namely transmitter, channel and receiver, are detailed. The specific code classes and their decoding algorithms that are used within this dissertation are introduced in the chapters in which they are applied to PUFs. For extensive details about coding theory, we refer to the standard literature, e.g., [Bos99, Bla03, LC04, KKS05].

2.2.1 Fundamentals

Error correction is used in a multitude of data transmission and data storage scenarios. Since data usually get corrupted by noise during transmission, error-correcting codes have to be applied in order to add redundancy to the information, which is used to detect or correct errors on the receiver's side. Beside classical scenarios in the field of communications engineering, error-correcting codes are also applied to network coding [ACLY00], distributed storage [DGW+10], as well as in the design of public-key cryptosystems, identification schemes and digital signature schemes [BBD09]. Furthermore, they can be used for quantum cryptographic key distribution or key agreement protocols over quantum channels [Djo12, LB13]. Likewise, many schemes in the field of *private information retrieval* are based on error-correcting codes [CGKS95].

We begin our discussion about coding theory with the definition of an error-correcting code. In the fields of computer science and information theory, the term "code" occurs in different contexts, most often related either to error correction (*error-correcting codes*) or source coding (*source codes*). Since we are only dealing with error-correcting codes within this dissertation, we use the word "code" and it implicitly means an error-correcting code.

Definition 2.13. (a) Let $\mathbb{F}_q$ be any finite field, where q is a prime power. A *code* $\mathcal{C}$ is a subset of the vectorspace $\mathbb{F}_q^n$, i.e., $\mathcal{C} \subseteq \mathbb{F}_q^n$. A code $\mathcal{C}$ is called a *linear code* of dimension k, when $\mathcal{C} \subseteq \mathbb{F}_q^n$ is a k-dimensional subspace of $\mathbb{F}_q^n$.

b) A linear code over $\mathbb{F}_q$ is most often denoted as $\mathcal{C}(q; n, k, d)$, where n is the *codeword length*, k the *dimension*, and

$$d = \min_{\substack{\boldsymbol{c},\boldsymbol{c}' \in \mathcal{C}, \\ \boldsymbol{c} \neq \boldsymbol{c}'}} \mathrm{dist}_{\mathrm{H}}(\boldsymbol{c}, \boldsymbol{c}') \tag{2.5}$$

the *minimum distance*[4]. When $q = 2$, $\mathcal{C}$ is called a *binary code*, often denoted as $\mathcal{C}(n, k, d)$ instead of $\mathcal{C}(q; n, k, d)$. The ratio $\frac{\log_q(|\mathcal{C}|)}{n}$ is called the code rate. The minimum distance is important, since it can be used to derive statements about the error detection

[4]For linear codes, the minimum distance equals the minimum weight $\min_{\boldsymbol{c} \in \mathcal{C}, \boldsymbol{c} \neq 0}\{\mathrm{wt}(\boldsymbol{c}) : \boldsymbol{c} \in \mathcal{C}\}$, where $\mathrm{wt}(\boldsymbol{c}) = |\{i : c_i \neq 0\}|$ is the Hamming weight of vector $\boldsymbol{c}$.

and error correction capabilities of a code. If $\mathcal{C}$ has minimum distance d, in total $d-1$ errors can be detected and $\lfloor \frac{d-1}{2} \rfloor$ errors can be corrected by using a *bounded minimum distance (BMD) decoder*. There are other types of decoders that are able to correct a number of errors beyond half-the-minimum distance, for example *list decoders* (used in Chapter 6), or *generalized-minimum-distance (GMD) decoders* (applied in Chapter 5.1).

c) In addition to errors, erasures can occur in some scenarios. In contrast to an error that changes the value of a code symbol into an other value of the same alphabet, an erasure turns a symbol into an additional symbol, denoted by Δ. Using an algebraic error and erasure decoding algorithm, τ errors and δ erasures can be corrected while $2\tau + \delta < d$. In this dissertations, erasures are used in Chapter 3.3 and Chapter 5.1.

d) In coding theory, *performance* describes the error correction capabilities of a code, while the term *complexity* is related to the runtime of a decoding algorithm.

Usually, due to their algebraic properties, linear codes are widely used in applications. Since linear codes are subspaces, concepts from linear algebra can directly be applied. This is useful for defining codes as well as for the design of efficient decoding algorithms. Non-linear codes, on the other hand, often have better properties, but cannot be decoded as efficiently as linear codes. In this dissertation only linear codes are considered.

Typically, two major families of error-correcting codes are distinguished, *block codes* and *convolutional codes*. Using a block code, the information sequence that is sent from transmitter to receiver is divided into blocks of length k. We denote such an information block as $\boldsymbol{i} \in \mathbb{F}_q^k$. Each block is encoded, transmitted and decoded independently from the other blocks. There exist two paradigms that can be used in order to decode block codes, namely *algebraic decoding* (which will be used in Chapter 5.1) and *iterative decoding* (which will be used in Chapter 4.2). In contrast to block codes, when using convolutional codes, the mapping from information to codewords also depends on previous information blocks. In this dissertation, block codes are applied to PUFs in Chapter 5.1, while convolutional codes are used in Chapter 5.2.

The standard model used for information transmission is shown in Figure 2.4. Information is sent over a noisy channel from a transmitter to a receiver. In order to protect information against disturbances, the transmitter uses an encoder to add additional symbols which are used to detect or correct errors in the transmitted information by using the decoder on the receiving side. Using this model, transmission of information can happen in two possible scenarios: First, information can be transmitted in space, i.e., over a distance from a transmitter to a receiver. Second, the channel can represent a storage medium where the transmitter stores information over time. The receiver, who can be equal to the transmitter in this scenario, represents the access to the information at a later point in time. We want to emphasize, that the noise which corrupts symbols exists due to physical reasons and not due to malicious adversaries. This perception differentiates coding theory from cryptology. The remainder of Chapter 2.2 details the components shown in Figure 2.4.

2.2.2 Transmitter

Let $k \leq n$. The transmitter adds $n-k$ redundancy symbols to an information word of length k, such that the resulting word is an element of the chosen error-correcting code $\mathcal{C}$. This is done in a component called *encoder*.

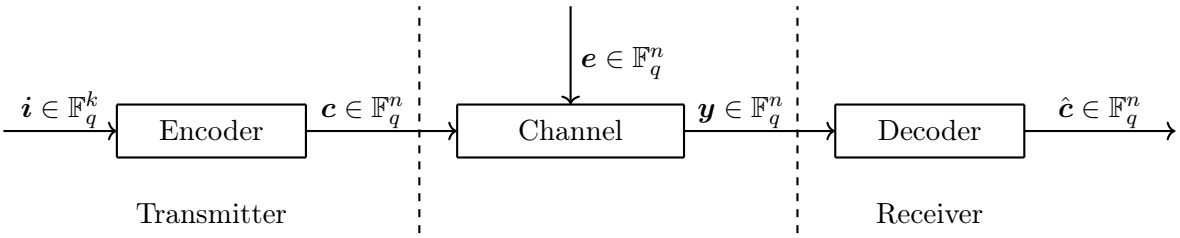


Figure 2.4: Information transmission scenario: A transmitter maps information $\boldsymbol{i}$ to a codeword $\boldsymbol{c}$ before the transmission over a noisy channel. The codeword contains redundancy which is used by the receiver's decoder in order to detect or correct errors in the received word $\boldsymbol{y}$.

Definition 2.14. An encoder is an injective function $\mathsf{enc} : \mathbb{F}_q^k \to \mathbb{F}_q^n$ that maps the information block $\boldsymbol{i} = (i_1, \ldots, i_k)$ to a code block $\boldsymbol{c} = (c_1, \ldots, c_n)$.

For linear codes, the encoder enc can be realized by using a generator matrix $\boldsymbol{G} \in \mathbb{F}_q^{k \times n}$, whose k rows compose a basis of the code $\mathcal{C}$. The resulting encoder is defined as

$$\begin{aligned} \mathsf{enc} : \mathbb{F}_q^k &\to \mathbb{F}_q^n, \\ \boldsymbol{i} &\mapsto \boldsymbol{i} \cdot \boldsymbol{G}. \end{aligned} \tag{2.6}$$

Using the generator matrix $\boldsymbol{G}$, the code can be defined as the set

$$\mathcal{C} = \{\boldsymbol{c} \in \mathbb{F}_q^n : \boldsymbol{c} = \boldsymbol{i} \cdot \boldsymbol{G}, \boldsymbol{i} \in \mathbb{F}_q^k\}. \tag{2.7}$$

2.2.3 Channel

The channel models the physical medium which is used to transmit information, e.g., radio channel, copper wire, fiber optics, or even a storage medium. During the transmission over a channel, errors occur due to noise and other disturbances, which are present because of physical reasons like for example crosstalk or thermal noise that occurs when transmitting information over a discrete-time channel. Often, this behavior is denoted as

$$\boldsymbol{y} = \boldsymbol{c} + \boldsymbol{e}, \tag{2.8}$$

where an error vector $\boldsymbol{e}$ is added to a codeword $\boldsymbol{c}$ during transmission. The elements of the error vector $\boldsymbol{e}$ specify, whether or not the code symbols at the corresponding positions were altered during transmission. Since the probability of less errors is always assumed to be larger than the probability of many errors, $\boldsymbol{e}$ usually is a low-weight vector[5], often also called *sparse*. The vector $\boldsymbol{y}$ is called *received word*.

[5]Different measures of weight exist, usually Hamming weight which counts the number of non-zero positions in a vector is used.

To describe the channel, a variety of *channel models* exist. We will explain the *Binary Symmetric Channel (BSC)* as well as the *Additive White Gaussian Noise (AWGN) Channel*, since these are the basic channel models used within this dissertation. The BSC is used when two symbols (usually "0" and "1") can be transmitted, and the probability p_b of an altered bit is the same for all transmitted symbols. As visualized in Figure 2.5, a bit is altered with *bit error probability* (*bitflip probability*, *crossover probability*) p_b, and is transmitted correctly with probability $1 - p_\mathrm{b}$.

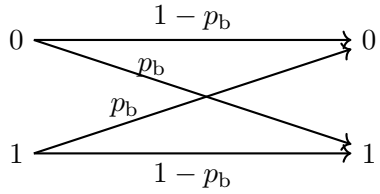


Figure 2.5: The binary symmetric channel (BSC) alters a bit with probability p_b and transmits it correctly with probability $1 - p_\mathrm{b}$.

Another channel that serves as a model for many scenarios in communications, is the time-discrete value-continuous memoryless *Additive White Gaussian Noise (AWGN)* channel. Using this channel model, the noise added to the codeword is a sample from a Gaussian distribution with mean $\mu = 0$ and variance σ^2, i.e., a sample of

$$\mathrm{pdf}(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{(x-\mu)^2}{2\sigma^2}}, \tag{2.9}$$

where pdf denotes the *probability density function* of the Gaussian distribution.

2.2.4 Receiver

To discuss the receiver's side, first some definitions are provided.

Definition 2.15. Let $\mathcal{V} \subseteq \mathbb{F}_q^n$ be a linear subspace of the vector space $\mathbb{F}_q^n$. We call

$$\mathrm{OC}(\mathcal{V}) := \left\{ \boldsymbol{u} \in \mathbb{F}_q^n : \boldsymbol{u}\boldsymbol{v}^\top = 0 \; \forall \boldsymbol{v} \in \mathcal{V} \right\} \tag{2.10}$$

the orthogonal complement of $\mathcal{V}$.

Remark 2.16 summarizes some facts from linear algebra and sets up notation, which is used throughout this dissertation.

Remark 2.16. (a) The dimension of the orthogonal complement $\mathrm{OC}(\mathcal{V})$ is

$$\dim(\mathrm{OC}(\mathcal{V})) = \dim(\mathbb{F}_q^n) - \dim(\mathcal{V}) = n - \dim(\mathcal{V}), \tag{2.11}$$

cf. e.g. [Mey00, Chapter 5.11].

(b) When $\mathcal{V}$ is a code, $\mathrm{OC}(\mathcal{V})$ is called the *dual code* of $\mathcal{V}$. Following the standard notation in coding theory, we also write $\mathcal{V}^\perp := \mathrm{OC}(\mathcal{V})$.

(c) Due to Remark 2.16(b), in coding theory often the term dual is used instead of orthogonal. For this reason, these terms are used interchangeably within this dissertation.

(d) Let $\boldsymbol{H}^{m\times n}$ be a matrix with row space $\langle \boldsymbol{H}\rangle := \{\boldsymbol{H}\cdot\boldsymbol{x} : \forall \boldsymbol{x} \in \mathbb{F}_q^n\}$. For convenience, we define $\mathrm{OC}(\boldsymbol{H}) := \mathrm{OC}(\langle \boldsymbol{H}\rangle)$.

(e) $\boldsymbol{H}$ is called *parity-check matrix* of a linear code $\mathcal{C}$, when it is a basis of $\mathrm{OC}(\mathcal{C})$.

(f) If $\boldsymbol{H}$ generates $\mathrm{OC}(\mathcal{C})$, but is not necessarily a basis, we call $\boldsymbol{H}$ a *decoding matrix* of $\mathcal{C}$. Every parity-check matrix is a decoding matrix, but the inverse statement is not true.

(g) The rows of a parity-check matrix (or decoding matrix, respectively) are called *parity-check equations*.

The receiver uses a parity-check matrix $\boldsymbol{H}$ to detect, whether or not a valid codeword was received. A parity-check matrix $\boldsymbol{H}$ is constructed, such that

$$\boldsymbol{H}\cdot\boldsymbol{y}^\top = \boldsymbol{0} \Leftrightarrow \boldsymbol{y} \in \mathcal{C}, \tag{2.12}$$

where $\boldsymbol{y}$ is the received word. Using the parity-check matrix $\boldsymbol{H}$, the corresponding code can be defined as the set

$$\mathcal{C} = \{\boldsymbol{c} \in \mathbb{F}_q^n : \boldsymbol{H}\cdot\boldsymbol{c}^\top = \boldsymbol{0}, \boldsymbol{c} \in \mathbb{F}_q^n\}. \tag{2.13}$$

Literature distinguishes two main decoding paradigms, *hard-decision decoding* and *soft-decision decoding*. Using hard-decision decoding, the decoding algorithm only uses the received symbols, whereas soft-decision decoding additionally uses reliability information about the received symbols. As quantitative measure of reliability, so called L-values (log-likelihood-ratios, LLRs) are used. For $1 \le i \le n$, let y_i be the received symbol, while c_i denotes the corresponding transmitted symbol. We define

$$L_{\mathrm{ch}}(y_i) = L(y_i|c_i) = \log_e\left(\frac{\mathrm{P}(y_i|c_i=0)}{\mathrm{P}(y_i|c_i=1)}\right) \tag{2.14}$$

as channel L-value. The specific value of $L(y_i|c_i)$ depends on the channel model and can be derived from the channel characteristics. We give examples for the calculation of channel L-values when using a BSC and an AWGN channel according to [Bos99, Chapter 7.2.2].

Using a BSC with bit error probability p_{b}, the channel L-value is calculated as

$$L(y_i|c_i) = \begin{cases} \log_e\left(\frac{1-p_{\mathrm{b}}}{p_{\mathrm{b}}}\right), \text{ if } y_i = 0 \\ \log_e\left(\frac{p_{\mathrm{b}}}{1-p_{\mathrm{b}}}\right), \text{ if } y_i = 1. \end{cases} \tag{2.15}$$

For the channel L-Value calculation of the AWGN channel, we apply BPSK-modulation[6]. Since the AWGN channel is used, the distribution of the received symbols is

$$\mathrm{P}(y_i|c_i) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\left(\frac{y_i-c_i}{\sigma}\right)^2}, \tag{2.16}$$

[6]BPSK (Binary Phase Shift Keying) modulation is defined by the mapping $c_i \mapsto (-1)^{c_i}$, i.e., $0 \mapsto (-1)^0 = 1$ and $1 \mapsto (-1)^1 = -1$.

(cf. Section 2.2.3). The channel L-value can be calculated according to Equation 2.14. First, we consider the case $y_i = 1$ and get

$$\begin{aligned} L(y_i|c_i) &= \log_e\left(\frac{\mathrm{P}(y_i|c_i=1)}{\mathrm{P}(y_i|c_i=-1)}\right) \\ &= \log_e\left(\frac{\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2}\left(\frac{1-1}{\sigma}\right)^2}}{\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2}\left(\frac{1+1}{\sigma}\right)^2}}\right) \\ &= \log_e\left(\frac{1}{e^{-\frac{2}{\sigma^2}}}\right) \\ &= \frac{2}{\sigma^2}. \end{aligned} \tag{2.17}$$

Next, we consider the case $y_i = -1$ and obtain

$$\begin{aligned} L(y_i|c_i) &= \log_e\left(\frac{\mathrm{P}(y_i|c_i=1)}{\mathrm{P}(y_i|c_i=-1)}\right) \\ &= \log_e\left(\frac{\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2}\left(\frac{-1-1}{\sigma}\right)^2}}{\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2}\left(\frac{-1+1}{\sigma}\right)^2}}\right) \\ &= \log_e\left(e^{-\frac{2}{\sigma^2}}\right) \\ &= -\frac{2}{\sigma^2}. \end{aligned} \tag{2.18}$$

In total this results in the channel L-values

$$L(y_i|c_i) = \underbrace{\frac{2}{\sigma^2}}_{:=L_{\mathrm{ch}}} \cdot y_i. \tag{2.19}$$

The L-value

$$L(c_i) = \log_e\left(\frac{\mathrm{P}(c_i=0|y_i)}{\mathrm{P}(c_i=1|y_i)}\right) \tag{2.20}$$

of code symbol c_i can be calculated based on the channel L-value L_{ch}. For a derivation we refer to [Bos99, Chapter 7.2.2].

We conclude the discussion about soft-decision decoding and L-values by providing an interpretation of the calculated L-values: The absolute value of $L(c_i)$ gives a measure of the reliability of the result. The sign of $L(c_i)$ determines the corresponding hard-decision.

2.3 Coding Theory for Physical Unclonable Functions

Due to the erroneous reproduction of PUF responses that was already discussed in Chapter 2.1, error correction is an indispensable component when PUFs are applied for cryptographic applications. Section 2.3.1 explains the framework in which error-correcting codes are used within the PUF scenario. Section 2.3.2 focuses on the selection of specific codes to be applied.

2.3.1 Secure Sketches and Fuzzy Extractors

When PUFs are used in cryptographic implementations, two problems occur. First, as explained in Section 2.1.1, responses are usually not perfectly reproducible. Second, responses are often not perfectly uniformly distributed. To tackle these problems, the concepts of secure sketches and fuzzy extractors have to be applied. Both concepts are formally defined in [DRS04, DORS08]. We use a more intuitive way for explaining their main principles.

A *secure sketch* is an algorithm that is used in order to guarantee error-free reproduction of PUF responses. Two phases of the algorithm are distinguished, namely initialization and reproduction. The *initialization phase*, which is assumed to take place in a secure environment, is executed only once. During initialization, an initial response $\boldsymbol{r}$, often called reference response (or golden response), is extracted from the PUF. Based on this response, which is assumed to be the secret, a binary vector $\boldsymbol{h}$ is generated and stored in a public storage. Since $\boldsymbol{h}$ is used later for reproducing the initial response $\boldsymbol{r}$ based on a noisy re-extracted response $\boldsymbol{r}'$, the vector $\boldsymbol{h}$ is called *helper data*. Since the helper data are allowed to be publicly available, the secure sketch has to guarantee, that not much information about the secret is leaked. There exist different methods in order to produce $\boldsymbol{h}$ based on the initial response $\boldsymbol{r}$, for example the code-offset construction as proposed in [JW99]. A random codeword of an error-correcting code $\mathcal{C}$ is chosen and a component-wise XOR operation of $\boldsymbol{c}$ and $\boldsymbol{r}$ is performed in order to obtain $\boldsymbol{h}$, i.e.,

$$\boldsymbol{h} = \boldsymbol{r} \oplus \boldsymbol{c}. \tag{2.21}$$

The *reproduction phase* is executed whenever the system wants the PUF to reproduce its initial response $\boldsymbol{r}$. A probably noisy response $\boldsymbol{r}'$ is extracted from the PUF. Since $\boldsymbol{r}'$ is expected to differ from the initial response $\boldsymbol{r}$ in some positions, it can be expressed by

$$\boldsymbol{r}' = \boldsymbol{r} \oplus \boldsymbol{e} \tag{2.22}$$

for some low-weight error vector $\boldsymbol{e}$. Since $\boldsymbol{r} = \boldsymbol{c} \oplus \boldsymbol{h}$ according to (2.21), the helper data $\boldsymbol{h}$ can be used to transform $\boldsymbol{r}'$ into the format codeword plus error, since

$$\begin{aligned} \boldsymbol{r}' &= \boldsymbol{r} \oplus \boldsymbol{e} \\ &= \boldsymbol{c} \oplus \boldsymbol{h} \oplus \boldsymbol{e}, \end{aligned} \tag{2.23}$$

and hence a componentwise XOR operation of $\boldsymbol{r}'$ and $\boldsymbol{h}$ results in $\boldsymbol{c} \oplus \boldsymbol{e}$. This is the format that is required by a decoding algorithm for code $\mathcal{C}$. If the weight of error vector $\boldsymbol{e}$, or equivalently the distance between $\boldsymbol{r}$ and $\boldsymbol{r}'$ is within the error correction capabilities of code $\mathcal{C}$, the initial response $\boldsymbol{r}$ can be reproduced. For practicability of a secure sketch, both initialization and reproduction need to be efficiently, i.e. in polynomial time, computable.

A secure sketch as explained so far, solves the problem of not perfectly reproducible responses by applying an error-correcting code $\mathcal{C}$. A fuzzy extractor additionally solves the problem of not perfectly distributed PUF responses by using a hash function. In both phases, initialization and reproduction, a hash function is applied in order to hash the response $\boldsymbol{r}$ to the final key. Note that a secure sketch is included in a fuzzy extractor. The term *helper data algorithm* is often used interchangeably with the term fuzzy extractor. Chapter 4 provides more details about secure sketches as well as specific examples on how they can be implemented.

Helper data are needed, since the initial response $\boldsymbol{r}$ in general is not a codeword. Assuming that the initial response is a codeword leads to a second approach, which we proposed in [MB17a]. This approach does only need an error-correcting code, but any further helper data can be omitted. Instead of using a chosen code, a code has to be constructed such that the initial response is a codeword. This approach will be studied in Chapter 4.2.

2.3.2 Error Correction for PUFs

In order to implement error correction for PUFs, the following information are of interest. First, the code designer needs to know the required key length as well as the error probability which is tolerated at most when reproducing a key. The *block error probability* P_{err} (*failure error probability*, *word error probability*) is the probability that a decoding result differs from the transmitted codeword. In the context of PUFs, this is the probability that the reference response $\boldsymbol{r}$ is reproduced erroneously. When PUFs are implemented on FPGAs, usually the goal is to obtain $P_{\text{err}} \leq 10^{-6}$ when providing methods for error correction, cf. e.g. [BGS+08]. For implementation on ASICs, P_{err} is typically desired to be $\leq 10^{-9}$. The specific target value of P_{err} depends on the underlying hardware platform. The threshold 10^{-6} for FPGAs can be derived from the reliability reports of FPGA manufacturers like Altera and Xilinx [LKA15, Xil19]. Using an error-correcting code that is guaranteed to correct at most t errors, decoding can fail when more than t errors occur. Hence, the block error probability for a BSC is calculated as

$$P_{\text{err}} = \sum_{i=t+1}^{n} \binom{n}{i} \cdot p_{\text{b}}^{i} \cdot (1-p_{\text{b}})^{n-i} = 1 - \sum_{i=0}^{t} \binom{n}{i} \cdot p_{\text{b}}^{i} \cdot (1-p_{\text{b}})^{n-i}, \tag{2.24}$$

and has to be $\leq 10^{-6}$ when an FPGA implementation is used. Detailed information about required key length and tolerated block error probability usually follow from the specification of the application for which the PUF is intended to be used.

Additionally, channel characteristics are useful for choosing a suitable code. At the very least, the worst-case probability of a bit error has to be known. Many studies, e.g. [MVHV12], assume that each bit might alter with the highest probability that was observed. Other studies use more detailed information about the channel. Soft information are applied for example in [MTV09a, MTV09b]. Chapter 3 of this dissertation deals with error and channel models.

There are essentially three criteria that are used in order to evaluate the quality of an error correction component for PUFs, namely required chip area, required size of the helper data storage and runtime. These criteria are possible to be contradictory, thus, a trade-off has to be found depending on the considered application. According to [HPS15], area considerations are much more important than runtime, when providing error correction for PUFs. A summary of code constructions, which were used for error correction in the context of PUFs in the literature, can be found in Appendix B, Table B.1.

3

Error and Channel Models

Traditionally, when working with PUFs, a *binary symmetric channel (BSC)* with a fixed bit error probability p_b is assumed as channel model. Figure 3.1 recalls from Section 2.2.3, that a bit which is transmitted over a BSC is altered with probability p_b, and is transmitted correctly with probability $1 - p_b$. This behavior is independent for all bits that are transmitted. [GKST07] performed practical experiments with SRAM PUFs in order to estimate p_b. Therefore, using four SRAM PUFs, 92 responses were extracted from each device and the number of altered bits was determined. A bit error probability of $p_b = 0.12$ was derived from the maximum number of altered bits that were observed during these experiments. The authors decided to make a conservative assumption and defined $p_b = 0.15$ as bit error probability, which they used in their work. In [BGS+08], also SRAM PUFs were used. The channel in that work was also modeled as BSC with bit error probability $p_b = 0.15$. In [MVHV12], ROPUFs were used for the implementation of a PUF-based cryptographic key generator. Similar to the approach in [GKST07], experiments were performed in order to estimate the bit error probability of the BSC. Based on experiments for different scenarios, the authors decided to work with the three different bit error probabilities $p_b = 0.12$, $p_b = 0.13$, and $p_b = 0.14$.

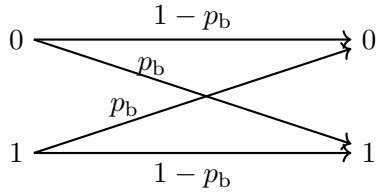


Figure 3.1: Transmission probabilities of the binary symmetric channel (BSC).

In [MTV09a], the channel model of a BSC with a fixed bit error probability was changed to a BSC with different bit error probabilities for the several response positions. An illustration of this channel model can be derived from the original explanation and is provided in Figure 3.2. Hence, the bit error probability for a response position is described by a random variable, whose underlying distribution was derived in [MTV09a] and is revisited in Section 3.1. It forms a basis for our results in Section 3.2. The study in [MTV09a] leads to the insight, that many SRAM cells can be modeled by a BSC with a bit error probability, significantly smaller than the average. Since there are only a few cells, whose bit error probabilities exceed the average, using a fixed worst-case bit error probability for all response bits is a too conservative assumption. On the other hand, considering adequate bit error probabilities for the several

response positions enables an improvement in the design of the error correction component. Note that in contrast to the fixed BSCs used in [GKST07, BGS+08, MVHV12], which are valid for the whole set of considered PUFs, the bit error probabilities used in [MTV09a] depend on a specific PUF instance. Hence, bit error probability p_{b_i} of response bit i is not the same for all PUF instances.

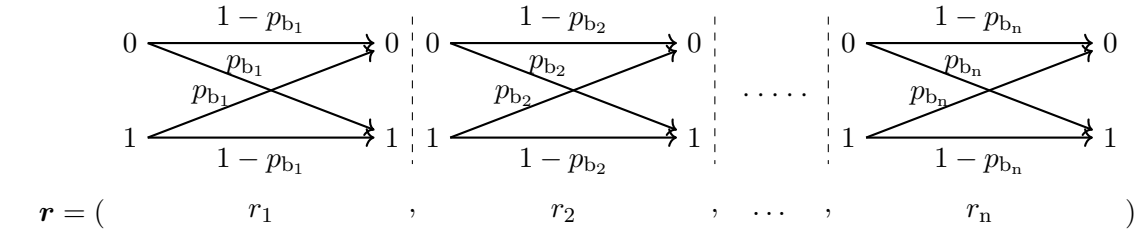


Figure 3.2: BSC with varying bit error probabilities $p_{\mathrm{b}_i} (i = 1, \ldots, n)$ for all bits $r_1, r_2, \ldots, r_n$, i.e., each bit r_i is modeled by an individual BSC with bit error probability p_{b_i}.

An essential factor, often not considered in the fields of security and hardware engineering, is knowing the details about the characteristics of the channel. This allows much more efficient implementations of error-correction techniques, due to the existence of codes that are specialized to certain channels. The main contributions of this chapter are studies of channel models for ROPUFs (Section 3.2) and DRAM PUFS (Section 3.3). We already published the ROPUF channel model as one of the contributions in [MPSB19]. Some of the results about DRAM PUFs are included in [MBS+19]. As preliminary study, the SRAM channel model from the literature [MTV09a] is revisited in Section 3.1.

3.1 Revisiting a Channel Model for SRAM PUFs

A channel model for SRAM PUFs was derived in the literature, cf. [MTV09a, Chapter 3]. Since we use an analog approach for deriving a channel model for ROPUFs in Section 3.2, we first revisit the SRAM approach in this section. We begin with setting up the notation: Every SRAM cell is defined by two parameters, M and N. The parameter M is a normal distributed random variable with mean μ_{M} and standard deviation σ_{M}, i.e.,

$$\mathrm{M} \sim \mathcal{N}(\mu_{\mathrm{M}}, \sigma_{\mathrm{M}}), \tag{3.1}$$

and represents the process variation of the inverters of the SRAM cell. Hence, it is a static component, which is captured once for every memory cell during the manufacturing process. The parameter N represents the noise, which is present when extracting responses. Hence, it is a dynamic component, which follows the normal distribution with mean 0 and standard deviation σ_{N}, i.e.,

$$\mathrm{N} \sim \mathcal{N}(\mu_{\mathrm{N}} = 0, \sigma_{\mathrm{N}}). \tag{3.2}$$

Let $r_i^{(t)}$ denote the response bit extracted from SRAM cell i at time instance t. It is

$$r_i^{(t)} = \begin{cases} 0, & m_i + n_i^{(t)} > T \\ 1, & m_i + n_i^{(t)} \leq T, \end{cases} \tag{3.3}$$

where $m_i \leftarrow \mathrm{M}$ i.i.d., $n_i^{(t)} \leftarrow \mathrm{N}$ i.i.d. for all i and t, and T is a threshold parameter, which depends on the specific SRAM technology that is used.

First, the probability, that the bit extracted from cell i at time instance t is "1", is calculated. This probability is called the *one-probability* of cell i and is denoted as p_{r_i}. It is

$$\begin{aligned} p_{\mathrm{r}_i} &:= P\left(r_i^{(t)} = 1\right) \\ &\underset{(3.3)}{=} P\left(m_i + n_i^{(t)} \leq T\right) \\ &= P\left(n_i^{(t)} \leq T - m_i\right) \\ &\underset{(*)}{=} P\left(\frac{n_i^{(t)} - 0}{\sigma_{\mathrm{N}}} \leq \frac{T - m_i}{\sigma_{\mathrm{N}}}\right) \\ &= \Phi\left(\frac{T - m_i}{\sigma_{\mathrm{N}}}\right), \end{aligned} \tag{3.4}$$

where Φ denotes the cumulative distribution function (cdf) of the standard normal distribution. Equality $(*)$ is correct due to normalization of the $\mathcal{N}(\mu_{\mathrm{N}} = 0, \sigma_{\mathrm{N}})$–distributed random variable N. The one-probability can be described by a random variable $\mathrm{P_r}$. Next, (3.4) is used in order to calculate the cumulative distribution function of $\mathrm{P_r}$ as

$$\begin{aligned} \mathrm{cdf}_{\mathrm{P_r}}(x) &= \mathrm{P}(\mathrm{P_r} \leq x) \\ &= \mathrm{P}\left(\Phi\left(\frac{T - m_i}{\sigma_{\mathrm{N}}}\right) \leq x\right) \\ &= \mathrm{P}\left(\frac{T - m_i}{\sigma_{\mathrm{N}}} \leq \Phi^{-1}(x)\right) \\ &= \mathrm{P}\left(\frac{T - m_i}{\sigma_{\mathrm{N}}} \cdot \sigma_{\mathrm{N}} \leq \sigma_{\mathrm{N}} \cdot \Phi^{-1}(x)\right) \\ &= \mathrm{P}\left(T - (T - m_i) \geq T - \sigma_{\mathrm{N}} \cdot \Phi^{-1}(x)\right) \\ &= \mathrm{P}\left(m_i \geq T - \sigma_{\mathrm{N}} \cdot \Phi^{-1}(x)\right) \\ &= \mathrm{P}\left(\frac{m_i - \mu_{\mathrm{M}}}{\sigma_{\mathrm{M}}} \geq -\frac{\sigma_{\mathrm{N}}}{\sigma_{\mathrm{M}}} \cdot \Phi^{-1}(x) + \frac{T - \mu_{\mathrm{M}}}{\sigma_{\mathrm{M}}}\right) \\ &= 1 - \mathrm{P}\left(\frac{m_i - \mu_{\mathrm{M}}}{\sigma_{\mathrm{M}}} \leq -\frac{\sigma_{\mathrm{N}}}{\sigma_{\mathrm{M}}} \cdot \Phi^{-1}(x) + \frac{T - \mu_{\mathrm{M}}}{\sigma_{\mathrm{M}}}\right) \\ &= 1 - \Phi\left(-\frac{\sigma_{\mathrm{N}}}{\sigma_{\mathrm{M}}} \cdot \Phi^{-1}(x) + \frac{T - \mu_{\mathrm{M}}}{\sigma_{\mathrm{M}}}\right) \\ &= \Phi\left(\underbrace{\frac{\sigma_{\mathrm{N}}}{\sigma_{\mathrm{M}}}}_{=:\lambda_1} \cdot \Phi^{-1}(x) - \underbrace{\frac{T - \mu_{\mathrm{M}}}{\sigma_{\mathrm{M}}}}_{=:\lambda_2}\right). \end{aligned} \tag{3.5}$$

By calculating the first derivative of $\mathrm{cdf}_{\mathrm{P_r}}$, the probability density function (pdf) of the one-

probability can be obtained by applying the chain rule:

$$\begin{aligned}\mathrm{pdf}_{\mathrm{P_r}}(x) &= \left[\Phi\left(\lambda_1 \cdot \Phi^{-1}(x) - \lambda_2\right)\right]' \\ &= \Phi'\left(\lambda_1 \cdot \Phi^{-1}(x) - \lambda_2\right) \cdot \left(\lambda_1 \cdot \frac{1}{\varphi(\Phi^{-1}(x))}\right) \\ &= \varphi\left(\lambda_1 \cdot \Phi^{-1}(x) - \lambda_2\right) \cdot \left(\lambda_1 \cdot \frac{1}{\varphi(\Phi^{-1}(x))}\right) \\ &= \frac{\lambda_1 \cdot \varphi(\lambda_1 \cdot \Phi^{-1}(x) - \lambda_2)}{\varphi(\Phi^{-1}(x))}, \end{aligned} \tag{3.6}$$

where φ denotes the pdf of the standard normal distribution.

Next, the error probability p_{e_i} of an SRAM cell i can be calculated. Let r_i denote the reference bit extracted from cell i during initialization. It is

$$r_i = \underset{r \in \{0,1\}}{\operatorname{argmax}} \left(\mathrm{P}(r_i^{(t)} = r)\right), \text{where } r = \begin{cases} 0, & \text{if } p_{\mathrm{r}_i} < \frac{1}{2} \\ 1, & \text{if } p_{\mathrm{r}_i} \geq \frac{1}{2}. \end{cases} \tag{3.7}$$

Further, let p_{e_i} denote the error probability of cell i, i.e.,

$$p_{\mathrm{e}_i} := \mathrm{P}(r_i^{(t)} \neq r_i) = \min\{p_{\mathrm{r}_i}, 1 - p_{\mathrm{r}_i}\}. \tag{3.8}$$

We can interpret p_{e_i} as sampled value of a random variable $\mathrm{P_e}$. Using Equations 3.7 and 3.8, the cumulative distribution function $\mathrm{cdf}_{\mathrm{P_e}}(x)$ can be obtained as

$$\begin{aligned}\mathrm{cdf}_{\mathrm{P_e}}(x) &= \mathrm{P}(\mathrm{P_e} \leq x) \\ &= \mathrm{P}(\min\{p_{\mathrm{r}_i}, 1 - p_{\mathrm{r}_i}\} \leq x). \end{aligned} \tag{3.9}$$

We distinguish two cases. In the first case, the minimum is p_{r_i}, i.e., $p_{\mathrm{r}_i} < 1 - p_{\mathrm{r}_i}$, which results in

$$\mathrm{P}(\mathrm{P_e} \leq x) = \mathrm{P}(\mathrm{P_r} \leq x) = \mathrm{cdf}_{\mathrm{P_r}}(x) \tag{3.10}$$

by definition of the cumulative distribution function. In the second case, we assume that $1 - p_{\mathrm{r}_i} < p_{\mathrm{r}_i}$ and thus obtain

$$\begin{aligned}\mathrm{P}(\mathrm{P_e} \leq x) = \mathrm{P}(1 - \mathrm{P_r} \leq x) &= 1 - \mathrm{P}(1 - \mathrm{P_r} > x) \\ &= 1 - \mathrm{P}(-\mathrm{P_r} > x - 1) \\ &= 1 - \mathrm{P}(\mathrm{P_r} \leq 1 - x) \\ &= 1 - \mathrm{cdf}_{\mathrm{P_r}}(1 - x). \end{aligned} \tag{3.11}$$

Combining these two cases, we get

$$\mathrm{cdf}_{\mathrm{P_e}}(x) = \begin{cases} \mathrm{cdf}_{\mathrm{P_r}}(x) + 1 - \mathrm{cdf}_{\mathrm{P_r}}(1 - x), & \text{if } x < \frac{1}{2} \\ 1, & \text{if } x \geq \frac{1}{2}. \end{cases} \tag{3.12}$$

Using $\mathrm{cdf}_{\mathrm{P_e}}(x)$, the probability density function

$$\mathrm{pdf}_{\mathrm{P_e}}(x) = \begin{cases} \mathrm{pdf}_{\mathrm{P_r}}(x) + \mathrm{pdf}_{\mathrm{P_r}}(1 - x), & \text{if } x < \frac{1}{2} \\ 0, & \text{if } x \geq \frac{1}{2} \end{cases} \tag{3.13}$$

can be derived.

[MTV09a] evaluated SRAM several times and compared the extracted bits to the theoretical model. The comparison yields that the model closely captures the real SRAM behavior, cf. [MTV09a, Figure 2]. This theoretical SRAM model from the literature provides the methodology for deriving a ROPUF model in Section 3.2 and will also be used when designing error correction for SRAM PUFs in Chapter 5.2.

3.2 Derivation of a Channel Model for Ring Oscillator PUFs

This section aims for transferring the approach by [MTV09a], that was revisited in Section 3.1, from SRAM PUFs to ROPUFs, in order to derive a channel model. For verification of the statements in this section, we use a set of real-world ROPUF data from [MCMS10], that are widely used as benchmark in the PUF community. The channel model, which we derive in this section, will later be used for the design of soft-decision secure sketches for ROPUFs (cf. Chapter 4).

3.2.1 Modelling a Ring Oscillator PUF

As explained in Chapter 2.1.1, Example 2.4, the intrinsic randomness of ROPUFs is caused by the delay behavior of inverters and wires which constitute the ring oscillators. According to [MCMS10], the delay d_α of a ring oscillator α can be modeled as sum of the three components d_{AVG}, d_{PV_α}, and $d_{\mathrm{NOISE}_\alpha}$, i.e.,

$$d_\alpha = d_{\mathrm{AVG}} + d_{\mathrm{PV}_\alpha} + d_{\mathrm{NOISE}_\alpha}. \tag{3.14}$$

The term d_{AVG} characterizes the average delay over the whole population of ring oscillators that are placed on one device. Hence, this term is constant for all ring oscillators on a device. The component d_{PV_α} captures the process variation. It is a static component, since for each ring oscillator it is measured only once during the manufacturing process. The term $d_{\mathrm{NOISE}_\alpha}$ captures the noise present at the time of performing a measurement. Since $d_{\mathrm{NOISE}_\alpha}$ changes over time, it must be considered as a dynamic component. In [MMS11], chip aging is added to Equation 3.14 as a further term d_{AGING}. Since it is reasonable to assume d_{AGING} to be equal for all ring oscillators on a device and generating responses is based on comparisons of frequencies, this term is ignored oftentimes in the literature and also in our studies.

3.2.2 Calculation of the One-Probability

A frequency of a particular ring oscillator can be described by the random variable

$$\mathrm{F} = \mathrm{F}_{\mathrm{PV}} + \mathrm{F}_{\mathrm{NOISE}}. \tag{3.15}$$

In this description,

$$\mathrm{F}_{\mathrm{PV}} \sim \mathcal{N}(f_{\mathrm{AVG}}, \sigma_{\mathrm{PV}}) \tag{3.16}$$

describes the process variation that is sampled once for each ring oscillator during the manufacturing process. The noise, represented by

$$\mathrm{F}_{\mathrm{NOISE}} \sim \mathcal{N}(0, \sigma_{\mathrm{NOISE}}), \tag{3.17}$$

is captured for every measurement. We consider PUF responses of length n, denoted as $\boldsymbol{r} = (r_1, \ldots, r_n)$. Response bit r_i at time t can be either "0" or "1", depending on the frequencies of the ring oscillators included in the comparison, i.e.,

$$r_i^{(t)} = \begin{cases} 0, & f_i + f_{\mathrm{NOISE}_i}^{(t)} \leq f_{i+1} + f_{\mathrm{NOISE}_{i+1}}^{(t)} \\ 1, & f_i + f_{\mathrm{NOISE}_i}^{(t)} > f_{i+1} + f_{\mathrm{NOISE}_{i+1}}^{(t)}, \end{cases} \tag{3.18}$$

where $f_i, f_{i+1} \leftarrow \mathrm{F_{PV}}$ and $f_{\mathrm{NOISE}_i}^{(t)}, f_{\mathrm{NOISE}_{i+1}}^{(t)} \leftarrow \mathrm{F_{NOISE}}$ are sampled i.i.d. for all i and t.

For response bit i the *one-probability* p_{r_i} is defined as

$$\begin{aligned} p_{\mathrm{r}_i} &:= \mathrm{P}\left(r_i^{(t)} = 1\right) \\ &\underset{(3.18)}{=} \mathrm{P}\left(f_i + f_{\mathrm{NOISE}_i}^{(t)} > f_{i+1} + f_{\mathrm{NOISE}_{i+1}}^{(t)}\right) \\ &= \mathrm{P}\left(\underbrace{f_i - f_{i+1}}_{=:\mathrm{F'_{PV}} \sim \mathcal{N}(0, \sqrt{2}\sigma_{\mathrm{PV}})} + \underbrace{f_{\mathrm{NOISE}_i}^{(t)} - f_{\mathrm{NOISE}_{i+1}}^{(t)}}_{=:\mathrm{F'_{NOISE}} \sim \mathcal{N}(0, \sqrt{2}\sigma_{\mathrm{NOISE}})} > 0\right) \\ &= \mathrm{P}\left(\mathrm{F'_{NOISE}} > f_{i+1} - f_i\right) \\ &= 1 - \mathrm{P}\left(\mathrm{F'_{NOISE}} \leq f_{i+1} - f_i\right) \\ &\underset{(*)}{=} 1 - \mathrm{P}\left(\frac{\mathrm{F'_{NOISE}}}{\sqrt{2}\sigma_{\mathrm{NOISE}}} \leq \frac{f_{i+1} - f_i}{\sqrt{2}\sigma_{\mathrm{NOISE}}}\right) \\ &= 1 - \Phi\left(\frac{f_{i+1} - f_i}{\sqrt{2}\sigma_{\mathrm{NOISE}}}\right) \\ &= \Phi\left(\frac{f_i - f_{i+1}}{\sqrt{2}\sigma_{\mathrm{NOISE}}}\right), \end{aligned} \tag{3.19}$$

where Φ denotes the cumulative distribution function of the standard normal distribution. The equality denoted with $(*)$ follows from the normalization of the $\mathcal{N}(0, \sqrt{2}\sigma_{\mathrm{NOISE}})$ distributed random variable $\mathrm{F'_{NOISE}}$. We can interpret the one-probabilities p_{r_i} as i.i.d. sampled values of a random variable $\mathrm{P_r}$.

Next, we derive the cumulative distribution function of $\mathrm{P_r}$ as

$$\begin{aligned} \mathrm{cdf}_{\mathrm{P_r}}(x) &= \mathrm{P}\left(\mathrm{P_r} \leq x\right) \\ &= \mathrm{P}\left(\Phi\left(\frac{\mathrm{F'_{PV}}}{\sqrt{2}\sigma_{\mathrm{NOISE}}}\right) \leq x\right) \\ &= \mathrm{P}\left(\frac{\mathrm{F'_{PV}}}{\sqrt{2}\sigma_{\mathrm{NOISE}}} \leq \Phi^{-1}(x)\right) \\ &= \mathrm{P}\left(\frac{\mathrm{F'_{PV}}}{\sqrt{2}\sigma_{\mathrm{PV}}} \leq \frac{\sqrt{2}\sigma_{\mathrm{NOISE}}}{\sqrt{2}\sigma_{\mathrm{PV}}} \cdot \Phi^{-1}(x)\right) \\ &= \Phi\left(\underbrace{\frac{\sigma_{\mathrm{NOISE}}}{\sigma_{\mathrm{PV}}}}_{=:\lambda} \cdot \Phi^{-1}(x)\right). \end{aligned} \tag{3.20}$$

The probability density function of $\mathrm{P_r}$ can be deduced by applying the chain rule to calculate the first derivative of $\mathrm{cdf}_{\mathrm{P_r}}$, resulting in

$$\begin{aligned}\mathrm{pdf}_{\mathrm{P_r}}(x) &= \left[\Phi\left(\lambda \cdot \Phi^{-1}(x)\right)\right]' \\ &= \Phi'\left(\lambda \cdot \Phi^{-1}(x)\right) \cdot \left(\lambda \cdot \frac{1}{\varphi(\Phi^{-1}(x))}\right) \\ &= \varphi\left(\lambda \cdot \Phi^{-1}(x)\right) \cdot \left(\lambda \cdot \frac{1}{\varphi(\Phi^{-1}(x))}\right) \\ &= \frac{\lambda \cdot \varphi(\lambda \cdot \Phi^{-1}(x))}{\varphi(\Phi^{-1}(x))},\end{aligned} \tag{3.21}$$

where φ denotes the probability density function of the standard normal distribution.

3.2.3 Calculation of the Error-Probability

To calculate the error probability, we proceed analog to the derivation of the SRAM channel model in [MTV09a] (cf. Section 3.1). Let the reference response bit at position i be

$$r_i = \underset{r \in \{0,1\}}{\mathrm{argmax}} \left(\mathrm{P}\left(r_i^{(t)} = r\right)\right), \text{where } r = \begin{cases} 0, \text{if } p_{\mathrm{r}_i} < \frac{1}{2} \\ 1, \text{if } p_{\mathrm{r}_i} \geq \frac{1}{2}. \end{cases} \tag{3.22}$$

The error probability p_{e_i} of response bit i is again defined as

$$p_{\mathrm{e}_i} = \mathrm{P}\left(r_i^{(t)} \neq r_i\right) = \min\{p_{\mathrm{r}_i}, 1 - p_{\mathrm{r}_i}\}, \tag{3.23}$$

and can be considered as i.i.d. sampled value of a random variable $\mathrm{P_e}$, for which we derive the cumulative distribution function

$$\begin{aligned}\mathrm{cdf}_{\mathrm{P_e}}(x) &= \mathrm{P}\left(\mathrm{P_e} \leq x\right) \\ &= \begin{cases} \mathrm{cdf}_{\mathrm{P_r}}(x) + 1 - \mathrm{cdf}_{\mathrm{P_r}}(1-x), & \text{if } x < \frac{1}{2} \\ 1, & \text{if } x \geq \frac{1}{2} \end{cases}\end{aligned} \tag{3.24}$$

in the same fashion as done in Equations 3.9–3.12, as well as the probability density function

$$\mathrm{pdf}_{\mathrm{P_e}}(x) = \begin{cases} \mathrm{pdf}_{\mathrm{P_r}}(x) + \mathrm{pdf}_{\mathrm{P_r}}(1-x), & \text{if } x < \frac{1}{2} \\ 0, & \text{if } x \geq \frac{1}{2}, \end{cases} \tag{3.25}$$

by using the first derivative of the cdf given in (3.24).

3.2.4 Results

We visualize the probability density functions of the one-probability and the error probability derived in Sections 3.2.2 and 3.2.3, and compare them to the real-world data from [MCMS10]. We want to emphasize, that from a stochastic point of view, the results we can obtain from the data set are very limited. This can be seen by the low density of the plots visualized in Figures 3.3a and 3.3c. The reason is the limited size of the data set, which consists of 193 devices with 100 readouts each (see also Chapter 2, Remark 2.6). We decided to use these data for three reasons:

1. A larger public data set with ROPUF data does not exist.

2. In the PUF scenario, conclusions are often drawn from a very limited number of devices, often less than 20 (cf. Appendix A, Table A.2 for a summary). Considering this fact, a data set containing 193 devices is already huge.

3. This data set is well-known within the PUF community and widely used as benchmark to compare results.

The probability density function of the one-probability is visualized in Figure 3.3b and captures the shape given by the real-world data in Figure 3.3a. We use

$$\lambda = \frac{\sigma_{\text{NOISE}}}{\sigma_{\text{PV}}} = \frac{0.028}{0.74} = 0.0378 \tag{3.26}$$

as parameter of the probability density function, according to estimates based on the data set (cf. Appendix A). This parameter was also used for generating the probability density function of the error probability visualized in Figure 3.3d, capturing the shape of the error probability resulting from the real-world data in Figure 3.3c.

The results obtained in this section will further be used in Chapter 4.3, in order to design soft-decision secure sketches for ROPUFs.

3.3 Derivation of Channel Models for DRAM PUFs

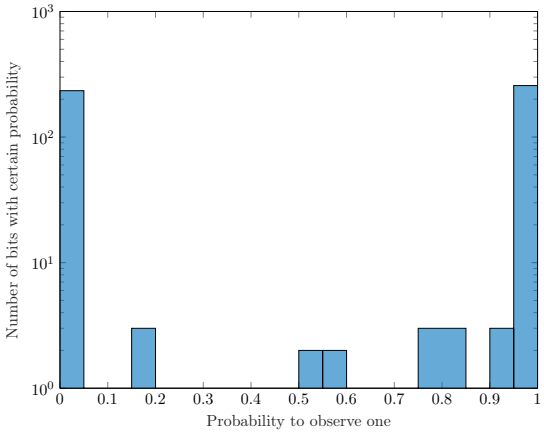
Usually, memory-based PUFs are using SRAM or similar technologies, like latches or flipflops (cf. Chapter 2, Example 2.7 and Remark 2.9). Since 2012, *Dynamic Random Access Memory (DRAM)* is used as an alternative memory technology to construct PUFs [FRC+12]. This section deals with the comparatively new direction of PUFs based on DRAM.

Nowadays, DRAM is included in a large number of mobile and embedded systems, cf. for example [SRK+17, Figure 1], and thus can be used in order to construct fully intrinsic PUFs which do not require any modifications on the underlying hardware. The main advantage, in comparison to SRAM, is the high density of memory cells that allows the reduction of chip-area that is needed to extract a certain amount of bits. Further, DRAM PUFs benefit from low cost and from the fact that no device startup is needed for extracting response bits.

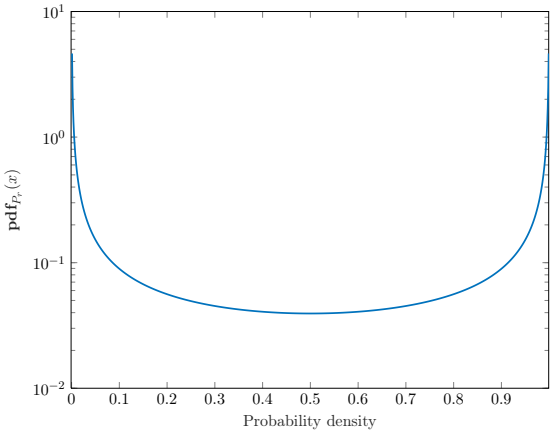
DRAM belongs to the family of volatile memories. A DRAM cell consists of a transistor and a capacitor and stores a bit according to the charge of the capacitor. DRAM cells are arranged in an array. The rows are connected by wordlines, which regulate the access to memory cells. The columns are connected by bitlines, which are used to charge and uncharge the cells. The charge in a cell gradually descends, a physical property that can result in an altered bit. The time until the bit stored in a memory cell changes its value is called the *retention time* of the cell. To prevent cells from changing their state, they are periodically recharged in intervals of 32 ms or 64 ms, depending on the specification of the used DRAM.

Up to now, literature provides four major approaches to derive PUF responses from DRAM:

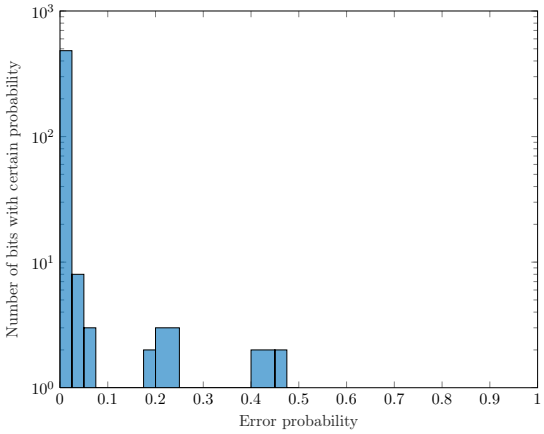
1. DRAM PUFs based on retention behavior: When pausing the refresh operation for a certain time interval, some of the DRAM cells leak their charge in that time period. A cell with such a behavior is called a *weak cell*, since the bit it is representing alters as soon as the cell is discharged. On the contrary, a cell that holds its charge, and hence,



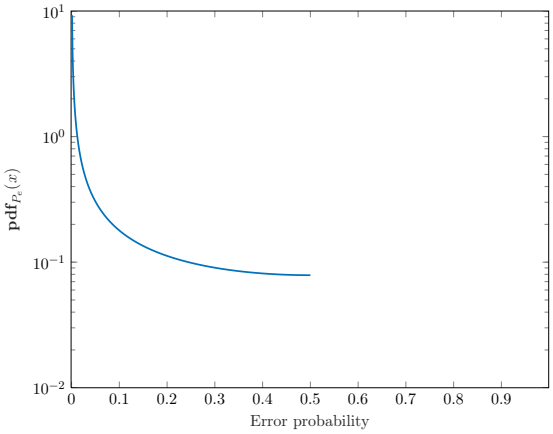
(a) One-probability according to the extended ROPUF data set from [MCMS10].



(b) One-probability according to the model derived in Section 3.2.2 ($\lambda = 0.0378$).



(c) Error-probability according to the extended ROPUF data set from [MCMS10].



(d) Error-probability according to the model derived in Section 3.2.3 ($\lambda = 0.0378$).

Figure 3.3: Comparison of the theoretical ROPUF channel model derived in Section 3.2 and the extended data set from [MCMS10].

does not alter its value, is called a *strong cell*. PUFs, based on retention behavior, are the most rigorous studied class of DRAM PUFs in the literature, cf. [FRC$^+$12, RFC$^+$13, LZLL14, RHHF16, SRR16, XSA$^+$16, SRR17, TZC$^+$17, SRK$^+$17, SXA$^+$18].

2. DRAM PUFs based on latency behavior: When performing read and write operations on DRAM cells, latency times exist to guarantee that there is enough time for each cell to process the respective operation. Due to random variations in the manufacturing process, the cells perform operations with varying paces. The latency times are selected such that all cells are able to process the operations. By reducing the timing parameters, some cells are prevented from successfully processing the operations. This behavior can be utilized in order to extract PUF responses. The DRAM latency behavior for constructing PUFs was studied in the literature, cf. [HSW$^+$15, TRT$^+$18, KPHM18].

3. DRAM PUFs based on startup values: Similar to SRAM PUFs, responses can be generated by exploiting the initialization values of DRAM cells. PUFs based on startup values are studied in the literature, cf. [TKXC15, TKYC17a, ETC17, TKYC17b].

4. DRAM PUFs based on row-hammering: Repeatedly accessing a row of the DRAM in short intervals is called row-hammering. As a side-effect of row hammering, the retention time of cells in rows adjacent to a hammered row might change [KDK$^+$14]. Originally, this effect was used in order to execute some kinds of fault analysis attacks, cf. for example [SD15, VDVFL$^+$16, XZZT16, RGB$^+$16, BM16]. According to literature, the first non-malicious application of the row-hammer effect is the construction of PUFs, cf. [SXA$^+$17, ZGS18].

An extensive literature research, summarizing results of the above mentioned publications concerning DRAM PUFs, was conducted within a bachelor's thesis [Bit18].

Before considering channel models for DRAM PUFs in Sections 3.3.1–3.3.3, we outline the scenario considered in this section according to Figure 3.4. A bit vector $\boldsymbol{x}$ is generated by using the measured DRAM data. TMV is used to produce a stabilized version of $\boldsymbol{x}$, denoted as $\widetilde{\boldsymbol{x}}$. Debiasing, which is studied in this section, deals with removing biases towards "0" or "1" from the data. The output of the debiasing function is defined to be the extracted response $\boldsymbol{r}$, which is further processed within a helper data algorithm (HDA). The latter step will be studied in Chapter 4. The dashed paths are not included in every implementation, since not all debiasing (helper data) algorithms need to use additional debiasing (helper) data.

The DRAM data used for our studies have been provided by the "Microelectronic Systems Design Research Group" located at the department of electrical and computer engineering at the technical university of Kaiserslautern [Sud18]. The data originate from an FPGA-based measurement platform that allows to conduct measurements on DRAM at different operating temperatures in a range from 25 °C up to 90 °C with an accuracy of ±2 °C. The measurement platform includes eight DDR3 *dual in-line memory modules (DIMMs)*, each of which consists of 16 devices, and each device in turn consists of 2^{32} memory cells. We partition each device into four sets of memory cells and define each of the resulting sets to be a PUF. Hence, the set of PUFs we use for our studies contains 128 devices, having 2^{30} memory cells each. Retention measurements were performed by disabling the refresh operation for 10 seconds. After that time, the memory cells are read. A bit vector $\boldsymbol{x}$ represents strong cells that are still charged after 10 seconds by a "0". Weak cells, that are discharged after that time, are represented

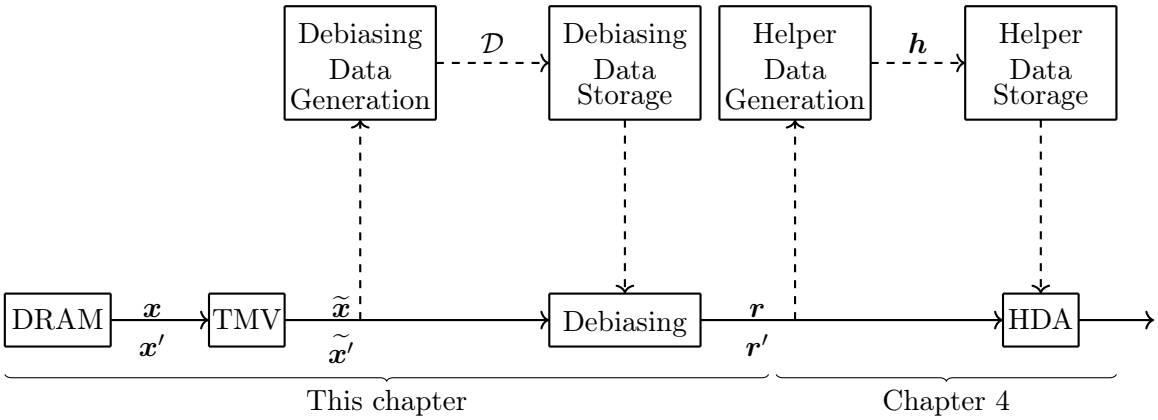


Figure 3.4: A bit vector $\boldsymbol{x}$ is generated from the DRAM data and stabilized by applying TMV. Debiasing removes biases towards "0" in the extracted bit vector and hence, forms the PUF response. The HDA unit provides solutions for the problem of noisy and non-uniformly distributed PUF responses.

by a "1". The set of retention measurements that was used for the studies conducted in this section comprises 50 retention measurements for each PUF at an operating temperature of 40 °C and 20 retention measurements for each PUF at an operating temperature of 37 °C. For a detailed description of the experimental setup we refer to [JMR+17].

In order to increase the stability of a bit vector $\boldsymbol{x}$, a noise reduction technique called *Temporal Majority Voting (TMV)* can be applied as an additional pre-processing step before the debiasing routine is executed. Using TMV, the bit vector $\widetilde{\boldsymbol{x}}$ is generated by performing m measurements instead of one and defining a memory cell to be weak, if it is classified as weak for at least θ of the m measurements. Otherwise, a cell is defined to be strong. Let $\widetilde{x}_i^{(\ell)}$ denote the classification of cell i into weak or strong for readout ℓ ($\ell = 1, \ldots, m$), i.e.,

$$\widetilde{x}_i^{(\ell)} = \begin{cases} 1, & \text{if cell } i \text{ is classified to be weak} \\ 0, & \text{if cell } i \text{ is classified to be strong} \end{cases}, \text{ for } \ell = 1, \ldots, m \text{ and } i = 1, \ldots, 2^{30}. \quad (3.27)$$

Hence, the components of the final bit vector $\widetilde{\boldsymbol{x}}$ after applying TMV are

$$\widetilde{x}_i = \begin{cases} 1 \text{ (weak)}, & \sum_{\ell=1}^{m} \widetilde{x}_i^{(\ell)} \geq \theta \\ 0 \text{ (strong)}, & \sum_{\ell=1}^{m} \widetilde{x}_i^{(\ell)} < \theta. \end{cases}, \text{ for } i = 1, \ldots, 2^{30}. \quad (3.28)$$

For the implementations of the debiasing algorithms discussed in this chapter, as long as not stated otherwise, the parameters $m = \theta = 1$ were chosen in order to reduce the time for producing responses[1].

In general, two types of errors can be distinguished. First, a cell that was classified as strong during initialization, might be classified as weak during reproduction. This event corresponds to a bit that alters from "0" to "1", what occurs with probability

$$\mathrm{P}(0 \longmapsto 1) \approx \frac{\#0 \longmapsto 1}{2^{30} - \mathrm{wt}(\widetilde{\boldsymbol{x}})}, \quad (3.29)$$

[1] In this case, we have $\widetilde{\boldsymbol{x}} = \boldsymbol{x}$.

Table 3.1: Hamming weight of the resulting bit vector $\widetilde{\boldsymbol{x}}$ and number of altered bits observed for different TMV setups with an operating temperature of 40 °C. TMV performs m measurements and classifies a DRAM cell to be weak if it behaves weak at least θ times [Bit18].

m	θ	$\mathrm{wt}(\widetilde{\mathbf{x}})$			$\#\,0 \longmapsto 1$			$\#\,1 \longmapsto 0$		
		min	avg	max	min	avg	max	min	avg	max
1	1	4373	7988.3	17245	26	303.39	4039	11	508.05	3124
5	1	4702	8654.6	17336	35	750.46	4410	14	229.66	1693
5	2	4591	8116.1	14022	30	379.82	1458	15	239.52	1433
5	3	4522	7899.1	13636	27	321.92	1303	19	220.05	1234
5	4	4447	7733.0	13243	22	290.46	1240	26	223.79	1199
5	5	4247	7538.6	12929	20	278.03	1090	40	246.95	1557

Table 3.2: Hamming weight of the resulting bit vector $\widetilde{\boldsymbol{x}}$ and number of altered bits observed for different TMV setups with an operating temperature of 37 °C. TMV performs m measurements and classifies a DRAM cell to be weak if it behaves weak at least θ times [Bit18].

m	θ	$\mathrm{wt}(\widetilde{\mathbf{x}})$			$\#\,0 \longmapsto 1$			$\#\,1 \longmapsto 0$		
		min	avg	max	min	avg	max	min	avg	max
1	1	1558	2829.0	6041	15	155.69	1922	6	97.77	367
5	1	1667	3024.8	6096	23	280.23	1392	6	46.65	394
5	2	1607	2859.8	5157	5	137.58	1033	5	47.00	366
5	3	1580	2804.4	5001	3	132.89	1133	3	41.97	237
5	4	1551	2756.1	4863	4	133.61	1209	2	41.76	144
5	5	1528	2699.9	4758	9	138.80	1202	2	46.10	124

where $\#0 \longmapsto 1$ in the nominator denotes the number of bits that change their value from "0" to "1" and the denominator consists of the total number of zeros in bit vector $\widetilde{\boldsymbol{x}}$. Analog, a cell that initially is classified as weak and behaves as strong during reproduction, leads to a change from "1' to "0". This event takes place with probability

$$\mathrm{P}(1 \longmapsto 0) \approx \frac{\#1 \longmapsto 0}{\mathrm{wt}(\widetilde{\boldsymbol{x}})}, \tag{3.30}$$

where the nominator includes the number of bits that alter from "1" to "0", while the denominator contains the total number of ones in bit vector $\widetilde{\boldsymbol{x}}$. Table 3.1 and Table 3.2 summarize the observed behavior for operating conditions of 40 °C and 37 °C, respectively. Comparing the two tables, we notice that the number of weak cells decreases when lowering the temperature. This behavior was already observed and studied in the literature, cf. [LJK+13].

The bit vectors $\widetilde{\boldsymbol{x}}$ generated in this fashion cannot be directly applied to helper data algorithms, since they are heavily biased towards "0". In [MvdLvdSW15, Section 2.4] it was

shown, that only a small bias can be tolerated in the inputs of a helper data algorithm. Hence, in addition to the problem of noise, we have to deal with the problem of biased data in the case of DRAM measurements. We apply techniques that implement debiasing for post-processing of the bit vectors $\widetilde{\boldsymbol{x}}$. In general, debiasing provides a solution to the following problem.

Problem 3.1. Given a binary source $Q = \{0, 1\}$ that generates "0" and "1" with constant, but unknown probabilities p_0 and p_1, respectively. The goal is to derive a uniformly distributed binary sequence from the output of Q.

The debiasing methods outlined in the subsequent sections provide solutions to Problem 3.1. The components responsible for helper data algorithms visualized in Figure 3.4 will be studied in Chapter 4, since they are not relevant for the derivation of channel models within this chapter.

3.3.1 Choose Length (CL) Debiasing

The basic idea of *Choose Length (CL) debiasing* is adopted from [SXA+18], in which also retention measurements are used to classify DRAM cells into weak and strong. A PUF response is generated by considering a random permutation of the cells' addresses and defining response bit r_i to be "1" when cell i is a weak cell and to be "0" when cell i is a strong cell. The response $\boldsymbol{r}$ that is generated in this way is further processed in an helper data algorithm. We modify this algorithm as explained in the following paragraphs.

The initialization phase is outlined in Algorithm 1. A binary response $\boldsymbol{r}$ of length n can be arbitrarily chosen (line 1). We generate two TMV-stabilized bit vectors $\widetilde{\boldsymbol{x}}_1$ and $\widetilde{\boldsymbol{x}}_2$. Each of these two bit vectors is generated with distinct operating conditions, as for example, different temperatures (lines 2–3). Using the DRAM measurements provided by [Sud18], $\widetilde{\boldsymbol{x}}_1$ is extracted at an operating temperature of 40 °C, while $\widetilde{\boldsymbol{x}}_2$ is produced at 37 °C. Memory cells, that retain the charged state in both bit vectors, are defined to be strong (line 4). Cells, that change their state twice, are defined to be weak (line 5). Cells, that change their state once, are ignored. For each response bit, an address of a memory cell is stored as debiasing data (lines 6–10). For each response bit "0", the address of a strong cell is stored, while for each response bit "1", the address of a weak cell is stored as debiasing data $\mathcal{D}$. In order to guarantee sufficient entropy, the cells that are used for storing addresses have to be chosen at random.

For reproduction according to Algorithm 2, the memory cells addressed by $\mathcal{D}$ are examined sequentially and their retention behavior is verified. If a cell decays within 10 seconds (weak cell), the corresponding response bit is set to "1", otherwise (strong cell) it is set to "0". The main difference to the algorithm in [SXA+18] is that the response can be chosen and does not depend on a random permutation of memory addresses. As it will turn out in Chapter 4, this property might be advantageous, when constructing secure sketches.

The observed error characteristics are summarized in Table 3.3. We notice, that all errors modify bits from "1" to "0", but not the other way around. Hence, the observed error behavior matches the model of the so-called *Z-Channel*, that is known in the literature as an asymmetric channel [Klø81]. Figure 3.5 provides a visualization of the Z-channel. This channel model has the property, that if a "0" is transmitted, the bit will correctly be received. However, if a "1" is transmitted, a bit error probability p_{b} exists, which turns a "1" into a "0". Figure 3.6 visualizes inter- and intra-response distances of CL debiasing, as defined in Chapter 2.1.2.

Algorithm 1: Choose Length (CL) debiasing: Initialization

Input: Refresh pause time t, Condition1, Condition2
Output: Response $\boldsymbol{r} = (r_1, \ldots, r_n)$, debiasing data $\mathcal{D}$ (lists of memory addresses)

$\boldsymbol{r} \in \{0,1\}^n \leftarrow$ Generate response (can be chosen arbitrarily)
$\widetilde{\boldsymbol{x}}_1 \leftarrow$ RetentionMeasurement(t, Condition1)
$\widetilde{\boldsymbol{x}}_2 \leftarrow$ RetentionMeasurement(t, Condition2)
Strong cells := {cells that behave strong in $\widetilde{\boldsymbol{x}}_1$ and $\widetilde{\boldsymbol{x}}_2$}
Weak cells := {cells that behave weak in $\widetilde{\boldsymbol{x}}_1$ and $\widetilde{\boldsymbol{x}}_2$}
for $i = 1, \ldots, n$ **do**
    if $r_i = 0$ **then**
        Randomly select and store an address of a strong memory cell into list $\mathcal{D}$
    else
        Randomly select and store an address of a weak memory cell into list $\mathcal{D}$
return $\boldsymbol{r} \in \{0,1\}^n$, $\mathcal{D}$

Algorithm 2: Choose Length (CL) debiasing: Reproduction

Input: List of memory addresses $\mathcal{D}$, refresh pause time t
Output: Response $\boldsymbol{r}' = (r'_1, \ldots, r'_n)$

$\widetilde{\boldsymbol{x}}' \leftarrow$ RetentionMeasurement(t)
$i = 1$
for *each* $d \in \mathcal{D}$ **do**
    if *cell* d *decayed in* $\widetilde{\boldsymbol{x}}'$ *within* t *seconds* **then**
        $r'_i = 1$
    else
        $r'_i = 0$
    $i = i + 1$
return *Response* $\boldsymbol{r}' \in \{0,1\}^n$

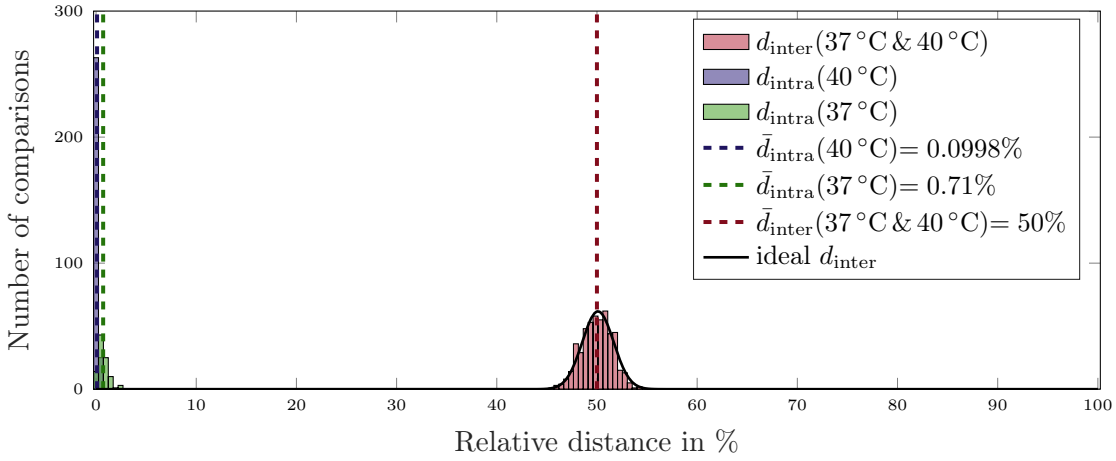


Figure 3.6: Inter- and intra-response distances when applying CL debiasing to the DRAM retention data with a randomly chosen initial response of length 1023. These results were produced by omitting TMV ($m = 1, \theta = 1$). The figure is used with kind permission of Sebastian Bitzer [Bit18].

Table 3.3: Bit error probabilities observed for CL debiasing at different operating conditions.

	$P(0 \longmapsto 1)$			$P(1 \longmapsto 0)$		
	min	avg	max	min	avg	max
40 °C	0.00%	0.00%	0.00%	0.00%	0.37%	5.04%
37 °C	0.00%	0.00%	0.00%	0.00%	3.03%	17.5%

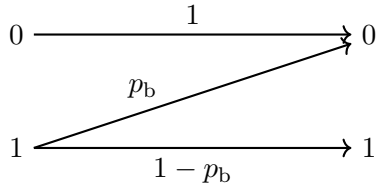


Figure 3.5: Transmission probabilities in a Z-channel according to [Kløve81].

3.3.2 Von Neumann (VN) Debiasing

Von Neumann (VN) debiasing was introduced in [VN51]. In [MvdLvdSW15], it was applied to PUFs for the first time. We use a toy example to explain the basic idea of VN debiasing. Given is a binary asymmetric source $\mathcal{Q}$, which has to be used in order to randomly generate equally distributed binary sequences. For example, assume that $\mathcal{Q}$ outputs "0" with probability $\frac{1}{4}$ and "1" with probability $\frac{3}{4}$. In order to generate a random, equally distributed binary sequence, two symbols are independently extracted from the source. For the given example, Table 3.4 shows the probabilities of the four possible events. In a random binary sequence, "0" and "1" have to occur with the same probability. Hence, only the pairs (0,1) and (1,0) are considered, since they occur with the same probability. If the source generates the pair (0,1), we extract a "0" as random bit. If the source produces (1,0), we extract a "1" as random bit. We refuse the outputs (0,0) and (1,1).

Table 3.4: Basic idea of Von Neumann (VN) debiasing, explained by using a binary asymmetric source that generates a "0" with probability $\frac{1}{4}$ and a "1" with probability $\frac{3}{4}$. Considering the pairwise outcomes x_i of the source, the pairs (0,1) and (1,0) occur with the same probability p_i. Hence, only these pairs are considered for generating a random sequence, which is constructed by extracting the first component of the corresponding pairs.

x_i	(0,0)	(0,1)	(1,0)	(1,1)
p_i	$\frac{1}{16}$	$\frac{3}{16}$	$\frac{3}{16}$	$\frac{9}{16}$

Initialization of VN debiasing is outlined in Algorithm 3. A bit vector $\boldsymbol{x}$ (or $\widetilde{\boldsymbol{x}}$ when TMV is applied) is extracted from the DRAM (cf. line 1) and sequentially divided into disjoint

pairs (cf. line 2). According to the explanation of VN debiasing given above, the algorithm stores pointers to the pairs (0,1) and (1,0), while pairs with equal components, i.e. (0,0) and (1,1), are ignored (cf. line 3). The list of pointers is stored as debiasing data $\mathcal{D}$ in the debiasing data storage. Response bit r_i is generated by taking the first component of the i-th pair referenced by $\mathcal{D}$ (cf. lines 5–7).

Algorithm 4 summarizes the reproduction of PUF responses. Bit-vector $\boldsymbol{x}'$ (or $\widetilde{\boldsymbol{x}}'$ when TMV is applied) is extracted from the DRAM (cf. line 1). Analog to the initial bit vector, $\boldsymbol{x}'$ is divided into pairs (cf. line 2). According to the available debiasing data $\mathcal{D}$, the algorithm chooses the n pairs that are indexed by a pointer (cf. line 3). If pointer i refers to a pair (1,0), response bit r'_i is set to "1". Similarly, if pointer i indexes a pair (0,1), response bit r'_i is set to "0". When pointer i returns a pair (0,0) or (1,1), the erasure symbol Δ is used as symbol r'_i. The algorithm returns the response $\boldsymbol{r}' \in \{0, 1, \Delta\}^n$, which is generated in this manner.

Algorithm 3: Von Neumann (VN) debiasing: Initialization

Input: Refresh pause time t, Condition
Output: Response $\boldsymbol{r} = (r_1, \ldots, r_n)$, debiasing data $\mathcal{D}$

1 $\widetilde{\boldsymbol{x}} \leftarrow$ RetentionMeasurement(t, Condition)
2 Divide $\widetilde{\boldsymbol{x}}$ into pairs
3 Store pointers to pairs (0,1) and (1,0) in data structure $\mathcal{D}$
4 $i = 1$ /* counts the stored pairs */
5 **for** *each pair* (x_1, x_2) *for which a pointer is stored in* $\mathcal{D}$ **do**
6 $r_i = x_1$
7 $i = i + 1$
8 **return** $\boldsymbol{r} \in \{0, 1\}^n$, $\mathcal{D}$

Algorithm 4: Von Neumann (VN) debiasing: Reproduction

Input: List of pointers $\mathcal{D}$, refresh pause time t
Output: Response $\boldsymbol{r}' = (r'_1, \ldots, r'_n)$

1 $\widetilde{\boldsymbol{x}}' \leftarrow$ RetentionMeasurement(t)
2 Divide $\widetilde{\boldsymbol{x}}'$ into pairs
3 Choose the n pairs for which a pointer is stored in $\mathcal{D}$
4 **for** $i = 1, \ldots, n$ **do**
5 **if** *Pair i = (1,0)* **then**
6 $r'_i = 1$
7 **else**
8 **if** *Pair i = (0,1)* **then**
9 $r'_i = 0$
10 **else**
11 $r'_i = \Delta$
12 **return** *Response* $\boldsymbol{r}' \in \{0, 1, \Delta\}^n$

The observed error characteristics are summarized in Table 3.5. We do not observe any

bits that change from "0" to "1" or vice versa. Instead, all disturbed bits are replaced by the erasure symbol Δ. This behavior can be modeled by an erasure channel as visualized in Figure 3.7. In an erasure channel, a "0" that is transmitted turns into an erasure symbol Δ with probability p_b and is transmitted correctly otherwise. Similar, a "1" that is transmitted turns into an erasure symbol Δ with probability q_b and is transmitted correctly otherwise. Bits that change their value do not occur in that model. In our studies on retention-based DRAM data, we observe $p_\mathrm{b} \approx q_\mathrm{b}$. To result in an inverted bit, VN debiasing requires both response bits in a pair to alter, an event which is unlikely to happen. We want to emphasize, that erasures are much easier to correct than errors, since their positions are directly known from the received word. A visualization of inter- and intra-response distances for VN debiasing is provided in Figure 3.8.

Table 3.5: Bit error probabilities observed for VN debiasing at different operating conditions.

	$\mathrm{P}(0 \longmapsto 1)$			$\mathrm{P}(1 \longmapsto 0)$			$\mathrm{P}(0 \longmapsto \Delta)$			$\mathrm{P}(1 \longmapsto \Delta)$		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max
40 °C	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.07%	4.88%	0.0%	1.0%	5.78%
37 °C	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.6%	11.51%	0.0%	2.38%	10.35%

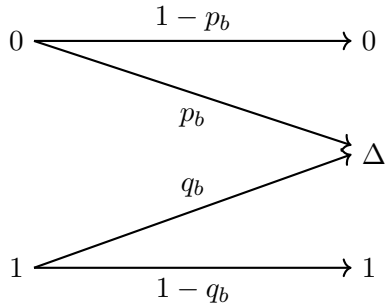


Figure 3.7: Transmission probabilities in a binary erasure channel, where Δ denotes the erasure symbol.

3.3.3 Other Debiasing Schemes

Besides CL and VN debiasing, we studied further debiasing methods. However, due to the resulting channels, which are unreliable at least under changing temperature conditions, they are all summarized in this section. In contrast to CL and VN debiasing, the debiasing algorithms discussed subsequently do not use debiasing data. Hence, they are examples for which the dashed edges to the units for debiasing data generation and debiasing data storage in Figure 3.4 are not used. In contrast to CL and VN debiasing, we applied TMV with parameters $m = 5$ and $\theta = 4$ (cf. Tables 3.1 and 3.2) to stabilize the bit vectors provided by the DRAM retention measurements.

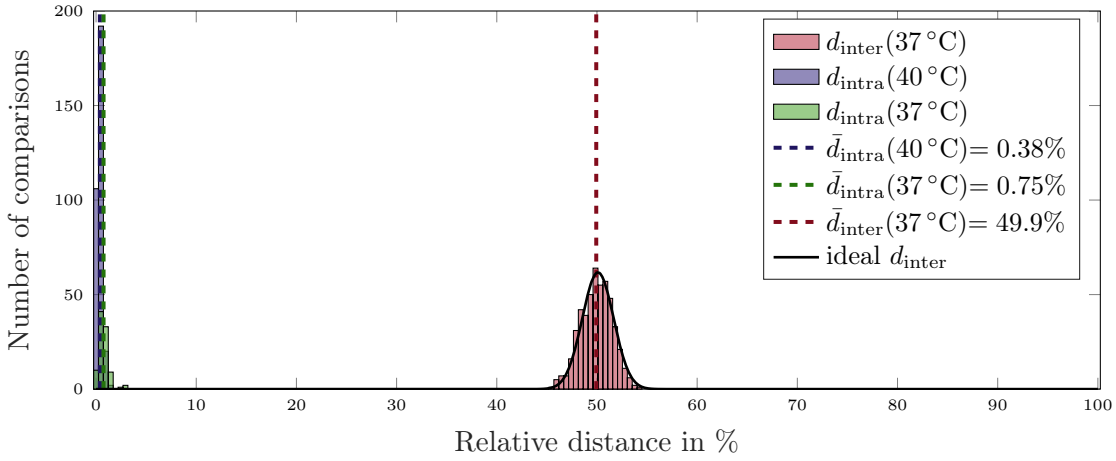


Figure 3.8: Inter and intra response distances when applying Von Neumann(VN) debiasing to the DRAM retention data. These results were produced by omitting TMV ($m = 1, \theta = 1$). The figure is used with kind permission of Sebastian Bitzer [Bit18].

Least Significant Bit (LSB) Debiasing

To the best of our knowledge, LSB debiasing has not been introduced in the literature so far. In order to generate a PUF response of length n, we use the DRAM retention measurements in order to identify weak cells. We choose the first n addresses that belong to weak cells, and connect their least significant bits (LSBs). The resulting bit sequence is defined to be the PUF response. LSB debiasing turns out to have outstanding properties concerning uniqueness. However, regeneration of responses is supremely disappointing, cf. Table 3.6 and Figure 3.9. LSB debiasing results in high bit error probabilities for both, bits that alter from "0" to "1" and bits that alter from "1" to "0".

Table 3.6: Bit error probabilities observed for LSB debiasing at different operating conditions.

	$\mathrm{P}(0 \longmapsto 1)$			$\mathrm{P}(1 \longmapsto 0)$		
	min	avg	max	min	avg	max
40 °C	36.10%	48.75%	56.22%	36.26%	48.19%	55.33%
37 °C	42.29%	50.11%	56.72%	44.29%	50.09%	55.64%

μ–to–λ Debiasing

In [AWSO17], μ–to–λ debiasing was proposed in order to deal with biased data in the context of PUFs. The main idea is to divide the biased data $\widetilde{\boldsymbol{x}}$ into sub-blocks of length μ. Each of these sub-blocks is mapped to a block of length λ, where $\lambda < \mu$. The best mapping can be found by solving an optimization problem with a branch and bound algorithm. Two special

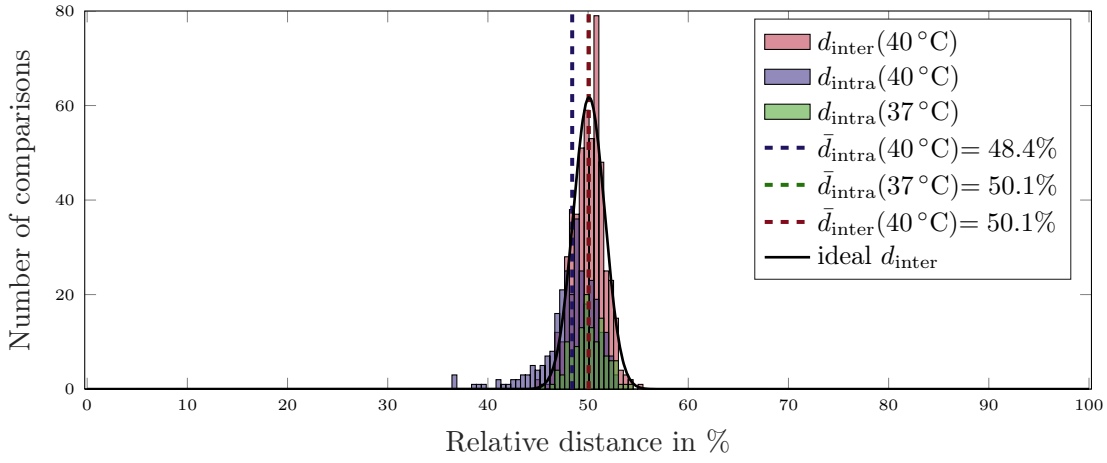


Figure 3.9: Inter and intra response distances when applying Least Significant Bit (LSB) debiasing to the DRAM retention data with a randomly chosen initial response of length 1023. These results were produced with TMV ($m = 5, \theta = 4$). The figure is used with kind permission of Sebastian Bitzer [Bit18].

cases of μ to λ debiasing are considered in this section, namely *OR debiasing* and *Exclusive OR (XOR) debiasing*.

OR debiasing implements μ–to–λ debiasing for $\mu = \frac{|\widetilde{\boldsymbol{x}}|}{\beta}$ and $\lambda = 1$. The parameter β depends on the available DRAM data and in our case is selected, such that

$$\left(1 - \frac{\mathrm{wt}(\widetilde{\boldsymbol{x}})}{2^{30}}\right)^{\beta} \approx \frac{1}{2}, \tag{3.31}$$

where $\mathrm{wt}(\widetilde{\boldsymbol{x}})$ is the Hamming weight of bit vector $\widetilde{\boldsymbol{x}}$ and 2^{30} is the total number of memory cells and hence, the length of $\widetilde{\boldsymbol{x}}$. The bit vector $\widetilde{\boldsymbol{x}}$ is divided into sub-blocks of length $\mu = \frac{|\widetilde{\boldsymbol{x}}|}{\beta}$. From each sub-block, $\lambda = 1$ response bit is derived. For this purpose, the bits are interpreted as boolean values and all bits of a block are linked by a logical OR operation.

As can be seen in Table 3.7, bits can alter their values from "0" to "1" as well as from "1" to "0". For some operating conditions, OR debiasing results in large bit error probabilities. Hence, when considering varying operating conditions, the error probabilities are too large in order to find a suitable method for error correction. Inter- and intra-response distances of OR debiasing are visualized in Figure 3.10.

Table 3.7: Bit error probabilities observed for OR debiasing at different operating conditions.

	$\mathrm{P}(0 \longmapsto 1)$			$\mathrm{P}(1 \longmapsto 0)$		
	min	avg	max	min	avg	max
40 °C	0.00%	2.23%	10.06%	0.00%	1.96%	10.17%
37 °C	0.00%	0.19%	0.78%	43.28%	56.72%	70.60%

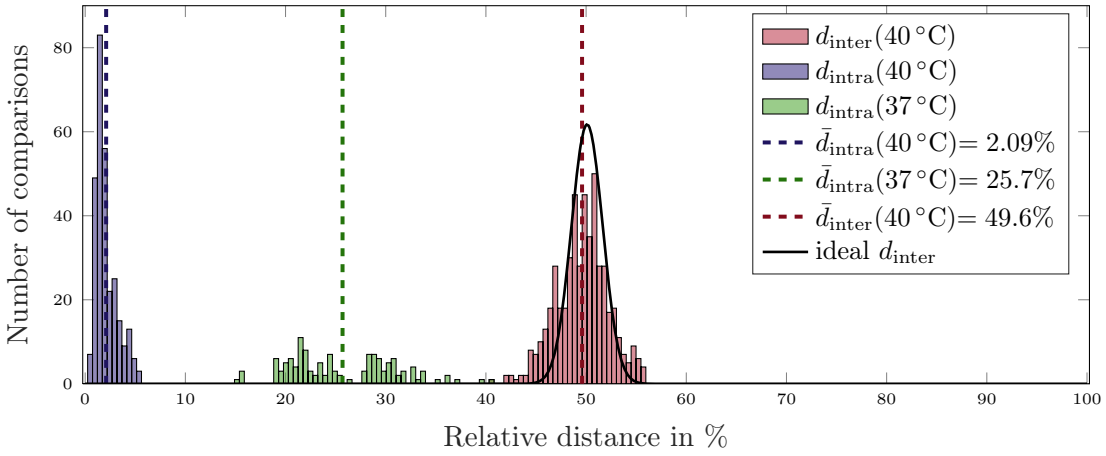


Figure 3.10: Inter and intra response distances when applying OR debiasing to the DRAM retention data with a randomly chosen initial response of length 1023. These results were produced with TMV ($m = 5, \theta = 4$). The figure is used with kind permission of Sebastian Bitzer [Bit18].

XOR debiasing in the context of PUFs was introduced in [AGM+15] and is a μ–to–λ debiasing for $\mu = 2$ and $\lambda = 1$. Hence, the bit vectors are divided into sub-blocks of length $\mu = 2$. For each sub-block, an output of length $\lambda = 1$ is produced by connecting both bits of a sub-block by using the XOR operator. We do not consider the XOR debiasing for our DRAM retention data due to the following reasons: Our data possess a strong bias towards "0", which cannot be eliminated by the described XOR operation. Also, in contrast to OR debiasing, a single bit error in a pair alters the generated response bit.

3.4 Concluding Remarks

The goal of this chapter was to derive channel models for PUFs. In coding theory, it is required to know the characteristics of the channel in order to select an error-correcting code. This prerequisite is often not fulfilled in the literature, when error correcting schemes for PUFs are proposed. Most often, the observed worst-case behavior is used to derive the bit error probability of a BSC. In [MTV09a], the authors considered a more detailed channel for the first time. Instead of a BSC that uses a fixed bit error probability for all PUFs and all response positions, individual bit error probabilities are obtained in the initialization phase for each PUF and each response position. The corresponding model was derived for SRAM PUFs in the literature.

In Chapter 3.2, we studied ROPUFs, the most often used PUF construction from the family of delay-based PUFs. Based on a public data set, we derived a channel model similar to the existing model for SRAM PUFs. For the one-probability as well as for the error probability, we derived cumulative distribution functions and probability density functions that model the underlying channel. The resulting channel model is not only useful in order to improve error correction, moreover it will be used in Chapter 4.3 to derive soft-decision secure sketches for ROPUFs. Both, the channel model and the secure sketches, have been published in [MPSB19].

In Chapter 3.3, we followed a comparatively new direction that constructs fully intrinsic

PUFs based on DRAM. In addition to the problems of noisy and not uniformly distributed PUF responses, which we have to deal with when using SRAM PUFs or ROPUFs, PUFs based on DRAM suffer from responses that are heavily biased. Therefore, we considered debiasing methods to be applied to the extracted DRAM data. We studied CL debiasing and VN debiasing for DRAM PUFs. Using CL debiasing, the behavior of the channel can be modeled by the so-called Z-channel. In this model, bits only change from "1" to "0" with bit error probability p_{b}, while a "0" is always transmitted correctly. Our proposal of VN debiasing prohibits inverted bits entirely with high probability. Instead, the channel turns some of the bits into erasure symbols. This behavior is an advantage for error correction, since there is no need for the decoder to identify erroneous positions in the received word. The erasure positions are obvious due to the erasure symbols in the received word, and only the correct bits have to be found for the given positions. A linear code with minimum distance d can correct $d-1$ erasures (when no errors are included), while only $\lfloor \frac{d-1}{2} \rfloor$ errors can be corrected. If both, τ errors and δ erasures are included in the received word, correction succeeds while $2\tau + \delta < d$ [Bos99, Chapter 3]. Error and erasure decoding will be considered in Chapter 5.1.

The drawback of CL and VN debiasing is the comparatively large number of extracted response bits, which are discarded during the initialization phase of the algorithms. This problem can, for example, be circumvented by using the techniques discussed in Section 3.3.3. Additionally, these algorithms do not require debiasing data. However, when comparing the results to CL and VN debiasing, we notice an unreliable reproduction of PUF responses, at least under some operating conditions. Summing up, Figures 3.6– 3.10 visualize the inter- and intra-response distances of the PUF responses generated by the debiasing techniques discussed in this chapter.

Future studies contain the verification of the derived channel models by using more retention measurements and also by using a larger amount of operating conditions. Also, we plan to propose error-correcting codes, which are suitable for the channel models derived based on the DRAM retention data. A first attempt is included in [MBS+19]. Furthermore, we intend to conduct studies based on row-hammer measurements.

4

Secure Sketches

Secure sketches are also often called helper data algorithms or fuzzy extractors. However, there exist subtle differences between those terms. Recall from Section 2.3.1, that dealing with PUFs includes two problems. First, the fuzziness of PUF responses, second their non-uniformity. A secure sketch is used to tackle the former problem. The terms helper data algorithm and fuzzy extractor are often treated as synonyms (e.g., [GKST07, BGS+08, MTV09a, MTV09b]) and additionally contain a solution for the problem of non-uniformity, usually by hashing the response to the final key. With other words, the functionality of a secure sketch can be understood as a subset of the functionality of a helper data algorithm, and contrariwise, a helper data algorithm can be constructed by extending a secure sketch.

From a logical perspective, the functionality of a helper data algorithm can be divided into two components. The *information reconciliation* component corresponds to a secure sketch and cares about the fuzziness of responses by providing an error-correction unit. The *privacy amplification* or *randomness extraction* component deals with the non-uniformity of responses by applying a hash function. Another concept, that occurs in the literature, is a *fuzzy embedder*, introduced in [BDH+10]. In contrast to a helper data algorithm, a fuzzy embedder is used to embed a key into a PUF response. Since we focus on error correction, and hence, ignore the non-uniformity problem, we decide to use the term secure sketch from now on. For this reason, the hash function is ignored in this chapter, in the description of the algorithms as well as in the corresponding visualizations.

The concepts of secure sketches and helper data algorithms were introduced by Jules and Wattenberg in 1999 [JW99], Linnartz et al. in 2003 [LT03], and Dodis et al. in 2004 [DRS04, DORS08] in the context of biometrics[1]. The generic structure of a secure sketch is shown in Figure 4.1. Within this chapter, this structure is used in order to explain specific implementations of secure sketches. As visualized in Figure 4.1, secure sketches always consist of two phases: *Initialization* and *reproduction*. The initialization phase is executed only once for an initial response $\boldsymbol{r}$ and aims for obtaining and storing so-called helper data $\boldsymbol{h}$ (cf. helper data generation and helper data storage components in Figure 4.1), which are later used during the reproduction phase, in order to correct erroneous responses. We emphasize, that helper data do not require a protected memory, since secure sketches have to be designed such that an attacker cannot reproduce the initial response by knowing the

[1] As PUFs can be understood as fingerprints of objects, the concepts of PUFs and human biometrics are closely related. Both techniques suffer from the fact that extracting the exact same information twice is possible only in rare cases. As stated in [DRS04], the concept of a fuzzy extractor is very general and can be used in all scenarios where we have no precise reproducibility, as well as no uniform distributed outputs.

stored helper data. Recall from Chapter 2.1.4, that storing a secret key instead would require a protected memory, what implies additional cost and chip area, as well as possibilities for attackers, for example, by performing methods from the field of reverse engineering. It can be assumed, that initialization is performed in a secure environment, e.g., in the factory during the manufacturing process. On the contrary, reproduction is executed multiple times in field, whenever the application requires to extract a response from the PUF.

During reproduction, the helper data which were generated during initialization are read from the helper data storage in order to map the erroneous response to a codeword plus error (preprocessing component), such that a decoding algorithm (decoding component) can be applied. Finally, the decoding result has to be mapped back to the initial response by again using the helper data. Since we restrict ourselves to the problem of fuzzy responses, we call the algorithm's output *key*. We want to emphasize again, that in a practical implementation of a secure sketch, the output (corrected response) has to be hashed to the final key.

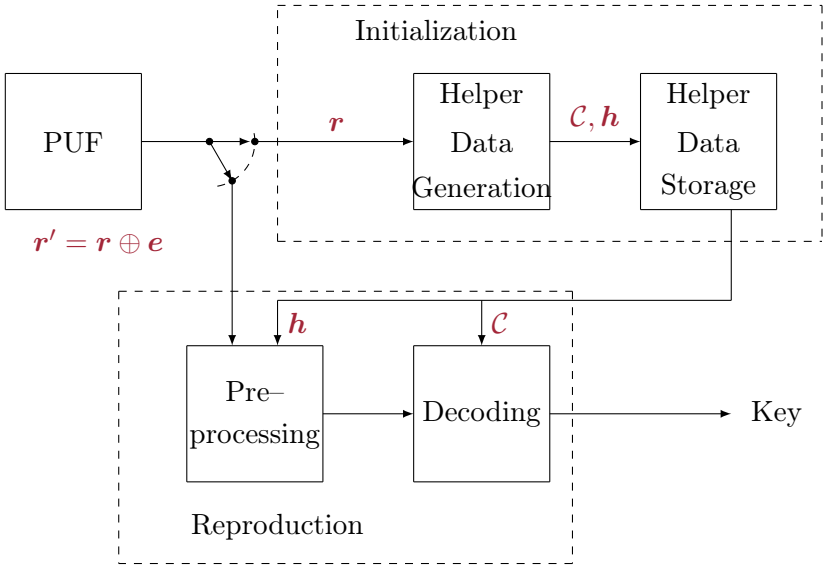


Figure 4.1: Generic structure of a secure sketch. Initialization is performed once in a secure environment in order to obtain helper data $\boldsymbol{h}$. These helper data are used in the reproduction phase, which aims for reproducing the initial response $\boldsymbol{r}$ from an erroneous (fuzzy) response $\boldsymbol{r}'$. Note that no protected memory is needed as helper data storage.

When designing secure sketches, the following criteria are crucial for evaluating the scheme: For an adequate security level, it is of utmost importance that the helper data do not leak substantial information about the produced key. Further, the area of the chip is aimed to be small, since a low price per chip is desired in practical implementations. Also, the amount of helper data that has to be stored and the complexity of the secure sketch have to be considered.

This chapter first gives a survey on existing secure sketches in Section 4.1. Our new proposed scheme is based on LDPC codes and is introduced, discussed and analyzed in Section 4.2. The proposal of the scheme was published in [MB17a]. Further construction methods for LDPC codes from literature, that can be applied in order to construct the

codes in our scheme, were implemented within a bachelor's thesis [Bal17]. The studies about practicability as well as security aspects have been published in [MPB18a]. In Section 4.3, two soft-decision secure sketches for ROPUFs are presented and evaluated. These algorithms have been published in [MPSB19], preliminary studies for that work were executed within a master's thesis [Stu17].

4.1 Classical Schemes

This section gives an overview of existing schemes that have been proposed in the literature. The most prominent schemes are the code-offset construction and the syndrome construction. Often, in the literature they are treated as different methods, however, we will formally proof that both schemes are equivalent. Also, we will briefly review existing soft-decision secure sketches, since we will present soft-decision secure sketches for ROPUFs in Section 4.3.

4.1.1 Code-Offset Construction

One of the most often used schemes (e.g. in [BGS+08]) is the code-offset construction, as proposed in [JW99][2]. In this section, we explain the code-offset construction according to Figure 4.2. During initialization, the initial response $\boldsymbol{r}$ is extracted from the PUF. In order to generate helper data, a codeword $\boldsymbol{c}$ of a preassigned code $\mathcal{C}$ is chosen uniformly at random and is component-wise XOR-ed with $\boldsymbol{r}$. The result

$$\boldsymbol{h} := \boldsymbol{r} \oplus \boldsymbol{c} \in \mathbb{F}_2^n \tag{4.1}$$

is called offset and stored in the helper data storage for later usage. The helper data consist of the n bit vector $\boldsymbol{h}$ and a representation of the code $\mathcal{C}$.

In the reproduction phase, a fuzzy response $\boldsymbol{r}'$ is extracted from the PUF. Due to its fuzziness, it can be interpreted as the initial response $\boldsymbol{r}$ plus some low-weight error vector $\boldsymbol{e}$, which we want to eliminate, i.e.,

$$\boldsymbol{r}' = \boldsymbol{r} \oplus \boldsymbol{e}. \tag{4.2}$$

Since $\boldsymbol{r} = \boldsymbol{c} \oplus \boldsymbol{h}$, it holds that

$$\boldsymbol{r}' = \boldsymbol{c} \oplus \boldsymbol{h} \oplus \boldsymbol{e}. \tag{4.3}$$

When we load the offset $\boldsymbol{h}$ from the helper data storage and add it to $\boldsymbol{r}'$ in the preprocessing stage, the result is

$$\boldsymbol{r}' \oplus \boldsymbol{h} = \boldsymbol{c} \oplus \boldsymbol{e}, \tag{4.4}$$

where $\boldsymbol{e}$ has low weight as described above. The resulting vector, which is named $\boldsymbol{y}$ in the figure, is used as input to a decoding algorithm for the code $\mathcal{C}$. If the distance of $\boldsymbol{r}$ and $\boldsymbol{r}'$ is within the error correction capabilities of $\mathcal{C}$, the decoder correctly produces the codeword

[2] In most publications dealing with error correction for PUFs, Dodis et al. [DRS04] is given as reference for the code-offset construction. However, the scheme already occurred in the work of Juels and Wattenberg from 1999 [JW99] and was rephrased in [DRS04] by providing a reference to the original work. The 1999 work from Juels and Wattenberg is also available in an extended version [JW13].

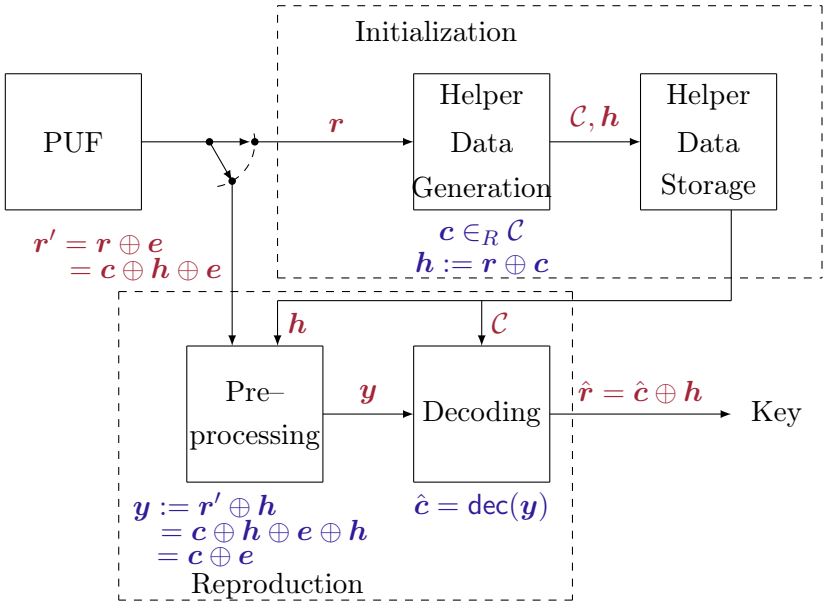


Figure 4.2: Code-offset construction according to [JW99]. In the initialization phase, a codeword of the specified code is randomly chosen, and the offset $\boldsymbol{h} = \boldsymbol{r} \oplus \boldsymbol{c}$ is calculated and stored as helper data. During reproduction, this offset is used in order to map the response $\boldsymbol{r}'$ into the format codeword plus low-weight error vector $\boldsymbol{e}$, which is the format that is required by a decoding algorithm.

$\boldsymbol{c}$, which was chosen during initialization. Using again the offset $\boldsymbol{h}$ for an XOR-operation on the output of the decoder restores the initial response $\boldsymbol{r} = \boldsymbol{c} \oplus \boldsymbol{h}$. In the PUF community, the pre-processing unit is often called syndrome decoding (cf. for example [HWRL+13]), but the term might be ambiguous from the perspective of coding theory, since it does not necessarily have to deal with the calculation of a syndrome. Hence, we decided to rename this component into "pre-processing".

Figure 4.3 intuitively visualizes how and why the code-offset construction works as intended. The rectangular box illustrates the n-dimensional binary space 2^n. The codewords of the applied code $\mathcal{C}$ are indicated by the gray dots inside the box. The spheres around the codewords represent the error-correction radius of $\mathcal{C}$. When a vector $\boldsymbol{y}$ that is located in one of the spheres is received, it will be decoded uniquely to the codeword in the center of the corresponding sphere (cf. BMD decoding, Definition 2.13). Assume that an initial response $\boldsymbol{r}$ is extracted from the PUF. Since $\boldsymbol{r}$ is considered to be a random vector, it is located anywhere in the space 2^n and is visualized by a red dot in Figure 4.3. We want to stress again, that $\boldsymbol{r}$ in general is not a codeword. If we obtain a noisy response $\boldsymbol{r}'$, having a distance from $\boldsymbol{r}$ that lies within the error-correction capability of $\mathcal{C}$, adding $\boldsymbol{h}$ to $\boldsymbol{r}'$ leads to a vector $\boldsymbol{y}$ inside the sphere of the chosen codeword $\boldsymbol{c}$ and thus can be uniquely decoded to $\boldsymbol{c}$.

Note that the public helper data leak some information. Knowing the helper data, the uncertainty is as large as the amount of codewords in the chosen code. An attacker ist able to perform a brute-force attack by using all possible, i.e. all 2^k many, codewords in a trial and error manner. In total, there are 2^n many possible response vectors $\boldsymbol{r}$. However, since an attacker who knows $\boldsymbol{c}$ also knows $\boldsymbol{r}$ by calculating $\boldsymbol{r} = \boldsymbol{c} \oplus \boldsymbol{h}$, the number of possible responses

reduces to 2^k. Hence, the code-offset construction implies an entropy loss of $n - k$ bit.

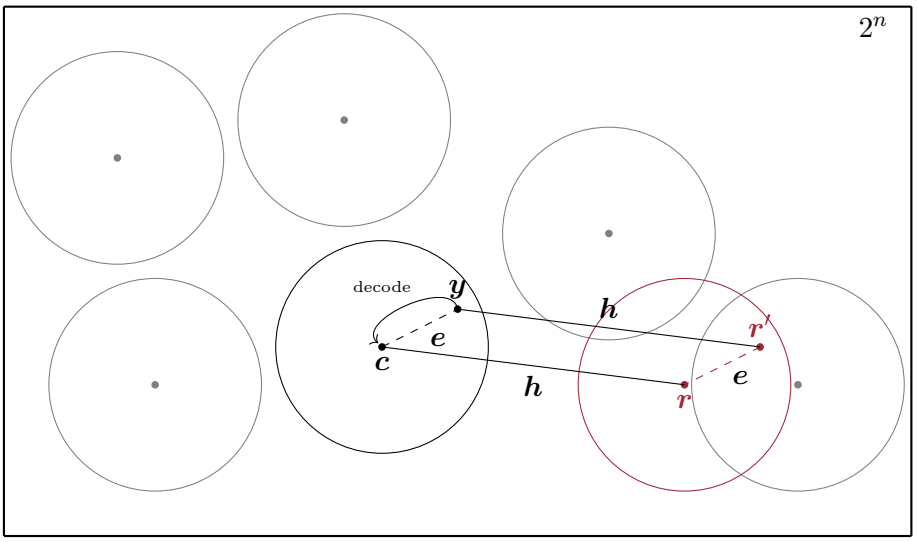


Figure 4.3: Correctness of the code-offset construction: If the distance between the responses $\boldsymbol{r}$ and $\boldsymbol{r}'$ is within the error correction capabilities of the chosen code, $\boldsymbol{r}$ can be regenerated by using the helper data $\boldsymbol{h}$.

A variant of the code-offset construction, called *Differential Sequence Coding (DSC)*, was introduced in [HWRL+13]. In addition to the offset vector $\boldsymbol{h}$, the helper data consist of an additional vector, which indicates the positions of highly reliable response bits. This reliability information might be available, for example, due to an extra step within the initialization process, where multiple responses are extracted and reliability information is calculated from those extractions. By selecting only response bits which are highly reliable and performing the code-offset construction, the overall channel is improved. This approach has advantages, when soft information exist and only a hard-decision decoder can be used, for example, due to technical reasons concerning the implementation.

4.1.2 Syndrome Construction

The syndrome construction was introduced in [LT03, DRS04] and is visualized in Figure 4.4. In the initialization phase, the initial response $\boldsymbol{r}$ is interpreted as received word. The helper data generation uses a parity-check matrix of the preassigned code $\mathcal{C}$ in order to calculate the syndrome $\boldsymbol{s}$, that serves as helper data. In this construction, the amount of helper data that have to be stored is the size of the syndrome vector, which is $n - k$, plus a representation of the code $\mathcal{C}$.

In the reproduction phase, $\boldsymbol{s}$ is component-wise XOR-ed to the syndrome calculated from the fuzzy response $\boldsymbol{r}'$. When the distance of $\boldsymbol{r}$ and $\boldsymbol{r}'$ lies within the error correction capability of $\mathcal{C}$, this XOR-ing results in the syndrome of the error $\boldsymbol{e}$ when $\boldsymbol{r}'$ is interpreted as $\boldsymbol{r} \oplus \boldsymbol{e}$. Hence, that syndrome can be used by a decoding algorithm in order to reproduce the initial response $\boldsymbol{r}$.

The syndrome construction is used, for example, in [MVHV12]. As already mentioned above, authors from outside the field of coding theory often use the word “syndrome” as

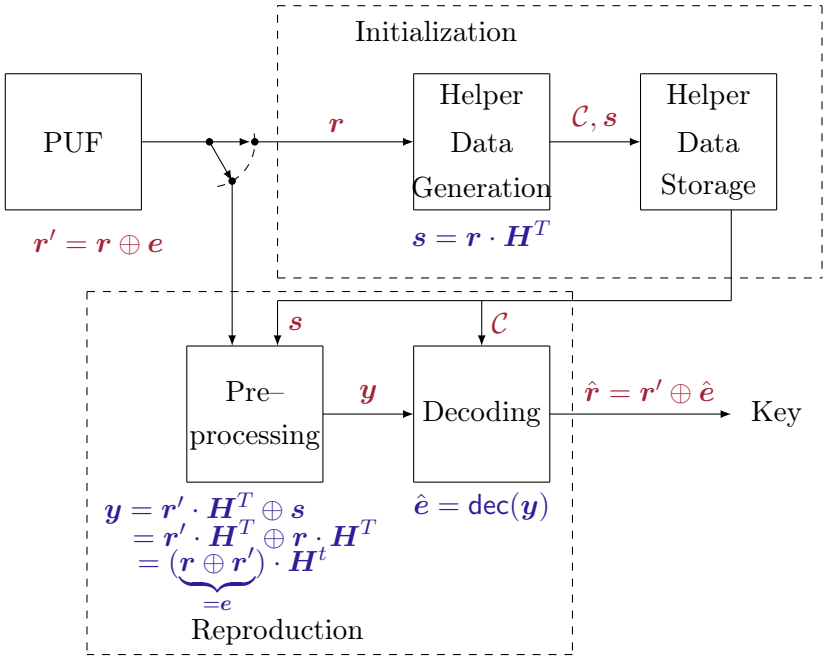


Figure 4.4: Visualization of the syndrome construction. In the initialization phase, the syndrome is calculated and stored as helper data. During reproduction, the syndrome is used in order to reproduce the initial response $\boldsymbol{r}$.

synonym for helper data, also when the helper data do not contain the syndrome (e.g. [HMSS12, HWRL+13]). This naming convention is most probably derived from the syndrome construction. We want to stress, that in general helper data are not necessarily the same as a syndrome as used in coding theory (e.g. compare with the code-offset construction in Section 4.1.1). Hence, in this dissertation we explicitly use the term helper data. As already mentioned in Section 4.1.1, in the context of secure sketches and helper data algorithms, we use the term "pre-processing" for the unit that is usually named "syndrome decoder" in the PUF literature.

Finally, we show that although literature often distinguishes between code-offset and syndrome construction, both schemes are equivalent. Thus, the entropy loss is the same as for the code-offset construction.

Theorem 4.1. Code-offset construction and syndrome construction are equivalent.

Proof. We begin with the direction from code-offset to syndrome construction. Using the code-offset construction, we construct an offset by calculating $\boldsymbol{h} = \boldsymbol{r} \oplus \boldsymbol{c}$. We calculate

$$\boldsymbol{H}\boldsymbol{h}^\top = \boldsymbol{H}(\boldsymbol{r} \oplus \boldsymbol{c})^\top = \boldsymbol{H}\boldsymbol{r}^\top \oplus \underbrace{\boldsymbol{H}\boldsymbol{c}^\top}_{=0} = \boldsymbol{H}\boldsymbol{r}^\top, \tag{4.5}$$

which is exactly the syndrome $\boldsymbol{s}$, which is calculated in the initialization of the syndrome construction.

To prove the other direction, we begin with the syndrome construction, where we calculate $\boldsymbol{s} = \boldsymbol{H}\boldsymbol{r}^\top$ during initialization. We consider the under-determined system of linear equations

$\boldsymbol{s} = \boldsymbol{H}\boldsymbol{r}'^{\top}$ and find any solution for $\boldsymbol{r}'$, which we interpret as offset, denoted by $\boldsymbol{h}$ in Figure 4.2 when explaining the code-offset construction. Note that the kernel of $\boldsymbol{H}$ is $\boldsymbol{r} + \mathcal{C}$. It exists a $\boldsymbol{c} \in \mathcal{C}$, such that $\boldsymbol{r}' = \boldsymbol{r} \oplus \boldsymbol{c}$. This proves the theorem. □

4.1.3 Pointer-based Methods

Pointer-based methods store pointers to specific bits of the PUF response. For example, sufficiently reliable response bits are chosen in order to reduce errors when embedding secrets into responses. Examples for so-called pointer-based methods are *Index-Based Syndrome Coding (IBS)* as introduced in [YD10] and *Complementary Index-Based Syndrome Coding* as proposed in [HMSS12]. Both methods were introduced in the context of fuzzy embedders. Since pointer-based methods are not used within this dissertation, an explanation is omitted and the interested reader can access the references given in this paragraph.

4.1.4 Secure Sketches Using Soft Information

The first soft-decision secure sketch was introduced in [MTV09a] for SRAM PUFs. It is based on the BSC model with varying bit error probabilities, that is also introduced in [MTV09a] and revisited in Chapter 3, Figure 3.2. The algorithm can be described by an extension of the code-offset scheme. In the initialization phase, the bit error probabilities p_{e_i} are collected for all SRAM cells $i = 1, \ldots, n$ and are stored in the helper data storage in addition to the offset $\boldsymbol{h}$ and the representation of the code $\mathcal{C}$. In the reproduction phase, the reliability values p_{e_i} are used as soft information for the decoding algorithm. A proof, that having the bit error probabilities p_{e_i} as public helper data does not imply an entropy loss, can be found in [MTV09a, Theorem 1]. Additionally, [MTV09b] provides an implementation of that soft-decision secure sketch. In Section 4.3, we propose and discuss two soft-decision secure sketches for ROPUFs.

4.2 A New Secure Sketch

This section contains one of the main contributions of this chapter. We propose a new secure sketch, which in contrast to the methods introduced in the literature, only uses an error-correcting code, but no further helper data. First, in Section 4.2.1 *Low-Density Parity-Check (LDPC) codes*, on which our scheme is based, are reviewed. Section 4.2.2 provides the intuitive idea of the new scheme. A formal description is given in Section 4.2.3. The practicability of the scheme is discussed in Section 4.2.4, while Section 4.2.5 deals with security considerations. Section 4.2.6 presents the results of simulations. Finally, in Section 4.2.7 we discuss possible applications of the scheme as well as its advantages and drawbacks in comparison to known schemes from literature.

4.2.1 Design and Iterative Decoding of Low-Density Parity-Check Codes

Binary low-density parity-check (LDPC) codes are well-known class of linear codes, introduced by Robert Gallager in [Gal63]. They are widely used in practical applications, since they can

be constructed as long codes that are efficiently decodable[3]. There are two types, namely regular and irregular LDPC codes. For both types, two equivalent approaches exist in order to define the codes.

The classical approach according to Gallager defines LDPC codes by a binary parity-check matrix of low density, i.e., a matrix $\boldsymbol{H} = (h_{ij})_{(i=1,\ldots,m;j=1,\ldots,n)}$, in which the number of ones is much smaller than the number of zeros. An LDPC code is called regular, if each column of $\boldsymbol{H}$ contains exactly $\gamma \in \mathbb{N}$ ones and each row contains exactly $\rho \in \mathbb{N}$ ones. An ensemble of regular LDPC codes is specified by the triple (n, γ, ρ). All codes describable by a parity-check matrix with n columns, $m = n \cdot \frac{\gamma}{\rho}$ rows, γ ones in each column, and ρ ones in each row belong to such an ensemble. If the parity-check matrix of a regular LDPC code consists of $n - k$ linearly independent rows, the code rate is $R = \frac{k}{n} = \frac{n-m}{n}$. In cases of a larger number of parity-check equations that are linearly dependent, the code rate increases. Hence, we obtain $R \geq \frac{n-m}{n} = 1 - \frac{\gamma}{\rho}$. In contrast to regular LDPC codes, irregular LDPC codes allow a different number of ones in the columns and rows, respectively. There exist a large amount of construction methods for LDPC codes, e.g., LDPC codes based on finite geometries [KLF00], LDPC codes based on Reed–Solomon codes [DXAGL03], Gallager's construction [Gal63], MacKay's construction [MN97] and the (modified) array structure construction [Fan01, EO02].

An alternative but equivalent approach, which is nowadays often used in order to describe LDPC codes as well as their decoding algorithms, is based on graph theory. Therefore, the parity-check matrix is represented as a bipartite graph, whose two sets of nodes are called *variable nodes* and *check nodes*. Each parity-check equation (row of $\boldsymbol{H}$) is represented by a check node. Similarly, each code symbol (column of $\boldsymbol{H}$) is represented by a variable node. An undirected edge between check node i and variable node j exists, if and only if, $h_{ij} = 1$. In other words, an undirected edge between check node i and variable node j exists, when code symbol j is in the support of the parity-check equation in row i. In the case of regular LDPC codes, each variable node has degree γ and each check node has degree ρ. For irregular LDPC codes, variable nodes as well as check nodes may have different degrees. Every parity-check matrix can be transformed into an equivalent bipartite graph and vice versa. Figure 4.5 exemplarily visualizes a bipartite graph that represents the code specified by the parity-check matrix

$$H = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}, \tag{4.6}$$

which is a regular LDPC code with $\gamma = 2$ ones per column and $\rho = 3$ ones per row[4]. Hence, each variable node (visualized by circles) of the graph has degree 2, while each check node (visualized by squares) has degree 3.

In order to decode LDPC codes, we explain the *bitflip algorithm* according to [BH86], which is adequate for hardware implementations in the PUF scenario. The bitflip algorithm,

[3] LDPC codes are, for example, contained in coding schemes for the standards DVB-S2 and IEEE 802.16e (Wi-MAX).

[4] This example is taken from [KKS05, Chapter 5.1]. Note that this code actually is not an LDPC code, since its parity-check matrix $\boldsymbol{H}$ is not of low density. However, real low-density matrices are only attainable for long codes, which are not suitable to be used as examples.

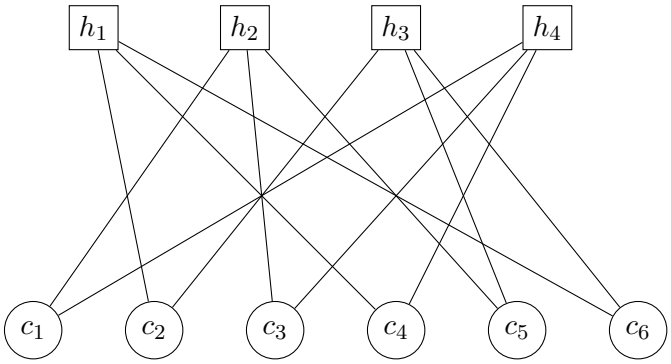


Figure 4.5: Bipartite graph that describes the code defined by the parity-check matrix $\boldsymbol{H}$, given in (4.6). The label h_i of a check node denotes parity-check equation i, which is given by row i of matrix $\boldsymbol{H}$. The labels c_j denote code symbol j and hence, column j of matrix $\boldsymbol{H}$.

as described in Algorithm 5, is a simple iterative method to decode LDPC codes. In the first iteration, the syndrome of the received vector $\boldsymbol{y}$ is calculated (cf. line 4). Recall from Chapter 2.2.1, Equation 2.12, that the received word $\boldsymbol{y}$ is a codeword if and only if its syndrome is the all-zero vector of length $n - k$. This in turn is the case, if and only if, the Hamming weight of the syndrome is zero (cf. line 5). In this case we are finished and decoding is performed successfully (cf. lines 6–8). If $\boldsymbol{y} \notin \mathcal{C}$, we modify $\boldsymbol{y}$ by inverting its bits, aiming for reducing the weight of the syndrome. By decreasing the weight of the syndrome in every iteration, the goal is to achieve a syndrome of weight zero, and thus a valid codeword, after some iterations. Hence, for each codeword position $i = 1, \ldots, n$, we modify $\boldsymbol{y}$ by inverting the bit in position i and thereby creating the vectors $\boldsymbol{y}^{(i)}$, which differ from $\boldsymbol{y}$ only in position i. Inverting the bit in position i of vector $\boldsymbol{y}$ is denoted as component-wise adding the i-th unit vector $\boldsymbol{u}_i$ to $\boldsymbol{y}$ in $\mathbb{F}_2$. After each bitflip, we compute and store the corresponding syndrome weight ω_i (cf. lines 9–12). For all n resulting vectors $\boldsymbol{y}^{(i)}$, we determine after which bitflip the weight of the syndrome receives the smallest value (cf. line 13). We terminate an iteration by flipping the corresponding bit in the original received vector and proceed with the next iteration (cf. line 14). The algorithm terminates as soon as $\boldsymbol{y}$ is turned into a valid codeword (cf. line 8), or when a predefined number of iterations is exceeded (break condition of the loop, cf. line 15).

The hard-decision bitflip decoder given in Algorithm 5 can be extended to soft-decision decoding. Thereto, we modify the calculations of ω_i in line 12 by

$$\omega_i = \mathrm{wt}(\boldsymbol{s}) + \Theta_i, \tag{4.7}$$

where Θ_i denotes soft information about position i, which in the PUF scenario, for example, can be gathered by performing multiple readouts. Assume, we extract m independent readouts from the PUF. If an error occurs in position i, with high probability only a subset of the m responses contains this error. Hence, the responses differ in position i. If all responses share a common value in position i, with high probability the bit in position i is error-free.

Algorithm 5: Bitflip algorithm for decoding LDPC code $\mathcal{C}$

Input: Vector $\boldsymbol{y} \in \mathbb{F}_2^n$, decoding matrix $\boldsymbol{H}$, $max\_iterations$
Output: Decoding result $\hat{\boldsymbol{c}}$

1 $iteration = 0$
2 **repeat**
3 $\quad iteration = iteration + 1$
4 $\quad \boldsymbol{s} = \boldsymbol{H} \cdot \boldsymbol{y}^\top$
5 $\quad \omega = \mathrm{wt}(\boldsymbol{s})$
6 $\quad$ **if** $\omega = 0$ **then**
7 $\quad\quad \hat{\boldsymbol{c}} = \boldsymbol{y}$
8 $\quad\quad$ **return** $\hat{\boldsymbol{c}}$
9 $\quad$ **for** $i = 1, \ldots, n$ **do**
10 $\quad\quad \boldsymbol{y}^{(i)} = \boldsymbol{y} + \boldsymbol{u}_i$
11 $\quad\quad \boldsymbol{s} = \boldsymbol{H} \cdot (\boldsymbol{y}^{(i)})^\top$
12 $\quad\quad \omega_i = \mathrm{wt}(\boldsymbol{s})$
13 $\quad$ Find $j \in \{1, \ldots, n\}$ with $\omega_j = \min_{i=1,\ldots,n} \omega_i$
14 $\quad \boldsymbol{y} = \boldsymbol{y} + \boldsymbol{u}_j$
15 **until** $iteration > max\_iterations$;
16 **return** $\hat{c}$

For all codeword positions $i = 1, \ldots, n$, we define the soft information

$$\Theta_i = \begin{cases} \delta_1, \text{if } \boldsymbol{r}_{1,i} = \ldots = \boldsymbol{r}_{m,i} \\ \delta_2, \text{otherwise,} \end{cases} \tag{4.8}$$

where $\delta_1 > \delta_2 > 0$. When adding the values Θ_i to the syndrome weight in line 12, we aim for increasing the weights ω_i for the reliable positions, such that they are not selected for modification in line 13. Figure 4.6 visualizes this idea for $m = 3$ extracted responses.

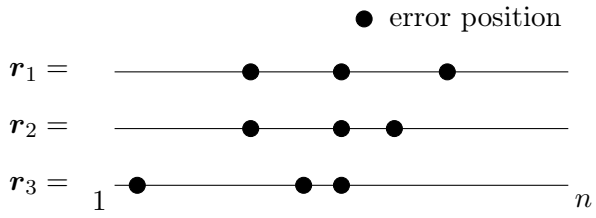


Figure 4.6: Multiple readouts (illustrated for $m = 3$ extracted responses) can be used to gather reliability information about the bit positions, since it is unlikely to happen, that a position is transmitted erroneously m times.

As an alternative to the bitflip algorithm, *message passing* algorithms are often used for iterative decoding. The family of message passing algorithms can be applied in different contexts like decoding, artificial intelligence or satisfiability testing. Usually, belief propaga-

tion is used in the context of decoding [Pea14, Gal63][5]. The explanation of message passing algorithms is usually based on the graph representation of an LDPC code.

4.2.2 Idea of the Secure Sketch

Figure 4.7 illustrates the structure of the new scheme in order to give an idea on an intuitive level. Using Figures 4.2 and 4.4 in order to compare the schemes discussed in Section 4.1 with our new scheme, we can directly observe that the new scheme does not contain a preprocessing unit. Since in general, the initial response is not a codeword, existing schemes use a preprocessing step in order to map a response to a codeword by using helper data. The main idea of the new scheme is to ensure that the initial response is a codeword of the applied error-correcting code. Then, the response extracted in the reproduction phase can directly be interpreted as codeword plus error, where the error vector indicates the positions in which initial and reproduced response differ. This is the form which is required as input to a decoding algorithm. Since responses are random, they cannot be forced to be a codeword of a specific code. A method to circumvent this problem, is to construct a binary linear code $\mathcal{C}$ based on a given response $\boldsymbol{r}$, such that $\boldsymbol{r}$ is a codeword of $\mathcal{C}$, i.e., $\boldsymbol{r} \in \mathcal{C}$. Formally, this problem can be stated as follows:

Problem 4.2. Given a vector $\boldsymbol{r} \in \mathbb{F}_2^n$ and a desired dimension $k < n$, find a binary linear code $\mathcal{C}(n, \approx k)$ with $\boldsymbol{r} \in \mathcal{C}$.

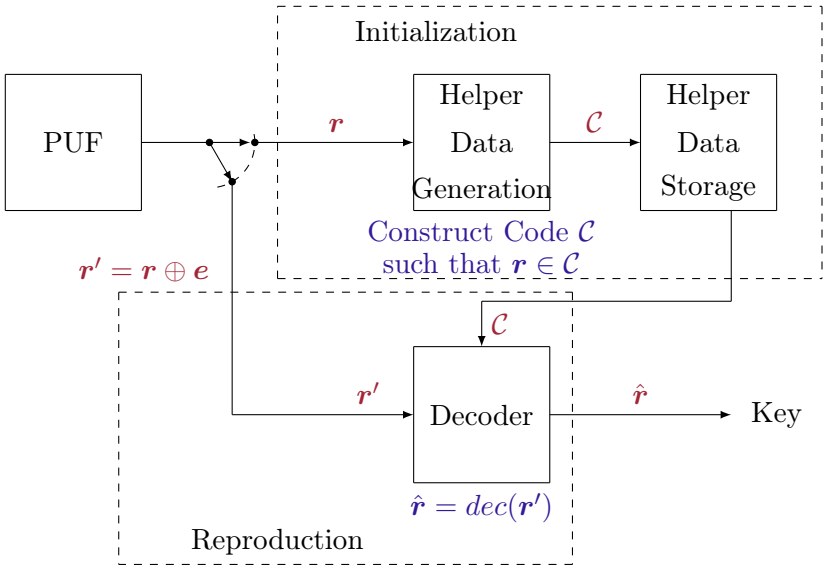


Figure 4.7: Visualization of the new scheme. In contrast to the schemes discussed in Section 4.1, no preprocessing unit is included since we ensure that the initial response $\boldsymbol{r}$ is a codeword of the used code $\mathcal{C}$.

As we will find out in Section 4.2.5, the dimension k affects the security level of the scheme.

[5]Currently, supported by master's theses [Raj18, Mar19], we study whether or not other message passing algorithms like survey propagation can be transferred from the field of satisfiability testing to decoding.

From Chapter 2.2.1 we know that

$$\boldsymbol{H} \cdot \boldsymbol{r}^\top = \mathbf{0} \Leftrightarrow \boldsymbol{r} \in \mathcal{C}. \tag{4.9}$$

We aim for constructing a parity-check matrix $\boldsymbol{H}$, such that $\boldsymbol{H} \cdot \boldsymbol{r}^\top = \mathbf{0}$ for a given initial response $\boldsymbol{r}$. We want the parity-check equations to be of low weight and hence, the matrix $\boldsymbol{H}$ to be a parity-check matrix of an LDPC code. The decision to use LDPC codes basically has three reasons: First, there exist efficient methods in order to represent low-density matrices, e.g., by representing each "1" by an ordered pair (row, column), which indicates its position in the matrix, cf. [Pis84, Chapter 1] or [DER17, Chapter 2]. Second, the existence of efficient iterative decoding algorithms that are suitable for implementations in hardware, as for example the bitflip algorithm outlined in Algorithm 5 and discussed in Section 4.2.1. Third, the minimum distance of an LDPC code is not as crucial for the error correction performance when comparing to algebraic codes. Using algebraic codes instead, would require that the responses have a Hamming weight, which is at least the minimum distance of the code. This would imply to discard PUFs, whose responses do not fulfill this property. In addition, this restriction might leak some side information, which can be utilized by attackers. The proposed code construction is explained in Section 4.2.3.

4.2.3 Algorithm

After discussing the idea of the secure sketch on an intuitive level in Section 4.2.2, this section provides a more formal description. Algorithm 6 explains the construction process of decoding matrix $\boldsymbol{H}$, which is part of the initialization phase. Algorithm 7 describes the reproduction phase of the secure sketch, which only consists of executing the decoding routine. Before studying these algorithms, we start by constructing $\boldsymbol{H}^{(\mathrm{I})}$, which is a decoding matrix of an LDPC code of length n (where n denotes the response length), that is used as input to Algorithm 6.

To construct $\boldsymbol{H}^{(\mathrm{I})}$, we use $j \in \mathbb{N}^+$ construction methods for LDPC codes in order to generate a set of j decoding matrices $\boldsymbol{H}_i^{(\mathrm{I})}$ $(i = 1, \ldots, j)$ of length-n LDPC codes[6]. $\boldsymbol{H}^{(\mathrm{I})}$ is formed as the union of these matrices, i.e.,

$$\boldsymbol{H}^{(\mathrm{I})} = \bigcup_{i=1}^{j} \boldsymbol{H}_i^{(\mathrm{I})}, \tag{4.10}$$

where we define the union of matrices as the vertical concatenation of their rows. $\boldsymbol{H}^{(\mathrm{I})}$ is the decoding matrix of an LDPC code with parameters $(n, k = n - \mathrm{rank}(\boldsymbol{H}^{(\mathrm{I})}))$. Since the construction of the matrices $\boldsymbol{H}_i^{(\mathrm{I})}$ (and hence $\boldsymbol{H}^{(\mathrm{I})}$) is independent from the initial response $\boldsymbol{r}$, only a subset of the parity-check equations is orthogonal[7] to $\boldsymbol{r}$. Let $m_{\boldsymbol{H}^{(\mathrm{I})}}$ denote the number of rows of decoding matrix $\boldsymbol{H}^{(\mathrm{I})}$ and note that $m_{\boldsymbol{H}^{(\mathrm{I})}}$ is much larger than the rank of $\boldsymbol{H}^{(\mathrm{I})}$, i.e., $m_{\boldsymbol{H}^{(\mathrm{I})}} \gg \mathrm{rank}(\boldsymbol{H}^{(\mathrm{I})})$. Table 4.1 contains the numbers of rows $m_{\boldsymbol{H}^{(\mathrm{I})}}$ and $\mathrm{rank}(\boldsymbol{H}^{(\mathrm{I})})$ for four binary linear (n, k) LDPC codes, that were constructed in the described way.

[6] To generate the decoding matrices, the construction methods mentioned in Section 4.2.1 can be used.

[7] Recall from linear algebra, that two vectors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ are orthogonal if their scalar product is zero, i.e., $\langle \boldsymbol{x}_1, \boldsymbol{x}_2 \rangle = 0$. Recall from Chapter 2.2.4, that in coding theory this property is often named *dual*, and hence, dual and orthogonal are treated as synonyms within this dissertation.

Table 4.1: Code examples: The first three examples are codes used in [MB17a], generated by Algorithm 6. The fourth example is the EG code of length 512 from [Bos13, Chapter 5.5]. For all four codes, the number of rows in their decoding matrices is much larger than their rank.

n	k	$m_{\boldsymbol{H}^{(\mathrm{I})}}$	$\mathrm{rank}(\boldsymbol{H}^{(\mathrm{I})})$
128	13	881	115
128	56	349	72
256	106	555	150
512	139	4672	373

Algorithm 6 uses the initial response $\boldsymbol{r}$, as well as the constructed matrix $\boldsymbol{H}^{(\mathrm{I})}$, in order to produce a decoding matrix $\boldsymbol{H}$ of an LDPC code $\mathcal{C}$, which contains $\boldsymbol{r}$ as codeword. Therefore, the algorithm adds the parity-check equations which fulfill the condition stated in Equation 4.9 to the final decoding matrix $\boldsymbol{H}$ and ignores all the other rows. $\boldsymbol{H}$, which only contains the rows from $\boldsymbol{H}^{(\mathrm{I})}$ that we decided to keep, is now a matrix that fulfills (4.9). Note that when deleting rows from a decoding matrix, the error-correction capability will decrease. If we choose, for example, $j = 1$ and thus construct only one matrix $\boldsymbol{H}_j^{(\mathrm{I})}$, we will eliminate too many parity-check equations when performing Algorithm 6 and cannot guarantee an error-correction capability that is sufficient for the considered scenario. This is the reason, why we need more rows than provided by only one decoding matrix, and hence, why we have to combine parity-check equations from different decoding matrices in order to construct the final decoding matrix $\boldsymbol{H}$. On the other hand, while increasing the number of rows of a decoding matrix, the rank of the matrix increases, and in turn, the dimension decreases. The construction process of $\boldsymbol{H}$ has to be stopped, when $\boldsymbol{H}$ possesses a desired dimension. Using that mechanism, the rate of the constructed code can be adjusted. We again emphasize, that the code construction process in Algorithm 6 is executed only once per PUF during the initialization phase of the proposed secure sketch. Its output matrix $\boldsymbol{H}$ serves as decoding matrix of a code $\mathcal{C}$ with the property $\boldsymbol{r} \in \mathcal{C}$.

Algorithm 6: LDPC code construction algorithm: Initialization phase

Input: Vector $\boldsymbol{r} \in \mathbb{F}_2^n$, dec. matrix $\boldsymbol{H}^{(\mathrm{I})}$ with $m_{\boldsymbol{H}^{(\mathrm{I})}}$ rows
Output: Decoding matrix $\boldsymbol{H}$ of a code $\mathcal{C}(n, k)$, such that $\boldsymbol{r} \in \mathcal{C}$

```
H initially is a matrix with n columns and 0 rows.
for i = 1, 2, ..., m_{H^(I)} do
    if i-th row h_i of H^(I) is orthogonal to r then
        Vertically append h_i to H.
return H
```

The matrix $\boldsymbol{H}$, which is produced as output of Algorithm 6 is used during the reproduction phase without any further helper data in order to reproduce the initial response $\boldsymbol{r}$ from a noisy version $\boldsymbol{r}'$. The reproduction phase, as outlined in Algorithm 7, consists of a single

Algorithm 7: Reproduction phase

Input: Re-extracted PUF response $\boldsymbol{r}' = \boldsymbol{r} + \boldsymbol{e} \in \mathbb{F}_2^n$, decoding matrix $\boldsymbol{H}$ produced by Algorithm 6

Output: Decoding result $\hat{\boldsymbol{r}} = \mathsf{dec}(\boldsymbol{r}', \boldsymbol{H})$

1 $\hat{\boldsymbol{r}} = \mathsf{dec}(\boldsymbol{r}', \boldsymbol{H})$

2 **return** $\hat{\boldsymbol{r}}$

step, namely of applying a decoding algorithm dec, which uses the noisy response $\boldsymbol{r}' = \boldsymbol{r} + \boldsymbol{e}$, as well as the decoding matrix $\boldsymbol{H}$ and produces a decoding result $\hat{\boldsymbol{r}}$. When the errors can be corrected by using code $\mathcal{C}$, decoding results in $\hat{\boldsymbol{r}} = \boldsymbol{r}$. In general, the decoding algorithm dec can be implemented by using an arbitrary method for decoding LDPC codes. For the usage in the context of PUFs, we recommend the bitflip algorithm that is stated in Section 4.2.1 due to its small implementation size. The bitflip algorithm is also used in the simulations summarized in Section 4.2.6.

The new secure sketch only needs an error-correcting code for reproducing the initial response. Hence, any pre-processing can be omitted. In contrast to classical schemes, the new secure sketch suffers from a larger complexity in the initialization phase. However, it is often assumed that the initialization phase does not necessarily have to be implemented on the device, cf. [GKST07, HPS15], which lessens the complexity overload when constructing a code for each device during the initialization process. Further, the construction has an advantage when using weak PUFs, for example, PUFs that only produce one response which can be used for identification purposes. Using the scheme also for strong PUFs would require to construct and store a code for every possible challenge response pair, which is not practical considering storage capacity.

4.2.4 Correctness and Practicability

In this section, we study correctness and practicability of the proposed secure sketch. First, we show, that the code resulting from the construction described in Section 4.2.3, is an LDPC code that contains the response vector as codeword. Second, we examine for which responses $\boldsymbol{r}$ it is possible to perform the code construction.

We start with proving that the constructed code $\mathcal{C}$ is an LDPC code with $\boldsymbol{r} \in \mathcal{C}$. First, we provide some notation for a more general lemma. Let $\boldsymbol{B}_K = \{\boldsymbol{b}_i\}$ be the rows of a decoding matrix of an LDPC code that was generated by using any construction method K. Further, we define

$$\boldsymbol{B}(\boldsymbol{r}) := \{\boldsymbol{b}_j \in \boldsymbol{B}_K : \langle \boldsymbol{b}_j, \boldsymbol{r} \rangle = 0\} \tag{4.11}$$

to be the rows from $\boldsymbol{B}_K$ that are orthogonal to the initial response $\boldsymbol{r}$. Let L denote the cardinality of $\boldsymbol{B}(\boldsymbol{r})$, and $\mathcal{C}_L$ the code defined by $\boldsymbol{B}(\boldsymbol{r})$.

Lemma 4.3. There exists a matrix $\boldsymbol{H} \subseteq \boldsymbol{B}(\boldsymbol{r})$ with $n - k$ rows, which is a parity-check matrix of an LDPC code $\mathcal{C}_L$, where $n - k = \mathrm{rank}(\boldsymbol{B}(\boldsymbol{r}))$, with $\boldsymbol{r} \in \mathcal{C}_L$.

Proof. It is $\boldsymbol{B}(\boldsymbol{r}) \subseteq \mathcal{C}_L^{\perp}$ by construction, since rows of $\boldsymbol{B}(\boldsymbol{r})$ are codewords of the dual code $\mathcal{C}_L^{\perp}$ (cf. Chapter 2, Remark 2.16). There exist $n-k$ parity-check equations in $\boldsymbol{B}(\boldsymbol{r})$ such that

$\text{rank}(\boldsymbol{H}) = n - k$. Hence, there exists a $k \times n$ matrix $\boldsymbol{G}$ that generates the code $\mathcal{C}_L$ and an information vector $\boldsymbol{i} \in \mathbb{F}_2^k$ such that $\boldsymbol{i} \cdot \boldsymbol{G} = \boldsymbol{r}$. □

The correctness of the new secure sketch is implied by the following theorem:

Theorem 4.4. The code $\mathcal{C}$ defined by the decoding matrix $\boldsymbol{H}$ that is constructed by Algorithm 6, is an LDPC code with $\boldsymbol{r} \in \mathcal{C}$.

Proof. The theorem directly follows from Lemma 4.3, which can be generalized to the code construction proposed in this section. Then, the rows of matrix $\boldsymbol{B}(\boldsymbol{r})$ are taken from different construction methods and hence $\boldsymbol{B}_K$ corresponds to $\boldsymbol{H}^{(\mathrm{I})}$ and $\boldsymbol{B}(\boldsymbol{r})$ corresponds to $\boldsymbol{H}$. □

If the distance between initial response $\boldsymbol{r}$ and re-extracted response $\boldsymbol{r}'$ is small enough, $\boldsymbol{r}'$ can be recovered by the code constructed in the LDPC-based secure sketch.

Next, we study the practicability of the proposed secure sketch. For this purpose, we examine for which responses the code construction can be performed. Let $\boldsymbol{h}_i$ be the i^{th} row of decoding matrix $\boldsymbol{H}^{(\mathrm{I})}$. We are interested in the probability, that $\boldsymbol{h}_i$ can be used for matrix $\boldsymbol{H}$, which according to the construction is the case if and only if $\langle \boldsymbol{h}_i, \boldsymbol{r} \rangle = 0$. To calculate that probability, we need to consider the positions of response $\boldsymbol{r}$ that are indexed by the support of $\boldsymbol{h}_i$, since the calculation of the scalar product only depends on that positions. Since $\boldsymbol{r}$ is assumed to be generated uniformly at random, i.e., $\boldsymbol{r} \sim U(\mathbb{F}_2^n)$, we conclude that

$$\mathrm{P_r}\left(\langle \boldsymbol{h}_i, \boldsymbol{r} \rangle = 0\right) = \frac{1}{2}, \tag{4.12}$$

and hence, each row of $\boldsymbol{H}^{(\mathrm{I})}$ can be used for constructing $\boldsymbol{H}$ with probability $\frac{1}{2}$.

We estimate the number of rows $m_{\boldsymbol{H}}$ of decoding matrix $\boldsymbol{H}$. Since $\boldsymbol{H}^{(\mathrm{I})}$ is sparse, the supports of two of its rows $\boldsymbol{h}_i \neq \boldsymbol{h}_j$ are most likely disjoint , i.e., $\mathsf{supp}(\boldsymbol{h}_i) \cap \mathsf{supp}(\boldsymbol{h}_j) = \emptyset$ with high probability. If these supports are not disjoint, we can expect the size of the overlap, $|\mathsf{supp}(\boldsymbol{h}_i) \cap \mathsf{supp}(\boldsymbol{h}_j)|$, to be small. From this observation we can conclude, that the events $\langle \boldsymbol{h}_i, \boldsymbol{r} \rangle = 0$ and $\langle \boldsymbol{h}_j, \boldsymbol{r} \rangle = 0$ essentially are statistically independent for any i and j. Hence, we assume that the number of rows $m_{\boldsymbol{H}}$ is binomially distributed with parameters $m_{\boldsymbol{H}^{(\mathrm{I})}}$ and $p = \frac{1}{2}$, i.e.,

$$m_{\boldsymbol{H}} \sim \mathsf{Bin}\left(m_{\boldsymbol{H}^{(\mathrm{I})}}, \frac{1}{2}\right). \tag{4.13}$$

This assumption is fulfilled by observation, as Example 4.5 shows.

Example 4.5. We choose the construction based on Euclidean Geometry (cf. Section 4.2.1) to generate an EG LDPC code of length 512 and dimension 139. The number of rows of the corresponding decoding matrix $\boldsymbol{H}^{(\mathrm{I})}$ is 4672 (cf. parameters given in [Bos13, Section 5.5.2]). We choose responses $\boldsymbol{r}$ uniformly at random and execute Algorithm 6 in order to generate a decoding matrix $\boldsymbol{H}$ from $\boldsymbol{H}^{(\mathrm{I})}$ and $\boldsymbol{r}$. We consider the statement given in (4.13). According to the theoretical derivation in this section, $m_{\boldsymbol{H}}$ should be approximately binomial distributed with parameters $m_{\boldsymbol{H}} = 4672$ and $p = \frac{1}{2}$. We simulated a sample size of 10^6 in order to compare with the corresponding theoretical cumulative distribution function (cdf). Figure 4.8 visualizes the comparison between the theoretical and the empirical cdf. As can be seen in the plot, the two curves almost coincide. Table 4.2 compares the estimated mean and variance with the corresponding theoretical values. The estimated values from the simulations are close to their theoretical counterparts.

The following two lemmas provide further theoretical statements.

Lemma 4.6. If $m_{\boldsymbol{H}^{(\mathrm{I})}} \gg \operatorname{rank}(\boldsymbol{H}^{(\mathrm{I})})$, then $\mathrm{P_r}\left(m_{\boldsymbol{H}} < \operatorname{rank}(\boldsymbol{H}^{(\mathrm{I})})\right)$ is negligible.

Proof. The lemma follows from the binomial distribution of $m_{\boldsymbol{H}}$, cf. (4.13). □

Lemma 4.7. $\operatorname{rank}(\boldsymbol{H}) \geq \operatorname{rank}(\boldsymbol{H}^{(\mathrm{I})}) - 1$ with high probability.

Proof. When $m_{\boldsymbol{H}^{(\mathrm{I})}} \gg \operatorname{rank}(\boldsymbol{H}^{(\mathrm{I})})$, the rows of $\boldsymbol{H}$ with high probability generate the vector space

$$\mathcal{V} := \langle \boldsymbol{H}^{(\mathrm{I})} \rangle \cap \mathrm{OC}(\boldsymbol{r}).$$

Using the dimensional formula for sub-vector spaces, it follows that

$$\begin{aligned}\operatorname{rank}(\boldsymbol{H}) = \dim(\mathcal{V}) &= \underbrace{\dim\left(\langle \boldsymbol{H}^{(\mathrm{I})} \rangle\right)}_{\substack{=\operatorname{rank}(\boldsymbol{H}^{(\mathrm{I})}) \\ \text{def.}}} + \underbrace{\dim(\mathrm{OC}(\boldsymbol{r}))}_{\substack{= \\ \text{Remark 2.16}}\ n-1} - \underbrace{\dim\left(\langle \boldsymbol{H}^{(\mathrm{I})} \rangle + \mathrm{OC}(\boldsymbol{r})\right)}_{\leq n} \\ &\geq \operatorname{rank}\left(\boldsymbol{H}^{(\mathrm{I})}\right) - 1.\end{aligned}$$

□

We checked the statement given in Lemma 4.7, by using the same code as in Example 4.5 In our simulations, it was always $\operatorname{rank}(\boldsymbol{H}) \geq \operatorname{rank}(\boldsymbol{H}^{(\mathrm{I})}) - 1$ as predicted by the lemma. For these simulations we reduced the sample size from 10^6 to 10^3 due to the complexity of rank computation.

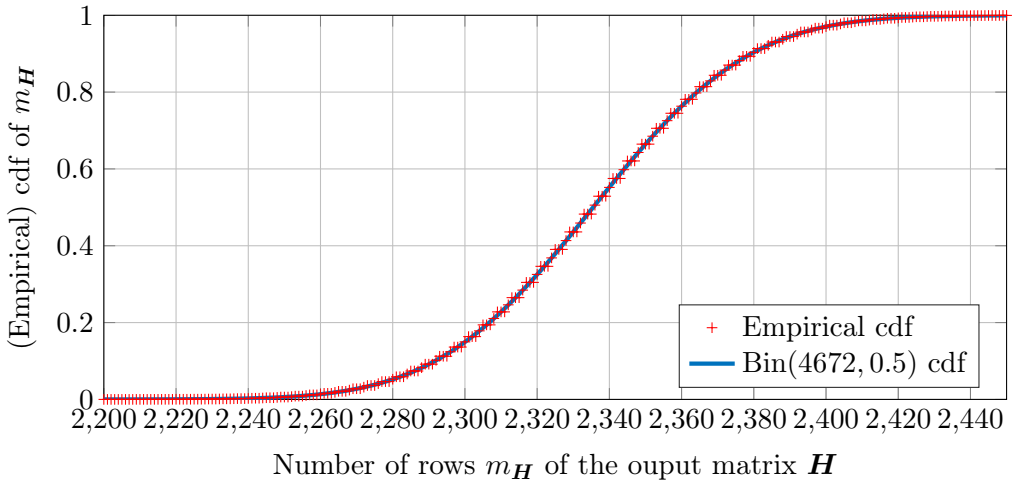


Figure 4.8: Comparison of the empirical cdf of the number of rows $m_{\boldsymbol{H}}$ of $\boldsymbol{H}$ with the cdf of a $\mathrm{Bin}(m_{\boldsymbol{H}^{(\mathrm{I})}}, 0.5)$ distributed random variable. The used sample size is 10^6, the number of rows of $\boldsymbol{H}^{(\mathrm{I})}$ is $m_{\boldsymbol{H}^{(\mathrm{I})}} = 4672$.

Table 4.2: Comparison between theoretical an estimated mean and variance for the code used in Example 4.5.

	Mean	**Variance**
Estimated	$\widehat{\mu} \approx 2335.99$	$\widehat{\sigma^2} \approx 1169.49$
Theoretical	$\mathrm{E}(m_{\boldsymbol{H}}) = \frac{m_{\boldsymbol{H}^{(\mathrm{I})}}}{2} = 2336$	$\mathrm{Var}(m_{\boldsymbol{H}}) = \frac{m_{\boldsymbol{H}^{(\mathrm{I})}}}{4} = 1168$

4.2.5 Security Considerations

In this section, we aim for assessing the security of the proposed scheme. We assume that an attacker wants to access the secret response $\boldsymbol{r}$. Since the helper data storage is public, he knows the decoding matrix $\boldsymbol{H}$. In addition, he knows that response $\boldsymbol{r}$ is a codeword of the code defined by $\boldsymbol{H}$, and hence, that there are at most $|\mathcal{C}| = 2^k$ many possibilities. These considerations suffice to obtain an upper bound of the attackers uncertainty $\mathrm{H}(\boldsymbol{r}|\mathcal{C})$. In the remainder of this section, we aim for deriving a lower bound on $\mathrm{H}(\boldsymbol{r}|\mathcal{C})$.

We still assume that an attacker, who knows the decoding matrix $\boldsymbol{H}$, wants to recover the initial response $\boldsymbol{r}$. We decompose the matrix $\boldsymbol{H}^{(\mathrm{I})}$ into two disjoint sub-matrices $\widetilde{\boldsymbol{H}}$ and $\overline{\boldsymbol{H}}$, where

$$\widetilde{\boldsymbol{H}} = \{\boldsymbol{h}_i \in \boldsymbol{H}^{(\mathrm{I})} : \langle \boldsymbol{h}_i, \boldsymbol{r} \rangle = 0\} \tag{4.14}$$

is the set of rows from $\boldsymbol{H}^{(\mathrm{I})}$ that can be used for the construction of $\boldsymbol{H}$ and

$$\overline{\boldsymbol{H}} = \{\boldsymbol{h}_i \in \boldsymbol{H}^{(\mathrm{I})} : \langle \boldsymbol{h}_i, \boldsymbol{r} \rangle \neq 0\} \tag{4.15}$$

is the set of rows that cannot be used. Note that by the construction given in Section 4.2.3, code $\mathcal{C}$ can be defined as $\mathrm{OC}\left(\widetilde{\boldsymbol{H}}\right)$, which denotes the orthogonal complement of $\widetilde{\boldsymbol{H}}$. From this observation directly follows that

$$\boldsymbol{r} \in \mathcal{C} := \mathrm{OC}\left(\widetilde{\boldsymbol{H}}\right). \tag{4.16}$$

Further, for $i = 1, \ldots, m_{\overline{\boldsymbol{H}}}$, we define the set

$$\mathcal{V}_i := \mathbb{F}_2^n \setminus \mathrm{OC}\left(\boldsymbol{h}_i\right), \text{ where } \boldsymbol{h}_i \in \overline{\boldsymbol{H}}, \tag{4.17}$$

and conclude that

$$\boldsymbol{r} \in \bigcap_{i=1}^{m_{\overline{\boldsymbol{H}}}} \mathcal{V}_i. \tag{4.18}$$

We want to stress, that (4.16) and (4.18) is exactly everything what an attacker knows. From (4.16) and (4.18) it follows that

$$\boldsymbol{r} \in \mathcal{C} \cap \left(\bigcap_{i=1}^{m_{\overline{\boldsymbol{H}}}} \mathcal{V}_i \right) =: \mathcal{S}. \tag{4.19}$$

Thus, the uncertainty of an attacker depends on the cardinality of the set $\mathcal{S}$. Due to (4.19), and since $r \in \mathcal{U}(\mathbb{F}_2^n)$, we have

$$\mathrm{H}(\boldsymbol{r}|\mathcal{C}) \geq \log_2(|\mathcal{S}|), \tag{4.20}$$

which is a lower bound on $\mathrm{H}(\boldsymbol{r}|\mathcal{C})$. Our aim is to calculate $\log_2(|\mathcal{S}|)$. For that purpose, we first consider the following lemma.

Lemma 4.8. A response $\boldsymbol{r}$ fulfills (4.16) and (4.18) if and only if

$$\boldsymbol{r}^* \in \mathrm{OC}\left(\boldsymbol{H}'\right),$$

where

$$\boldsymbol{r}^* := [\boldsymbol{r}, 1] \in \mathbb{F}_2^{n+1} \text{ and } \boldsymbol{H}' := \begin{bmatrix} \widetilde{\boldsymbol{H}} & \mathbf{0}_{m_{\widetilde{\boldsymbol{H}}} \times 1} \\ \overline{\boldsymbol{H}} & \mathbf{1}_{m_{\overline{\boldsymbol{H}}} \times 1} \end{bmatrix} \in \mathbb{F}_2^{(m_{\widetilde{\boldsymbol{H}}} + m_{\overline{\boldsymbol{H}}}) \times (n+1)}.$$

Proof. Using the condition stated in (4.16) we have

$$\begin{aligned} \boldsymbol{r} \in \mathcal{C} := \mathrm{OC}\left(\widetilde{\boldsymbol{H}}\right) &\Leftrightarrow \langle \boldsymbol{h}_i, \boldsymbol{r} \rangle = 0, \ \forall i = 1, \dots m_{\widetilde{\boldsymbol{H}}}, \boldsymbol{h}_i \in \widetilde{\boldsymbol{H}} \\ &\Leftrightarrow \langle [\boldsymbol{h}_i, \mathbf{0}_{m_{\widetilde{\boldsymbol{H}}} \times 1}], [\boldsymbol{r}, \mathbf{1}_{m_{\overline{\boldsymbol{H}}} \times 1}]^\top \rangle = 0 \ \forall i = 1, \dots m_{\widetilde{\boldsymbol{H}}}, [\boldsymbol{h}_i, 0] \in [\widetilde{\boldsymbol{H}}, \mathbf{0}_{m_{\widetilde{\boldsymbol{H}}} \times 1}] \\ &\Leftrightarrow [\boldsymbol{r}, 1] \in \mathrm{Kern}[\widetilde{\boldsymbol{H}}, \mathbf{0}_{m_{\widetilde{\boldsymbol{H}}} \times 1}]. \end{aligned}$$

With the condition given in (4.18) we obtain

$$\begin{aligned} \boldsymbol{r} \in \bigcap_{i=1}^{m_{\overline{\boldsymbol{H}}}} \mathcal{V}_i &\Leftrightarrow \langle \boldsymbol{h}_i, \boldsymbol{r} \rangle = 1, \ \forall i = 1, \dots m_{\overline{\boldsymbol{H}}}, \boldsymbol{h}_i \in \overline{\boldsymbol{H}} \\ &\Leftrightarrow \langle [\boldsymbol{h}_i, \mathbf{1}_{m_{\widetilde{\boldsymbol{H}}} \times 1}], [\boldsymbol{r}, \mathbf{1}_{m_{\overline{\boldsymbol{H}}} \times 1}]^\top \rangle = 0 \ \forall i = 1, \dots m_{\overline{\boldsymbol{H}}}, [\boldsymbol{h}_i, 1] \in [\overline{\boldsymbol{H}}, \mathbf{1}_{m_{\overline{\boldsymbol{H}}} \times 1}] \\ &\Leftrightarrow [\boldsymbol{r}, 1] \in \mathrm{Kern}[\overline{\boldsymbol{H}}, \mathbf{1}_{m_{\overline{\boldsymbol{H}}} \times 1}]. \end{aligned}$$

Note that the rows of the two matrices $[\widetilde{\boldsymbol{H}}, \mathbf{0}_{m_{\widetilde{\boldsymbol{H}}} \times 1}]$ and $[\overline{\boldsymbol{H}}, \mathbf{1}_{m_{\overline{\boldsymbol{H}}} \times 1}]$ are exactly the rows of matrix $\boldsymbol{H}'$. Thus, we can conclude that the conditions stated in (4.16) and(4.18) are fulfilled if and only if $[\boldsymbol{r}, 1] \in \mathrm{Kern}\, \boldsymbol{H}'$, and hence, $\boldsymbol{r} \in \mathrm{OC}\left(\boldsymbol{H}'\right)$. □

Using Lemma 4.8 and the rank nullity theorem from linear algebra, we can prove the following theorem which gives us a lower bound on the attacker's uncertainty.

Theorem 4.9. $\mathrm{H}(\boldsymbol{r}|\mathcal{C}) \geq n - \mathrm{rank}(\boldsymbol{H}')$

Proof. From (4.20) we know that $\mathrm{H}(\boldsymbol{r}|\mathcal{C}) \geq \log_2(|\mathcal{S}|)$, where $\mathcal{S}$ is defined as in (4.19). We investigate $\log_2(|\mathcal{S}|)$, which can be expressed by using $\boldsymbol{H}'$, i.e.,

$$\begin{aligned} \log_2(|\mathcal{S}|) &= \dim\left(\mathrm{OC}\left(\boldsymbol{H}'\right)\right) - 1 \\ &= (n+1) - \mathrm{rank}\left(\boldsymbol{H}'\right) - 1 \\ &= n - \mathrm{rank}\left(\boldsymbol{H}'\right). \end{aligned}$$

Hence, we have proven that $\mathrm{H}(\boldsymbol{r}|\mathcal{C}) \geq \log_2(|\mathcal{S}|) = n - \mathrm{rank}(\boldsymbol{H}')$. □

Our next step is to consider practical values for this lower bound. To get a sufficiently secure result, we want $\operatorname{rank}(\boldsymbol{H}')$ to be small. The following lemma provides a lower bound on the rank of $\boldsymbol{H}'$ in case of a successful code construction. Recall from Lemma 4.7, that in a successful code construction, we have $\operatorname{rank}(\boldsymbol{H}) \geq \operatorname{rank}(\boldsymbol{H}^{(\mathrm{I})}) - 1$ with high probability.

Theorem 4.10. $\operatorname{rank}(\widetilde{\boldsymbol{H}}) \geq \operatorname{rank}(\boldsymbol{H}^{(\mathrm{I})}) - 1 \Longrightarrow \operatorname{rank}(\boldsymbol{H}') \leq \operatorname{rank}(\boldsymbol{H}^{(\mathrm{I})}) + 2$

Proof. Using that $\operatorname{rank}(\widetilde{\boldsymbol{H}}) \geq \operatorname{rank}(\boldsymbol{H}^{(\mathrm{I})}) - 1$, we get that $\langle \widetilde{\boldsymbol{H}} \rangle \cap \langle \overline{\boldsymbol{H}} \rangle$ is a subspace of $\langle \overline{\boldsymbol{H}} \rangle$ of dimension

$$\dim\left(\langle \widetilde{\boldsymbol{H}} \rangle \cap \langle \overline{\boldsymbol{H}} \rangle\right) = \underbrace{\operatorname{rank}\left(\widetilde{\boldsymbol{H}}\right)}_{\geq \operatorname{rank}(\boldsymbol{H}^{(\mathrm{I})})-1} + \operatorname{rank}\left(\overline{\boldsymbol{H}}\right) - \underbrace{\dim\left(\langle \widetilde{\boldsymbol{H}} \rangle + \langle \overline{\boldsymbol{H}} \rangle\right)}_{= \operatorname{rank}(\boldsymbol{H}^{(\mathrm{I})})} \geq \operatorname{rank}\left(\overline{\boldsymbol{H}}\right) - 1.$$

Hence, the vector space $\langle \overline{\boldsymbol{H}} \rangle$ can be written as direct sum of the form

$$\langle \overline{\boldsymbol{H}} \rangle = \langle \boldsymbol{h} \rangle + \left(\langle \widetilde{\boldsymbol{H}} \rangle \cap \langle \overline{\boldsymbol{H}} \rangle\right),$$

where $\boldsymbol{h} \in \langle \overline{\boldsymbol{H}} \rangle$. Hence, all rows of the lower half of matrix $\boldsymbol{H}'$, i.e., sub-matrix $\begin{bmatrix} \overline{\boldsymbol{H}} & \mathbf{1}_{m_{\overline{\boldsymbol{H}}} \times 1} \end{bmatrix}$, are of one of the following two types:

(i) $[\boldsymbol{h} + \tilde{\boldsymbol{h}}, 1]$, where $\tilde{\boldsymbol{h}}$ is in the span of the rows of $\widetilde{\boldsymbol{H}}$.

(ii) $[\tilde{\boldsymbol{h}}, 1]$, where $\tilde{\boldsymbol{h}}$ is in the span of the rows of $\widetilde{\boldsymbol{H}}$.

We first consider rows having the form specified by type (i). If there exists at least one row $\boldsymbol{h}'_1 = [\boldsymbol{h} + \tilde{\boldsymbol{h}}', 1]$ of that type, all such rows are in the span of $\begin{bmatrix} \widetilde{\boldsymbol{H}} & \mathbf{0}_{m_{\widetilde{\boldsymbol{H}}} \times 1} \end{bmatrix}$ and $\boldsymbol{h}'_1$ since

$$[\boldsymbol{h} + \tilde{\boldsymbol{h}}, 1] = \underbrace{[\boldsymbol{h} + \tilde{\boldsymbol{h}}', 1]}_{= \boldsymbol{h}'_1} + \underbrace{[\tilde{\boldsymbol{h}} - \tilde{\boldsymbol{h}}', 0]}_{\in \langle \begin{bmatrix} \widetilde{\boldsymbol{H}} & \mathbf{0}_{m_{\widetilde{\boldsymbol{H}}} \times 1} \end{bmatrix} \rangle}.$$

A similar argument holds for rows having the form specified by type (ii), if there is at least one such row $\boldsymbol{h}'_2$. Hence, the rows of $\boldsymbol{H}'$ are spanned by the rows of the matrix $\langle \begin{bmatrix} \widetilde{\boldsymbol{H}} & \mathbf{0}_{m_{\widetilde{\boldsymbol{H}}} \times 1} \end{bmatrix} \rangle$ and $\boldsymbol{h}'_1$ and $\boldsymbol{h}'_2$ and its rank is

$$\operatorname{rank}\left(\boldsymbol{H}'\right) \leq \operatorname{rank}\left(\begin{bmatrix} \widetilde{\boldsymbol{H}} & \mathbf{0}_{m_{\widetilde{\boldsymbol{H}}} \times 1} \end{bmatrix}\right) + 2 = \operatorname{rank}\left(\widetilde{\boldsymbol{H}}\right) + 2.$$

□

Lemma 4.7, Theorem 4.9, and Theorem 4.10 give us the following corollary, which provides a lower bound on $H(\boldsymbol{r})$ in case of a successful code construction.

Corollary 4.11. $\operatorname{rank}(\widetilde{\boldsymbol{H}}) \geq \operatorname{rank}(\boldsymbol{H}^{(\mathrm{I})}) - 1 \Longrightarrow \mathrm{H}(\boldsymbol{r}|\mathcal{C}) \geq k - 2$

Proof. From Theorem 4.9 we know that $\mathrm{H}(\boldsymbol{r}|\mathcal{C}) \geq \log_2(|\mathcal{S}|) = n - \mathrm{rank}(\boldsymbol{H'})$. From Theorem 4.10 we know that $\mathrm{rank}(\boldsymbol{H'}) \leq \mathrm{rank}(\boldsymbol{H}^{(\mathrm{I})}) - 1$ if $\mathrm{rank}(\widetilde{\boldsymbol{H}}) \geq \mathrm{rank}(\boldsymbol{H}^{(\mathrm{I})}) - 1$. The latter statement is true with high probability due to Lemma 4.7. Hence, we have

$$\begin{aligned}
\mathrm{H}(\boldsymbol{r}|\mathcal{C}) \geq \log_2(|\mathcal{S}|) &= n - \underbrace{\mathrm{rank}(\boldsymbol{H'})}_{\leq \mathrm{rank}(\boldsymbol{H}^{(\mathrm{I})})+2} \\
&\underset{Thm.\ 4.10}{\geq} n - (\mathrm{rank}(\boldsymbol{H}^{(\mathrm{I})}) + 2) \\
&= n - \mathrm{rank}(\boldsymbol{H}^{(\mathrm{I})}) - 2 \\
&= n - (n-k) - 2 \\
&= k - 2.
\end{aligned}$$

□

Using Corollary 4.11 we can conclude that in most cases the uncertainty of $\boldsymbol{r}$ for an attacker can be lower-bounded by $k - 2$. To summarize, in this section we have shown, that

$$n - \mathrm{rank}(\boldsymbol{H'}) \leq \mathrm{H}(\boldsymbol{r}|\mathcal{C}) \leq k,$$

which is

$$k - 2 \leq \mathrm{H}(\boldsymbol{r}|\mathcal{C}) \leq k$$

with high probability in a practical scenario. In comparison, the entropy loss when using the code-offset construction is $n - k$. If we have $\mathrm{H}(\boldsymbol{r}) = n$, it is

$$\mathrm{H}(\boldsymbol{r}|\boldsymbol{h}) = \mathrm{H}(\boldsymbol{r}) - (n-k) = k.$$

Thus, concerning entropy loss, the two schemes are equivalent up to a small constant.

4.2.6 Results

We summarize results, which have been obtained by constructing and simulating LDPC codes, according to the construction method introduced and discussed in the preceding sections.

First, recall that we claimed in Section 4.2.3, that deleting rows from a decoding matrix worsens the error-correction behavior. This is confirmed by Figure 4.9, showing the results of simulations, which visualize the impact on the block error probability, when deleting rows from a decoding matrix for several bit error probabilities (compare "full EG", which is the EG(2,8) LDPC code based on euclidean geometry and constructed by using the parameters given in [Bos13, Section 5.5.2], and "selected EG", which only uses the rows that are orthogonal to a given PUF response $\boldsymbol{r}$). Adding more rows to the selected parity-check equations reduces the block error probability, and hence, improves the error correction capability. All simulation results that are shown in that graph are gained by using Algorithm 5 for hard-decision decoding.

Figure 4.10 visualizes the block error probability of three LDPC codes of length 128, which differ in their dimension. We simulated up to 15 observed block errors, for each we chose a simulation size of 10^6 samples. For those results, only hard-decision decoding was used. As

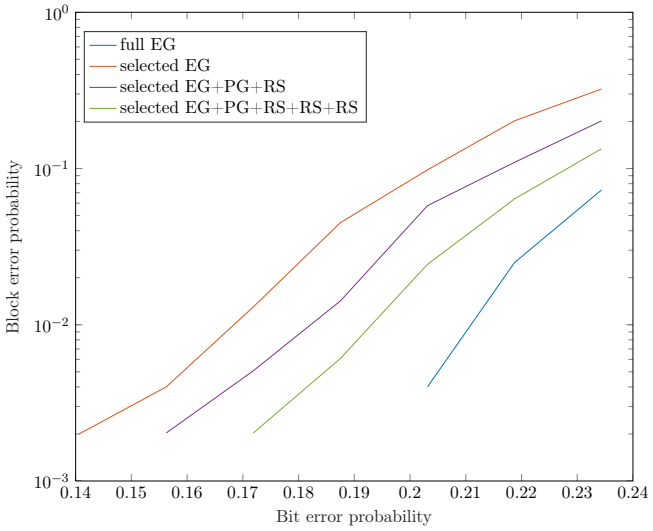


Figure 4.9: Error correction capabilities of LDPC codes with decoding matrices of different cardinalities. "Full EG" denotes the EG(2,8) LDPC code of length 64, based on Euclidean geometry. "Selected EG" only contains those rows, that are orthogonal to a given PUF response $\boldsymbol{r}$. The other plots represent extensions of the "selected EG" decoding matrix, gained by adding rows that are orthogonal to $\boldsymbol{r}$, generated based on Reed–Solomon codes and projective geometry.

can be seen in the figure, the performance of the code does not solely depend on the amount of redundancy, but on the chosen construction methods, used to generate the corresponding decoding matrices. Insights on which construction methods to use and how to combine parity-check equations in order to obtain codes with a good performance is an open problem. In the following, we list the construction methods that were used for constructing the three LDPC codes considered in Figure 4.10:

- $k = 8$
  - Gallager construction [Gal63]
  - Array structure construction [Fan01]
  - Advanced array structure construction [EO02]
- $k = 28$
  - Gallager construction [Gal63]
  - Array structure construction [Fan01]
  - Advanced array structure construction [EO02]
  - Two LDPC codes, constructed based on Reed–Solomon codes [DXAGL03], Table 4.3 summarizes the parameters that were used to generate the two codes
- $k = 63$
  - Gallager [Gal63]

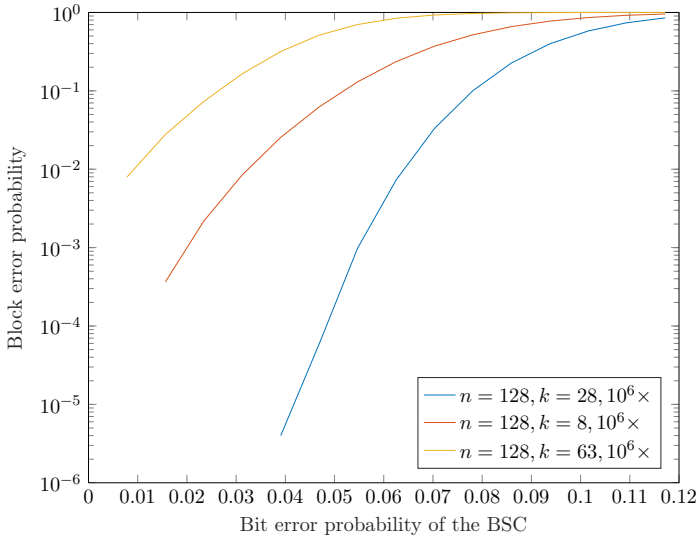


Figure 4.10: Comparison of LDPC codes of length 128 with three different dimensions, constructed by using selected rows as proposed in this chapter. These plots visualize the block error probability of hard-decision decoding with Algorithm 5.

– Array structure construction [Fan01]

Table 4.3: Parameters of the Reed–Solomon based LDPC codes of length 64. The naming of the parameters follows [DXAGL03], the work, which introduced the construction method.

Parameter	RS1	RS2
p	2	2
r	4	4
ρ	4	4
γ	6	8
#rows	96	128
rank($\boldsymbol{H}$)	48	49

Figure 4.11 compares two LDPC codes, which were constructed as detailed in this chapter and applied to a common PUF scenario, which uses BCH codes by following the standard hard-decision decoding paradigm. The block error probability of two exemplary BCH codes in that scenario is visualized in the plots. To outperform the BCH codes with good LDPC codes, it is required to use soft information gained from multiple independent readouts. Under the latter premise, the block error probabilities of our LDPC codes can be lower in comparison to the BCH scenario. The codes used here are constructed by using the methods based on Reed–Solomon codes, Euclidean and projective geometries. Using only hard-decision decoding for our LDPC codes results in a weak performance compared to BCH codes.

However, the performance of the constructed LDPC codes depends on the construction methods used for creating the corresponding decoding matrices $\boldsymbol{H}$. As can be seen in Figure 4.12, which visualizes the performance of LDPC codes generated by combining several

Irregular Repeat Accumulate Codes as proposed in [JKM00] and implemented in [Ste19, Appendix A.1], the constructed codes can show a worse performance than BCH codes with similar parameters. The identification of requirements, which have to be fulfilled by the constructed LDPC codes in order to guarantee a certain error-correction performance and to outperform BCH codes, is still an open problem.

4.2.7 Discussion

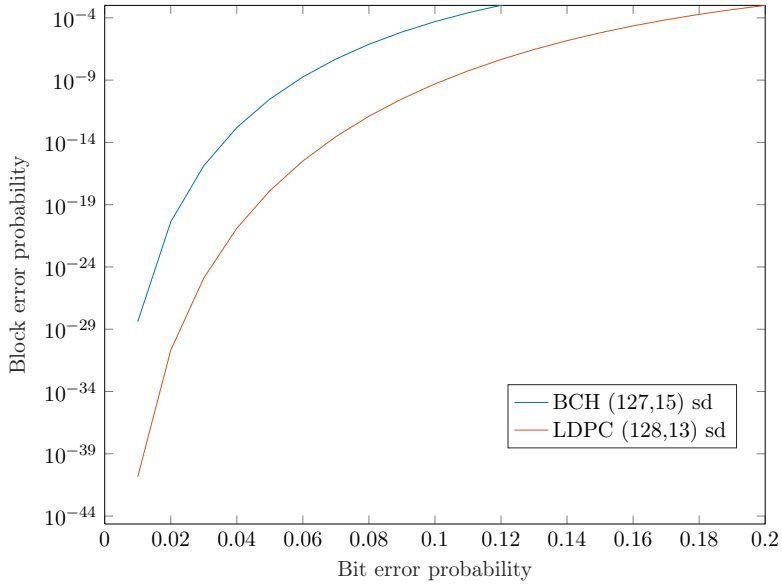
Before listing the advantages of the scheme introduced in this chapter, we want to emphasize again, that it is not suitable for all kinds of PUF applications. The scheme is not practical to scenarios which use a challenge-response system, like for example authentication, since the scheme would require to construct an error-correcting code for each possible challenge-response pair. However, there are many applications that use only a single challenge, for example identification for counterfeit prevention or Physical Obfuscated Keys (POKs). In applications where this requirement is met, the scheme possesses the following benefits: First, there exist side-channel attacks which aim on the helper data. Hence, a method that does not use helper data complicates those attacks. For example, the side-channel analysis given in [MSSS11, Section 5.2] cannot be used in order to attack this scheme. Second, since no pre-processing unit is needed, complexity as well as chip area is reduced in the reproduction phase. In comparison to other methods, the scheme is a conceptual new method for implementing a secure sketch for PUFs. In addition, it rises the interesting theoretical question for methods to construct codes, such that a given vector is one of its codewords (cf. Problem 4.2). To solve this problem for other code classes than LDPC codes is an open problem.

4.3 Soft-Decision Secure Sketches for ROPUFs

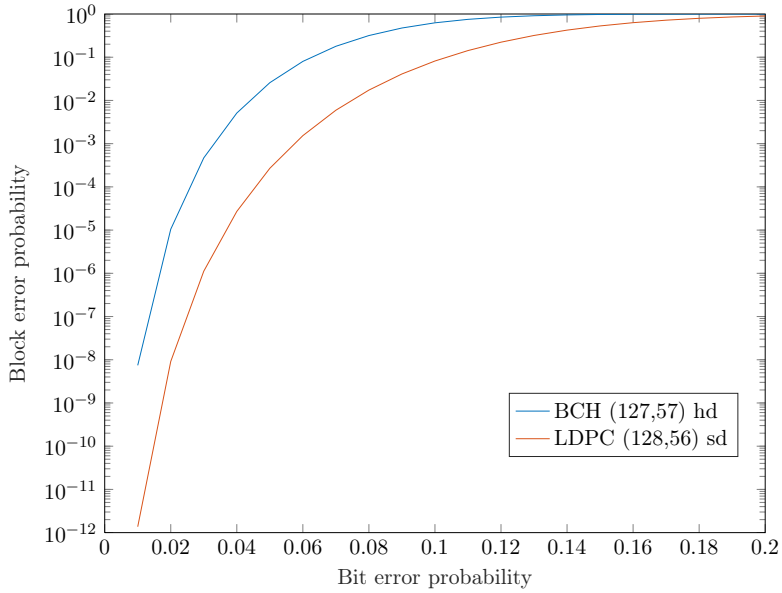
While the secure sketch in Chapter 4.2 is proposed independently of the underlying PUF construction, this section deals with soft-decision secure sketches for ROPUFs that are based on the code-offset construction. In addition to the aforementioned helper data, soft-decision secure sketches need reliability information. In general, using soft information results in a better error-correcting performance or a higher code rate. In the case of secure sketches, this behavior leads to a smaller min-entropy loss and is hence a promising technique to be used for error correction in the context of PUFs. Note that disjoint pairs of ring oscillators have to be used for generating PUF responses in both secure sketches discussed in the following. Recall that preliminaries on soft-decision decoding can be found in Section 2.2.4.

4.3.1 A Soft-Decision Secure Sketch for ROPUFs based on the Binary Symmetric Channel

In this section, we adapt a soft-decision secure sketch, which was proposed for SRAM PUFs in [MTV09a], to ROPUFs. The underlying channel model is a BSC with individual bit error probabilities p_{e_i} for all response positions r_i, where $i = 1, \ldots, n$. We again refer to Figure 3.2 for a visualization of that channel model. Let the vector $\boldsymbol{p}_e = (p_{e_1}, \ldots, p_{e_n})$ denote the bit error probabilities of the response bits $r_1, \ldots, r_n$. In order to use these reliability information while executing the secure sketch, it has to be assumed that $\boldsymbol{p}_e$ is known and available via the public helper data storage. In a practical scenario, $\boldsymbol{p}_e$ can be obtained during the initialization

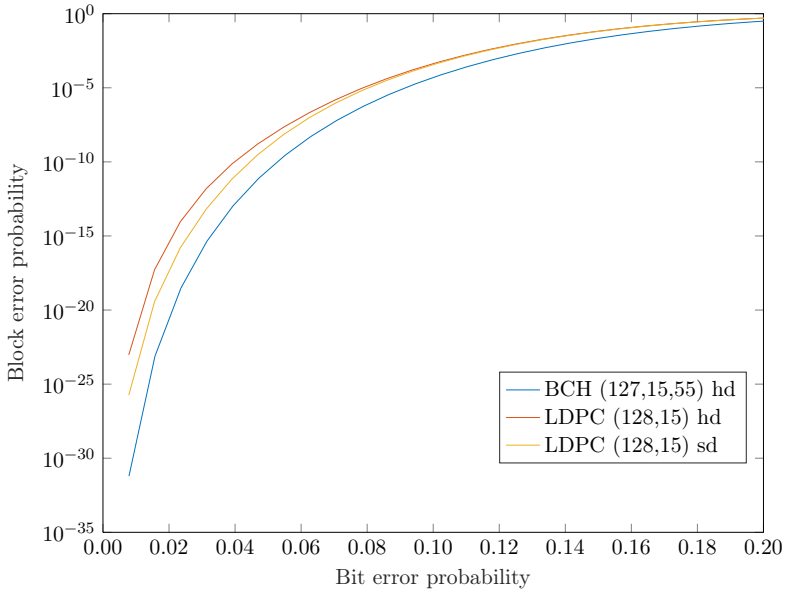


(a) LDPC code of length $n = 128$ and dimension $k = 13$, constructed by using the constructions based on Reed–Solomon codes and finitie geometries, compared to a hard-decision BCH scenario.

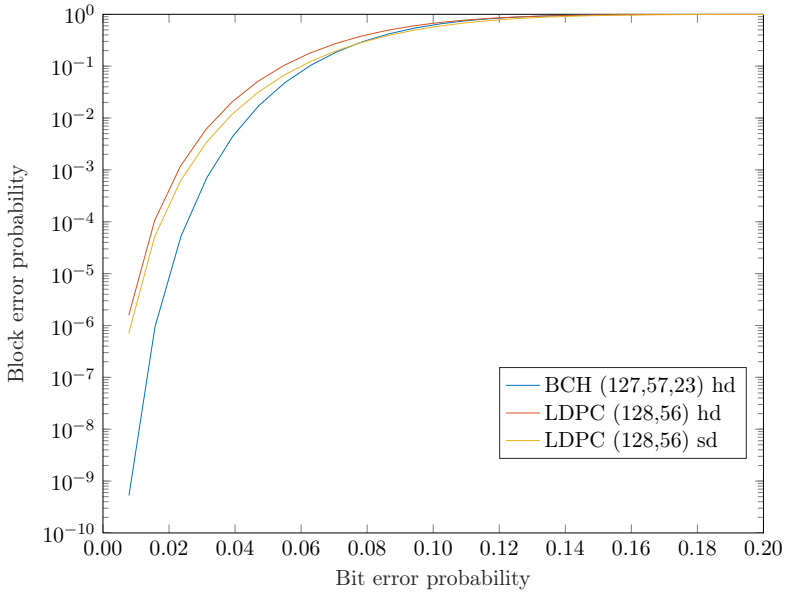


(b) LDPC Code of length $n = 128$ and dimension $k = 56$, constructed by using the constructions based on Reed–Solomon codes and finite geometries, compared to a hard-decision BCH scenario.

Figure 4.11: Comparison of two PUF scenarios. The BCH codes are used with hard-decision decoding as classically done in the PUF context. To result in a better block error probability with the codes constructed in this chapter, soft information, that can be generated from multiple independent response extractions, have to be used.



(a) LDPC code of length $n = 128$ and dimension $k = 15$, constructed by combining parity-check equations of irregular repeat accumulate codes, compared to a hard decoded BCH code.



(b) LDPC code of length $n = 128$ and dimension $k = 56$, constructed by combining parity-check equations of irregular repeat accumulate codes, compared to a hard decoded BCH code.

Figure 4.12: Comparison of two PUF scenarios. The BCH codes are used with hard-decision decoding as classically done in the PUF context. We compare to LDPC codes with similar parameters for both, hard and soft-decision decoding.

phase, by extracting responses several times and analyzing the extracted bit sequences. For example, a value p_{e_i} can be determined by the relative frequency of errors at position i, when comparing all extracted responses. The more responses are extracted, the more precise the reliability information. This implies a higher complexity, however, it has to be emphasized that this is not dramatic, since it has to be done only once during initialization. Alternatively, theoretical models can be applied in order to estimate $\boldsymbol{p}_e$. For this purpose, we can use the channel model derived for ROPUFs in Chapter 3.2.

Recall from Equation 3.8 that

$$p_{\mathrm{e}_i} = P(r_i^{(t)} \neq r_i) = \min\{p_{\mathrm{r}_i}, 1 - p_{\mathrm{r}_i}\} = \begin{cases} p_{\mathrm{r}_i}, & \text{if } r_i = 0 \\ 1 - p_{\mathrm{r}_i}, & \text{if } r_i = 1. \end{cases} \tag{4.21}$$

Hence, it follows from Equation 3.19 that

$$p_{\mathrm{e}_i} = \begin{cases} \Phi\left(\frac{f_i - f_{i+1}}{\sqrt{2}\sigma_{\mathrm{NOISE}}}\right), & \text{if } r_i = 0 \\ \Phi\left(\frac{f_{i+1} - f_i}{\sqrt{2}\sigma_{\mathrm{NOISE}}}\right), & \text{if } r_i = 1. \end{cases} \tag{4.22}$$

For each response bit r_i $(i = 1, \ldots, n)$, the soft information

$$s_i^{(t)} = (-1)^{h_i \oplus r_i^{(t)}} \cdot (\log(1 - p_{\mathrm{e}_i}) - \log(p_{\mathrm{e}_i})) \tag{4.23}$$

is calculated as proposed by [MTV09a, Chapter 5]. The secure sketch proposed in this section is exactly the code-offset construction, the only difference is that the values calculated in (4.23) are used as input to the decoder. Having $\boldsymbol{p}_e$ as public helper data does not lead to any restrictions concerning security, if we ensure, that no ring oscillator is used twice for generating a response.

Theorem 4.12. Revealing $\boldsymbol{p}_e$ does not induce any additional min-entropy loss compared to hard-decision secure sketches.

Proof. The proof can be adapted in a straightforward manner from [MTV09a, Theorem 1], where the theorem was proven in the SRAM scenario. □

4.3.2 A New Soft-Decision Secure Sketch for ROPUFs based on the AWGN Channel

Due to the normal distribution of ring oscillator frequencies, the AWGN channel is a suitable channel model when working with ROPUFs. For brevity, in this section we write f_{min} and f_{max} to denote the minimum and the maximum of the frequencies f_i and f_{i+1}, i.e.,

$$f_{min} = \min\{f_i, f_{i+1}\} \text{ and } f_{max} = \max\{f_i, f_{i+1}\}. \tag{4.24}$$

Let

$$f_i^{(t)} \sim \mathcal{N}(f_i, \sigma^2_{\mathrm{NOISE}}) \text{ and } f_{i+1}^{(t)} \sim \mathcal{N}(f_{i+1}, \sigma^2_{\mathrm{NOISE}}) \tag{4.25}$$

be the measurements of ring oscillator i and ring oscillator $i+1$ at time t. In the following, we derive a formula for calculating channel L-values as explained in Section 2.2.4. It is

$$\begin{aligned} L(r_i^{(t)}|r_i) &= \log_e\left(\frac{\mathrm{P}(r_i^{(t)}|r_i=0)}{\mathrm{P}(r_i^{(t)}|r_i=1)}\right) \\ &= \log_e\left(\frac{p(f_i^{(t)}, f_{i+1}^{(t)}|f_i=f_{min}, f_{i+1}=f_{max})}{p(f_i^{(t)}, f_{i+1}^{(t)}|f_i=f_{max}, f_{i+1}=f_{min})}\right), \end{aligned}$$

where $p(\cdot)$ denotes the probability density function of the normal distribution. Since the frequencies f_i and f_{i+1} are independent, we get

$$L(r_i^{(t)}|r_i) = \log_e\left(\frac{p(f_i^{(t)}|f_i=f_{min})\cdot p(f_{i+1}^{(t)}|f_{i+1}=f_{max})}{p(f_i^{(t)}|f_i=f_{max})\cdot p(f_{i+1}^{(t)}|f_{i+1}=f_{min})}\right).$$

Using the standard calculation rules for logarithms, this equality can be transformed to

$$\begin{aligned} L(r_i^{(t)}|r_i) &= -\frac{(f_i^{(t)}-f_{min})^2}{2\sigma^2_{\mathrm{NOISE}}} - \frac{(f_{i+1}^{(t)}-f_{max})^2}{2\sigma^2_{\mathrm{NOISE}}} + \frac{(f_i^{(t)}-f_{max})^2}{2\sigma^2_{\mathrm{NOISE}}} + \frac{(f_{i+1}^{(t)}-f_{min})^2}{2\sigma^2_{\mathrm{NOISE}}} \\ &= \frac{f_{max}-f_{min}}{\sigma^2_{\mathrm{NOISE}}}\cdot\left(f_{i+1}^{(t)}-f_i^{(t)}\right) \\ &= L_{\mathrm{ch}}\cdot y_i, \end{aligned} \tag{4.26}$$

where $L_{\mathrm{ch}} = \frac{|f_{i+1}-f_i|}{\sigma^2_{\mathrm{NOISE}}}$ and $y_i = f_{i+1}^{(t)} - f_i^{(t)}$.

Algorithm 8 summarizes the initialization phase of the AWGN soft-decision secure sketch for ROPUFs. Figure 4.13 provides a visualization of the quantities used in the description of the algorithm. First, N measurements of all the ξ ring oscillators on a device are obtained. These N measurements are denoted by the vectors $\boldsymbol{F}_1, \ldots \boldsymbol{F}_N$, where

$$\boldsymbol{F}_j = (f_1^{(j)}, \ldots, f_\xi^{(j)})$$

lists the frequencies measured for the ξ ROs during measurement $j \in \{1, \ldots, N\}$ (cf. line 1 in Algorithm 8 and columns in Figure 4.13). These measurements are used in order to calculate the average frequencies of the ξ ROs as well as σ_{NOISE}, the standard deviation of the noise (cf. line 2 and rows in Figure 4.13). The reference response bits r_i are derived for all $1 \le i \le n$ based on the average frequencies of the ring oscillators (cf. line 3). The helper data vector $\boldsymbol{h}$ is produced according to the code-offset scheme (cf. line 4, where $\boldsymbol{r}$ denotes the initial response). Also, the channel L-values are calculated according to (4.26) in line 5.

The reproduction phase is outlined in Algorithm 9 and is the exact same than for the soft-decision secure sketch proposed in Section 4.3.1, i.e., the reproduction phase of the code-offset construction modified by using the L-values as input to the decoding algorithm.

4.3.3 Comparison of Soft-Decision and Hard-Decision Secure Sketches

In this section, we compare the soft-decision secure sketches proposed in Section 4.3.1 (denoted as "SD 1") and Section 4.3.2 (denoted as "SD 2"). To simulate the algorithm, we use

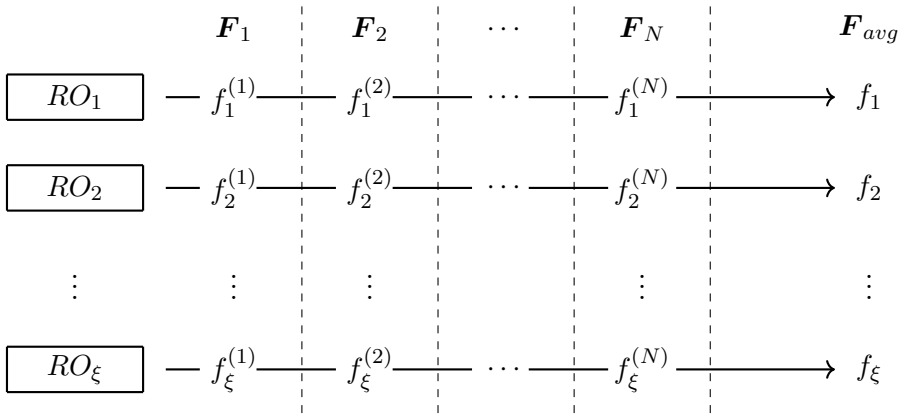


Figure 4.13: Frequencies measured in the initialization phase of the secure sketch: All ξ ring oscillators (ROs) are evaluated N times, obtaining the N measurement vectors $\boldsymbol{F}_1, \boldsymbol{F}_2, \ldots, \boldsymbol{F}_N$. These measurements are used to calculate the average frequencies $f_1, f_2, \ldots, f_\xi$ and the standard deviation σ_{NOISE}, which are in turn used to generate the initial response and the channel L-values (cf. Algorithm 8).

Algorithm 8: AWGN soft information secure sketch for ROPUFs: Initialization phase

Input: –

Output: Helper data $\boldsymbol{h}$ and L_{ch}

1 Perform N measurements of the ξ ring oscillators
$\boldsymbol{F}_1 = (f_1^{(1)}, \ldots, f_\xi^{(1)}), \ldots, \boldsymbol{F}_N = (f_1^{(N)}, \ldots, f_\xi^{(N)})$

2 Use $\boldsymbol{F}_1, \ldots, \boldsymbol{F}_N$ to calculate estimations of average frequencies f_ι $(1 \leq \iota \leq \xi)$ and standard deviation σ_{NOISE}

3 Use values f_ι $(1 \leq \iota \leq \xi)$ to derive the reference response $\boldsymbol{r} = (r_1, \ldots, r_n)$

4 $\boldsymbol{h} = \boldsymbol{r} \oplus \boldsymbol{c}$

5 $L_{\text{ch}} = \frac{|f_{\iota+1} - f_\iota|}{\sigma^2_{\text{NOISE}}}$

6 **return** $\boldsymbol{h}, L_{\text{ch}}$

convolutional codes for error correction[8]. Note that in this context, it is sufficient to consider the decoder as black box. In this chapter it is insignificant to know, how convolutional codes work[9]. We only need the intuition, that increasing a certain parameter μ, which is called constraint length, increases the error correction capability, but also increases the decoding complexity when using Viterbi algorithm, the most popular decoding algorithm for convolutional codes [Vit67]. We will again find that behavior in Chapter 5.2. To generate a response, the 512 ring oscillators on a device are pairwise compared (we emphasize again that disjoint pairs have to be used), resulting in a response of length 256. Note that this length is much shorter than the response length usually used in the literature. However, this is due to the available ROPUF data, which do not allow to extract more response bits, as long as each ring

[8] As it will turn out in Chapter 5.2.4, convolutional codes are a good choice when dealing with the ROPUFs based on [MCMS10].

[9] Since convolutional codes are applied when considering how to implement the error correction unit in Chapter 5, an introduction will be provided there.

Algorithm 9: AWGN soft information secure sketch for ROPUFs: Reproduction phase

Input: Helper data $\boldsymbol{h}$ and L_{ch}
Output: Response $\hat{\boldsymbol{r}}$

Obtain frequency measurements $\boldsymbol{F}' = (f'_1, \ldots, f'_\xi)$
Use $\boldsymbol{F}'$ to derive response $\boldsymbol{r}^{(t)}$
$\boldsymbol{y} = \boldsymbol{r}^{(t)} \oplus \boldsymbol{h}$
$\hat{\boldsymbol{c}} = \text{Decode}(\boldsymbol{y}, L_{\text{ch}})$
$\hat{\boldsymbol{r}} = \hat{\boldsymbol{c}} \oplus \boldsymbol{h}$
return $\hat{\boldsymbol{r}}$

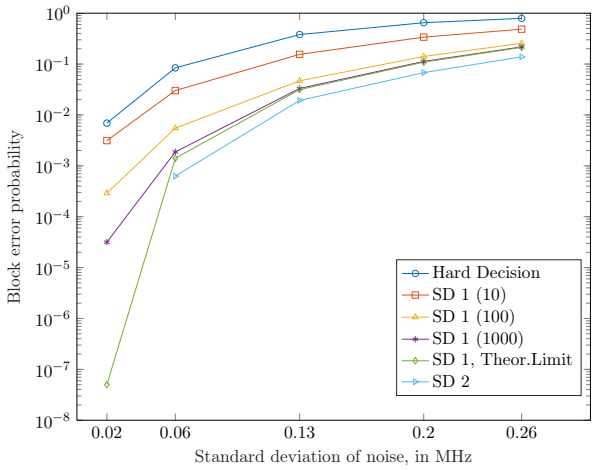
oscillator is only used in one comparison. Figure 4.14 presents simulation results of rate $\frac{1}{2}$ convolutional codes, that differ regarding their constraint lengths ($\mu = 1, 2, 4, 5, 6$ are used).

Each plot in Figure 4.14 is generated by reproducing the reference response $8 \cdot 10^7$ times, for three different secure sketches, namely hard-decision code-offset, "SD1" and "SD2". The plots show the corresponding block error rates (BERs) for different noise levels. According to the results of analyzing the given data set (cf. Appendix A, Figure A.3), the noise standard deviations $\sigma_{\text{NOISE}} \in \{0.0175, 0.0575, 0.1301\}$ are chosen. The noise standard deviations $\sigma_{\text{NOISE}} \in \{0.2026, 0.2598\}$ are arbitrarily chosen in addition.

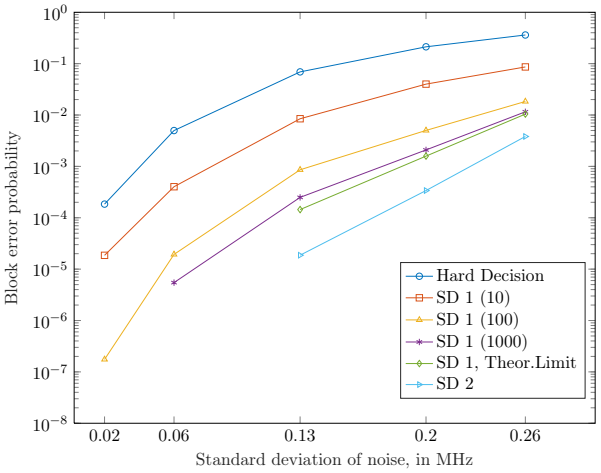
The following secure sketches are applied: The curves labeled with "hard decision" are generated by using the standard code-offset construction without any modifications. "SD 1" labels the curves generated by the soft-decision secure sketch proposed in Section 4.3.1. This algorithm is applied for 10, 100 and 1000 readouts taken during initialization for estimating reliability information. The frequency samples that are used to derive the responses are obtained by using the model proposed in Chapter 3.2. A fourth plot for "SD 1", indicating the theoretic limit according to (4.22), was generated. "SD 2" denotes the curve gained by using the soft-decision secure sketch suggested in Section 4.3.2.

The plots in Figure 4.14 enable the following observations: First, as expected, using the code-offset construction without soft information results in the weakest performance. Second, comparing the different versions of "SD 1" implies, that the number of measurements taken during initialization significantly influences the quality of the reliability information estimated based on these measurements. We observe that the "SD 1" plots converge towards the theoretical limit with an increasing number of readouts during initialization. Third, we find that "SD 2" outperforms the other algorithms.

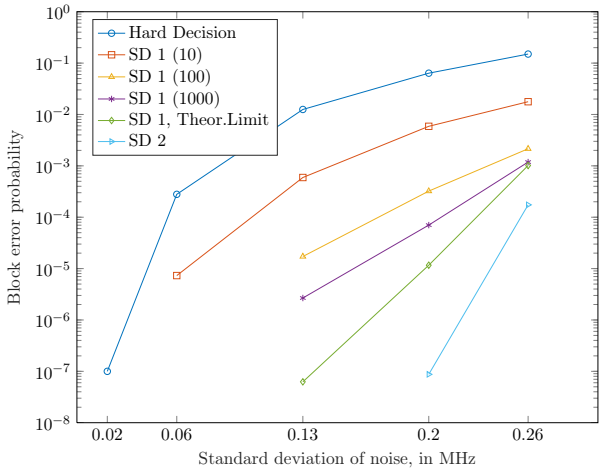
For an interpretation of the results, we have to recall from Section 2.3.2, that usually a block error rate $\leq 10^{-6}$ is required when considering FPGA-based implementations. Due to the analysis of the data set, the most realistic scenario of noise is $\sigma_{\text{NOISE}} = 0.0575$. For this noise level, all codes considered in this study do not reach the required block error rate, while applying the hard-decision code-offset construction. Using "SD 1", constraint length $\mu \geq 4$ should be used for an application with 100 or more extracted readouts during initialization (cf. Figure 4.14c). Increasing the constraint length to $\mu = 6$ allows to decrease the number of initial readouts to ten. Note that when using $\mu = 6$ all soft-decision variants fulfill the requirements, also for a higher amount of noise (cf. Figure 4.14e). The theoretical limit for "SD 1" works also when using $\mu = 1$. Using secure sketch "SD 2", constraint length $\mu \geq 2$ already fulfills the requirement (cf. Figure 4.14b).



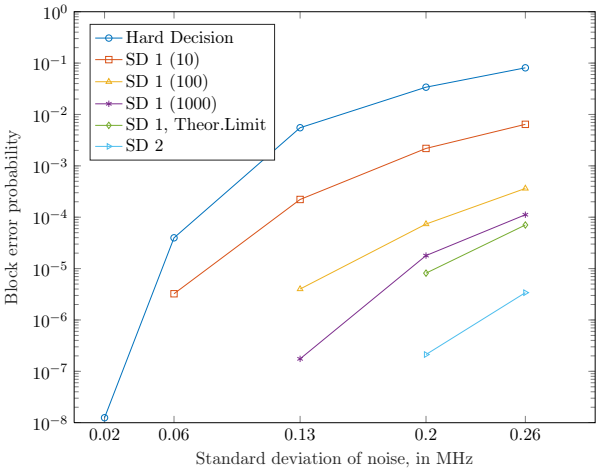
(a) $(2, 1, [1])$ convolutional code.



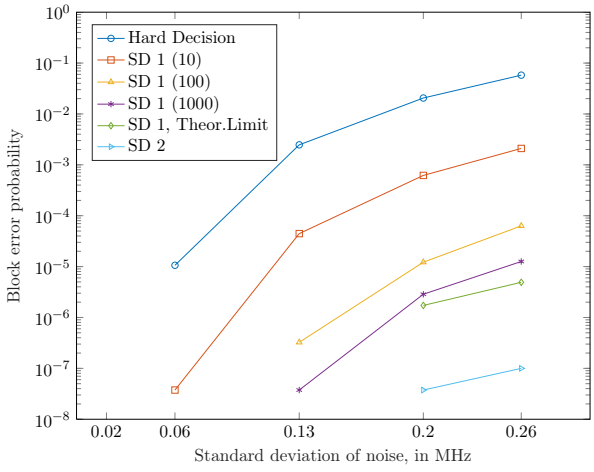
(b) $(2, 1, [2])$ convolutional code.



(c) $(2, 1, [4])$ convolutional code.



(d) $(2, 1, [5])$ convolutional code.



(e) $(2, 1, [6])$ convolutional code.

Figure 4.14: Comparison of different decoding techniques using a $(2, 1, [\mu])$ convolutional code for a constraint length $\mu \in \{1, 2, 4, 5, 6\}$ (simulation size: $80 \cdot 10^6$).

4.4 Concluding Remarks

In this chapter, we focused on secure sketches. After revisiting secure sketches from literature in Chapter 4.1, we introduced a secure sketch that only needs an error-correcting code, but no further helper data, in Chapter 4.2. The presented scheme suffers from the fact that a code has to be constructed for every PUF response that needs to be reproduced during operation. Hence, the scheme cannot be used for all applications. In cases where the scheme can be used, benefits are the prevention of attacks that aim on the helper data and the elimination of the preprocessing unit of the secure sketch. It is still an open problem, how to construct and combine parity-check equations such that the resulting LDPC code shows an adequate performance. Furthermore, the construction raises interesting theoretical questions for further research, like the constructability of error-correcting codes that contain a given vector as one of its codewords.

The choose length (CL) debiasing scheme for PUFs based on DRAM, proposed in Chapter 3.3.1, reveals a further approach to solve the problem of having a PUF response, which is a codeword of an error-correcting code. The response in CL debiasing can be arbitrarily chosen, and hence, can be selected as a codeword of a code $\mathcal{C}$. For security reasons, it is required, that $\mathcal{C}$ has a sufficiently large cardinality. Also, depending on the channel model, a suitable number of errors must be correctable. Finally, to preserve the uniqueness property, the codewords of $\mathcal{C}$ need to have a sufficiently large distance to each other in the applied metric.

Chapter 4.3 was dealing with soft-decision secure sketches for ROPUFs. Using soft information is a powerful concept in coding theory and was not applied to PUFs for a long time, most probably because it is not well-known within the hardware security community. One of the secure sketches proposed in this chapter is based on an AWGN channel, which conveniently models the normal distribution of the ring oscillator frequencies. Our results show that soft-decision secure sketches significantly improve the reliability when reconstructing PUF responses, in comparison to classical hard-decision secure sketches. The AWGN secure sketch considerably outperforms the other techniques. Using convolutional codes, this observation implicates the applicability of codes with a shorter constraint length and hence, with a lower decoding complexity. In addition to the results in this chapter, the supremacy of soft information will further be exploited in Chapter 5.2 when using convolutional codes in order to provide error correction for PUFs.

5

Error Correction for Physical Unclonable Functions

While Chapter 4 was dealing with secure sketches, this chapter focuses on a specific part of those algorithms, namely the error-correction component. The goal is the improvement of previous solutions that were suggested by the PUF community. We apply concepts from coding theory, which have not been used for error correction in the PUF scenario so far.

Before presenting specific code constructions, we discuss the general aims of error correction schemes designed for PUFs. Since PUFs usually occur within embedded systems, error correction has to be implemented in hardware. Hence, area restrictions have to be taken into consideration when developing a code construction. Especially, with regards to decoding, the reduction of chip area needs to be considered and is more important than time constraints in most application scenarios. Since PUF responses are assumed to be binary (possibly after an appropriate transformation), the final code construction has to be a binary code. The code parameters, i.e. length, dimension and minimum distance, have to be derived from the requirements of the application that the PUF is supposed to support.

When choosing the codeword length n, it has to be considered, that n bits have to be extracted from the PUF, what might consume much time and area in case of large n (recall from Figure 4.2 in Chapter 4, that the code-offset scheme needs response and codeword to have the same length). The dimension of the code has to be greater or equal to the desired key length, since the secret is derived from the information encoded into the codeword. Note that in case of a source with small entropy, a larger dimension can be used in combination with a hash function in order to achieve keys with good cryptographic properties. The required minimum distance can be derived by studying the channel model. Finally, the block error probability P_{err} states the probability that a response is reproduced incorrectly and has to be smaller than a threshold that is given by the underlying hardware platform (compare Chapter 2, Section 2.3.2). In contrast to other applications, the dimension k needs to be large enough to prevent a significant entropy loss. For example, using the code-offset construction, we have $\mathrm{H}(\boldsymbol{r}|\boldsymbol{h}) = \mathrm{H}(\boldsymbol{r}) - (n-k)$. If $\mathrm{H}(\boldsymbol{r}) < n$, k needs to be larger in order to obtain a certain security level.

This chapter is divided into two sections: First, in Section 5.1, block codes are used for error correction in the PUF scenario. The used code classes are briefly described in Section 5.1.1. In total, four code constructions are proposed. Two of them, $\mathcal{C}_1$ and $\mathcal{C}_4$, are generalized concatenated codes using Reed–Muller codes (cf. Section 5.1.2) and Reed–Solomon codes (cf. Section 5.1.3), respectively. The other two code constructions, $\mathcal{C}_2$ and $\mathcal{C}_3$, are ordinary

concatenated codes based on Reed–Solomon and Reed–Muller codes. After presenting the constructions using block codes, in Section 5.2 convolutional codes are applied. The possibility of efficient hardware implementations and good error correction properties qualifies them for being used for PUFs. Parts of this chapter have been published: The generalized concatenated code construction based on Reed–Muller codes was published in [MPB+14]. Implementations in software and hardware were realized in a diploma thesis [Kür14], a part of the results thereof was published in [HKS+15]. Furthermore, this construction has later been used in a master's thesis [Man18] for the error correction component in the design and development of a completely modularized PUF coding chain. The results thereof were published in [MHK+19]. The ordinary and generalized code constructions based on Reed–Solomon codes were published in [PMB+15]. Results that use methods from the field of convolutional codes (cf. Section 5.2) are contained in [MB17b, MPB18b].

5.1 Block Codes for PUFs

This section deals with applying block codes for error correction in PUFs. Reed–Muller and Reed–Solomon codes are used as components in ordinary as well as in generalized concatenated codes. While constructions based on ordinary concatenated codes already exist for error correction in PUFs (e.g. [BGS+08, MVHV12]), generalized concatenated codes are applied for the first time in this context.

We summarize our aims for the code construction proposed in Chapter 5.1. The underlying basis for our constructions is [MVHV12], which proposes a reference implementation of a "cryptographic key generator based on a PUF" and is a work that is very well known in the PUF community. A key of length 128 is generated, using an ordinary concatenated code of length 2226, based on a BCH code and a repetition code. A block error probability in the order of 10^{-9} was achieved. With our code constructions, we aim for several things: In order to be comparable with [MVHV12], we assume a binary symmetric channel with bit error probability $p_\mathrm{b} = 0.14$ and aim for generating a 128 bit key. Hence, our code constructions need to have a dimension ≥ 128. Furthermore, we aim for a code length less than 2226 and a block error probability less than 10^{-9}. All code constructions proposed in this section significantly outperform the construction suggested in [MVHV12]. An overview from the literature, which summarizes other code constructions that were applied for PUFs, is provided in Appendix B, Table B.1.

5.1.1 Classes of Block Codes

This section briefly introduces the code classes, which are used to construct the codes $\mathcal{C}_1$ – $\mathcal{C}_4$. For details about the codes, references to the literature are provided.

Reed–Muller Codes

Reed–Muller codes, introduced in [Mul54, Ree54], are a widely used class of binary linear block codes. For example, they were applied in the context of several NASA space missions[1]. An

[1]The $\mathcal{RM}(1,5)$ code (cf. Figure 5.1a) was used in the Mariner 6, 7 and 9 missions, when photographs from the Mars were sent from the Mariner space probes to earth [Mas92].

intuitive method to define Reed–Muller codes recursively is the *Plotkin construction* [Plo60][2].

Definition 5.1. Let $\mathcal{C}_u(n, k_u, d_u)$ and $\mathcal{C}_v(n, k_v, d_v)$ be linear codes. The code $\mathcal{C}$ is defined as

$$\mathcal{C} = \{(\boldsymbol{u}|\boldsymbol{u}+\boldsymbol{v}) : \boldsymbol{u} \in \mathcal{C}_u, \boldsymbol{v} \in \mathcal{C}_v\}, \tag{5.1}$$

where $\boldsymbol{u}+\boldsymbol{v}$ denotes component–wise addition and the symbol "|" denotes concatenation. In literature, this construction is called *Plotkin construction* or $(\boldsymbol{u}|\boldsymbol{u}+\boldsymbol{v})$ *construction.*

Theorem 5.2. An error-correcting code $\mathcal{C}$, that is constructed according to Definition 5.1, is a $(2n, k_u + k_v, \min\{2d_u, d_v\})$ code.

Proof. For a proof, we refer to [Bos99, Chapter 5.2]. □

Next, Definition 5.1 is used to define Reed–Muller codes, by specifying which codes are chosen as $\mathcal{C}_u$ and $\mathcal{C}_v$.

Definition 5.3. Let $0 \leq r < m$. A *Reed–Muller code* $\mathcal{RM}(r, m)$ of order r can be generated by applying the Plotkin construction from Definition 5.1 to the codes visualized in Figure 5.1a. To construct $\mathcal{RM}(r, m)$, we use $\mathcal{C}_u := \mathcal{RM}(r, m-1)$ and $\mathcal{C}_v := \mathcal{RM}(r-1, m-1)$. The corresponding excerpt from Figure 5.1a, which visualizes this relationship for arbitrary r and m is highlighted in Figure 5.1b. A Reed–Muller code with parameters r and m is recursively defined as

$$\mathcal{C} := \mathcal{RM}(r, m) = \{(\boldsymbol{u}|\boldsymbol{u}+\boldsymbol{v}) : \boldsymbol{u} \in \mathcal{RM}(r, m-1), \boldsymbol{v} \in \mathcal{RM}(r-1, m-1)\}. \tag{5.2}$$

Reed–Muller codes of order 0, i.e. $\mathcal{RM}(0, m)$, are the binary repetition codes of length 2^m. Reed–Muller codes $\mathcal{RM}(m-1, m)$ are the binary single parity-check codes of length 2^m.

Repetition codes and single parity-check codes are used to terminate the recursion, when defining and decoding Reed–Muller codes.

Theorem 5.4. An $\mathcal{RM}(r, m)$ code has length 2^m, dimension $k = \sum_{i=0}^{r} \binom{m}{i}$ and minimum distance $d = 2^{m-r}$.

Proof. The proof directly follows from Theorem 5.2. □

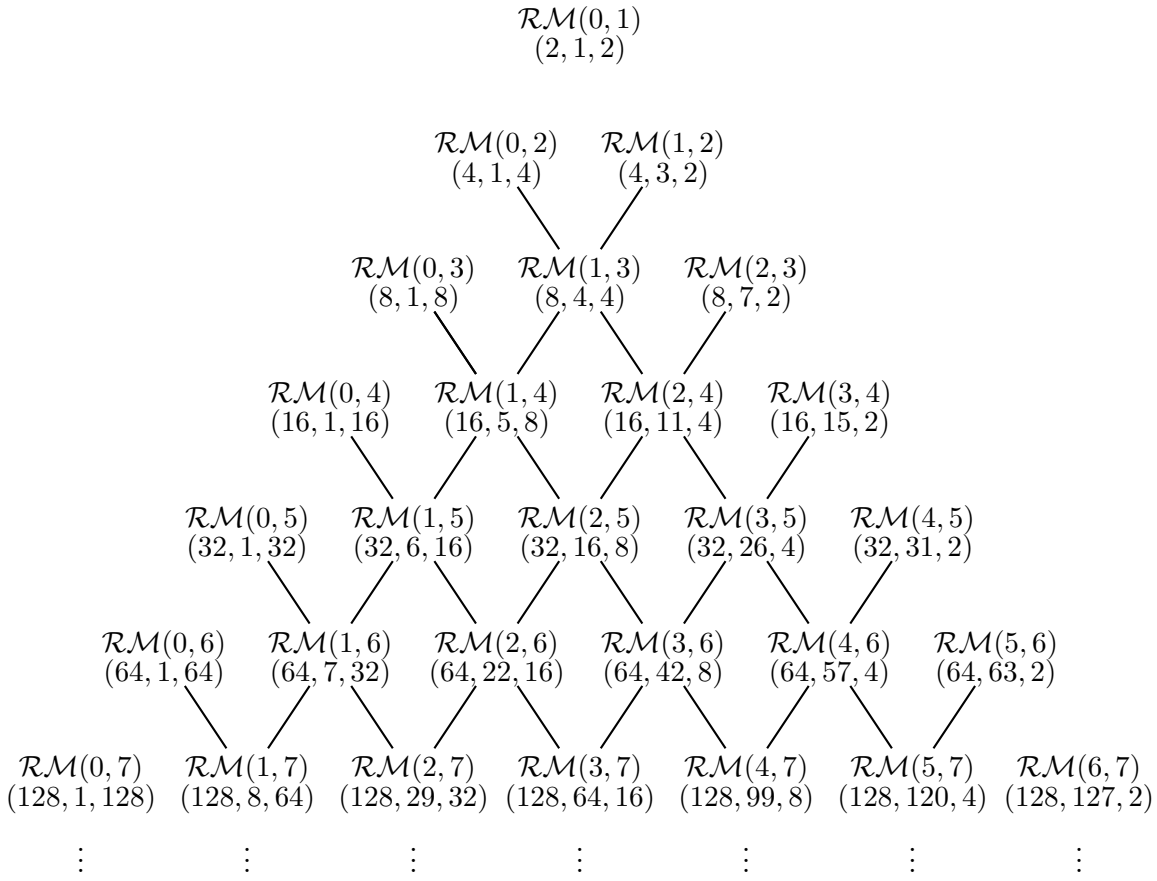
In order to encode an information vector using a code $\mathcal{C} = \mathcal{RM}(r, m)$, the generator matrix

$$\boldsymbol{G} = \begin{pmatrix} \boldsymbol{G}_u & \boldsymbol{G}_u \\ \boldsymbol{0} & \boldsymbol{G}_v \end{pmatrix} \tag{5.3}$$

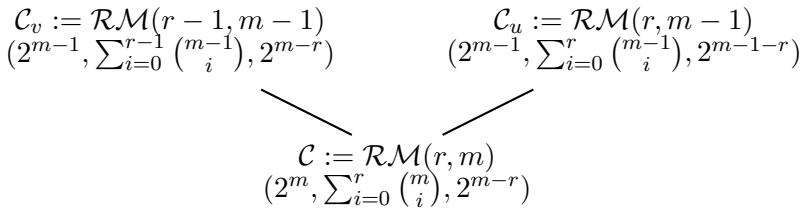
is used, where $\boldsymbol{G}_u$ is a generator matrix of the code $\mathcal{C}_u = \mathcal{RM}(r, m-1)$ and $\boldsymbol{G}_v$ is a generator matrix of the code $\mathcal{C}_v = \mathcal{RM}(r-1, m-1)$. The recursion terminates with $1 \times n_v$ generator matrix

$$\boldsymbol{G}_v = \begin{pmatrix} 1 & 1 & \cdots & 1 \end{pmatrix} \in \mathbb{F}_2^{2^m} \tag{5.4}$$

[2]The original definition of Reed–Muller codes was given based on boolean algebra [Mul54].



(a) Plotkin construction for the recursive definition of Reed–Muller codes. The recursion terminates with repetition codes $\mathcal{RM}(0,m)$ in the leftmost diagonal, and with single parity-check codes $\mathcal{RM}(m-1,m)$ in the rightmost diagonal.



(b) Excerpt from Figure 5.1a: $\mathcal{RM}(r,m)$ is constructed recursively, by concatenating the codewords $\boldsymbol{u} \in \mathcal{RM}(r,m-1)$ and $\boldsymbol{u}+\boldsymbol{v}$ for all codewords $\boldsymbol{v} \in \mathcal{RM}(r,m-1)$.

Figure 5.1: Recursive structure used for defining, encoding and decoding Reed–Muller codes.

for a repetition code of length n_v, and with a $k_u \times n_u$ generator matrix

$$\boldsymbol{G}_u = \begin{pmatrix} 1 & 0 & & 0 & 1 \\ 0 & 1 & & 0 & 1 \\ 0 & 0 & & 0 & 1 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & & 0 & \vdots \\ 0 & 0 & & 1 & 1 \end{pmatrix} \in \mathbb{F}_2^{2^{m-1} \times 2^m} \tag{5.5}$$

for an (n_u, k_u) single parity-check code, respectively.

For decoding, Algorithm 10 can be used. In the description of the algorithm, addition of vectors is understood to be component-wise. The symbol $\otimes$ denotes an erasure. An extensive example of decoding Reed–Muller codes recursively is given in [Bos12, Chapter 8.4.3]. We modified the algorithm therein to be able to decode τ errors and δ erasures if $2\tau + \delta < d$. Also an efficient modification of Algorithm 10, which allows the use of soft-information, exists. This so-called *Generalized Multiple Concatenated (GMC) algorithm* can be applied for generalized multiple concatenated codes, which can also be used to define Reed–Muller codes in an alternative but equivalent way [SB94].

Algorithm 10: Recursive decoding algorithm for Reed–Muller codes [Bos12, Chapter 8.4.3].

Input: Received vecor $\boldsymbol{y} = (\boldsymbol{y}_u | \boldsymbol{y}_v) = (\boldsymbol{u} + \boldsymbol{e}_u | \boldsymbol{u} + \boldsymbol{v} + \boldsymbol{e}_v) \in \{0, 1, \otimes\}^{2^m}$
Output: Decoding result

1 **if** *r=0* **then**
2 Decode $\boldsymbol{y}$ with a decoding algorithm for repetition code $\mathcal{RM}(r, m)$, let $\hat{\boldsymbol{v}}$ be the decoding result.
3 **return** $\hat{\boldsymbol{v}}$
4 **if** *r=m-1* **then**
5 Decode $\boldsymbol{y}$ with a decoding algorithm for single parity-check code $\mathcal{RM}(r, m)$, let $\hat{\boldsymbol{u}}$ be the decoding result.
6 **return** $\hat{\boldsymbol{u}}$
7 Decode $\boldsymbol{y}_u + \boldsymbol{y}_v = \boldsymbol{v} + \boldsymbol{e}_u + \boldsymbol{e}_v$ by calling this algorithm recursively for code $\mathcal{RM}(r-1, m-1)$ to obtain the decoding result $\hat{\boldsymbol{v}}$.
8 Decode $\boldsymbol{y}_v + \hat{\boldsymbol{v}} = \boldsymbol{u} + \boldsymbol{e}_v$ by calling this algorithm recursively for code $\mathcal{RM}(r, m-1)$ to obtain the decoding result $\hat{\boldsymbol{u}}_1$
9 Decode $\boldsymbol{y}_u + \boldsymbol{u} + \boldsymbol{e}_u$ by calling this algorithm recursively for code $\mathcal{RM}(r, m-1)$ to obtain the decoding result $\hat{\boldsymbol{u}}_2$
10 For $i = \{1, 2\}$, choose the $\hat{\boldsymbol{u}}_i$ (calculated in steps 8 and 9), such that $\mathrm{dist}_{\mathrm{H}}(\boldsymbol{y}, (\hat{\boldsymbol{u}}_i | \hat{\boldsymbol{u}}_i + \hat{\boldsymbol{v}}))$ is minimal.
11 **return** $(\hat{\boldsymbol{u}}_i | \hat{\boldsymbol{u}}_i + \hat{\boldsymbol{v}})$

Generalized Minimum Distance (GMD) decoding is an approach introduced in [FJ66b] in order to decode beyond half the minimum distance. Soft information are used to obtain the reliabilities of the symbols in the received word. A list of modified received words is generated.

In iteration i, each word in that list is generated by transforming the i least reliable symbols of the received word into erasures ($i = 0, \ldots, \lfloor \frac{d+1}{2} \rfloor$). Each of the modified received words is decoded by using an error and erasure decoding algorithm, and hence, a list of candidate solutions is generated. A soft-decision metric is calculated for all candidate solutions and the candidate with the best metric is chosen by the algorithm as decoding result. An extensive explanation of GMD decoding is given in [LC04, Chapter 10.4.1]. An analysis as well as modifications of GMD decoding are, for example, provided in [Bos99, Chapter 7.4.1] and [Sen11].

Reed–Solomon Codes

Reed–Solomon codes were proposed in [RS60] and became one of the most often used class of codes[3]. One advantage of Reed–Solomon codes results from flexibility in choosing code length n and dimension k, which makes it possible to adjust the code rate arbitrarily. Also, Reed–Solomon codes have the maximum possible minimum distance when n and k are given, i.e., they fulfill the Singleton bound $d \leq n - k + 1$ with equality. Furthermore, efficient algebraic decoding algorithms exist.

First, we define Reed–Solomon codes as *evaluation codes*. The name evaluation code arises from the fact, that the code is defined by evaluation of polynomials.

Definition 5.5. Let $\mathbb{F}_q$ be a finite field and $\alpha_1, \ldots, \alpha_n$ distinct elements from $\mathbb{F}_q \setminus \{0\}$, for example choose α to be an element of order n. A Reed–Solomon ($\mathcal{RS}$) code of length n and dimension k over the field $\mathbb{F}_q$ is defined as

$$\mathcal{C} = \{f(\alpha_1), \ldots, f(\alpha_n) : f \in \mathbb{F}_q[x], \mathsf{deg}(f) < k\}. \tag{5.6}$$

The polynomial $f = i_0 + i_1 x + \ldots i_{k-1} x^{k-1}$ (where $i_j \in \mathbb{F}_q$ for $j = 0, \ldots k-1$) is the so-called information polynomial, whose k coefficients are the k information symbols. Such a code is often denoted as $\mathcal{RS}(q; n, k)$ or $\mathcal{RS}(q; n, k, d)$, where q is the size of the finite field over which the code is defined.

Remark 5.6. In the field of engineering, often the discrete Fourier transformation (DFT) is used to define Reed–Solomon codes. This leads to an equivalent definition, however, proofs can often be simplified by using properties of the DFT for justification.

As for every linear code, for encoding and decoding a generator matrix and a parity-check matrix exist. However, Reed–Solomon codes are cyclic codes, which are usually encoded by multiplying the information polynomial with a generator polynomial and decoded with the help of a parity-check polynomial. For Reed–Solomon codes

$$g(x) = \prod_{i=k}^{n-1} (x - \alpha^{-i}), \ \mathsf{deg}(g) = n - k \tag{5.7}$$

[3] For example, Reed–Solomon codes are used for data storage on CDs/DVDs, data transmission in several space missions (Voyager 2, Galileo, Huygens), QR codes, NATO military radiocommunication and QR codes. Reed–Solomon are also a part of concatenated schemes used in DVB and DSL.

is the generator polynomial and

$$h(x) = \prod_{i=0}^{k-1}(x - \alpha^{-i}),\ \mathsf{deg}(h) = k \tag{5.8}$$

is the parity-check polynomial.

There exists a large variety of algebraic decoding algorithms for Reed–Solomon codes. The problem of algebraically decoding Reed–Solomon codes is reduced to solving a so-called key equation, which is done in two steps. First, the error positions in the received word are determined. In a second step, the error values are calculated. Many BMD decoders that allow algebraic decoding in order to correct up to $\lfloor \frac{d-1}{2} \rfloor$ errors exist. The most famous ones are the Peterson–Gorenstein–Zierler algorithm [Pet60, GZ61], the Berlekamp–Massey algorithm [Ber66], and the Welch–Berlekamp algorithm [WB86]. List decoders allow to correct beyond $\lfloor \frac{d-1}{2} \rfloor$ errors by increasing the radius of the correction spheres and returning a list of codeword candidates in cases where no unique decoding result exists. Well-known list decoders are the Sudan algorithm [Sud97], the Guruswami-Sudan Algorithm [GS98], and Wu's algorithm [Wu08].

Other methods to decode beyond half the minimum distance are to use interleaved Reed–Solomon codes [BKY03, SSB06b], power decoding [SSB06a] or power decoding up to the Johnson radius [Nie15].

Definition 5.7. Let $\mathcal{C}_1, \ldots, \mathcal{C}_\ell$ be (not necessarily distinct) Reed-Solomon codes of length n and dimensions $k_1, \ldots, k_\ell$ over a finite field $\mathbb{F}_q$. An *interleaved Reed–Solomon code* is defined as

$$\mathcal{C}(\ell; n, k_1, \ldots, k_\ell) = \left\{ \boldsymbol{c} = \begin{pmatrix} \boldsymbol{c}_1 \\ \vdots \\ \boldsymbol{c}_\ell \end{pmatrix} : \boldsymbol{c}_i \in \mathcal{C}_i, i = 1, \ldots, \ell \right\}. \tag{5.9}$$

If $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_\ell$ are codewords of the same Reed–Solomon code, i.e., $\mathcal{C}_i = \mathcal{C}_j \forall i \neq j$, the interleaved Reed–Solomon code is called *homogeneous*. If different Reed–Solomon codes of the same length but maybe of different dimensions are used, the resulting interleaved Reed–Solomon code is called *heterogeneous*.

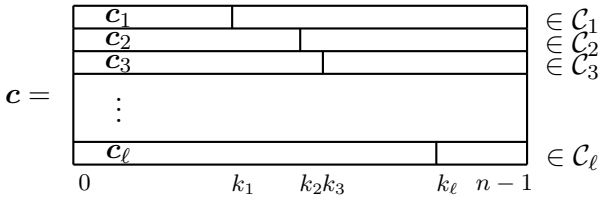


Figure 5.2: Structure of a codeword of a heterogeneous interleaved Reed–Solomon code.

The structure of a heterogeneous interleaved Reed–Solomon codeword is visualized in Figure 5.2. Encoding is performed by successively encoding in ℓ normal Reed–Solomon codes. Consequently, the rows of the resulting matrix consist of codewords of the Reed–Solomon codes $\mathcal{C}_1, \ldots, \mathcal{C}_\ell$. The columns of the matrix can be interpreted as symbols of an extension

field $\mathbb{F}_{q^\ell}$. Interleaved Reed–Solomon codes are convenient for scenarios in which burst errors occur. A burst error can be denoted as an error vector of length n which consists of symbols of $\mathbb{F}_{q^\ell}$. As a consequence, burst errors have an impact on columns of $\boldsymbol{c}$ and hence, alter all codewords in the same positions. As an alternative to decoding every codeword separately, collaborative decoding can be used to exploit the described property and thus increasing the error-correction radius. Note that Reed–Solomon codes as defined in Definition 5.5 are a special case of interleaved Reed–Solomon codes for $\ell = 1$.

In the code construction that will be presented in Section 5.1.3, power decoding as proposed in [SSB06a] is applied. Using an $\mathcal{RS}(q; n, k)$ code, the main idea of power decoding is to transform the received word into a received word of a heterogeneous interleaved Reed–Solomon code. To explain the idea, we first need the following definition.

Definition 5.8. Let $p(x) = p_0 + p_1 x + \cdots + p_{n-1}x^{n-1}$ be a polynomial with coefficients from a finite field $\mathbb{F}_q$. We define

$$p^{\langle i \rangle} := p_0^i + p_1^i x + \cdots + p_{n-1}^i x^{n-1}. \tag{5.10}$$

Let $\mathcal{C}$ be an $\mathcal{RS}(q; n, k)$ code. We define

$$\mathcal{C}^{\langle i \rangle} := \{c^{\langle i \rangle}(x) : c(x) \in \mathcal{C}\}. \tag{5.11}$$

Theorem 5.9. Let $\mathcal{C} = \mathcal{RS}(q; n, k)$ and $\mathcal{C}^{\langle i \rangle}$ as defined in Definition 5.8. Further, let $\boldsymbol{c}$ be any codeword of $\mathcal{C}^{\langle i \rangle}$. If $i(k-1)+1 \leq n$, $\boldsymbol{c}$ is a codeword of Reed–Solomon code

$$\mathcal{C}^{(i)} := \mathcal{RS}(q; n, i(k-1)+1, n-i(k-1)). \tag{5.12}$$

Proof. For a formal proof, we refer to [SSB06a, Lemma 1]. □

Using power decoding, the matrix

$$\boldsymbol{Y} = \begin{pmatrix} y^{\langle 1 \rangle}(x) \\ y^{\langle 2 \rangle}(x) \\ \vdots \\ y^{\langle \ell \rangle}(x) \end{pmatrix} \tag{5.13}$$

is constructed from the received word $y(x) = y_0 + y_1 x + \cdots + y_{n-1}x^{n-1}$, where $y^{\langle i \rangle}(x)$ is generated according to (5.10). The parameter ℓ has to be chosen, such that $\ell(k-1)+1 \leq n$ (cf. Theorem 5.9). Figure 5.3 illustrates this transformation.

Note that error positions are not modified by powering coefficients, but might get annihilated. The matrix $\boldsymbol{Y}$ can be interpreted as an erroneous codeword of a heterogeneous interleaved Reed–Solomon code. For decoding heterogeneous interleaved Reed–Solomon codes, a modified multi sequence shift-register synthesis algorithm, proposed by Schmidt and Sidorenko in [SS06], can be used.

Basically, power decoding has two benefits. First, an implementation can be realized easily by using shift-register synthesis. Second, as proven in [SSB06a, Chapter 4], decoding beyond half the minimum distance is possible for codes with low rates, for which

$$\tau_\ell = \left\lfloor \frac{2\ell n - \ell(\ell+1)k + \ell(\ell-1)}{2(\ell+1)} \right\rfloor \tag{5.14}$$

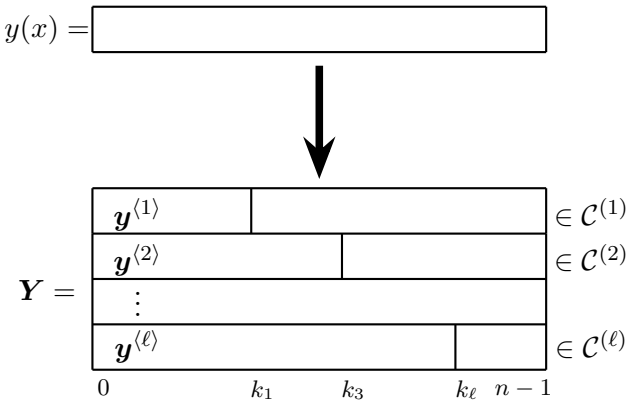


Figure 5.3: The received word $y(x)$, being an erroneous codeword of $\mathcal{RS}(q; n, k)$, is transformed into an erroneous codeword of a heterogeneous interleaved Reed–Solomon code.

errors can be corrected with high probability for $\ell \leq \ell_{max}$, where ℓ_{max} is upper bounded by

$$\ell_{max} \leq \frac{\sqrt{(k+3)^2 + 8(k-1)(n-1)} - (k+3)}{2(k-1)}. \tag{5.15}$$

Figure 5.4 compares the maximum decoding radius for power decoding of Reed–Solomon codes and the decoding radius when applying half the minimum distance decoding. For codes with rates $\leq \frac{1}{3}$ the decoding radius can be extended by using power decoding instead of bounded minimum distance decoding. An extensive overview of algorithms to decode Reed–Solomon codes can be found in [BB13].

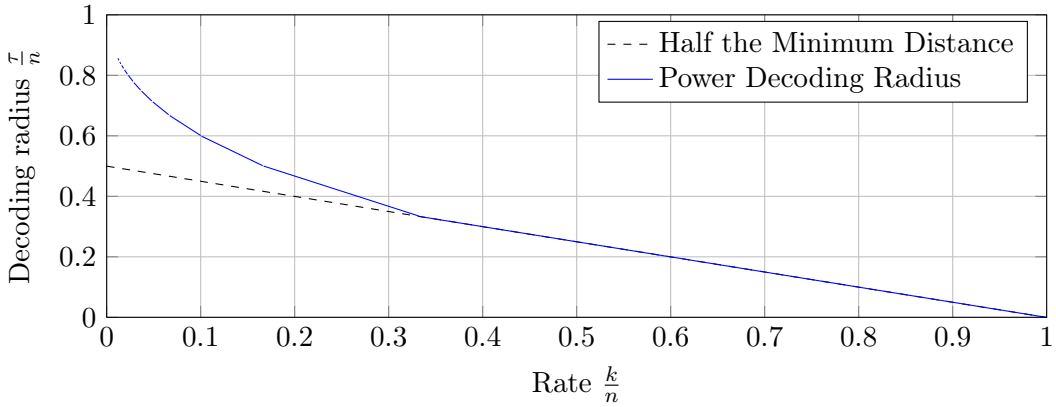


Figure 5.4: Maximum decoding radius for power decoding of Reed–Solomon codes in comparison to half the minimum distance decoding. For low-rate codes, power decoding enables to extend the decoding radius significantly.

Another famous class of cyclic codes are BCH codes, introduced by Bose, Ray-Chaudhuri and Hocquenghem [BRC60b, BRC60a, Hoc59]. For decoding, the same methods as for Reed–Solomon codes can be applied. A BCH code of length $n = 32$, dimension $k = 11$, and minimum distance $d = 12$. is applied in Section 5.1.3 for constructing generalized concatenated code $\mathcal{C}_4$.

Code Concatenation

There are two types of code concatenation, namely *ordinary code concatenation* [FJ66a] and *generalized code concatenation* [BZ74]. The general idea of ordinary code concatenation is to encode information first with an outer code $\mathcal{A}(n_o, k_o, d_o)$ and then to apply an inner code $\mathcal{B}(n_i, k_i, d_i)$ to the positions of the codeword generated by the outer code.[4] Using generalized code concatenation, an inner code $\mathcal{B}^{(1)}$ is partitioned into several disjoint sub-codes $\mathcal{B}_i^{(2)}$ such that $\mathcal{B}^{(1)} = \dot{\bigcup}_i \mathcal{B}_i^{(2)}$. These sub-codes are again partitioned until the partitions generated in this way only consist of one codeword each. The inner code should be chosen, such that the minimum distance within the partitions is as large as possible. The edges of a partition tree are numbered with binary labels, which are used to uniquely identify a codeword of the inner code $\mathcal{B}^{(1)}$, when traversing the partition tree from the root to a leaf, in which according to the construction only one codeword of $\mathcal{B}^{(1)}$ is stored. The partitions of level i are protected with an outer code $\mathcal{A}^{(i)}$. Hence, in contrast to ordinary concatenated codes, several outer codes are used. Examples for both, ordinary concatenated codes and generalized concatenated codes follow as the main contributions of this chapter in Section 5.1.2 and Section 5.1.3, respectively.

The idea of concatenated codes is to construct long codes, by concatenating short codes. The advantage in comparison to long codes that are constructed directly (without using the concepts of concatenation) is a reduced decoding complexity. This arises from the fact, that only short codes have to be decoded, which can be performed more efficiently than decoding a single long code. We emphasize the main advantage of generalized code concatenation in comparison to ordinary code concatenation, that is also relevant for the application in the field of PUFs: Using the exact same parameters for length n and dimension k, generalized code concatenation allows to construct codes with a larger minimum distance than attainable by ordinary code concatenation. To illustrate this statement, we use an example from [ZSB99]: For given length $n = 63$ and dimension $k = 24$, an ordinary concatenated code with minimum distance $d = 12$ is constructed. On the other hand, the construction of a generalized concatenated code results in a code with minimum distance $d = 16$. The statement can also be given from another perspective: For given length n and minimum distance d, generalized code concatenation allows the construction of codes with a larger dimension k (and thus a higher code rate and more codewords) than ordinary code concatenation does. Table 5.1 summarizes how to calculate the parameters of a (generalized) concatenated code, based on the parameters of the inner and outer codes.

When explaining generalized code concatenation, literature usually uses examples instead of abstract descriptions, since there exists much theory which leads to a complicated notation and goes beyond the scope of most textbooks as well as beyond the scope of most works in which generalized code concatenation is applied. Hence, also in this dissertation we use our code constructions to explain partitioning of an inner code, encoding and decoding. For extensive information about concatenated codes, the interested reader is referred to the literature: [ZSB99], [Bos99, Chapter 9] and [LC04, Chapter 15] provide comprehensive introductions to (generalized) concatenated codes.

In Section 5.1.2, we use Reed–Muller codes as introduced in Section 5.1.1 in order to construct a generalized concatenated code for a PUF scenario. In Section 5.1.3, Reed–Solomon

[4] Also, it is possible to use $\ell > 1$ outer codes which have the same redundancy (cf. [Bos99, Example 9.3]). Since the ordinary concatenated codes which we construct in this chapter only use one outer code, we do not go further into details about the use of $\ell > 1$ outer codes.

Table 5.1: Parameters of ordinary and generalized concatenated codes, based on the parameters of inner codes $\mathcal{B}^{(\ell)}$ and outer codes $\mathcal{A}^{(\ell)}$ $(\ell = 1, \ldots, l)$.

	$\mathcal{B}^{(\ell)}(p; n_i, k_i^{(\ell)}, d_i^{(\ell)})$, $\mathcal{A}^{(\ell)}(p_\ell^m; n_o, k_o^{(\ell)}, d_o^{(\ell)})$, $\ell = 1, \ldots, l$
Ordinary concatenated code $(\ell = 1)$	$n = n_i \cdot n_o$ $k = k_i \cdot k_o$ $d \geq d_o \cdot d_i$
Generalized concatenated code $(\ell = 1, \ldots, l)$	$n = n_i \cdot n_o$ $k = \sum_{\ell=1}^{l} m_\ell \cdot k_o^{(\ell)}$ $d \geq \min_{\ell=1,\ldots,l} \left\{ d_i^{(\ell)} \cdot d_o^{(\ell)} \right\}$

codes as defined in Section 5.1.1 are used for the construction of ordinary as well as generalized concatenated codes to be applied in the same scenario.

5.1.2 Error Correction for PUFs Using Reed–Muller Codes and Generalized Code Concatenation

The first code construction proposed in this chapter, is a generalized concatenated code based on Reed–Muller codes, which are an appropriate choice due to efficient decoding algorithms that are suitable for hardware implementations. We briefly recall the requirements which the constructed code aims to fulfill, according to the reference code construction given in [MVHV12]: Required is a code of length $n < 2226$, dimension $k \geq 128$, and a block error probability $P_{\text{err}} < 10^{-9}$.

Code Construction

The generalized concatenated code constructed in this section uses the code $\mathcal{B}^{(1)}(16, 5, 8) = \mathcal{RM}(1, 4)$ as inner code. Also, two outer codes are applied, namely the $\mathcal{A}^{(1)}(2^4; 128, 8, 64) = RM(1, 7)$ and the $\mathcal{A}^{(2)}(128, 99, 8) = \mathcal{RM}(4, 7)$, in order to construct a generalized concatenated code $\mathcal{C}_1(2048, 131)$.

The first step of constructing a generalized concatenated code is to partition the inner code. In general, this is not a trivial task and much research to solve this problem has been carried out, cf. for example [Bos99, Chapter 9.2.2] for an overview of possible methods. The partitioning of the construction proposed in this section is visualized in Figure 5.5. The inner code $\mathcal{B}^{(1)}(16, 5, 8) = \mathcal{RM}(1, 4)$ is partitioned such that the minimum distance of the codewords in all partitions is as large as possible. Therefore, first, the inner code $\mathcal{B}^{(1)}(16, 5, 8)$ is partitioned into 16 sub-codes, enumerated by $\mathcal{B}_i^{(2)}$, where i is the 4-digit binary representation of the numbers from 0 to 15, used to label the partitions. These 16 sub-codes, each containing two of the 32 codewords of $\mathcal{B}^{(1)}$, represent the second level in the partition tree. Note that $\mathcal{B}^{(1)} = \dot{\bigcup}_i \mathcal{B}_i^{(2)}$. We choose $\mathcal{B}_{0000}^{(2)}$ to be the repetition code of length 16, in order to maximize the minimum distance. The other 15 sub-codes in level 2

are generated by taking the distinct cosets of $\mathcal{B}^{(2)}_{0000}$. The edges between level 1 and level 2 are labeled with the 4-digit binary identifiers i of the codes $\mathcal{B}^{(2)}_i$. These labels are used for encoding and decoding and are protected with outer code $\mathcal{A}^{(1)}(2^4; 128, 8, 64)$, which is the Reed–Muller code $\mathcal{RM}(1, 7)$. The partitioning of the second level codes $\mathcal{B}^{(2)}_i$ is trivial, since they consist of only two codewords each. Hence, the 16 sub-codes in level 2 are divided into two partitions each. In total, we obtain 32 codes in level 3, each containing only one codeword. The edges of the partitions from level 2 to level 3 are labeled with 0 or 1, since one bit is sufficient in this case to uniquely identify a codeword in level 3. The codes in level 3 are denoted by $\mathcal{B}^{(3)}_{i,j}$, where i is the 4-digit binary identifier of the labeling from level 1 to level 2 and j is the 1-digit binary identifier of the labeling from level 2 to level 3. The labeling from level 2 to level 3 is protected with an outer code $\mathcal{A}^{(2)}(128, 99, 8)$, which is the Reed–Muller code $\mathcal{RM}(4, 7)$. Note that the labeling allows to uniquely identify a codeword, when starting at the root of the partition tree and traversing the tree until a leaf is reached, by following the edges indicated by a label. The generalized concatenated code $\mathcal{C}_1$ constructed in this section has length $n_i \cdot n_o = 16 \cdot 128 = 2048$ and dimension 131 (cf. Table 5.1).

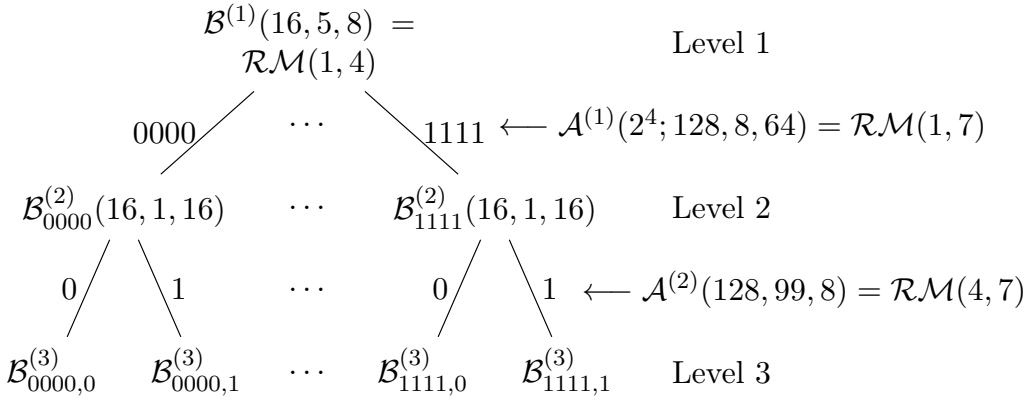


Figure 5.5: Partitions of the inner code $\mathcal{B}^{(1)}(16, 5, 8) = \mathcal{RM}(1, 4)$. The outer codes $\mathcal{A}^{(1)}(2^4; 128, 8, 64) = \mathcal{RM}(1, 7)$ and $\mathcal{A}^{(2)}(128, 99, 8) = \mathcal{RM}(4, 7)$ are used to protect the labeling of the partitions. The labels on the edges allow to uniquely identify each codeword of the inner code $\mathcal{B}^{(1)}$.

Encoding

The encoding process is visualized in Figure 5.6 and is performed in three steps, denoted by (a) – (c). The generalized concatenated code $\mathcal{C}_1(2048, 131)$ constructed above is used to encode 131 information bits, which are separated into the 8×4 and 99×1 Matrices I and II, respectively. In step (a), the first 32 information bits are encoded. For that purpose, the encoder of outer code $\mathcal{A}^{(1)}(2^4; 128, 8, 64) = \mathcal{RM}(1, 7)$ is applied column-wise to Matrix I. Result of that operation is the 128×4 Matrix III, whose columns are codewords of the code $\mathcal{A}^{(1)}$. The number of columns is determined by the four bits, used to label the edges between the first two levels in the partition tree. In step (b), the remaining 99 information symbols are encoded, using the outer code $\mathcal{A}^{(2)}(128, 99, 8) = \mathcal{RM}(4, 7)$. The 128×1 Matrix IV represents the corresponding codeword. Next, the codeword Matrices III and IV are

read row-wise in step (c). For each row, the four bits in Matrix III define a path from the inner code $\mathcal{B}^{(1)}$ (root of the partition tree), to one of its partitions in the second level, by following the edge labeled with these four bits. Matrix IV, generated by the encoder of $\mathcal{A}^{(2)}$, for each row dictates the path to use from the chosen second level code to a third level code. Note that in the third level, each code consists of only one codeword, which is chosen to compose the corresponding row in the resulting 128×16 Matrix V. After performing this codeword selection for all 128 rows, Matrix V is the codeword of the generalized concatenated code $\mathcal{C}_1(2048, 131)$. Table 5.2 provides a brief summary of the encoding process described in this section.

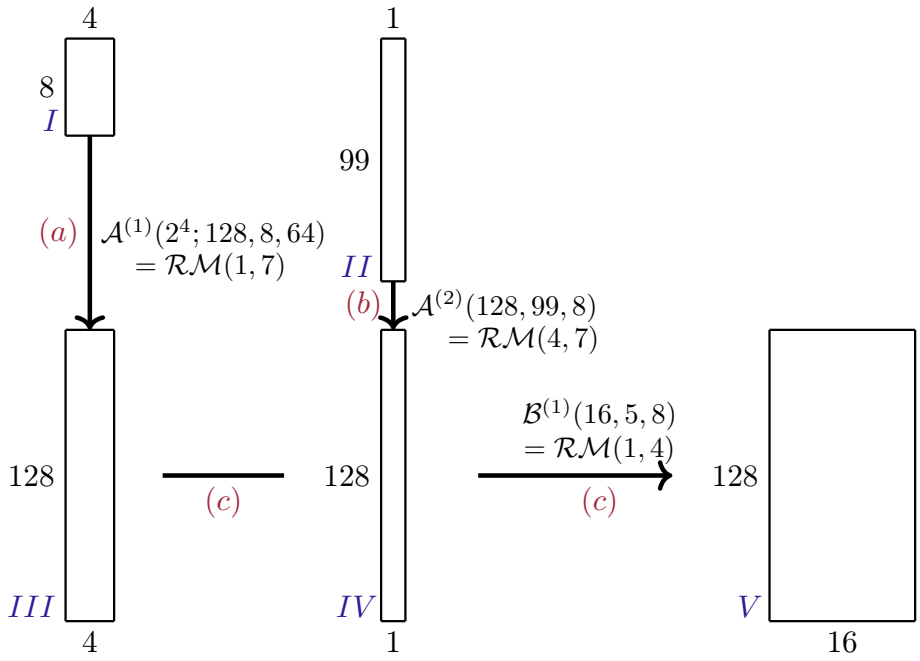


Figure 5.6: Illustration of the three encoding steps. First, $4 \cdot 8$ information bits are mapped column-wise to codewords of outer code $\mathcal{A}^{(1)}$. Second, 99 information bits are encoded by using outer code $\mathcal{A}^{(2)}$. Third, codewords of the inner code $\mathcal{B}^{(1)}$ are chosen by following the labels given by the rows of Matrices III and IV to a leaf of the partition tree.

Decoding

This paragraph describes, how to decode the generalized concatenated code $\mathcal{C}_1(2048, 131)$. The description of the decoding process is structured into two stages. Stage 1 aims for recovering the information bits encoded with outer code $\mathcal{A}^{(1)}$ (recall Figure 5.6, step (a)), while stage 2 deals with recovering the information bits encoded with outer code $\mathcal{A}^{(2)}$ (recall Figure 5.6, step (b)).

We start the description with stage 1 of the decoding process according to Figure 5.7. Let the 128×16 Matrix I represent the received word, i.e., a codeword of code $\mathcal{C}_1$ plus an error which occurred in the BSC during transmission. Remember from the encoding process

Table 5.2: Brief description of the encoding process of generalized concatenated code $\mathcal{C}_1$ and legend to Figure 5.6.

Matrix/Step	Description
I	(8×4)–matrix containing 32 information bits.
(a)	Column-wise encoding in outer code $\mathcal{A}^{(1)}(128, 8, 64) = \mathcal{RM}(1, 7)$.
III	Result of step (a): (128×4)–matrix, columns are codewords of $\mathcal{A}^{(1)}$. Each row provides the first partition index i (4 bits) for encoding in step (c), i.e., for choosing $\mathcal{B}_i^{(2)}$.
II	(99×1)–matrix containing 99 information bits.
(b)	Encoding in outer code $\mathcal{A}^{(2)}(128, 99, 8) = \mathcal{RM}(4, 7)$.
IV	Result of step (b): (128×1)–matrix which is codeword of $\mathcal{A}^{(2)}$. Each row provides the second partition index j (1 bit) for encoding in step (c), i.e., for choosing $\mathcal{B}_{i,j}^{(3)}$.
(c)	Takes row–wise partition indices i (from Matrix *III*) and j (from Matrix *IV*) and writes the only codeword contained in $\mathcal{B}_{i,j}^{(3)}$ in the corresponding row of Matrix *V*.
V	Final result: The codeword of the generalized concatenated code $\mathcal{C}_1(2048, 131)$.

described above, that each row in that matrix is a codeword of the inner code $\mathcal{B}^{(1)}$, plus an error. Hence, step (a) decodes row-wise in $\mathcal{B}^{(1)}$. Since $\mathcal{B}^{(1)}$ in our construction only consists of $2^5 = 32$ codewords, maximum likelihood decoding can efficiently be performed. In Matrix *II*, each row consists of either the corresponding decoding result (codeword of code $\mathcal{B}^{(1)}$) or a length-16 sequence of erasures in cases where the decoder is not able to uniquely determine the codeword closest to the corresponding received word. This happens for example when two codewords with the same distance to the received word exist. Additionally, the Hamming distance of the received row and the decoding result is stored, to be later used as soft information in GMD decoding (cf. step (c)). In step (b), each row is re-mapped to the length-4 bit sequence, which labels the partition from level 1 to level 2 that includes the codeword of that row. This can be done by taking the information parts of the codewords in the rows of Matrix *II*. Each row of Matrix *III* either consists of the label to the second level code containing the codeword in the corresponding row of Matrix *II*, or of a length-4 sequence of erasures. Note that each column of Matrix *III* consists of a codeword of the outer code $\mathcal{A}^{(1)} = \mathcal{RM}(1, 7)$ (cf. Figure 5.6, Matrix *III*). In step (c), decoding of $\mathcal{A}^{(1)}$ is applied to all four columns. We used GMD decoding in our simulations, internally applying the decoder explained in Algorithm 10 as an error and erasure decoding algorithm. However, the use of GMD decoding is optional, any error and erasure decoding algorithm can be applied. The

erasures are transfered recursively, until a repetition or parity-check code, for which decoding erasures can be done easily, is reached. If decoding fails, we define the decoding algorithm to terminate by returning a decoding failure. In this case, the decoding process fails in stage 1. The decoding results are represented by the columns of Matrix IV, thus each column of Matrix IV is a codeword of $\mathcal{A}^{(1)} = \mathcal{RM}(1,7)$. Matrix IV is used for two purposes: First, recall from encoding (Figure 5.6, step (a)), that we use $\mathcal{A}^{(1)} = \mathcal{RM}(1,7)$ in order to encode eight information bits. Thus, we can extract eight information bits from each of the four columns of Matrix IV in step (d). Hence, the first $8 \cdot 4 = 32$ bits of the information are recovered. Second, the rows in Matrix IV provide partition information, which is used for the next decoding step.

In stage 2 of the decoding process, we want to recover the remaining 99 information bits, which were encoded using the outer code $\mathcal{A}^{(2)} = \mathcal{RM}(4,7)$ (cf. Figure 5.6, step (b)). We again start with the received word, represented by Matrix I and decode row-wise in code $\mathcal{B}_i^{(2)}$. The index i is determined by the labeling given by the corresponding row of Matrix IV. Analog to the first step of stage 1, maximum likelihood decoding can efficiently be applied in step (e), due to the small number of codewords in code $\mathcal{B}_i^{(2)}$, namely only two in this case. Analog to step (a), soft information (to be used in step (g)) is obtained. The rows of Matrix V either contain the decoding results or a length-16 sequence of erasures in cases where unique maximum likelihood decoding is not possible. In step (f), every row of Matrix V is either re-mapped to the 1-bit index which labels the corresponding level-3 partition of code $\mathcal{B}_i^{(2)}$, or to an erasure. This re-mapping is represented by the 128×1 Matrix VI. Note that this is a codeword of the outer code $\mathcal{A}^{(2)} = \mathcal{RM}(4,7)$. Analog to stage 1, GMD decoding is applied in step (g), using the soft information generated in step (e). If the decoding in this step fails, we define the algorithm to return a decoding failure. In this case, the decoding process fails in stage 2. Matrix VII contains the corresponding decoding result and hence a codeword of code $\mathcal{A}^{(2)} = \mathcal{RM}(4,7)$. Finally, in step (h), the remaining 99 information bits can be recovered. The decoding process described in this paragraph is briefly summarized in Table 5.3.

Analysis

This paragraph aims for analyzing the block error probability P_{err} of the constructed generalized concatenated code $\mathcal{C}_1(2048, 131)$ by deriving an upper bound of P_{err}.

Recall, that decoding of generalized concatenated codes is performed in several stages, one stage per outer code $\mathcal{A}$. Let these stages be consecutively numbered with $1, \cdots, r$, where r is the number of outer codes used in a generalized concatenated code. In code $\mathcal{C}_1(2048, 131)$, two outer codes $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$ are used, and hence, the decoding process consists of two stages. Let S_i be the event that decoding fails in stage i. The overall decoding process fails, if at least one of the stages of the decoding process fails, i.e., when at least one of the events S_i occurs. Using the union bound, an upper bound for P_{err} can be calculated by

$$P_{\text{err}} = \mathrm{P}\left(\bigcup_{i=1}^{r} S_i\right) \leq \sum_{i=1}^{r} \mathrm{P}\left(S_i\right). \tag{5.16}$$

For the code $\mathcal{C}_1(2048, 131)$ considered in this section, we have to study the events S_1 and S_2.

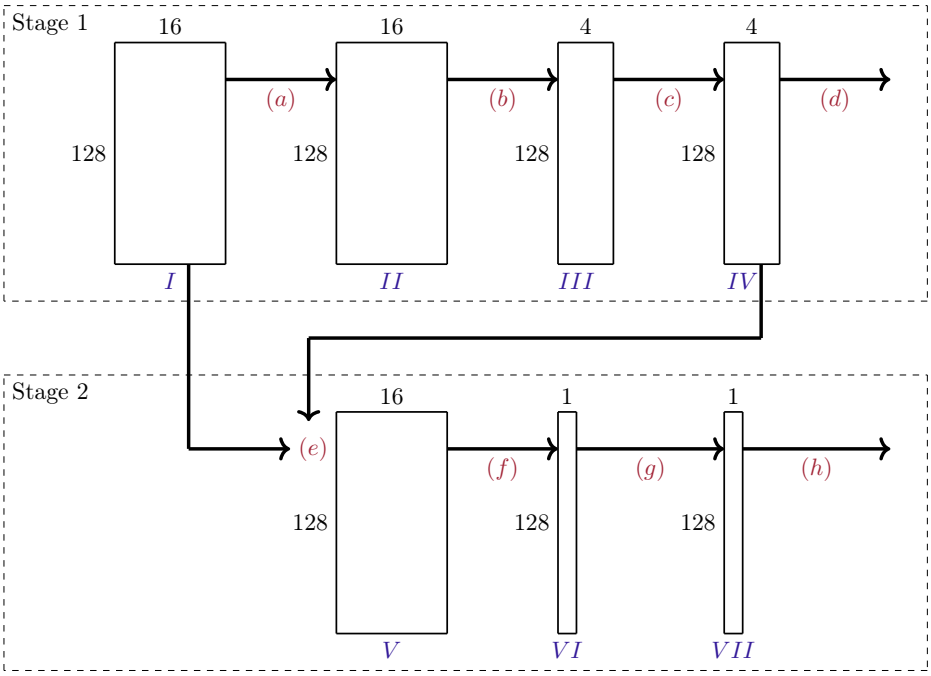


Figure 5.7: Illustration of the algorithm, used to decode the generalized concatenated code $\mathcal{C}_1$. Also refer to Table 5.3. for a summary of the several steps.

Since erasures can occur in the decoding process, the BSC with bit error probability $p = 0.14$ needs to be transformed into a binary error and erasure channel. Maximum likelihood decoding of the inner code $\mathcal{B}^{(1)}(16, 5, 8) = \mathcal{RM}(1, 4)$ was used in simulations in order to obtain the probability

$$\mathrm{P}(\text{error}) = 0.020698 \tag{5.17}$$

for the occurrence of an error and the probability

$$\mathrm{P}(\text{erasure}) = 0.155532 \tag{5.18}$$

for the occurrence of an erasure. We begin with studying the event S_1, which corresponds to stage 1 of the decoding algorithm, cf. decoding steps (a)–(d) in Figure 5.7. Recall, that an error and erasure decoder of outer code $\mathcal{A}^{(1)}(128, 8, 64) = \mathcal{RM}(1, 7)$ can decode correctly while $2\tau+\delta < 64$, where τ is the number of errors and δ is the number of erasures. Considering this condition, we calculate the probability that S_1 occurs, i.e., that decoding fails in stage 1 and we obtain

$$\begin{aligned}\mathrm{P}(S_1) &= \mathrm{P}(2\tau + \delta \geq 64) \\ &= \sum_{i=0}^{128} \mathrm{P}(\delta = i) \cdot \mathrm{P}(2\tau \geq 64 - i | \delta = i) \\ &\approx 9.51 \cdot 10^{-12}.\end{aligned} \tag{5.19}$$

Next, we consider event S_2, which corresponds to stage 2 of the decoding process, cf. decoding steps (e)–(h) in Figure 5.7. The approach to calculate the probability that event S_2 occurs is the exact same than for event S_1. We obtain

$$\mathrm{P}(S_2) \approx 1.48 \cdot 10^{-9}. \tag{5.20}$$

Since decoding fails when at least S_1 or S_2 occurs, we can use (5.19) and (5.20) to derive the upper bound

$$\begin{aligned} P_{\text{err}} &\leq \mathrm{P}(S_1) + \mathrm{P}(S_2) \\ &= 9.51 \cdot 10^{-12} + 1.48 \cdot 10^{-9} \\ &\approx 1.49 \cdot 10^{-9} \end{aligned} \tag{5.21}$$

according to (5.16).

The analysis in this paragraph was done by assuming a unique (i.e., $2\tau + \delta < d$) error and erasure decoder. The block error probability P_{err} can further be decreased by applying GMD decoding, which decodes beyond half the minimum distance. It is hard to analytically describe this behavior, thus simulations were performed and a block error probability

$$P_{\text{err}} \approx 5.37 \cdot 10^{-10} \tag{5.22}$$

was achieved for GMD decoding. To simulate such a large amount of instances, we wrote a highly optimized C program that needs 1 ms for one iteration. Using a simulation size of 10^{11} iterations, approximately 1150 days are needed, which can be reduced to approximately 23 days by using 50 kernels for computations.

Summary

Table 5.4 summarizes the advantages of the code construction proposed in this section, in comparison to the construction published in [MVHV12]. Using the generalized concatenated code $\mathcal{C}_1$ allows a decreased block error probability P_{err}. For a binary symmetric channel with bit error probability $p = 0.14$, simulations using GMD decoding have shown a block error probability in the order of $\approx 5.37 \cdot 10^{-10}$, which is smaller than the block error probability of $\approx 10^{-9}$ determined in [MVHV12]. Using an arbitrary error and erasure decoder, as shown in our analysis, P_{err} is at least in the order of the code suggested in [MVHV12]. Also, the code length is reduced from 2226 to 2048. Hence, fewer bits have to be extracted from the PUF. In addition, the decoder uses the binary field only. This is advantageous, since it is easier to implement operations over the binary field, than over larger fields.

Table 5.4: Improvements of the generalized concatenated code $\mathcal{C}_1(2048, 131)$ based on Reed–Muller codes, compared to the ordinary concatenated code construction in [MVHV12]. Our new code construction benefits from a shorter codeword length and a smaller block error probability when using GMD decoding. Additionally, only operations over the binary field have to be implemented.

Code construction	P_{err}	n	k	$\frac{k}{n}$	Largest Field
[MVHV12]	$\approx 10^{-9}$	2226	174	0.078	$\mathbb{F}_{2^8}$
GC RM $\mathcal{C}_1$ (Section 5.1.2)	$\approx 1.48 \cdot 10^{-9}$	2048	131	0.064	$\mathbb{F}_2$
GMD simulations	$\approx 5.37 \cdot 10^{-10}$				

When using Reed–Muller codes, the dimension, and hence, the code rate, cannot be chosen arbitrarily. This drawback restricts the use of Reed–Muller codes as outer codes in generalized concatenated schemes, in scenarios where a flexible design is desired. For this reason, Reed–Solomon codes are used in Section 5.1.3. Besides an arbitrarily adjustable code rate, Reed–Solomon codes are maximum distance separable (MDS), i.e., for given length and dimension, the minimum distance is maximized.

5.1.3 Error Correction for PUFs Using Reed–Solomon Codes and (Generalized) Code Concatenation

In this section, we propose three concatenated code constructions using Reed–Solomon codes. Concerning their parameters, Reed–Solomon codes are more flexible than Reed–Muller codes. Since they are MDS codes, they possess excellent error correction capabilities. Also many efficient decoding algorithms exist.

Ordinary Concatenated Code Constructions

We present two ordinary concatenated codes. Let $\mathcal{C}_2$ denote the first ordinary concatenated code, which we construct based on an outer Reed–Solomon code and an inner Reed–Muller code. We choose $\mathcal{B}(32, 6, 16) = \mathcal{RM}(1, 5)$ as inner code. Remember, that one of the requirements for the final code is a dimension of at least 128. Hence, as outer code we suggest to use a Reed–Solomon code of dimension $k_o = 22$, which leads to dimension

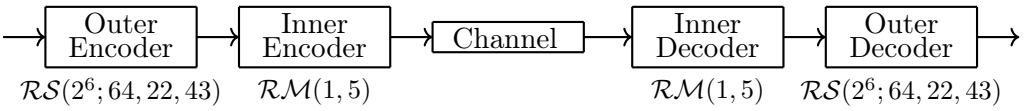
$$k = 6 \cdot k_o = 132 \tag{5.23}$$

of the ordinary concatenated code $\mathcal{C}_2$. Since we want to construct a code of length less than 2226, we choose $n_o = 64$ and obtain a concatenated code of length

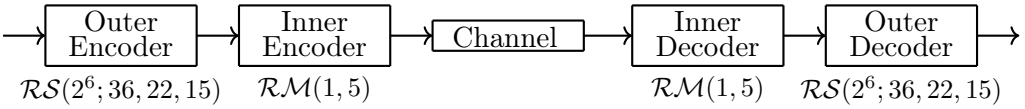
$$n_o \cdot n_i = 64 \cdot 32 = 2048. \tag{5.24}$$

This results in outer Reed–Solomon $\mathcal{A}(2^6; 64, 22, 43)$. Based on these codes, we construct an ordinary concatenated code $\mathcal{C}_2$ of length $n = 2048$ and dimension $k = 132$.

This code construction is visualized in Figure 5.8a, the corresponding encoding process is illustrated in Figure 5.9. Matrix I represents the information. The dimension of the outer



(a) Ordinary concatenated code $\mathcal{C}_2$ using inner code $\mathcal{B}(32, 6, 16) = \mathcal{RM}(1, 5)$ and outer code $\mathcal{A} = \mathcal{RS}(2^6; 64, 22, 43)$.



(b) Ordinary concatenated code $\mathcal{C}_3$ using inner code $\mathcal{B}(32, 6, 16) = \mathcal{RM}(1, 5)$ and outer code $\mathcal{A} = \mathcal{RS}(2^6; 36, 22, 15)$.

Figure 5.8: Ordinary concatenated codes based on Reed–Muller and Reed–Solomon codes.

Reed–Solomon code is 22, which is the number of rows of the information matrix. Since the code is defined over the extension field $\mathbb{F}_{2^6}$, each information symbol can be represented by 6 bits, visualized by the number of columns of the information matrix. Hence, the 22 information symbols can be represented by $22 \cdot 6$ bits (Matrix *I*). The symbols of this matrix are encoded using the outer Reed–Solomon code $\mathcal{A}(2^6; 64, 22, 43)$ in step (*a*). The corresponding codeword is represented by Matrix *II*, in which each row is one of the 64 symbols of the codeword. Again, each symbol is represented by using 6 bits. In a second step (*b*), the inner code is applied row-wise to Matrix *II*. Row i of Matrix *III* represents the 32-bit codeword of the inner Reed–Muller code $\mathcal{B}(32, 6, 16)$, that corresponds to the 6-bit information of row i in Matrix *II*. Hence, in total we obtain a codeword of length $64 \cdot 32 = 2048$ of the ordinary concatenated code $\mathcal{C}_2(2048, 132)$.

Before we continue with the explanation of the decoding process, we want to use this example to clarify the difference between ordinary and generalized concatenated codes. In contrast to generalized concatenated codes, when using ordinary concatenated codes there exists no partitioning of the inner code. The decoding result of the outer code (cf. Figure 5.9, Matrix *II*) is directly used to encode row-wise by using an encoder for the inner code. Using instead a generalized concatenated code, the rows are used to choose a codeword of the inner code, by following the corresponding path from the root to a leaf in the partition tree (recall Section 5.1.2, Table 5.2).

For decoding of code $\mathcal{C}_2$, assume Matrix *III* in Figure 5.9 plus an error to be the received word. We know, that each row in that matrix is a codeword of $\mathcal{B}(32, 6, 16) = \mathcal{RM}(1, 5)$ plus an error. Hence, we decode row-wise to obtain the corresponding codewords. We extract the information from each codeword, thereby we have recovered the rows of Matrix *II*. Like in the encoding process each row is a symbol of the field $\mathbb{F}_{2^6}$. Hence, in Matrix *II* we have a codeword of the outer Reed–Solomon code $\mathcal{A}$ (plus an error in case decoding of the Reed–Muller codewords was done erroneously). Decoding of the Reed–Solomon code and extract the information results in the information matrix (cf. Matrix *I*).

We analyze the block error probability P_{err} of code $\mathcal{C}_2(2048, 132)$. The inner code, by applying maximum likelihood decoding, transforms the BSC with bit flip probability $p = 0.14$ into a binary error and erasure channel. We performed simulations on the inner code to obtain the probability $\mathrm{P}(\text{error}) = 0.00317$ for the occurrence of an error as well as the prob-

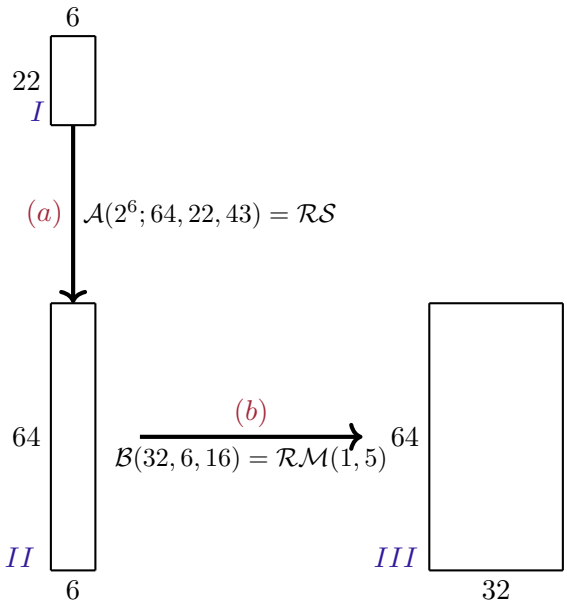


Figure 5.9: Illustration of the encoding steps using the ordinary concatenated code Figure 5.8a. First, 22 symbols from $\mathbb{F}_{2^6}$ (represented by 6 Bits each, Matrix *I*) are encoded in $\mathcal{A}$. Second, each symbol of the resulting codeword (Matrix *II*) is encoded in $\mathcal{B}$.

ability P(erasure) = 0.017605 for the occurrence of an erasure. Analog to the analysis of the generalized concatenated code based on Reed–Muller codes in Section 5.1.3 (cf. Equations 5.19 – 5.21), we calculated a block error probability $P_{\text{err}} \approx 6.79 \cdot 10^{-37}$ for our ordinary concatenated code.

Note that the block error probability $P_{\text{err}} \approx 6.79 \cdot 10^{-37}$ is much smaller that the required 10^{-9}. Hence, we can try to further decrease the codeword length n. Due to the flexibility of Reed–Solomon codes, this can be done arbitrarily. We replace the outer code $\mathcal{A}(2^6; 64, 22, 43)$ by an $\mathcal{RS}(2^6; 36, 22, 15)$ code as visualized in Figure 5.8b, in order to obtain the ordinary concatenated code $\mathcal{C}_3(1152, 132)$. Encoding and Decoding works exactly as described above. Even the same decoder as for $\mathcal{RS}(2^6; 64, 22, 43)$ can be used, when declaring some of the codeword positions as erasures. Using this modification, the code length reduces to

$$n = n_o \cdot n_i = 36 \cdot 32 = 1152. \tag{5.25}$$

By also modifying the analysis described above, we obtain block error probability

$$P_{\text{err}} \approx 1.19 \cdot 10^{-10}. \tag{5.26}$$

By changing the parameters of the Reed–Solomon code, a trade-off between codeword length and block error probability can be adjusted. The dimension can also be adjusted, e.g., to counteract low entropy sources.

Generalized Concatenated Code Construction

Using the ordinary concatenated codes $\mathcal{C}_2$ and $\mathcal{C}_3$, constructed as described in the previous paragraphs, enables to reduce the codeword length significantly. Note that in [MVHV12], a concatenated code of length 2226 was used. Our code $\mathcal{C}_3(1152, 132)$ does not only have a shorter codeword length, but even a smaller block error probability. This paragraph presents a generalized concatenated code, that further decreases the codeword length to 1024.

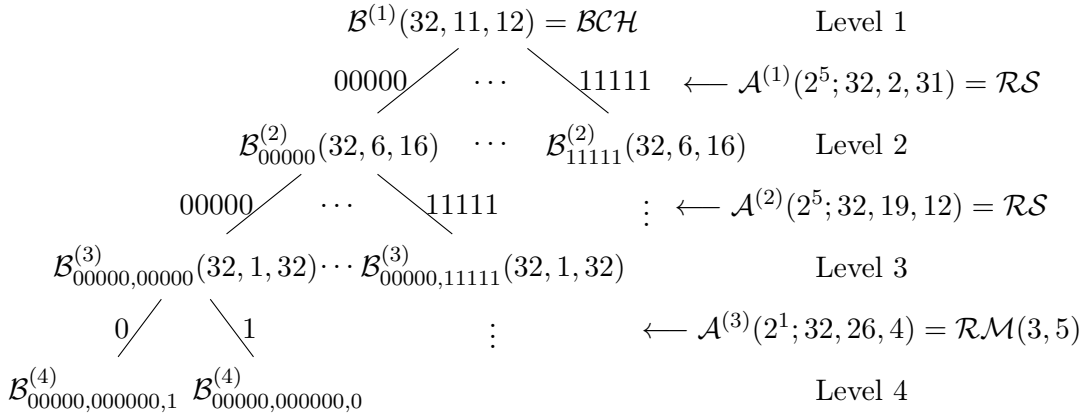


Figure 5.10: Partition tree of the generalized concatenated code $\mathcal{C}_4$. The inner code $\mathcal{B}^{(1)}$ is partitioned, the labeling of the partitions is protected by outer Reed–Solomon and Reed–Muller codes, respectively.

Let $\mathcal{C}_4$ denote the generalized concatenated code constructed in this paragraph. Figure 5.10 represents the partition tree, which consists of four levels. The inner code used in this construction is an extended BCH code of length 32, dimension 11 and minimum distance 12. This code is denoted by $\mathcal{B}^{(1)}$ and partitioned into 32 sub-codes of $2^6 = 64$ codewords each. The labels on the edges from level 1 to level 2 identify these partitions and are protected by a Reed–Solomon code $\mathcal{A}^{(1)}(2^5; 32, 2, 31)$. Each of the sub-codes in level 2 is further divided into 32 sub-codes, each containing $2^1 = 2$ codewords. The first partition in level 3 is the repetition code of length 32, the others partitions are the corresponding disjoint cosets of this repetition code. The labeling of the edges from level 2 to level 3 is protected by the outer Reed–Solomon code $\mathcal{A}^{(2)}(2^5; 32, 19, 12)$. Finally, the cardinality-2 partitions in level 3 are further divided into two sub-partitions each. The sub-codes in level 4 contain only one codeword each. Hence a 1-bit labeling, which is protected by the outer Reed–Muller code $\mathcal{A}^{(3)}(2^1; 32, 26, 4)$, is sufficient to identify these codewords.

Encoding is visualized in Figure 5.11. In total, 131 bits of information are encoded by using three outer codes. The first 10 bits, represented by Matrix I, are encoded by using the Reed–Solomon code $\mathcal{A}^{(1)}(2^5; 32, 2, 31)$. Each of the two rows is interpreted as the binary representation of a symbol of the finite field $\mathbb{F}_{2^5}$, over which the Reed–Solomon code $\mathcal{A}^{(1)}$ is defined. The encoding result is represented by Matrix IV, again each row is interpreted as the binary representation of a symbol from $\mathbb{F}_{2^5}$. The second part of the information, 95 bits, is arranged in the 19×5 Matrix II. These information bits are encoded by the outer Reed–Solomon code $\mathcal{A}^{(2)}(2^5; 32, 19, 12)$ and the result is represented by Matrix V. Again, the rows of Matrices II and V are interpreted as the binary representation of symbols from $\mathbb{F}_{2^5}$. The last

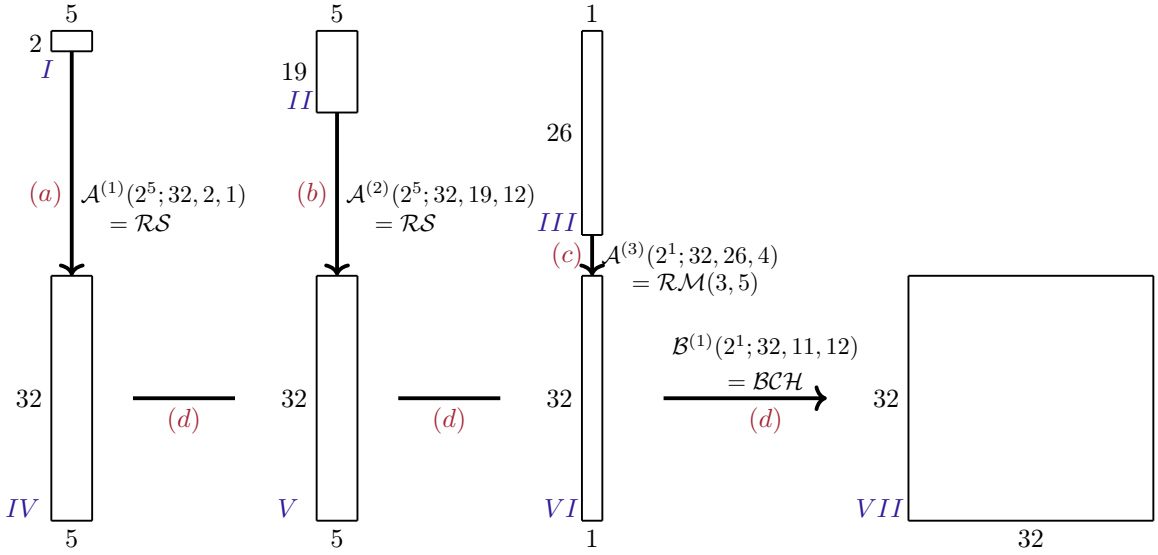


Figure 5.11: Encoding of generalized concatenated code $\mathcal{C}_4(1024, 131)$.

26 information bits, visualized in Matrix III, are encoded by the outer code $\mathcal{A}^{(3)}(2^1; 32, 26, 4)$, which is the Reed–Muller code $\mathcal{RM}(3, 5)$. Matrix VI represents the encoding result. In contrast to the previous matrices, the rows are interpreted as bits, since a binary code is used for encoding. Finally, a codeword of code $\mathcal{C}_4$ is generated by reading Matrices IV, V and VI row-wise. Note that the 5 bits in a row of Matrix IV select an edge from the root (inner code $\mathcal{B}^{(1)}$) to a partition in level 2 of the partition tree (cf. Figure 5.10). Similarly, the corresponding row in Matrix V selects an edge from the chosen level 2 partition to a partition in level 3. The bit in the corresponding row of Matrix VI provides the last part of the path to a leaf of the partition tree, which represents a codeword. The codeword chosen by this path through the partition tree is placed into the corresponding row of Matrix VII. After repeating this process for all of the 32 rows of Matrices IV, V and VI, Matrix VII provides a codeword of the generalized concatenated code $\mathcal{C}_4(1024, 131)$.

In contrast to the generalized concatenated code $\mathcal{C}_1$ proposed in Section 5.1.2, the construction of code $\mathcal{C}_4$ uses three outer codes. Hence, decoding consists of three stages. The description of these three stages as well as of the corresponding steps are given in Table 5.5. A detailed description of the decoding process in the text is omitted, since the structure is exactly the same as for code $\mathcal{C}_1$, for which an extensive explanation was given in Section 5.1.2. Note that due to the low rates of the used Reed–Solomon codes, power decoding as shown in Section 5.1.1 is suitable for decoding.

Analysis of the error correction capabilities of code $\mathcal{C}_4$ works analog to the analysis of code $\mathcal{C}_1$ in Section 5.1.2. Recall, that we denote by S_i the event that decoding fails in stage i of the decoding process. Using code $\mathcal{C}_4$, we have three decoding steps. Hence, the events S_1, S_2, and S_3 have to be considered in the analysis of the block error probability. In stage 1 of the decoding process, maximum likelihood decoding in the inner code $\mathcal{B}^{(1)}$ is efficiently possible since $|\mathcal{B}^{(1)}| = 2048$. The BSC with $p = 0.14$ is transformed into an error and erasure channel.

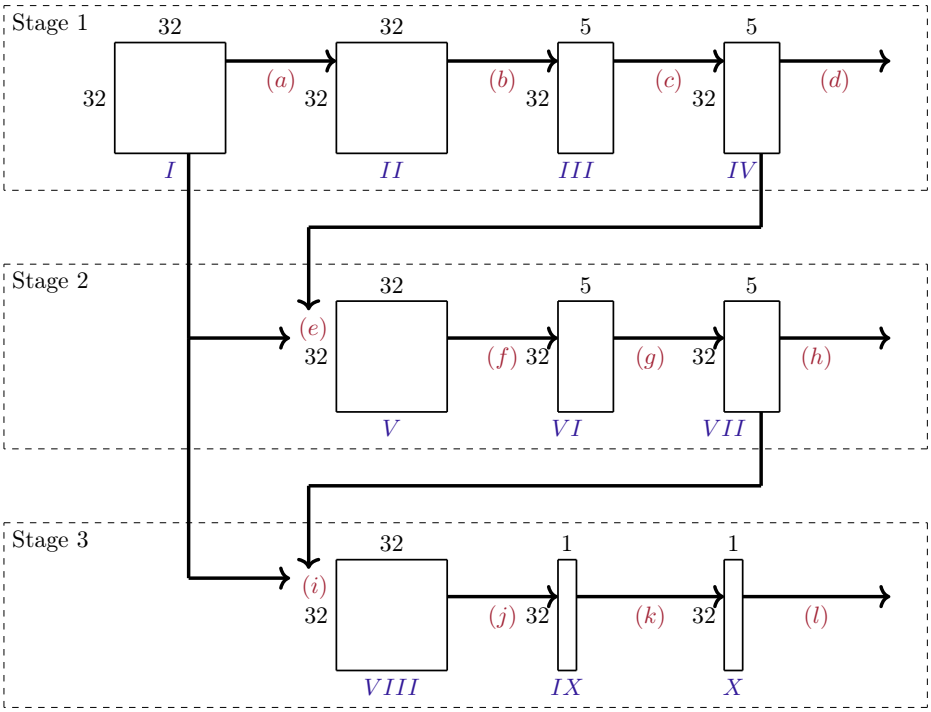


Figure 5.12: Illustration of the algorithm, used to decode the generalized concatenated code $\mathcal{C}_4$. Decoding consists of three stages, since three outer codes are used in the code construction.

By simulations we obtain the probabilities

$$\begin{aligned} \mathrm{P}(\text{error}) &= 0.037808 \text{ and} \\ \mathrm{P}(\text{erasure}) &= 0.174488. \end{aligned} \tag{5.27}$$

for the occurrence of an error or erasure, respectively. We obtain

$$\mathrm{P}(S_1) \approx 1.03 \cdot 10^{-8} \tag{5.28}$$

when applying half the minimum distance decoding. Note that this decoding error probability is too high for the given requirements, however it can be decreased to

$$P(S_1) \approx 1.48 \cdot 10^{-11} \tag{5.29}$$

when using power decoding as explained in Section 5.1.1. In stage 2, decoding in Reed–Solomon code $\mathcal{A}^{(2)}(2^5; 32, 19, 12)$ transforms the BSC in an error and erasure channel with

$$\begin{aligned} \mathrm{P}(\text{error}) &= 0.0032167 \text{ and} \\ \mathrm{P}(\text{erasure}) &= 0.0175397, \end{aligned} \tag{5.30}$$

again determined by simulations. The resulting probability that event S_2 occurs, and hence, decoding in stage 2 fails is

$$\mathrm{P}(S_2) \approx 3.11 \cdot 10^{-10}. \tag{5.31}$$

Finally, for stage 3 of the decoding process we obtain

$$\mathrm{P}(S_3) \approx 2.13 \cdot 10^{-11}. \tag{5.32}$$

In total, the block error probability P_{err} is upper bounded by

$$P_{\mathrm{err}} \leq \mathrm{P}(S_1) + \mathrm{P}(S_2) + \mathrm{P}(S_3) \approx 3.47 \cdot 10^{-10}, \tag{5.33}$$

which fulfills the stated requirement.

Summary on Block Codes for PUFs

Table 5.6 summarizes the results, including the ones from Section 5.1.2 (that are already included in Table 5.4) in order to give a comprehensive summary. Basis of our code constructions is [MVHV12], which suggests an ordinary concatenated code $\mathcal{C}(2226, 174)$, constructed by using a BCH code as outer code, and a repetition code as inner code. In the PUF scenario, a BSC with $p = 0.14$ is used and a block error probability of $P_{\mathrm{err}} \approx 10^{-9}$ is obtained. The largest field used for that construction is $\mathbb{F}_{2^8}$, due to the BCH code applied in the construction.

In a first approach towards improving these results, we use generalized code concatenation in order to construct code $\mathcal{C}_1(2048, 131)$ based on Reed–Muller codes, which are chosen due to the existence of efficient decoding algorithms that are also suitable for efficient hardware implementations. Applying code $\mathcal{C}_1$ enables to decrease the codeword length from 2226 in [MVHV12] to 2048. Also the block error probability decreases from $\approx 10^{-9}$ to $\approx 5.3710^{-10}$ when GMD decoding is applied. A further advantage of code $\mathcal{C}_1$ is, that only the binary field is used.

We further improve the code construction by exploiting the flexibility of Reed–Solomon codes. We propose two ordinary concatenated codes $\mathcal{C}_2$ and $\mathcal{C}_3$, based on Reed–Solomon codes. $\mathcal{C}_2$ uses a Reed–Solomon code of length 64 and dimension 22 over the field $\mathbb{F}_{2^6}$ as outer code. As inner code, Reed–Muller code $\mathcal{RM}(1, 5)$ of length 32 and dimension 6 is used. The result of the construction $\mathcal{C}_2$ also is a code of length 2048 (same length as code $\mathcal{C}_1$), however the block error probability is significantly reduced from $\approx 10^{-10}$ to $\approx 6.79 \cdot 10^{-37}$. Note that operations over the field $\mathbb{F}_{2^6}$ have to be implemented for code $\mathcal{C}_2$. Since a block error probability in the range of 10^{-37} is much smaller than needed, a shorter Reed–Solomon code can be applied. We follow this idea, when constructing the ordinary concatenated code $\mathcal{C}_3$. In contrast to code $\mathcal{C}_2$, the codeword length of the outer Reed–Solomon is reduced from 64 to 36, by keeping the dimension of 22. Code $\mathcal{C}_3$ has codeword length 1152 and obtains an error probability of $\approx 1.19 \cdot 10^{-10}$. Note that 1152 is almost half of the codeword length used in [MVHV12]. Due to the flexibility of Reed–Solomon codes, a trade-off between n and P_{err} can be found according to the specification of a given application.

Finally, we suggest a generalized concatenated code $\mathcal{C}_4$, based on outer Reed–Solomon and Reed–Muller codes, as well as on an extended BCH code as inner code. Using $\mathcal{C}_4$ further decreases the codeword length to 1024. We emphasize that this is half of the lengths of codes $\mathcal{C}_1$ and $\mathcal{C}_2$, and less than half of the length used in the initial construction given in [MVHV12]. Disadvantage of code $\mathcal{C}_4$ in comparison to code $\mathcal{C}_1$ is the field size $\mathbb{F}_{2^5}$. To choose between the codes $\mathcal{C}_1$ and $\mathcal{C}_2$, a decision can be made by comparing codeword length over the field size.

Implementations of some of the code constructions proposed in this section exist and were done by research groups familiarly with hardware implementations. In [Kür14] two software

implementations as well as a hardware implementation of the generalized concatenated code $\mathcal{C}_1$, based on Reed–Muller codes (cf. Section 5.1.2), were realized. The hardware implementation of the decoder leads to an area reduction of about 37% in comparison to [MVHV12]. [HKS+15] presents an area-optimized VLSI implementation of the Reed–Muller based generalized concatenated code $\mathcal{C}_1$ proposed in Section 5.1.2. The decoder implements the steps shown in Figure 5.7 and Table 5.3, respectively. To decode the inner codes, standard array decoders are used [Bos99, Chapter 1.3]. For the outer $\mathcal{RM}(1,7)$ and $\mathcal{RM}(4,7)$ codes, Reed-Decoders are applied, since they are very memory efficient [Ree54]. For a detailed description of the functionality and architecture of the decoders we refer to [HKS+15, Chapters 4–5]. In contrast to earlier implementations of Reed–Muller decoders for PUFs, e.g. [MTV09b], no large intermediate results have to be stored due to a serialized decoder architecture. In comparison to the considered reference implementation in [MVHV12], up to 50% less slices are needed for the decoder at the cost of a small increase of the key error probability, which is $1.49 \cdot 10^{-9}$ (cf. upper bound derived in Section 5.1.2). In [Man18], a completely modularized PUF coding chain implemented on a system on chip (SoC) is developed. For the channel coding module, also the generalized concatenated code $\mathcal{C}_1$ from Section 5.1.2 is used. Results of that work have been published in [MHK+19].

Table 5.3: Description of the decoding process of $\mathcal{C}_1$ and legend to Figure 5.7. $\otimes$ denotes an erasure, $\otimes^\ell$ denotes a sequence of length ℓ consisting of erasures.

Stage	Block/Step	Description
1	I	(128×16)–matrix containing the received word. Rows are codewords of $\mathcal{B}^{(1)} = \mathcal{RM}(1,4)$ plus error.
	(a)	Row–wise ML decoding in $\mathcal{B}^{(1)}$. Result: $c \in \mathcal{B}^{(1)}$ or $\otimes^{16}$.
	II	Result of step (a). Rows are codewords of $\mathcal{B}^{(1)}$ or $\otimes^{16}$.
	(b)	Remapping of every row (codewords of $\mathcal{B}^{(1)}$) to index (4 bits) of the partition which contains the codeword. If erasure: $\otimes^4$.
	III	Result of step (b). Rows are $\in \{0,1\}^4 \cup \{\otimes^4\}$.
	(c)	Column–wise error–erasure decoding (optional: GMD using soft information obtained from step (a)) in $\mathcal{A}^{(1)}(128,8,64) = \mathcal{RM}(1,7)$. If decoding fails: Declare failure of algorithm.
	IV	Result of step (c). Columns are codewords of $\mathcal{RM}(1,7)$. Rows give indices i (4 bits) to specify the partition $\mathcal{B}_i^{(2)}$ in which the rows must be decoded in step (e).
	(d)	Extraction of the first $32 = 4 \cdot 8$ information bits (each column of IV is a codeword of a $\mathcal{A}^{(1)}(128,8,64)$ code which corresponds to exactly one information word of length 8).
2	(e)	Row–wise ML decoding in $\mathcal{B}_i^{(2)}$ (i denotes the partition index for each row given by the corresponding row of Matrix IV).
	V	Result of step (e). Rows are codewords of $\mathcal{B}_i^{(2)}$ or $\otimes^{16}$.
	(f)	Remapping of every row (codewords of $\mathcal{B}_i^{(2)}$) to index j (1 bit) of the partition $\mathcal{B}_{i,j}^{(3)}$ of $\mathcal{B}_i^{(2)}$ which contains the codeword. If erasure: $\otimes$.
	VI	Result of step (f).
	(g)	Error–erasure decoding (optional: GMD using soft information obtained from step (e)) of the column in $\mathcal{A}^{(2)} = \mathcal{RM}(4,7) = \mathcal{C}(128,99,8)$. If decoding fails: Declare failure of algorithm.
	VII	Result of step (g). Column contains codeword of $\mathcal{A}^{(2)}$.
	(h)	Extraction of remaining 99 information bits which correspond to the $\mathcal{A}^{(2)}(128,99,8)$ codeword in Matrix VII.

Table 5.5: Description of the decoding process of $\mathcal{C}_4$ and legend to Figure 5.12. $\otimes$ denotes an erasure, $\otimes^\ell$ denotes a sequence of length ℓ consisting of erasures.

Stage	Block/Step	Description
1	I	(32×32)–matrix containing the received word. Rows are codewords of $\mathcal{B}^{(1)}(2^1; 32, 11, 12)$ plus error.
	(a)	Row–wise ML decoding in $\mathcal{B}^{(1)}$. Result: $c \in \mathcal{B}^{(1)}$ or $\otimes^{32}$.
	II	Result of step (a). Rows are codewords of $\mathcal{B}^{(1)}$ or $\otimes^{32}$.
	(b)	Remapping of every row (codewords of $\mathcal{B}^{(1)}$) to index (5 bits) of the partition which contains the codeword. If erasure: $\otimes^5$.
	III	Result of step (b). Rows are $\in \{0,1\}^5 \cup \{\otimes^5\}$.
	(c)	Error–erasure decoding in $\mathcal{A}^{(1)}(2^5; 32, 2, 1)$. If decoding fails: Declare failure of algorithm.
	IV	Result of step (c). Codewords of $\mathcal{A}^{(1)}$. Rows give indices i (5 bits) to specify the partition $\mathcal{B}_i^{(2)}$ in which the rows must be decoded in step (e).
	(d)	Extraction of the first $10 = 2 \cdot 5$ information bits.
2	(e)	Row–wise ML decoding in $\mathcal{B}_i^{(2)}$ (i denotes the partition index for each row given by the corresponding row of Matrix IV).
	V	Result of step (e). Rows are codewords of $\mathcal{B}_i^{(2)}$ or $\otimes^{32}$.
	(f)	Remapping of every row (codewords of $\mathcal{B}_i^{(2)}$) to index j (5 bits) of the partition $\mathcal{B}_{i,j}^{(3)}$ of $\mathcal{B}_i^{(2)}$ which contains the codeword. If erasure: $\otimes^5$.
	VI	Result of step (f).
	(g)	Error–erasure decoding in $\mathcal{A}^{(2)}(2^5; 32, 19, 12)$. If decoding fails: Declare failure of algorithm.
	VII	Result of step (g). Matrix contains codeword of $\mathcal{A}^{(2)}$.
	(h)	Extraction of $95 = 5 \cdot 19$ information bits.
3	(i)	Row–wise ML decoding in $\mathcal{B}_{i,j}^{(3)}$ (j denotes the partition index for each row given by the corresponding row of Matrix VII).
	$VIII$	Result of step (i). Rows are codewords of $\mathcal{B}_{i,j}^{(3)}$ or $\otimes^{32}$.
	(j)	Remapping of every row (codewords of $\mathcal{B}_{i,j}^{(3)}$) to index k (1 bit) of the partition $\mathcal{B}_{i,j,k}^{(4)}$ of $\mathcal{B}_{i,j}^{(3)}$ which contains the codeword. If erasure: $\otimes$.
	IX	Result of step (f).
	(k)	Error–erasure decoding in $\mathcal{A}^{(3)}(2^1; 32, 26, 4)$. If decoding fails: Declare failure of algorithm.
	X	Result of step (g). Matrix contains codeword of $\mathcal{A}^{(3)}$.
	(l)	Extraction of the remaining 26 information bits.

Table 5.6: Comparison between the code constructions $\mathcal{C}_1, \ldots, \mathcal{C}_4$ proposed in this chapter and [MVHV12]. P_{err} is the upper bound of the block error probability. in case of $\mathcal{C}_1$, additionally the block error probability obtained by simulations using Generalized Minimum Distance (GMD) decoding, is stated.

Code (Section)	P_{err}	n	k	$\frac{k}{n}$	Largest Field
[MVHV12]	$\approx 10^{-9}$	2226	174	0.078	$\mathbb{F}_{2^8}$
GC RM $\mathcal{C}_1$ (Section 5.1.2)	$\approx 1.48 \cdot 10^{-9}$	2048	131	0.064	$\mathbb{F}_2$
GMD simulations	$\approx 5.37 \cdot 10^{-10}$				
RS $\mathcal{C}_2$ (Section 5.1.3)	$\approx 6.79 \cdot 10^{-37}$	2048	132	0.064	$\mathbb{F}_{2^6}$
RS $\mathcal{C}_3$ (Section 5.1.3)	$\approx 1.19 \cdot 10^{-10}$	1152	132	0.115	$\mathbb{F}_{2^6}$
GC RS $\mathcal{C}_4$ (Section 5.1.3)	$\approx 3.47 \cdot 10^{-10}$	1024	131	0.128	$\mathbb{F}_{2^5}$

5.2 Convolutional Codes for PUFs

Section 5.1 was dealing with the construction of block codes for error correction in the context of PUFs. This section, on the other hand, considers convolutional codes and is structured as follows: In Section 5.2.1, an overview about studies that previously applied convolutional codes to PUFs is given. Weaknesses are revealed and possible improvements are outlined. Section 5.2.2 briefly introduces convolutional codes as well as related techniques, which are used in the remainder of this chapter. Section 5.2.3 contains approaches to improve reliability when applying convolutional codes for PUFs as one of the chapter's main contributions. Simulation results are presented and compared to results from literature. In this section, SRAM PUFs are assumed as underlying PUF construction. Responses are generated according to the theoretical model from [MTV09a], which was already revisited in Chapter 3 in the context of channel and error models. In Section 5.2.4, convolutional codes are applied to the ROPUF data from [MCMS10]. We show that this code class is a reasonable choice for an error correction component, when considering the given ROPUFs. Finally, Section 5.2.5 summarizes our work about using convolutional codes for PUFs. The results obtained by using the approaches discussed in Section 5.2.3 have been published in [MPB18b], while the results using the real-world ROPUF data in Section 5.2.4 have been published in [MB17b].

5.2.1 Error Correction for PUFs Using Convolutional Codes

This section briefly summarizes previous studies from literature that apply convolutional codes to PUFs. In [HWRL+13], convolutional codes are used for the first time in the context of PUFs. The reasons for choosing them are twofold: First, they have good error correction properties. Second, they are easily implementable in hardware. For example, [HLS14] proposes a hardware implementation of the Viterbi decoding algorithm, particularly designed for PUFs that are realized on FPGAs. This implementation is optimized concerning chip area and power consumption.

[HÖSB16] constitutes the basis of our work with convolutional codes. The authors aim at decreasing the decoding failure rate and hence, increasing the key reliability, by essentially two approaches: First, reliability information about the several response bits are known and used to select only those response bits which possess a certain reliability. Thereby, the entire channel is improved by consciously ignoring unreliable bits. Second, the reliability of the decoding output is estimated by applying an algorithm called "simplified ROVA" [FH07]. This algorithm is combined with the concept of multiple readouts. A response is extracted to produce a key as usual. However, if the simplified ROVA algorithm detects that the decoding result is not reliable enough, a defined amount of new readouts is extracted and the most reliable one is selected as result. Alternatively, a certain amount of readouts is performed directly, and the one which is classified as most reliable by simplified ROVA is chosen. Thus, [HÖSB16] successfully increases the reliability of key reproduction, however, at increased cost. First, more response bits than needed have to be extracted from the PUF, such that a selection of enough reliable bits can be performed. Hence, the PUF must support the generation and extraction of sufficiently many response bits, which requires an appropriate chip area. Also, depending on the PUF construction, extraction of bits can be time-consuming, and hence, extracting less bits in general is preferred over extracting a larger number of bits. Furthermore, in order to evaluate the reliability of the decoding result,

simplified ROVA operates on the reliabilities of the several bits. Calculating reliabilities on a bit level is an overhead, considering that we are interested in the reliability of the whole word.

We present approaches that can be used to improve the results of [HÖSB16]. First of all, it has to be emphasized, that in all of the mentioned studies that deal with convolutional codes for PUFs ([HWRL+13, HLS14, HÖSB16]), hard-decision decoding using the Viterbi algorithm, is applied. Our approaches use reliability information not only to select reliable response bits, but additionally to generate soft information that is used at the input of the Viterbi algorithm. Thereby, the decoding failure rate can be decreased, or alternatively more unreliable bits can be included in favor of wasting less extracted response bits in order to obtain the same decoding failure probability. In all our approaches we do not use simplified ROVA, since we consider the algorithm to be computational overhead. Instead, we apply list decoding by using a variant of the Viterbi algorithm that was proposed in [SSZB04]. We further improve the results obtained by these methods by combining them with multiple readouts. In our simulations, a behavior that is well-known in coding theory shows up. Using convolutional codes with a large memory length results in a smaller error probability. Also, the results in [HÖSB16] indicate, that it would be advisable to increase the memory length. However, the complexity of Viterbi algorithm grows exponentially with the memory length and thus makes the use of large memory lengths impractical. This problem can be circumvented by applying sequential decoding, that has a complexity which is independent of the memory length and rather depends on the number of errors in the received sequence. Due to this advantage, we apply sequential coding, in particular the Fano algorithm as proposed in [Fan63].

Some of the techniques used in this section are novel in the area of PUFs. Sequential decoding and Viterbi list decoding are used for the first time in the context of PUFs. Soft information at the input of the decoding algorithm was already used in the PUF scenario in [MTV09b], however, for a concatenation of short block codes instead of convolutional codes. Applying soft information at the decoder's input for convolutional codes was briefly implied in [MTV09b], however, convolutional codes were generally classified as inappropriate, since comparatively short code sequences are used when dealing with PUFs, and convolutional codes are known to present their full strength for long code sequences. However, [HWRL+13, HLS14, HÖSB16] proposed to use convolutional codes for PUFs, thus showing their practicability in that scenario.

5.2.2 Convolutional Codes

Convolutional codes were introduced in [Eli55] and became a popular code class that is often used in practical applications as for example GSM, UMTS and LTE. Their popularity essentially arises from the following properties: First, the Viterbi algorithm, used for decoding of convolutional codes, provides efficient maximum-likelihood decoding. Second, it is comparatively simple to use soft information at the decoder's input as well as to generate soft output. Third, encoding and decoding routines are very suitable for hardware implementations, what makes them also useful in the field of PUFs. Extensive theory about convolutional codes can be found in the literature [JZ15, Bos99, LC04]. This section briefly summarizes the knowledge that is necessary in order to follow the contents in the remainder of this chapter. Table 5.7 lists the convolutional codes used for simulations within this chapter. Subsequently, the code

Table 5.7: Convolutional codes used in our simulations to obtain the results presented in this chapter. Usually good convolutional codes are determined by computer search and listed in tables, e.g. [Lar73, Joh75].

			polynomials used (in octal)		
k	n	ν	g_2	g_1	g_0
1	2	2		5	7
1	2	6		133	171
1	2	7		247	371
1	2	10		3645	2671
1	2	14		63057	44735
1	2	16		313327	231721
1	3	6	133	165	171
1	3	7	225	331	367
1	3	8	557	663	711
1	3	9	1117	1365	1633
1	3	10	2353	2671	3175

in the first row of this table is used as an example to illustrate the concepts applied in this section.

Analog to block codes, the information sequence is partitioned into blocks of length k, and each block is mapped to a codeword of length $n > k$. Convolutional codes differ from block codes, since generating codeword $\boldsymbol{c}_r$ does not solely depend on information block $\boldsymbol{i}_r$, but in addition on the μ previous information blocks $\boldsymbol{i}_{r-1}, \ldots, \boldsymbol{i}_{r-\mu}$.

The encoding process of block codes is generalized to

$$\boldsymbol{c}_r = \boldsymbol{i}_r \boldsymbol{G}_0 + \boldsymbol{i}_{r-1} \boldsymbol{G}_1 + \cdots + \boldsymbol{i}_{r-\mu} \boldsymbol{G}_\mu, \tag{5.34}$$

where $\boldsymbol{G}_0, \boldsymbol{G}_1, \ldots, \boldsymbol{G}_\mu$ are $(k \times n)$ generator matrices. Equivalently, this can be written as $\boldsymbol{c}_r = (\boldsymbol{i}_r, \boldsymbol{i}_{r-1}, \ldots \boldsymbol{i}_{r-\mu}) \cdot (\boldsymbol{G}_0^\top, \boldsymbol{G}_1^\top, \ldots, \boldsymbol{G}_\mu^\top)^\top$. An encoder for a convolutional code of rate $R = \frac{k}{n}$ can be implemented by using a linear shift register with k inputs and n outputs. The information sequence is divided into k subsequences, one for each input of the shift register. When processing a bit at each input, the preceding bits are shifted by one position in the memory elements that follow the corresponding inputs of the shift register. In most cases, the memory elements are pre-initialized with zeros. In each step, one bit per output is produced. The number of memory elements at input i is called the *constraint length* ν_i of input i. The total sum of memory elements in the encoder is denoted as *overall constraint length* $\nu = \sum_{i=1}^{k} \nu_i$. The *memory length* is defined as $\mu = \max_i\{\nu_i\}$ and can be interpreted as the number of previous blocks that is used to encode a certain information block. Within this work only convolutional codes of rate $R = \frac{1}{2}$ and $R = \frac{1}{3}$ are used. Hence, corresponding encoders have always one input, and either two or three outputs. Note that in this case constraint length, overall constraint length and memory length are the same and the terms can be used interchangeably. The parameters of a convolutional code are specified by the triple $(n, k, [\nu])$.

Figure 5.13 visualizes an encoder of the (2,1,[2]) convolutional code listed in Table 5.7. The

generator matrices implemented by this encoder can be derived from the given generators $g_0 = (7)_8 = (111)_2$ and $g_1 = (5)_8 = (101)_2$ and hence, are $\boldsymbol{G}_0 = (1,1)$, $\boldsymbol{G}_1 = (1,0)$ and $\boldsymbol{G}_2 = (1,1)$. From the perspective of system theory, an encoder for a convolutional code can be described as a linear time-invariant (LTI) system with k inputs an n outputs. The generators g_j are determined by the impulse response at output j. As an alternative to encode by using (5.34), the sequence at output j can be calculated by convolution of the input sequences with the corresponding impulse responses of the system observed at output j.

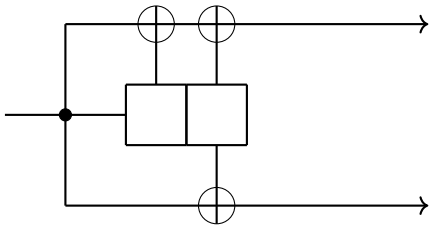


Figure 5.13: Encoder of a rate $R = \frac{1}{2}$ convolutional code with constraint length $\nu = 2$ and generators $g_0 = (7)_8 = (111)_2$, $g_1 = (5)_8 = (101)_2$.

Usually, information sequences of infinite length are assumed in the theory of convolutional codes. However, when considering error correction for PUFs (as well as in a multitude of other practical applications), we consider sequences of finite length. There exist essentially three methods how to deal with finite sequences when using convolutional codes:

1. *Truncation:* After an information sequence of finite length is encoded, the process simply stops to feed any symbols into the encoder. The drawback of that solution is, that the last encoded information bits do not contribute to $\mu + 1$ code symbols and hence, these last encoded bits are less protected.

2. *Termination:* After an information sequence of finite length is encoded, a number of μ zeros is inserted to reset the state of the encoder to its initial state. This repairs the problem caused by truncation, however, the code rate suffers, which is negligible in most practical scenarios, for example when dealing with long information sequences.

3. *Tailbiting:* In contrast to initialize the shift register with zeros, its final state is used for initialization. This requires pre-computation of the final state, thus knowing in advance the complete information sequence.

The Viterbi algorithm is the most often used decoder for convolutional codes. It operates on an undirected graph, called *trellis*. Each path through the trellis represents a code sequence. Basically, every linear code can be represented by such a trellis, but the trellis representation is particularly suited for convolutional codes due to two reasons: First, convolutional codes can be represented in a very compact form, which can be attained by merging edges that represent common substrings of a code sequence. This enables to simultaneously conduct calculations for several code sequences. Second, using convolutional codes, the segments of the trellis are repeating, after a short construction phase. Therefore, the implementation of the data structures and algorithms can efficiently be realized. Note that the smaller the number of nodes and edges in a trellis, the smaller the computational complexity. In order to construct a trellis for a convolutional code, an encoder is transformed into a state diagram,

which in turn is used to derive the trellis. Figure 5.14 visualizes a state diagram for the $(2, 1, [2])$ code given in Table 5.7. Since the encoder in Figure 5.13 has two memory elements, there are four possible states in the state diagram, namely "00", "10", "01", and "11". Each of these states is represented by one node in the state diagram. The diagram contains two outgoing edges per node, one for input "0" (dashed edges) and one for input "1" (solid edges). The edges lead to the state of the register after processing the input bit. The edges are labeled with the output, produced after processing the corresponding input bit. Representing the state diagram unwrapped in time, starting in the all-zero state "00", results in the trellis visualized in Figure 5.15.

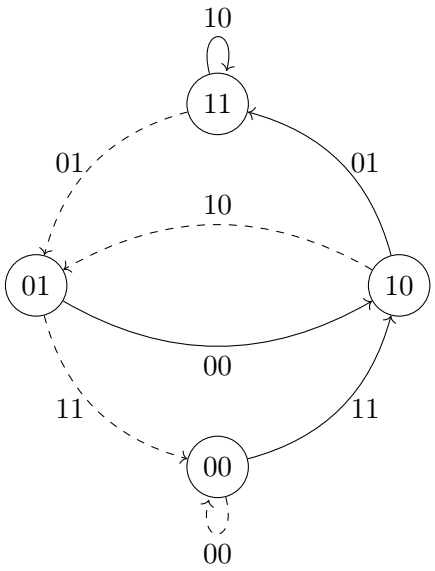


Figure 5.14: State diagram of a rate $R = \frac{1}{2}$ convolutional code with constraint length $\nu = 2$ and generators $g_0 = (7)_8$, $g_1 = (5)_8$. For example, assume the encoder is in state "10" and information bit "1" waits at the input to be encoded. The encoder outputs the code symbols "01" and moves to state "11" due to the right shift that is carried out in the register.

The Viterbi algorithm for decoding convolutional codes can be described based on the trellis representation of the code. Since the Viterbi decoder performs efficient ML decoding, it implements the decoding rule

$$\hat{\boldsymbol{c}} = \underset{\boldsymbol{c} \in \mathcal{C}}{\operatorname{argmax}} P(\boldsymbol{y}|\boldsymbol{c}), \tag{5.35}$$

which aims to find the code sequence which was most likely transmitted when receiving the sequence $\boldsymbol{y}$. For explaining the Viterbi algorithm, we denote the transmitted code sequence by $\boldsymbol{c} = (c_1, c_2, \dots)$ and the received sequence by $\boldsymbol{y} = (y_1, y_2, \dots)$. To find the most likely transmitted code sequence, the decoder aims to identify the path, which represents that sequence in the trellis. An *edge metric* as well as a *node metric* is defined as a measure of the distance between $\boldsymbol{y}$ and a valid code sequence $\boldsymbol{c}$. For all edges, the edge metric is defined as

$$\lambda_i = P(y_i|c_i) = n - \operatorname{dist}_{\mathrm{H}}(c_i, y_i). \tag{5.36}$$

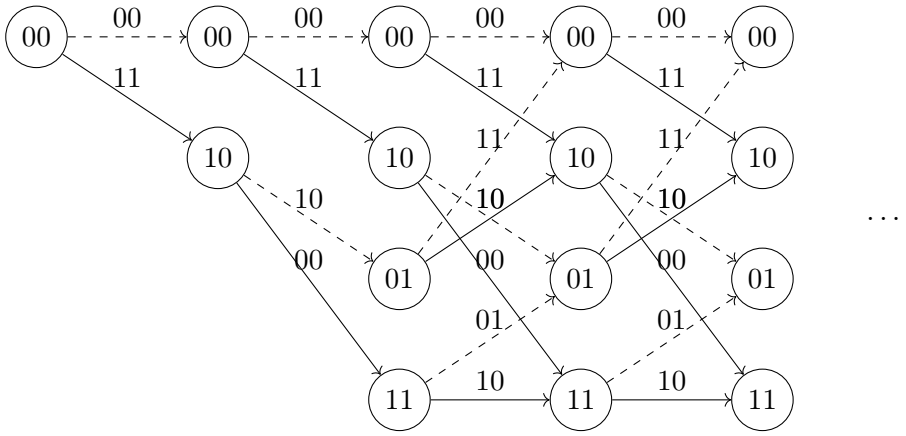


Figure 5.15: Trellis diagram derived from the state diagram in Figure 5.14. Each trellis segment consists of the nodes representing the encoder's states. Analog to the state diagram, two edges leave at each node, representing the information symbols "0" (dashed edges) and "1" (solid edges), respectively. The edges are labeled with the code sequences produced by the decoder. Time proceeds from left to right.

The smaller the distance between c_i and y_i, the larger the edge metric. The node metric is defined as the sum of all edge metrics of edges being part of the most probable path to a considered node v, i.e.,

$$\Lambda = \sum_{i=1}^{\eta} \lambda_i, \tag{5.37}$$

where η denotes the length of the considered path. To calculate the node metric for a specific node v, all incoming edges are considered. For each incoming edge i, the edge metric λ_i is added to the node metric of the node in the previous segment that is connected to the considered edge. The edge which maximizes the node metric for node v is added to the *survivor path* (most likely path) to that node.

An extension of the Viterbi algorithm that allows to perform list decoding was proposed in [SSZB04]. The modification consists of additionally storing the difference between the metric of the survivor and the second likely path in each node. The smaller that number, the less reliable the decision made at that node. After the calculation of the survivor path, we search for the node with the smallest difference value on that path. Since the decision at that node is the most unreliable one, the non-surviving edge of that node is chosen. To generate an alternative path through the trellis, we keep on following that edge back, until the path merges again with the survivor path. To generate a list of $\mathcal{L}$ paths, after updating the differences, we $\mathcal{L}-1$ times apply the strategy of searching the node with the lowest difference on the already chosen paths.

There exist extensions of the Viterbi decoder, which aim for assessing the reliability of the decoding result. *Soft-Output Viterbi Algorithm (SOVA)* is a modification, such that in addition to the most likely path in the trellis, the algorithm calculates the reliabilities of the decoding decisions for each symbol [HH89]. Instead of evaluating the reliability symbol-wise, *Reliability Output Viterbi Algorithm (ROVA)* calculates the reliability of the overall sequence

[RB98]. While ROVA exactly calculates the desired reliability, *simplified ROVA* as introduced in [FH07] only approximates ROVA. Hence, it produces worse values than ROVA in favor of a much lower decoding complexity. Simplified ROVA uses reliability values of the individual symbols to calculate the reliability of the overall sequence. This approach is used in the context of PUFs in [HÖSB16]. Since we do not use extensions to calculate the reliability of the decoded sequence, we omit to go into further detail about their functionality and refer the interested reader to the references, provided above for all three algorithms.

The Viterbi decoder is popular, since it efficiently performs maximum likelihood decoding. However, its complexity grows exponentially with an increasing constraint length ν. Results in [HÖSB16] as well as our own research indicates, that when convolutional codes are applied for PUFs, a comparatively large constraint length is required. The complexity of *sequential decoding*, introduced in [Woz57], instead of the constraint length, depends on the number of errors in a received sequence. In contrast to the Viterbi decoder, sequential decoding does not provide maximum likelihood decoding. Different algorithms for implementing sequential decoding exist. The most famous ones are the *ZJ algorithm*, named after its inventors Zigangirov and Jelinek [Zig66, Jel69], as well as the *Fano algorithm* [Fan63]. We consider the Fano algorithm to be the better choice when working with PUFs, since it is memory efficient and, hence, is suitable for hardware implementations. The ZJ algorithm in contrast is slightly faster when decoding received sequences which include a larger number of errors, however more memory is needed due to the implementation and maintenance of a stack-like data structure. In [JZ15, Section 6.10], the Fano algorithm is denoted to be the "most practical sequential decoding algorithm".

In contrast to the Viterbi algorithm, Fano's algorithm does not operate on a trellis, but uses a code tree as underlying data structure. To provide an example, Figure 5.16 illustrates the code tree which can be constructed from the encoder given in Figure 5.13. Each path starting at the root of the tree represents a code sequence. Each edge is labeled with a code block that results when encoding an information symbol. Being at a node, we follow the edge to its left child node if the next information bit is "0", and the edge to its right child node in cases where the next information bit is "1". For example, the information sequence $(0, 1, 1, \dots)$ results in the code sequence $(00\ 11\ 01\dots)$. Meaningful examples for decoding with the Fano algorithm are extensive and, hence, we refer to [LC04, Chapter 13.2] for this purpose.

The Fano algorithm belongs to the family of iterative backtracking algorithms. In order to perform a depth-first search, the code tree is traversed from the root until a leaf node is reached. The algorithm aims for maximizing the so-called Fano metric

$$M(y_i|c_i) = \begin{cases} \log_2 2p_i - R, & \text{if } y_i \neq c_i \\ \log_2 2(1-p_i) - R, & \text{otherwise,} \end{cases} \tag{5.38}$$

defined for a BSC with bit error probabilities p_i and a code of rate R. Being at a node v, the algorithm decides, based on that metric, to which child node of v it proceeds. Therefore, the metric of the corresponding edge is added to the current value of the metric and the algorithm continues with the next iteration. Based on a threshold T, the algorithm notices when the currently examined path becomes unlikely. If there are paths starting at node v, which were not examined yet, these paths can successively be considered. If all paths that start at node v have already been considered and result in a metric that is lower than threshold T, the

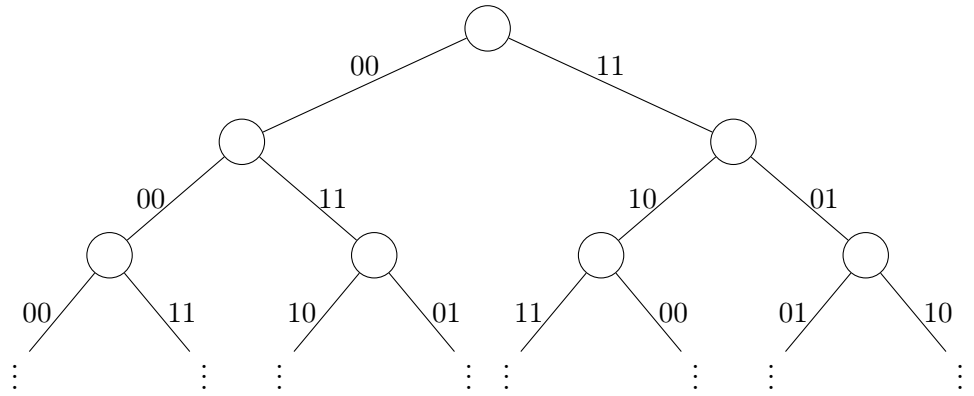


Figure 5.16: The paths in a code tree represent the code sequences, which result when encoding an information sequence. The encoding process starts at the root of the tree. Encoding an information bit "0", we proceed to the left child node and produce the code bits indicated by the label of the corresponding edge. Similar, for encoding a "1", we proceed to the right child node. Sequential decoding algorithms operate on this data structure instead of a trellis diagram.

algorithm performs backtracking to the previous level. If the metric of the predecessor of v is also lower than T, the threshold is lowered by using a threshold increment Γ, that has to be optimized in simulations. When the algorithm visits a node for the first time, the threshold T is increased by the maximum multiple of Γ, such that the value of the metric is still larger than the threshold. For a detailed description of the Fano algorithm including extensive examples, we refer to [LC04, Chapter 13]. In addition to the standard literature about coding theory, which was already mentioned in the introduction of this chapter, details about convolutional codes can be found in [JZ15].

5.2.3 Improving the Reliability when Applying Convolutional Codes for PUFs

Before explaining our approaches, we outline the setup, which forms the underlying basis for all of them. As channel model, a BSC with an individual bit error probability p_{b_i} for each bit position $i = 1, \ldots, n$, as explained in Chapter 3 and visualized in Figure 3.2, is assumed. Recall that this model was derived in [MTV09a], according to the properties of SRAM cells. In all approaches, we extract $s \in \mathbb{N}$ response bits from the PUF. Then we select $n \leq s$ out of the s response bits, which belong to BSCs that are reliable enough. Therefore, a constant p_T is used as threshold, and a response bit i is defined to be reliable, when $p_{\mathrm{b}_i} < p_T$. For example, if $p_T = 0.1$, n response bits that correspond to BSCs with bit error probabilities $p_{\mathrm{b}_i} < 0.1$ are selected. This enables to ignore response bits that are too unreliable and, hence, to improve the overall channel. According to the specification given in [HÖSB16], we aim for a block error probability of at least 10^{-6}. Table 5.7 lists the convolutional codes that were used in simulations verifying our approaches, together with their generators. Usually, good convolutional codes are determined by computer search, a selection of often used candidates can be found in [Bos99, Chapter 8.8]. The implementations of convolutional codes, encoder and the Viterbi decoder for our studies are adopted from a convolutional code library developed by [Sch02], the list Viterbi variant as well as Fano algorithm for sequential decoding were

added within our work. Before we continue with outlining our approaches in the following paragraphs, we want to stress that none of our approaches uses simplified ROVA or other methods that estimate the reliability of a regenerated PUF response.

Approach 1: Using Soft Information at the Input of Viterbi Algorithm

Recall that [HÖSB16] uses reliability information, only to select and to refuse unreliable response bits. In Approach 1, reliability information are additionally used to calculate soft information

$$s_i^{(t)} = (-1)^{h_i \oplus r_i^{(t)}} \cdot (\log(1 - p_{e_i}) - \log(p_{e_i})) \tag{5.39}$$

for each cell i at time instance t. In Formula 5.39, r_i and h_i denote bit i from the response and the helper data, respectively. The probability p_{e_i} denotes the error probability of SRAM cell i. There are different ways how to obtain the error probabilities p_{e_i}. For the experiments considered in this section, p_{e_i} is obtained from the theoretical model described in Chapter 3.1. In a scenario with practical SRAM PUFs, multiple readouts can be performed in an enrollment phase in order to estimate the error probabilities p_{e_i} for the several SRAM cells. As an alternative to multiple readouts during enrollment, reliability information can be gathered by using only one single measurement. This possibility was introduced in the literature [VdLPVdS12]. Note that Formula 5.39 was already given in [MTV09a], however, there it was not applied to convolutional codes, but to soft-decision decoding of short linear block codes.

In our first approach, we use the same scenario as [HÖSB16] and select reliable response bits according to three different threshold values $p_T \in \{0.3, 0.2, 0.1\}$. The difference to the reference simulation is, that we do not use simplified ROVA, but soft information at the input of the Viterbi decoder instead of hard-decision decoding. Results of simulations conducted with this setup are contained in column "**SD**, $\mathcal{L} = 1$"[5] of Table 5.8. When comparing with the results from [HÖSB16] in column 4, it can be noticed, that the block error probabilities are around the same magnitude, or even better in our approach. Consider, for example, threshold $p_T = 0.1$ for convolutional code $(2, 1, [6])$. Note that the block error probability in our approach is $2.15 \cdot 10^{-3}$, which is below the bound of $1.98 \cdot 10^{-2}$ obtained in [HÖSB16]. Similarly, using a $(2, 1, [7])$ code results in block error probability $7.63 \cdot 10^{-4}$, compared to $4.86 \cdot 10^{-3}$ in [HÖSB16]. Therefore, we can conclude, that the use of soft information at the input of Viterbi algorithm is a good candidate to replace the simplified ROVA algorithm.

Approach 2: Additional Use of List Decoding

We extend Approach 1 by replacing the Viterbi decoder by a list decoding variant as proposed in [SSZB04] and explained in Section 5.2.2. Note that, like in all our approaches, soft information at the decoder's input (as explained for Approach 1) are still used. When applying list decoding with a sufficiently large list size $\mathcal{L}$, the threshold p_T can be increased without any loss in the block error probability. To illustrate this statement, we consider for example the convolutional code $(2, 1, [6])$ with threshold $p_T = 0.1$ in Table 5.8. In the reference [HÖSB16], a block error probability upper bounded by $1.98 \cdot 10^{-2}$ is obtained. Using Approach 2, we obtain a block error probability in the same order when increasing the threshold to $p_T = 0.2$

[5] The parameter $\mathcal{L}$ describes the list size when list decoding is performed (cf. Approach 2). $\mathcal{L} = 1$ means a list size of one and is, therefore, simply not a list decoding at all.

Table 5.8: Comparison of block error probabilities. Using rate $\frac{1}{2}$ codes with hard-decision decoding (HD), soft-decision decoding (SD) and list size $\mathcal{L}$. Extraction means the number of simulated key extractions. (* only 500.000 extractions were simulated in [HÖSB16].)

Code	$\mathbf{p_T}$	**Extractions**	**Ref.[HÖSB16]**	**HD**, $\mathcal{L}=3$	**HD**, $\mathcal{L}=4$	**SD**, $\mathcal{L}=1$	**SD**, $\mathcal{L}=3$	**SD**, $\mathcal{L}=4$
(2,1,[6])	0.3	10.000.000	4.11e-01*	6.72e-01	6.66e-01	1.86e-01	1.29e-01	1.27e-01
	0.2	10.000.000	6.98e-02*	1.83e-01	1.80e-01	3.62e-02	2.09e-02	2.05e-02
	0.1	10.000.000	≤1.98e-02	1.04e-02	1.02e-02	2.15e-03	1.09e-03	1.07e-03
(2,1,[7])	0.3	10.000.000	–	4.98e-01	4.91e-01	1.18e-01	7.92e-02	7.74e-02
	0.2	10.000.000	–	8.37e-02	8.18e-02	1.80e-02	1.01e-02	9.90e-03
	0.1	10.000.000	≤4.86e-03	2.68e-03	2.62e-03	7.63e-04	3.80e-04	3.73e-04
(2,1,[10])	0.3	10.000.000	–	3.05e-01	2.98e-01	2.97e-02	1.85e-02	1.79e-02
	0.2	10.000.000	–	2.66e-02	2.57e-02	2.28e-03	1.21e-03	1.17e-03
	0.1	10.000.000	–	3.20e-04	3.09e-04	3.90e-05	1.80e-05	1.80e-05
(2,1,[14])	0.3	500.000	–	1.20e-01	1.16e-01	4.17e-03	2.54e-03	2.45e-03
	0.2	500.000	–	3.37e-03	3.24e-03	8.60e-05	4.20e-05	4.00e-05
	0.1	500.000	–	1.60e-05	1.60e-05	**<2e-06**	**<2e-06**	**<2e-06**
(2,1,[16])	0.3	500.000	–	1.56e-03	1.5e-03	2.00e-05	**<2e-06**	**<2e-06**
	0.2	500.000	–	**<2e-06**	1.5e-03	**<2e-06**	**<2e-06**	**<2e-06**
	0.1	500.000	–	**<2e-06**	**<2e-06**	**<2e-06**	**<2e-06**	**<2e-06**

(and hence use more of the extracted PUF bits in comparison to threshold $p_T = 0.1$) when using list decoding of list size $\mathcal{L} = 3$ (cf. column "**SD**, $\mathcal{L} = 3$"). We can conclude, that using Approach 2, less PUF bits have to be refused to result in the same error correction performance. Figure 5.17 shows that, when using threshold $p_T = 0.1$, roughly half of the extracted bits are wasted. By increasing the threshold to $p_T = 0.2$, only about $\frac{1}{3}$ of the extracted bits are refused.

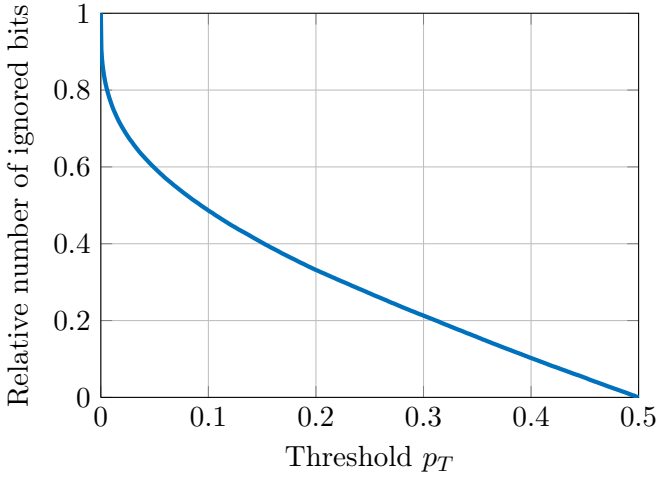


Figure 5.17: Relative number of refused bits for a given threshold p_T, when using the SRAM model from [MTV09a].

Approach 3: Additional Use of Multiple Readouts

We further extend Approach 2 (recall: soft information at the input of the Viterbi algorithm as well as list decoding) by the concept of multiple readouts. The results of simulations for this setup are summarized in Table 5.9, where $\mathcal{L}$ again denotes the list size and m describes the number of readouts. We want to stress three observations: First, we compare the use of multiple readouts between [HÖSB16] and Approach 3. We consider the threshold $p_T = 0.1$ and the convolutional codes $(2, 1, [6])$ and $(2, 1, [7])$, for which [HÖSB16] provides results when using $m = 3$ readouts. In both cases, the defined requirement of obtaining a block error probability of at least 10^{-6} is fulfilled. Using soft information at the input of the Viterbi algorithm and list decoding with list size $\mathcal{L} = 3$ allows to reduce the number of readouts by one in order to still fulfill the required block error probability. Second, also by comparing [HÖSB16] and Approach 3, we can omit list decoding and use the same amount of readouts as [HÖSB16] in order to result in a similar block error probability. Third, we compare Approach 2 ($m = 1$) and Approach 3. Using $\mathcal{L} = 3$ and threshold $p_T = 0.1$ for the code $(2, 1, [6])$ in Approach 2 yields a block error probability of $1.09 \cdot 10^{-3}$, whereas using $m = 2$ readouts in Approach 3 results in a block error probability of $6 \cdot 10^{-6}$. We can conclude, that the use of multiple readouts results in an essential improvement over Approach 2.

In Figure 5.18a, multiple variants are compared for different threshold values p_T. The conclusions that can be drawn from the plots summarize the observations described so far:

Table 5.9: Comparison of block error probabilities using rate $\frac{1}{2}$ codes with hard-decision decoding (HD), soft-decision decoding (SD), list size $\mathcal{L}$ and m readouts. Extraction means the number of simulated key extractions.

Code	$\mathbf{p_T}$	**Extractions**	**Ref.[HÖSB16]** $m = 1$	**Ref.[HÖSB16]** $m = 3$	**HD**, $\mathcal{L} = 3$ $m = 3$	**SD**, $\mathcal{L} = 1$, $m = 3$	**SD**, $\mathcal{L} = 3$, $m = 2$	**SD**, $\mathcal{L} = 3$, $m = 3$
(2,1,[6])	0.3	500.000	4.11e-01	–	6.32e-01	7.81e-03	3.58e-02	4.93e-03
		10.000.000	–	–	6.32e-01	7.86e-03	3.59e-02	4.91e-03
	0.2	500.000	4.11e-01	–	1.84e-01	5.40e-05	1.35e-03	3.40e-05
		10.000.000	–	–	1.84e-01	6.10e-05	1.36e-03	3.20e-05
	0.1	10.000.000	≤1.98e-02	**7.6e-07**	3.27e-02	**<1e-07**	**6.00e-06**	**<1e-07**
(2,1,[7])	0.3	10.000.000	–	–	4.89e-01	2.19e-03	1.49e-02	1.32e-03
	0.2	10.000.000	–	–	1.05e-01	**8.00e-06**	3.44e-04	**4.00e-06**
	0.1	10.000.000	≤4.86e-03	**<1e-07**	2.66e-02	**<1e-07**	**1.00e-06**	**<1e-07**
(2,1,[10])	0.3	10.000.000	–	–	3.25e-01	4.70e-05	1.02e-03	2.85e-04
	0.2	10.000.000	–	–	6.43e-02	**<1e-07**	**5.00e-06**	**<1e-07**
	0.1	10.000.000	–	–	3.98e-02	**<1e-07**	**<1e-07**	**<1e-07**

Using soft information at the input of the Viterbi algorithm improves the reference results. A further improvement is possible by applying list decoding. The additional use of multiple readouts results in significant improvements. Furthermore, we can see in the figure, that using hard-decision decoding, even when combined with list decoding, performs worse than the reference implementation.

Figure 5.18b compares rate $\frac{1}{2}$ convolutional codes of different constraint lengths ν. The figure also contains additional plots for the codes $(2, 1, [6])$ and $(2, 1, [7])$ when using $m = 3$ readouts. This shows, that to a certain extend, a longer constraint length can be replaced by using multiple readouts. Both of the mentioned codes perform better than a code with constraint length 10 and one readout, but worse than using a code with constraint length 16 and one readout.

Our simulations non-surprisingly confirm the well-known fact, that convolutional codes with a large constraint length possess a better error correction performance in comparison to convolutional codes with a short constraint length. Unfortunately, the runtime of the Viterbi decoding algorithm increases exponentially with the constraint length. Hence, convolutional codes with a large constraint length are impractical when using the Viterbi algorithm for decoding. This problem is tackled in Approach 4, where sequential decoding is applied in order to circumvent this drawback.

Approach 4: Sequential Decoding

As already stated in the discussion of Approach 3, convolutional codes with a comparatively large constraint length ν are required to obtain the desired block error probability of at least 10^{-6}. However, decoding becomes infeasible for the Viterbi decoder when using convolutional codes $(n, k = 1, [\nu])$ with large ν. According to [Bos99, Chapter 8.8], $\nu \leq 8$ is adequate when using Viterbi decoder, whereas sequential decoding is realistic up to $\nu \approx 100$. Table 5.10 shows results of simulations performed with the Fano algorithm[6]. A block length of 14 or more fulfills the given requirement of a block error probability $\leq 10^{-6}$. Using this approach, efficient decoding is possible for those constraint lengths.

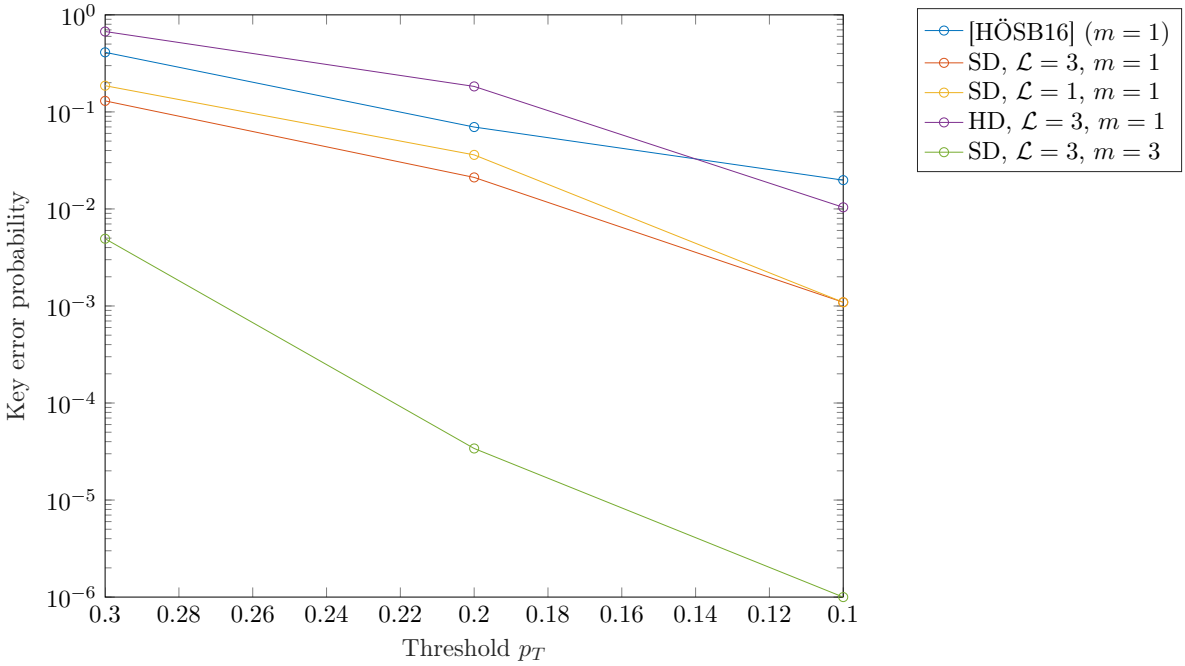
Table 5.10: Performance of sequential decoding using Fano's algorithm with soft information input, $m = 1$ readout and threshold value $p_T = 0.1$.

Code	Extractions	SD
(2,1,[14])	500.000	9e-06
(2,1,[16])	500.000	2e-06
(2,1,[20])	10.000.000	<1e-07
(2,1,[24])	10.000.000	<1e-07

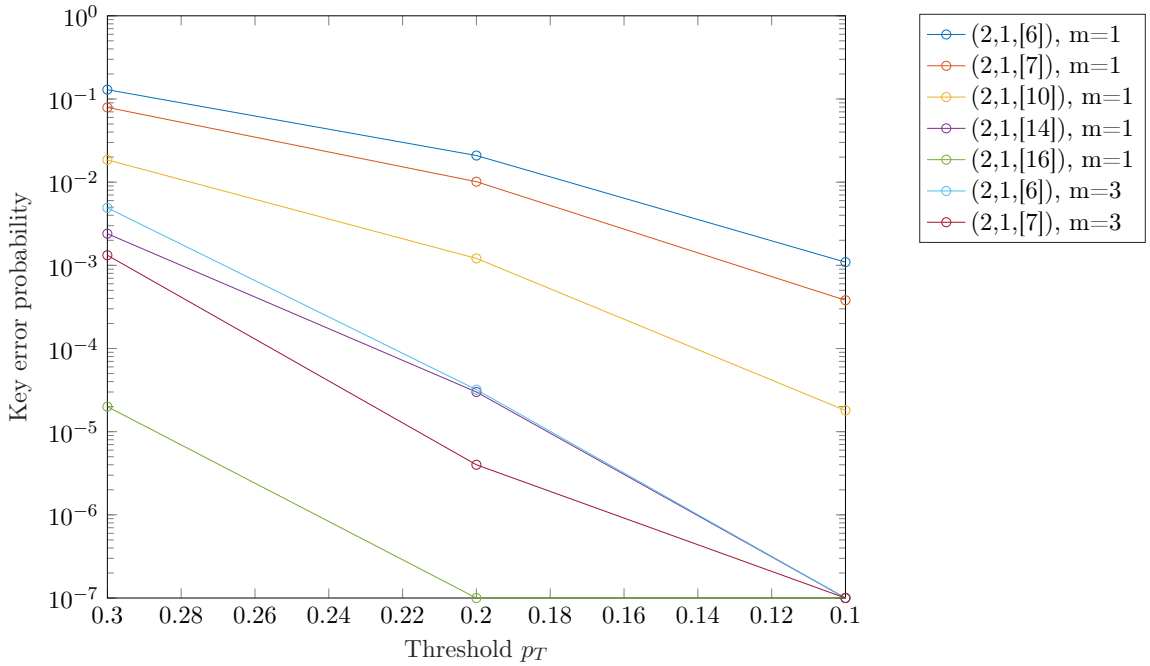
Approach 5: Without Refusing Unreliable Bits

The idea to use only response bits with a certain reliability, as for example done in [HÖSB16], has the drawback, that more bits than needed have to be extracted from the PUF. This results

[6]The implementation of the Fano decoding algorithm as well as the execution of simulations occurred in the context of a master's thesis [Fan17].



(a) $(2, 1, [6])$ code with soft input (SD) or hard input (HD), list decoding (list size $\mathcal{L}$), and m readouts.



(b) Comparison of convolutional codes with different constraint lengths. The applied coding techniques in these simulations are the use of soft information at the input of the Viterbi algorithm and list decoding with list size $\mathcal{L} = 3$. For two of the codes, additionally simulations with using $m = 3$ readouts were performed.

Figure 5.18: Comparison of the techniques studied in Approaches 1 to 3.

Table 5.11: Comparison of key error probabilities when using rate $\frac{1}{3}$ codes and all response bits, even the unreliable ones. The parameters $\mathcal{L}$ and m denote list size and number of readouts. Extraction means the number of simulated key extractions.

Code	**Extractions**	**SD**, $\mathcal{L}=1$	**HD**, $\mathcal{L}=3$	**SD**, $\mathcal{L}=3$	**SD**, $\mathcal{L}=3$, $m=3$
(3,1,[6])	10.000.000	3.98e-02	6.59e-01	2.24e-02	5.70e-05
(3,1,[7])	10.000.000	1.73e-02	5.93e-01	9.43e-03	**6.00e-06**
(3,1,[8])	10.000.000	9.72e-03	5.09e-01	5.14e-03	**1.00e-06**
(3,1,[9])	10.000.000	5.07e-03	4.28e-01	2.65e-03	**<1e-07**
(3,1,[10])	10.000.000	2.30e-03	3.39e-01	1.17e-03	**<1e-07**

in an increased consumption of chip area and time. Recall Figure 5.17 for a visualization of the amount of wasted bits for different thresholds p_T. The idea of this approach is to use all extracted bits. By using the techniques applied in our first three approaches, we expect to observe that the reliable bits compensate the unreliable ones, or in terms of coding theory, that the good channels compensate the bad channels. Using threshold $p_T \approx 0.2$, Figure 5.17 shows that $\approx \frac{1}{3}$ of the extracted bits are discarded. Hence, in this approach we use codes of rate $\frac{1}{3}$ instead of $\frac{1}{2}$, to result in a fair comparison with [HÖSB16].

Table 5.11 shows the block error probabilities obtained in simulations when using all extracted PUF response bits and applying convolutional codes of rate $\frac{1}{3}$. We compare the results of this approach with the results obtained by using Approach 1. Therefore, consider the block error probabilities in Table 5.8 for the convolutional codes (2,1,[6]), (2,1,[7]) and (2,1,[10]), when using threshold $p_T = 0.2$, and note that they are in the same order than the results obtained from Approach 5, as listed in Table 5.11.

Figure 5.19 compares the amount of required response bits, when using [HÖSB16] or Approach 5, respectively. According to Figure 5.17, in the former approach, roughly half of the extracted bits are refused for threshold $p_T = 0.1$, due to a low reliability. The other half of the extracted bits defines the codeword length. When using a code of rate $\frac{1}{2}$, information part and redundancy part have the same amount of bits in the codeword. On the other hand, Approach 5 does not refuse any response bits. Using a code of rate $\frac{1}{3}$ results in the same error correction performance when using soft information at the decoder's input and list decoding, but requires a smaller number of response bits.

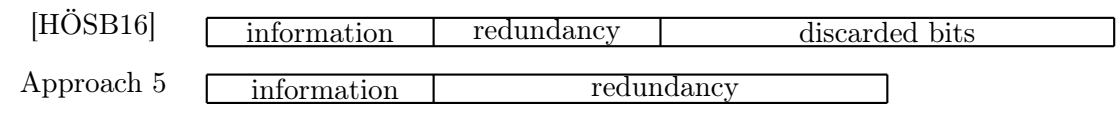


Figure 5.19: Comparing the number of response bits required to fulfill the desired block error probability in the scenario discussed above, when using the methods proposed in [HÖSB16] or Approach 5 as explained in this chapter, respectively.

5.2.4 Applying Convolutional Codes to Key Extraction using ROPUFs

In Section 5.2.3, we used SRAM responses for our discussion of convolutional codes in the PUF scenario, generated based on a mathematical model. In this section, we apply convolutional codes to the real-word ROPUF data from [MCMS10]. The data set contains both frequencies, those measured at stable environmental conditions and those measured at changing temperature and supply voltage conditions. The first contribution of this section is to demonstrate, that applying convolutional codes is reasonable for practical PUFs. Note that in [MCMS10], error correcting codes were not addressed. Hence, as second contribution of this section, the convolutional codes applied in Section 5.2 can be seen as a proposal for error correcting codes, which can be applied to the ROPUFs constructed in [MCMS10]. Before discussing our results, we briefly outline the setup of our simulations. According to [MCMS10], responses of length 511, which were generated by pairwise comparing the frequencies of adjacent ring oscillators, are used. The standard code-offset scheme as explained in Chapter 4 (c.f. Figure 4.2) is applied as secure sketch. From the 100 measurements, which are available per device, the average is chosen as reference response, while the 100 samples are used as responses that are extracted in the reproduction phase.

We begin our study by using the data gathered at stable operating conditions, measured at 25°C ambient temperature and 1.2V supply voltage. For this scenario, the data set provides responses of 193 PUFs, where the relative response intra-distance lies in between 0.38% and 1.39%. These numbers are related to a very good channel. For each PUF, 100 measurements are contained in the data set. Table 5.12 shows the amount of PUFs, for which reproduction of the key fails at least once when reproducing the responses 100 times. Performing hard-decision (HD) decoding and using a code of rate $R = \frac{1}{2}$, a constraint length of $\nu = 10$ is necessary in order to result in an error-free reproduction in all test instances. Using soft information at the input of Viterbi algorithm (SD), choosing constraint length $\nu = 2$ is sufficient. Decreasing the rate to $R = \frac{1}{3}$, the constraint length when using hard-decision decoding can be decreased from $\nu = 10$ to $\nu = 3$.

Table 5.12: Number of PUFs for which key regeneration fails at least once. E.g., the entry "33" for code rate $R = \frac{1}{2}$ and constraint length $\nu = 2$ means that key reconstruction fails, at least once, for 33 out of 193 devices when regenerating the 100 responses.

	$\nu = 2$ (HD)	$\nu = 3$ (HD)	$\nu = 6$ (HD)	$\nu = 10$ (HD)	$\nu = 2$ (SD)
$R = \frac{1}{2}$	33	15	1	0	0
$R = \frac{1}{3}$	1	0	0	0	0

In the next step, we investigate the five PUFs in the data set, which have been evaluated at different temperature and supply voltage conditions. Table 5.13 shows the nine temperature/supply voltage combinations as well as the amount of failures, when regenerating the key 100 times for each of the five PUFs under the corresponding environmental conditions. For these simulations, a $(2, 1, [6])$ convolutional code was used and hard-decision decoding was chosen to obtain the results summarized in Table 5.13. Decreasing the code rate to $R = \frac{1}{3}$ allows to correct all test cases, besides the extreme case with a supply voltage of 1.96V, while still using hard-decision decoding. To decode all test cases correctly, a $(2, 1, [2])$ code is sufficient, when using soft information at the input of Viterbi algorithm. Hence, soft information

can be used to prevent the problems occurring for changing environmental conditions and the constraint length can be kept small.

Table 5.13: Average amount of failures when regenerating the key 100 times under varying operating conditions using hard input ($R = \frac{1}{2}$, $\nu = 6$).

	PUF 1	PUF 2	PUF 3	PUF 4	PUF 5
0.96V, 25°C	100	100	79	100	100
1.08V, 25°C	36	18	2	26	58
1.20V, 25°C	0	0	0	0	0
1.20V, 35°C	0	0	0	0	0
1.20V, 45°C	0	0	0	0	0
1.20V, 55°C	0	0	0	0	0
1.20V, 65°C	0	0	0	0	0
1.32V, 25°C	6	52	0	0	47
1.44V, 25°C	59	99	54	91	96

As already stressed previously, the considered data set contains too few data for drawing conclusions that are reliable from a statistical point of view (even when it is large in comparison to other studies concerning PUFs, cf. summary in Appendix A and Table A.1). It is not intended to extend the data set to a statistically meaningful number like ,e.g., 10^8 responses. The reason is, that a single ring oscillator needs 250ms per measurement and hence, the overall process would require a time of 9 months [Sch17]. To circumvent that problem, we follow a suggestion by [Sch17] and approximate a larger amount of responses by modeling a ring oscillator that is calibrated by using the real measurements according to [MMS11]. As already mentioned in Chapter 3 when discussing channel and error models, a frequency f_i of ring oscillator i can be decomposed into

$$f_i = f_{avg} + f_{PV} + f_{aging} + f_{noise}, \tag{5.40}$$

where f_{avg} is the average delay (static component), f_{PV} the deviation caused by process variation (static component), f_{aging} the deviation caused by aging (dynamic component), and f_{noise} the deviation due to noise (dynamic component). The average and process variation components are calculated by using the average of the 100 provided readouts in order to remove the noise. For the generation of new responses $f_{aging} \sim N(-6.75, 0.05)$ and $f_{noise} \sim N(0, \sigma)$ with $\sigma = \sqrt{\frac{1}{99}\sum_{k=1}^{100}(f_k - f_{avg})^2}$ is used according to [MMS11], where these distributions were calculated by performing accelerated aging[7]. Figure 5.20 shows for an exemplary PUF, that the synthetic responses preserve the behavior of the real error pattern. For example, consider the peaks, which are around the same bit positions in both pictures. We applied this model in order to generate 10^7 synthetic responses for which simulation results are provided in Table 5.14.

[7] Accelerated aging is an approach to simulate aging effects by stressing the devices with high temperatures and voltages.

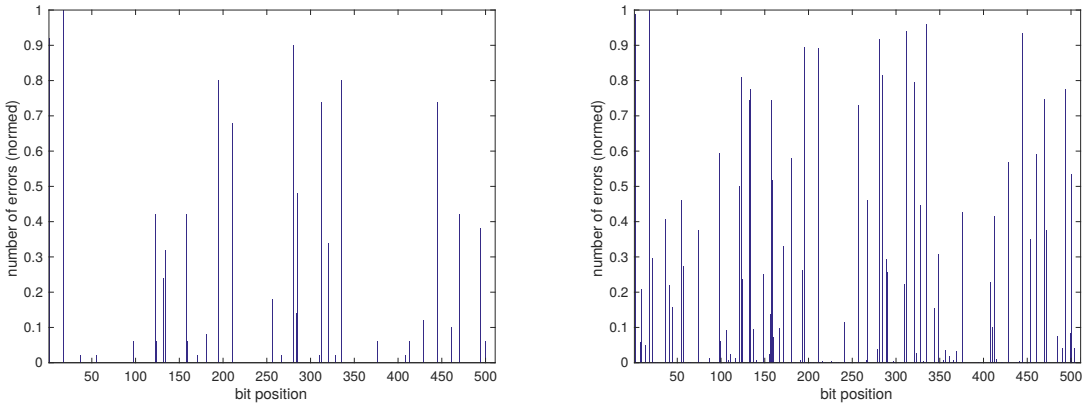


Figure 5.20: Comparison of error pattern between real data (left) and synthetic data (right).

Table 5.14: Block error probabilities when using synthetic responses. Hard-decision (HD) and soft-decision (SD) decoding is applied for both, convolutional codes of rates $R = \frac{1}{2}$ and $R = \frac{1}{3}$. For each of them, constraint lengths $\nu \in \{2, 3, 6, 10\}$ are used.

	$\nu = 2$	$\nu = 3$	$\nu = 6$	$\nu = 10$
$R = \frac{1}{2}$ (HD)	1.33e-01	4.99e-02	1.99e-03	2.19e-06
$R = \frac{1}{2}$ (SD)	3.82e-02	8.89e-04	2.70e-06	0
$R = \frac{1}{3}$ (HD)	4.28e-03	2.78e-03	0	0
$R = \frac{1}{3}$ (SD)	5.44e-05	0	0	0

5.2.5 Summary on Convolutional Codes for PUFs

We applied convolutional codes for error correction in PUFs. We can conclude, that previous approaches, which use convolutional codes in the same context, can be improved by applying advanced techniques from the field of coding theory. In our studies, which are based on a mathematical response model for SRAM PUFs (Section 5.2.3), we have shown that using soft information at the input of the Viterbi algorithm can improve the block error probability significantly. In a first approach, based on a scenario from literature [HÖSB16], we determined, that the use of soft information at the input of the Viterbi decoding algorithm can replace simplified ROVA without degrading, but rather improving the resulting block error probability. In a second approach, we additionally replaced the ordinary Viterbi decoder with a corresponding list decoding variant. With a suitable list size, the threshold used in [HÖSB16] as well as in our approaches, can be increased and hence less extracted bits have to be discarded. Additionally, multiple readouts are used in our third approach. Replacing Viterbi algorithm with sequential decoding, a more efficient runtime can be achieved even for larger constraint lengths, since sequential decoding provides an alternative to Viterbi algorithm with a runtime independent of the constraint length. Hence, convolutional codes with larger constraint lengths can be applied when using sequential decoding. In a final approach, we do not refuse any extracted response bits. By using soft information, list decoding, and

multiple readouts, our simulations revealed that the reliable bit positions compensate for the unreliable ones. The usage of all extracted response bits avoids an increased chip area and leads to a decreased time consumption that is required to extract the PUF response. Working with real-world ROPUF data in Section 5.2.4 leads to results, which also confirm the assumption, that using a BSC with a fixed bit error probability is a too conservative assumption for practical scenarios.

5.3 Concluding Remarks

In this chapter, we have proposed and evaluated several code constructions, that can be applied in secure sketches when implementing error correction for Physical Unclonable Functions. We have shown, that known constructions, based on block codes, can be improved significantly by designing concatenated codes, based on suitably chosen component codes. Our proposed code constructions either reduce the block error probability or the codeword length considerably. Our contribution relating to block codes can be stated as follows:

- We presented four concatenated code constructions $\mathcal{C}_1, \ldots, \mathcal{C}_4$, which significantly improve the reference implementation given in [MVHV12], concerning codeword length or block error probability, as well as the required chip area in hardware implementations. For decoding, Generalized Minimum Distance and Power Decoding is applied for the first time in the context of PUFs.

We also studied, how to improve approaches based on convolutional codes. Therefore, we applied concepts like soft information input, list decoding, and sequential decoding. We summarize our contributions relating to convolutional codes:

- In the literature, hard-decision decoding by using the Viterbi algorithm is used for decoding of convolutional codes in the field of PUFs. For the first time in that context, we applied techniques that are well known in coding theory, but widely unknown by the PUF community. Our simulations confirm, that applying soft information at the input of the Viterbi algorithm as well as using a list decoding variant of the Viterbi algorithm significantly improves previous results. This implies, that the use of algorithms that compute the reliability of a decoded sequence can be omitted.

- By using soft input for the Viterbi algorithm, list decoding as well as multiple readouts, the pre-selection of reliable response bits can be omitted, when using a convolutional code with a suitable constraint length.

- Sequential decoding is applied for error correction in the context of PUFs for the first time, thereby enabling codes with much larger constraint length, which are necessary in the considered scenario, as indicated by previous studies.

- We showed by using a collection of real-world ROPUF data, that applying convolutional codes works for practical PUFs.

- We proposed convolutional codes as suitable choice in order to extend the PUFs from [MCMS10] with an error correction component.

6

Attacks and Countermeasures

In this chapter, we study approaches that make PUFs resistant against side-channel attacks. One of the considered methods is based on masking techniques. The other suggestion is the modification of the decoding algorithm, such that a constant runtime, which is independent of the received word, can be achieved, and hence, deducing on the secret by observing the decoder's runtime is impossible. Some of the presented methods focus on Reed–Solomon and Reed–Muller codes, and hence, can directly be applied to the concatenated code constructions proposed in Chapter 5. A further contribution of this chapter is the application of list decoding of Reed–Solomon codes in the context of PUFs. According to the authors' best knowledge, list decoding was not considered in this context before.

The chapter is structured as follows: Section 6.1 deals with attacks on PUFs in general. In Section 6.2, we briefly discuss list decoding of Reed–Solomon codes. Section 6.3 is about the prevention of side-channel attacks and is sub-divided into two proposals. First, in Section 6.3.1, masking techniques are applied. These methods aim for hiding the codeword, which is used in a secure sketch, from an attacker. Section 6.3.2 suggests to apply constant time decoding algorithms for Reed–Solomon as well as Reed-Muller codes. We have published the contributions discussed in this Chapter in [PMWZB17].

6.1 Attacks on PUFs

We begin with structuring different types of attacks on cryptosystems in general and briefly discuss their influence when PUFs are used. As visualized in Figure 6.1, we differentiate *classical cryptanalysis* from *implementation attacks*[1]. Classical cryptanalysis aims for exploiting mathematical weaknesses in cryptographic algorithms in order to break them. When breaking an algorithm independent of an implementation by using classical cryptanalysis, the algorithm itself is broken and consequently all implementations of that algorithm are obsolete. Implementation attacks, in contrast, deal with studying weaknesses in a specific implementation of a cryptosystem. When breaking a specific implementation of a cryptographic algorithm, other implementations still remain secure.

Dealing with implementation attacks, active and passive attacks are distinguished[2]. In case of active attacks, a chip is physically altered. Examples for active attacks are *reverse engineering* approaches, which aim for gaining knowledge about a secret by disassembling the chip, or *fault attacks*, which aim for changing the behavior of the system by inducing errors.

[1] Implementation attacks are also often called *physical attacks* in the literature.
[2] Also combined implementation attacks exist.

When applying those kinds of attacks to PUFs, properties on the microscopic level that are used to extract randomness are altered, thus changing the challenge-response behavior. Due to this behavior, PUFs are called tamper-resistant.

Using passive attacks, the functioning of a device is not changed, information are only collected by passive approaches like observing or measuring. The most famous kind of passive implementation attacks are so-called *side-channel attacks*, which emerged in the 1990s [Koc96, KJJ99]. During calculations that depend on the secret key, the physical behavior of a device and its environment is correlated with the value of the key. Through measurements of properties like, for example, execution time, power consumption [KJJ99, MOP08] or electromagnetic radiation, a so-called side-channel can be established. By monitoring and analyzing the signals observed via such a side-channel, the attacker aims for reconstructing the secret key. Side-channel attacks on PUFs usually focus on the helper data, e.g., [MSSS11]. Instead, we consider a side-channel attack on the decoding algorithm as well as possible countermeasures. Side-channel attacks are a huge field of current research. Introductions can for example be found in [Geb09, Chapter 8] and [Ver10, Chapter 2]. There exist some studies about side-channel analysis in the field of PUFs. In [MSSS11], electro-magnetic emission of ROPUFs was studied. Attacks on software implementations of Reed–Solomon and BCH codes were examined in [KS10]. [DW09] analyzes the power consumption when storing a codeword of a parity-check code and studies the amount of information that is leaked via the side-channel.

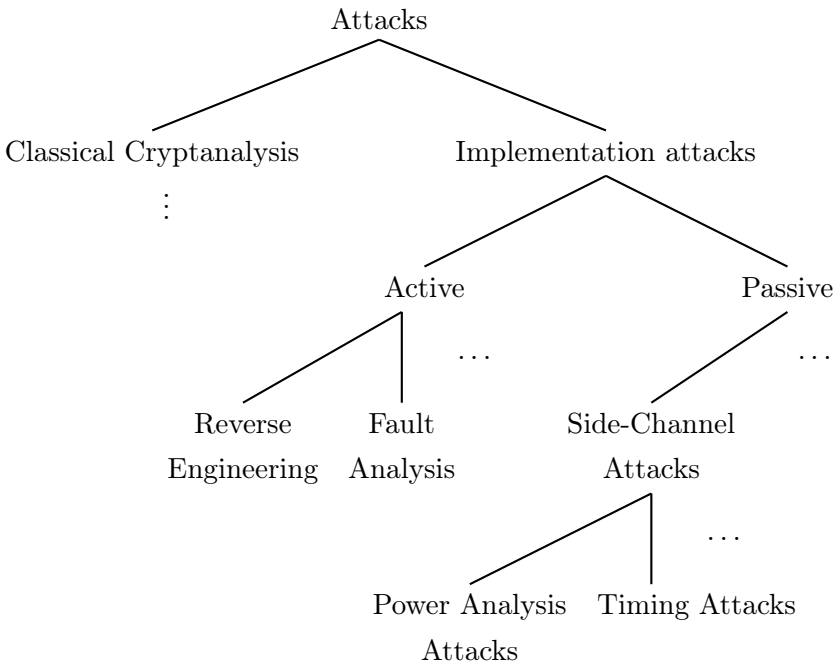


Figure 6.1: Simplified classification of attacks on cryptosystems. We distinguish classical cryptanalysis and implementation attacks. In this chapter, we focus on side-channel attacks, which are a special type of passive implementation attacks. The goal of side-channel attacks is to get knowledge about the secret key, by passively measuring and analyzing timing behavior or power supply, while the cryptographic key is processed by the cryptosystem.

An alternative classification groups implementation attacks into *dynamic* and *static* attacks. An attack is called dynamic, if it can only be performed during runtime. Hence, the correct moment to attack the system has to be determined. Static attacks can also be executed while the device is disabled, which makes attacks very flexible, since they can be performed at any time. These kinds of attacks are very promising when a key has to be recovered that is stored in a non-volatile memory. Recall from Chapter 2.1.4, that PUFs can be used to reproduce a key when it is needed instead of storing it permanently. Hence, static attacks can not be applied for recovering the key when using PUFs and dynamic attacks are of primary interest.

6.2 List Decoding of Reed–Solomon Codes

Recall Definition 5.5 from Chapter 5.1.1, where we gave the definition of Reed–Solomon codes. This section briefly discusses list decoding, since this technique is used in Section 6.3.2, when we develop a decoding algorithm which is resistant against timing attacks.

BMD decoding can decode uniquely if $\tau \leq \lfloor \frac{d-1}{2} \rfloor$ errors occur, since the error correction spheres around the codewords do not overlap for these values of τ. The idea of list decoding is to extend the radius of these spheres. Thus, some of the spheres overlap and consequently, for some of the received words, there exists more than one option to decode. In such cases, instead of one unique decoding result, a list of polynomial size, which contains possible codeword candidates is returned. Due to the polynomial size of this list, ML decoding can efficiently be performed on the codeword candidates.

The Guruswami–Sudan algorithm [GS98], which was proposed as an extension of the Sudan algorithm[3], is one of the most prominent list decoding algorithms for Reed–Solomon codes. The Guruswami–Sudan algorithm aims for determining a bivariate polynomial $Q(x, y)$ according to Problem 6.1, in order to be able to correct $\tau < n - \sqrt{n(k-1)}$ errors.

Problem 6.1. For a given vector $\boldsymbol{y} = (y_1, \ldots, y_n) \in \mathbb{F}_q^n$, find a non-zero bivariate polynomial $Q(x, y) \in \mathbb{F}_q[x, y]$ of the form

$$Q(x, y) = \sum_{j=0}^{\ell} Q_j(x) y^j,$$

such that for given integers s, τ and ℓ:

1. (α_i, y_i) are zeros of $Q(x, y)$ of multiplicity s,
   $\forall i = 1, \ldots, n$,
2. $\mathsf{deg} Q_j(x) \leq s(n - \tau) - 1 - j(k - 1)$, $\forall j = 0, \ldots, \ell$.

The algorithm consists of two steps. The goal of the first step is to interpolate the polynomial $Q(x, y)$, while the second step aims for finding roots. The algorithm calculates all polynomials with the property

$$(y(x) - f(x)) | Q(x, y), \tag{6.1}$$

[3]The Sudan algorithm is of historic importance since it was the first method to algebraically decode beyond half-the-minimum distance in polynomial time [Sud97]. The drawback of Sudan algorithm is, that it is only applicable to codes of a rate $\leq \frac{1}{3}$. However, its computational complexity is lower in comparison to the Guruswami–Sudan algorithm. The Sudan algorithm reaches the same error-correction radius as power decoding (c.f. Section 5.1.1) and can be seen as a special case of the Guruswami–Sudan algorithm.

where $y(x)$ represents the received word and $f(x)$ is the evaluation polynomial. According to a proof in [GS98], the calculated list includes all evaluation polynomials $f(x)$ that are encoded to codewords $\boldsymbol{c} \in \mathcal{C}$ with $\mathrm{dist_H}(\boldsymbol{c}, \boldsymbol{y}) < \tau$. The interpolation step can for example be performed by using the algorithms proposed in [Ale02] or [ZGA11]. The root-finding step can be implemented by the Roth–Ruckenstein algorithm [RR00], that also will be used in Section 6.3.2. Also, efficient VLSI implementations exist, which are interesting for applying the algorithm to practical PUFs [GKKG02].

Example 6.2. We consider the ordinary concatenated code $\mathcal{C}_2(2; 2048, 132, \geq 688)$ as constructed in Chapter 5. Recall that $\mathcal{C}_2$ is an ordinary concatenated code consisting of an outer Reed–Solomon code $\mathcal{RS}(2^6; 64, 22, 43)$ and an inner Reed–Muller code $\mathcal{RM}(1, 5)$. In Chapter 5.1.3, we have observed, that using this code construction results in a block error probability $P_{\mathrm{err}} \approx 6.79 \cdot 10^{-37}$. Furthermore, we have shown that ML decoding of $\mathcal{RM}(1, 5)$ transforms the channel into a binary error and erasure channel with error probability P(error) = 0.00317 and erasure probability P(erasure) = 0.017605. We now consider the Reed–Solomon code $\mathcal{RS}(2^6; 64, 22, 43)$. Due to minimum distance $d = 43$, it is possible to decode uniquely up to 21 errors when using BMD decoding. Performing list decoding with the Guruswami–Sudan algorithm allows to increase this number and to correct up to $\lceil n - \sqrt{n(k-1)} \rceil - 1 = 27$ errors. If we additionally have erasures, the Guruswami–Sudan algorithm only considers non-erased positions in the interpolation step. Let τ and δ again denote the number of errors and erasures, respectively. Then, the block error probability can be calculated as

$$P_{\mathrm{err}} = \sum_{i=0}^{n} \mathrm{P}(\delta = i)\mathrm{P}(\tau \geq n - i - \sqrt{(n-i)(k-1)}) \tag{6.2}$$

$$\approx 3.5308 \cdot 10^{-46}. \tag{6.3}$$

We notice that this number is significantly smaller than the block error probability $P_{\mathrm{err}} \approx 6.79 \cdot 10^{-37}$ which results from BMD decoding. Similarly, the block error probabilities of the ordinary concatenated code $\mathcal{C}_3$ and the generalized concatenated code $\mathcal{C}_4$ can be decreased.

6.3 Preventing Side-Channel Attacks on PUFs

Figure 6.2 visualizes the attacker model considered in this chapter. The illustration of the reproduction process is depicted in a slightly different way, when comparing to the visualizations of the helper data algorithms discussed in Chapter 4 (e.g. Figure 4.1). Here, the process is logically divided into the components *pre-processing*, *decoding*, and *post-processing*. The pre-processing unit prepares the extracted PUF response to have the format codeword plus error, which is required by the decoding algorithm. The post-processing unit reproduces the original response from the decoder's output and hashes the result to the final key. The function φ can be used to mask the decoder's input according to [MSS13]. Since the target of the attack considered in this chapter is the decoding algorithm, the other components are assumed to be secure. Note that the helper data storage always can be considered as attackable, since it is allowed to be a public storage by definition.

We study two approaches for the prevention of side-channel attacks. The first countermeasure uses a function φ as visualized in Figure 6.2, in order to apply masking techniques (cf. Section 6.3.1). In a second approach, we consider the decoder. The decoding time may

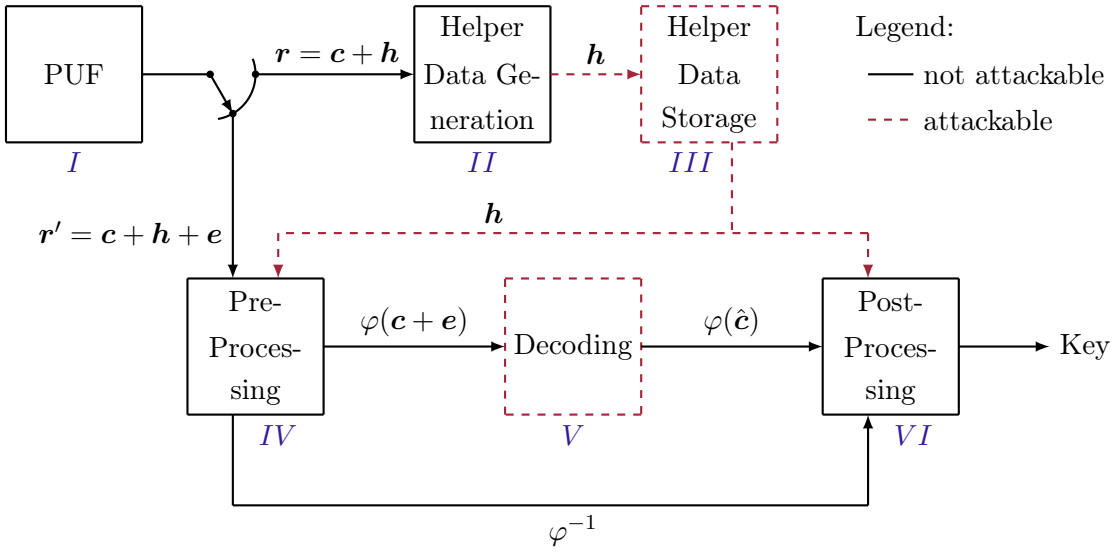


Figure 6.2: Since attacks on the decoding algorithm are considered, we define some of the components of the helper data algorithm as not attackable. The helper data storage is assumed to be public, and hence, can always considered to be attackable.

depend on the received word. In order to prevent timing attacks, we aim for designing a decoding algorithm that has a constant runtime, which is independent of the received word (cf. Section 6.3.2).

6.3.1 Masking Techniques

Codeword masking was proposed and used in [MSS13] in the context of preventing differential power analysis (DPA) attacks on PUFs. However, it also protects an implementation against timing attacks. The main idea is to make calculations, independent of the used codeword $\boldsymbol{c}$ by randomly adding up vectors. Hence, dependencies between input data and intermediate data are destroyed and the codeword $\boldsymbol{c}$ is hidden from an attacker.

Figure 6.3 visualizes the code-offset scheme as discussed in Chapter 4, extended by codeword masking according to [MSS13], where a random codeword $\boldsymbol{c}_r \in \mathcal{C}$ is added to $\boldsymbol{c} + \boldsymbol{e}$ in order to hide $\boldsymbol{c}$. For this purpose, the function φ is defined as

$$\varphi(\boldsymbol{c} + \boldsymbol{e}) = \boldsymbol{c} + \boldsymbol{c}_r + \boldsymbol{e}. \tag{6.4}$$

Since $\boldsymbol{c}_r$ is chosen randomly, for an attacker it seems like a random codeword of code $\mathcal{C}$ is used. The decoder uses (6.4) as input and delivers $\boldsymbol{c} + \boldsymbol{c}_r$.

[MSS13] does not provide a proof, that the uncertainty, which remains when we assume an attacker with access to the helper data $\boldsymbol{h}$ as well to the masked word $\boldsymbol{c} + \boldsymbol{c}_r + \boldsymbol{e}$, is large enough. For this reason, we prove in the following theorem, that even when an attacker finds any method to obtain $\boldsymbol{e}$ and $\boldsymbol{c} + \boldsymbol{c}_r$, the entropy to know $\boldsymbol{c}$ still remains large enough, since $\boldsymbol{c}_r$ is chosen randomly.

Theorem 6.3. $H(\boldsymbol{r} \mid (\boldsymbol{c}_r + \boldsymbol{c} + \boldsymbol{e}, \boldsymbol{h})) \geq H(\boldsymbol{r}) - (n - k)$

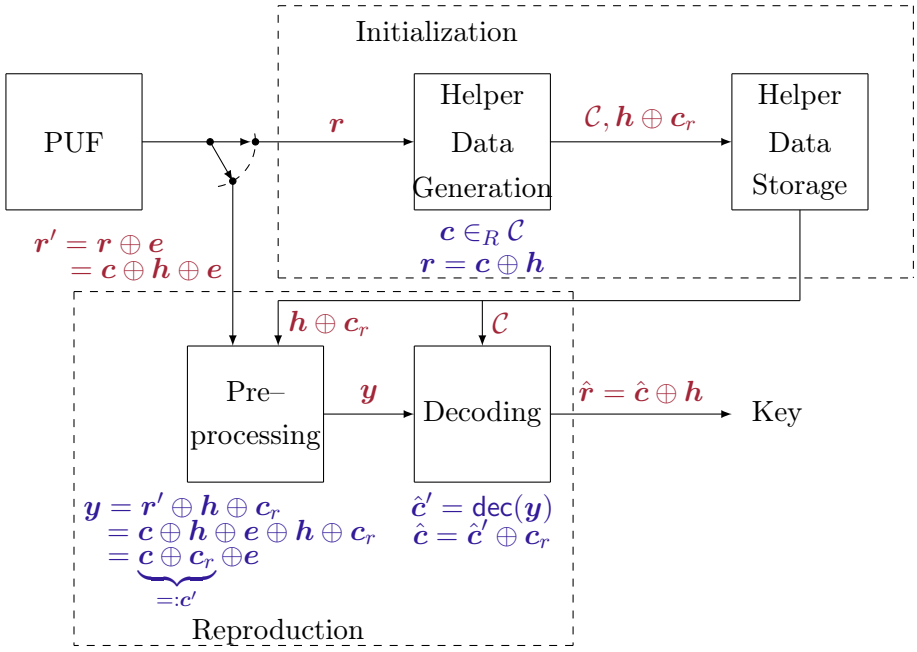


Figure 6.3: Code-offset scheme with codeword masking, which adds a random codeword $\boldsymbol{c}_r$ to the helper data to hide the codeword $\boldsymbol{c}$ from an attacker, who is able to retrieve $\varphi(\boldsymbol{c}+\boldsymbol{e})$.

Proof. We know that $\boldsymbol{r}, \boldsymbol{c}, \boldsymbol{c}_r, \boldsymbol{e}$ are pairwise independent. Also, the codewords $\boldsymbol{c}$ and $\boldsymbol{c}_r$ are drawn from the code uniformly at random, so

$$H(\boldsymbol{c}+\boldsymbol{e}) = H(\boldsymbol{c}+\boldsymbol{c}_r+\boldsymbol{e}). \tag{6.5}$$

In general, it holds that

$$H(\boldsymbol{c}+\boldsymbol{c}_r+\boldsymbol{e}, \boldsymbol{h}) \leq H(\boldsymbol{c}+\boldsymbol{c}_r+\boldsymbol{e}) + H(\boldsymbol{h}). \tag{6.6}$$

Since we can compute $(\boldsymbol{r}, \boldsymbol{c}_r+\boldsymbol{c}+\boldsymbol{e}, \boldsymbol{c})$ from $(\boldsymbol{r}, \boldsymbol{c}_r+\boldsymbol{c}+\boldsymbol{e}, \boldsymbol{h})$ and vice versa, we have

$$\begin{aligned} H(\boldsymbol{r}, \boldsymbol{c}_r+\boldsymbol{c}+\boldsymbol{e}, \boldsymbol{h}) &= H(\boldsymbol{r}, \boldsymbol{c}_r+\boldsymbol{c}+\boldsymbol{e}, \boldsymbol{c}) \\ &= H(\boldsymbol{r} \mid (\boldsymbol{c}_r+\boldsymbol{c}+\boldsymbol{e}, \boldsymbol{c})) + H(\boldsymbol{c}_r+\boldsymbol{c}+\boldsymbol{e}, \boldsymbol{c}) \\ &= H(\boldsymbol{r}) + H(\boldsymbol{c}_r+\boldsymbol{c}+\boldsymbol{e}, \boldsymbol{c}) \\ &= H(\boldsymbol{r}) + H(\boldsymbol{c}_r+\boldsymbol{c}+\boldsymbol{e} \mid \boldsymbol{c}) + H(\boldsymbol{c}) \\ &= H(\boldsymbol{r}) + H(\boldsymbol{c}_r+\boldsymbol{e}) + H(\boldsymbol{c}). \end{aligned} \tag{6.7}$$

Hence, we obtain

$$\begin{aligned} H(\boldsymbol{r} \mid (\boldsymbol{c}_r+\boldsymbol{c}+\boldsymbol{e}, \boldsymbol{h})) &= H(\boldsymbol{r}, \boldsymbol{c}_r+\boldsymbol{c}+\boldsymbol{e}, \boldsymbol{h}) - H(\boldsymbol{c}_r+\boldsymbol{c}+\boldsymbol{e}, \boldsymbol{h}) \\ &\overset{(6.6),(6.7)}{\geq} H(\boldsymbol{r}) + H(\boldsymbol{c}_r+\boldsymbol{e}) + H(\boldsymbol{c}) - H(\boldsymbol{c}+\boldsymbol{c}_r+\boldsymbol{e}) - H(\boldsymbol{h}) \\ &\overset{(6.5)}{=} H(\boldsymbol{r}) + H(\boldsymbol{c}) - H(\boldsymbol{h}) \\ &= H(\boldsymbol{r}) + k - H(\boldsymbol{e}) \\ &\geq H(\boldsymbol{r}) - (n-k), \end{aligned} \tag{6.8}$$

which is the same as for the code-offset construction. □

Note that if $H(\boldsymbol{r}) = n$, then $H(\boldsymbol{r} \,|\, (\boldsymbol{c}_r + \boldsymbol{c} + \boldsymbol{e}, \boldsymbol{h})) \geq k$. If using a masking scheme, it is of utmost importance not to change the Hamming weight of the error vector $\boldsymbol{e}$, because otherwise, the hardness of the decoding problem is changed. The only further masking operations, besides codeword masking, which fulfill this requirement, are the Hamming-metric isometries. Considering $\mathbb{F}_2^n$, these isometries are exactly all permutations of the positions $1, \ldots, n$. Let π denote such a permutation and let $\boldsymbol{c}$ be a codeword of a Reed–Solomon code $\mathcal{C}$. Then $\pi(\boldsymbol{c})$ is also a codeword of a Reed–Solomon code that is defined by a permutation of the code locators α_i of $\mathcal{C}$. Thus, $\pi(c + e) = \pi(c) + \pi(e)$ with $\mathrm{wt}(\pi(\boldsymbol{e})) = \mathrm{wt}(\boldsymbol{e})$, we obtain $\pi(c)$ from $\pi(\boldsymbol{c} + \boldsymbol{e})$ with a decoder for Reed–Solomon codes in all cases where we can correct the error $\boldsymbol{e}$ in $\boldsymbol{c} + \boldsymbol{e}$. We distinguish two cases, depending on whether or not the permutation π is an element of the automorphism group of the code. If π is not an element of the automorphism group of the code, the decoder has to know π. Otherwise, the masking technique is equivalent to codeword masking, since $\pi(\boldsymbol{c}) - \boldsymbol{c}$ is again a codeword.

6.3.2 Constant-Time Decoding

This section deals with the development of a constant time decoding algorithm. Since the runtime of that algorithm does not depend on the received word, it can be used in order to prevent timing attacks on the decoding routine.

Before discussing the runtime of the specific operations of a decoding algorithm, we have to start from a more general point of view. Error-correcting codes usually are defined over finite fields, more precisely, over extension fields $\mathbb{F}_{p^m}$ (p prime, $m \in \mathbb{N}$). In practical applications, most often $p = 2$ and thus extension fields over the binary field are used. Hence, algebraic decoding algorithms perform operations over finite fields. First, these operations have to be resistant against timing attacks. For small fields, like usually used for error-correcting codes (e.g. $\mathbb{F}_{2^6}$ or $\mathbb{F}_{2^8}$), look-up tables can be used for operations that are constant in time. For larger field sizes, the work of [PT15] can be used, which considers the protection of finite field operations against side-channel attacks in the context of elliptic-curve cryptography. Hence, we can assume that finite field operations are possible to be performed with a constant runtime. Next, we can turn our attention to the actual decoding algorithms.

Decoding Reed–Solomon Codes in Constant Time

We start with decoding Reed–Solomon codes, which were proposed as outer codes in the concatenated schemes in Chapter 5.1.3. Recall that list decoding with the Guruswami–Sudan algorithm consists of an interpolation step and a root-finding step.

The interpolation step aims for finding a bivariate polynomial

$$Q(x, y) = \sum_{\eta=0}^{\ell} Q_\eta(x) y^\eta = \sum_{\eta=0}^{\ell} \sum_{\mu=0}^{d_\eta} Q_{\eta,\mu} x^\mu y^\eta, \tag{6.9}$$

where $d_\eta = s(n - \tau) - 1 - \eta(k - 1)$, such that the two properties stated in Problem 6.1 are fulfilled. This corresponds to finding a non-zero solution $Q_{\eta,\mu} \in \mathbb{F}_q$ for $0 \leq \mu \leq d_\eta$ and

$0 \leq \eta \leq \ell$ of the system

$$\sum_{\eta=0}^{\ell} \sum_{\mu=0}^{d_\eta} \binom{\eta}{h} \binom{\mu}{j} Q_{\eta,\mu} \alpha_i^{\mu-j} r_i^{\eta-h} = 0,$$

for $i = 0, \ldots, n$ and $h + j < s$. There exist algorithms that exploit the structure in order to efficiently provide a solution to Problem 6.1. These algorithms are asymptotically faster than naive approaches. However, their runtime depends on the received word $\boldsymbol{y}$, which in turn might reveal side information. To circumvent the drawback of revealing side information, we accept a larger runtime and use the naive approach of Gaussian elimination. To achieve a constant runtime, we always execute a row operation. If an element is already zero, we add a zero-row. This results in the same number of additions and multiplications, independent of the vector $\boldsymbol{y}$, and hence, no side information are revealed.

For the root-finding step, we consider the Roth–Ruckenstein algorithm according to [RR00], which is outlined in Algorithm 11.

Algorithm 11: RR $(Q(x,y), g(x), i, \mathcal{L})$ [RR00]

Input: $Q(x,y) = \sum_{\eta=0}^{\ell} Q_\eta(x) y^\eta$, $g(x)$, i, global list $\mathcal{L}$

1 **if** i=k **then return**;
2 $M(x,y) \leftarrow Q(x,y)/x^r$ with $r \in \mathbb{N}$ maximal ;
3 $p(y) \leftarrow M(0,y)$;
4 Find roots of $p(y)$;
5 Remove $g(x)$ from the global list $\mathcal{L}$;
6 **for** *each root* γ **do**
7 Add $g(x) + \gamma x^i$ to the global list $\mathcal{L}$;
8 RR $\left(M(x, x(y-\gamma)), g(x) + \gamma x^i, i+1, \mathcal{L}\right)$;

Roth–Ruckenstein is a recursive algorithm. The initial call uses the parameters

$$Q(x,y), \quad g(x), \quad i, \quad \mathcal{L},$$

where $Q(x,y)$ is the polynomial calculated in the interpolation step, $g(x)$ is a polynomial used to recursively calculate coefficients $0, 1, 2, \ldots k-1$ of all y-roots of $Q(x,y)$, i is a natural number denoting the recursion level (starting in level 0), and $\mathcal{L}$ is a global list in which polynomials calculated during the execution of the algorithm are stored. At the end, $\mathcal{L}$ includes all y-roots of the initial $Q(x,y)$ and no further elements.

At the beginning of the algorithm, we ensure that $k-1$ is the largest recursion level which can occur (cf. line 1). In line 2, the bivariate polynomial $M(x,y)$ is calculated by dividing $Q(x,y)$ with x^r for the largest possible $r \in \mathbb{N}$ that divides $Q(x,y)$. Line 3 defines the polynomial $p(y)$ to be $M(0,y)$. The calculation of roots in line 4 can be carried out with any suitable algorithm, for example by using exhaustive search. In each iteration, $g(x)$ is removed from the list $\mathcal{L}$ (cf. line 5). For each root γ of $p(y)$, which is calculated in line 4, the polynomial $g(x) + \gamma x^i$ (where $i > deg(g(x))$) is appended to $\mathcal{L}$ (cf. line 7) and delivered to the next recursion level in line 8. If the polynomial $p(y)$ does not possess any roots, a

polynomial of the form

$$g(x) + \sum_{i=\mathsf{deg}(g(x))+1}^{k-1} \gamma_i x^i,$$

which is a y-root of $Q(x, y)$ does not exist and consequently, the recursion terminates. For an extensive explanation and analysis of list decoding and the Roth–Ruckenstein algorithm, we refer to [Rot06, Chapter 9].

To obtain a constant runtime that does not depend on the received word $\boldsymbol{y}$, we modify Algorithm 11 as follows:

1. The i-th recursion level of all recursive calls is computed before starting any calculations at recursion depth $i + 1$.

2. After completing calculations at recursion depth i for all recursive calls, random univariate polynomials of degree $\leq i$ are stored in list $\mathcal{L}$, such that $\mathcal{L}$ always contains $\ell(k-1)$ polynomials. Furthermore, the random polynomials are marked as random. This approach is visualized in Figure 6.4

3. At recursion depth $i+1$, for all polynomials in $\mathcal{L}$, the corresponding bivariate polynomial $M(x, x(y-\gamma))$ serves as input for the algorithm. For all random polynomials in $\mathcal{L}$, the algorithm is called with a random bivariate polynomial of y-degree $\leq \ell$. In this case, the result of the calculation is not stored in $\mathcal{L}$.

After executing the algorithm, all elements in $\mathcal{L}$ that are not marked as random are the same elements than calculated with the unmodified Roth–Ruckenstein algorithm. From this property, we can conclude the correctness of the modified Roth–Ruckenstein algorithm.

We proof that the modified Roth–Ruckenstein algorithm always performs the same number of field operations and that this number is independent of $Q(x, y)$.

Theorem 6.4. Consider Algorithm 11, including the modifications listed above. The algorithm

$$\mathrm{RR}\left(Q(x, y), 0, 0, \{0\}\right)$$

calls $\mathrm{RR}(\cdot)$ exactly $\ell^2(k-1)$ times.

Proof. The original Roth–Ruckenstein algorithm calls itself $\leq \ell(k-1)$ times (cf. [RR00]). Hence, the number of *non-random* entries in list $\mathcal{L}$ will never be $\geq \ell(k-1)$. At recursion depth i, for $i = 1, \ldots, k$, $\mathrm{RR}(\cdot)$ is called exactly $\ell(k-1)$ times since $|\mathcal{L}| = \ell(k-1)$. □

Theorem 6.5. The number of multiplications and additions needed by Algorithm 11 for fixed parameters is independent of $Q(x, y)$.

Proof. We know that

$$\mathsf{deg}p(y) \leq \ell,$$

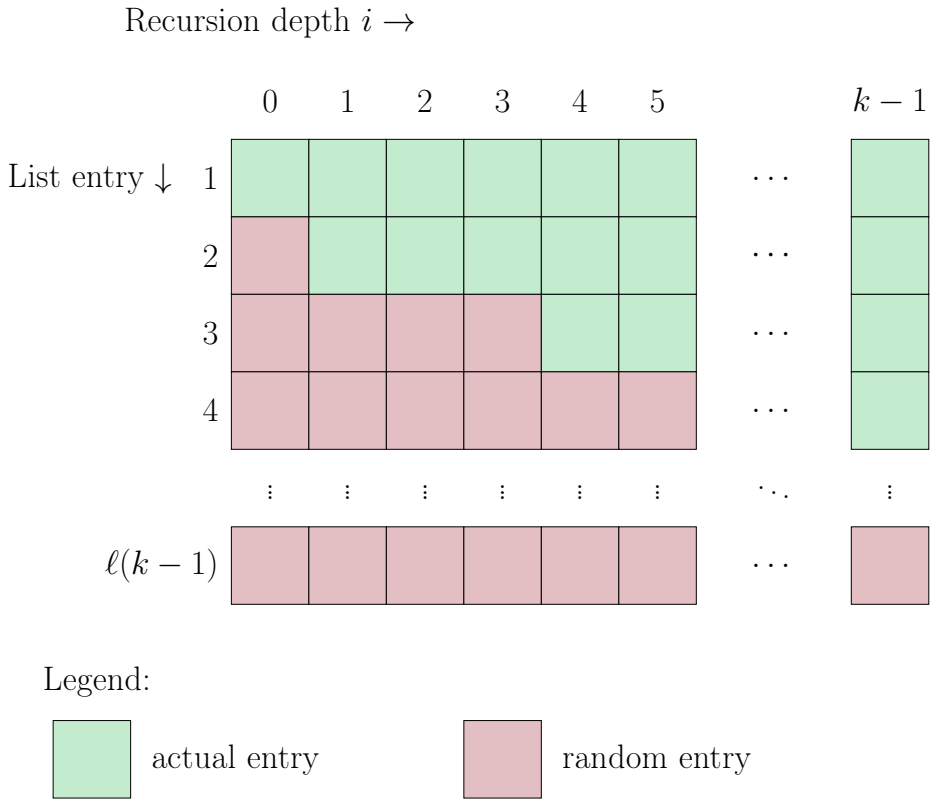


Figure 6.4: As one property of the modified Roth-Ruckenstein algorithm, the list $\mathcal{L}$ always contains $\ell(k-1)$ polynomials. To achieve this property, random univariate polynomials of degree $\leq i$ are stored in $\mathcal{L}$ at each recursion depth. Those polynomials are marked as random and are visualized by the red list entries. The green list entries represent the actual polynomials.

so evaluation corresponds to $\ell + 1$ multiplications and ℓ additions of field elements. Finding the roots of polynomial $p(y)$ can be accomplished by evaluating it at all elements of $\mathbb{F}_q$. In recursion depth i, we have

$$\mathsf{deg} M_\eta(x) \leq \max_\mu \{\mathsf{deg} Q_\mu\} + \ell i.$$

Hence, the calculation of $M(x, x(y-\gamma))$ can be performed in constant time, since we can treat $M_\eta(x)$ as a polynomial that exactly has degree

$$\max_\mu \{\mathsf{deg} Q_\mu\} + \ell i.$$

Finding r depends on the data structures. Obtaining $Q(x,y)/x^r$ and $M(0,y)$ efficiently requires no computation. □

Decoding Reed–Muller Codes in Constant Time

In the concatenated codes constructed in Chapter 5, the inner codes $\mathcal{B}$ have a comparably small dimension, and hence, a low cardinality. This allows to efficiently perform maximum

likelihood decoding. For this purpose, we calculate the distances $d_1, \dots d_{2^k}$ between the received word $\boldsymbol{c} + \boldsymbol{e}$ and the 2^k codewords of the inner code $\mathcal{B}$. Let π denote a random permutation of the indices $\{1, \dots, 2^k\}$ of the codewords of the inner code $\mathcal{B}$. Further, let $(d_{\pi(1)}, \dots d_{\pi(2^k)})$ be the ordered list of Hamming distances of the received word $\boldsymbol{c}+\boldsymbol{e}$ and the permuted list of codewords. We prove that if an attacker manages to extract the ordered list of Hamming distances, the uncertainty of the codewords is not influenced.

Theorem 6.6. $H(\boldsymbol{c} \,|\, (d_{\pi(1)}, \dots, d_{\pi(2^k b)})) = H(\boldsymbol{c})$.

Proof. Let $\mathcal{B}$ denote the inner code of a concatenated scheme. Since

$$d_{\pi(i)} = \mathrm{dist_H}(\boldsymbol{c} + \boldsymbol{e}, \boldsymbol{c}_{\pi(i)}) = \mathrm{dist_H}(\boldsymbol{c}' + \boldsymbol{c} + \boldsymbol{e}, \boldsymbol{c}' + \boldsymbol{c}_{\pi(i)}) \tag{6.10}$$

for any codeword $\boldsymbol{c}' \in \mathcal{B}$ and we can define another permutation π' such that

$$\boldsymbol{c}_{\pi'(i)} = \boldsymbol{c}' + \boldsymbol{c}_{\pi(i)} \tag{6.11}$$

(adding a codeword is a bijection on the code),

$$d_{\pi(i)} = \mathrm{dist_H}(\boldsymbol{c}' + \boldsymbol{c} + \boldsymbol{e}, \boldsymbol{c}_{\pi'(i)}). \tag{6.12}$$

We conclude that the uncertainty of choosing a codeword $\boldsymbol{c}'$ remains. □

6.4 Concluding Remarks

Side-channel attacks and their countermeasures gained considerable attention during the last years and provide a large field of active research. It is of utmost importance to protect implementations of cryptographic systems against attackers, which use side-channels like timing behavior, power consumption or electromagnetic radiation. In this chapter, we considered timing attacks on PUFs. In particular, we studied side-channel attacks, which extract information from the timing behavior of the decoding algorithm. Following this perception, we differ from literature, which most often considers side-channel attacks on the helper data.

First, we recalled a technique for codeword masking according to [MSS13], in order to protect an implementation from side-channel attacks in general. The main idea is to randomly add up vectors in order to result in a calculation that is independent of the received word. In addition to the original work in [MSS13], we provided a formal proof that the masking scheme has no impact on the uncertainty.

The Guruswami–Sudan decoding algorithm that uses the modified Roth–Ruckenstein algorithm, which we proposed in this chapter, provides resilience against side-channel attacks on the runtime of the decoding algorithm. Additionally, the algorithm allows to increase the number of errors that can be corrected, since it implements list decoding, a decoding paradigm, which is able to correct errors beyond half-the-minimum distance and which is applied for the first time to PUFs. There exist techniques to further increase the decoding performance, that have not been investigated concerning their suitability in the context of PUFs. These techniques comprise list recovery according to [GR05] and a soft-decision variant of the Guruswami–Sudan algorithm developed by Kötter and Vardy in [KV03]. The latter can also be utilized with the modified Roth–Ruckenstein algorithm.

In summary, we proposed a faster implementation, which requires less chip area and at the same time provides resistance to side-channel attacks. The countermeasures discussed in this chapter indicate a bunch of future studies. Until now, there exists no hardware implementation of the modified Roth–Ruckenstein algorithm that was proposed in this chapter. Also, studying the resilience of a side-channel attack for a corresponding implementation has to be done within further research. In addition to timing attacks, DPA attacks on the decoding algorithm need to be studied. A possible approach can be the combination of the methods proposed in this chapter and the DPA-resistant logic styles discusses in [WMG18]. In addition, it is important to isolated study DPA resistance of field operations.

7

Conclusion

Physical Unclonable Functions are usually studied from the perspective of hardware engineering. In this dissertation we have covered many aspects from the perspective of coding theory, since the application of error-correcting codes is indispensable in order to guarantee a reliable reproduction of PUF responses.

In Chapter 3, we started with the derivation of channel models for ROPUFs and DRAM PUFs. Knowledge about a channel is mandatory in order to select or to design a suitable error-correcting code. Studying the underlying channel has been often neglected in the literature. Instead, a binary symmetric channel with a fixed worst-case bit error probability was assumed. First, based on real world data, we derived a channel model for ROPUFs. We applied a methodology analog to the derivation of an SRAM channel model proposed in the literature. We derived cumulative distribution functions and probability density functions for both, the one-probability and the error probability of a response bit. In addition to the code design, knowledge about the channel can be used when constructing secure sketches.

Second, we followed a comparatively new direction and generated PUF responses from ordinary DRAM. As additional problem that occurs when using DRAM, the extracted data are biased, and hence, cannot be used directly as input for a secure sketch. Thus, we proposed debiasing methods. Two of them resulted in error models that have not been obtained for PUFs before, namely a Z-channel and an erasure channel.

Chapter 4 dealt with secure sketches, which are indispensable in order to correct fuzzy PUF responses. In most proposals, helper data are used to map a response to a codeword of an error-correcting code, and hence, to enable error correction. We discussed and analyzed an approach that only needs an error-correcting code without any further helper data. Helper data can be omitted, when the initial PUF response is a codeword of the error-correcting code that was chosen to implement the error correction unit. Our approach to provide this guarantee is to design an error-correcting code in the initialization phase of the secure sketch, depending on the initial PUF response. To present the concept, we used LDPC codes due to the simplicity of the code construction. The problem, whether or not other code classes can be used to construct a code, such that a given vector is a codeword, has not yet been solved.

Furthermore, based on the derived ROPUF channel model, we proposed two soft-decision secure sketches for ROPUFs. The first one works analog to a soft-decision secure sketch that was proposed for SRAM PUFs in the literature. In general, soft-decision secure sketches improve the decoding performance and consequently also the reliability, when regenerating PUF responses. The second secure sketch is based on an AWGN channel and considerably outperforms previous results from literature.

In Chapter 5, we focused on the error correction component of a secure sketch. We proposed

code constructions for block codes as well as for convolutional codes in order to improve results from literature. Concerning block codes, we proposed several concatenated codes based on Reed–Muller, Reed–Solomon and BCH codes. Using these concatenated codes, we showed how to improve an existing reference implementation from the literature concerning codeword length, block error probability and required chip area. For the first time, the concepts of generalized concatenated codes, generalized minimum distance decoding and power decoding were used for decoding in the context of PUFs.

Further, we showed that scenarios based on convolutional codes can be improved by using soft information as input to the decoding algorithm and by applying the list decoding paradigm. Further, previous results in the literature imply, that using convolutional codes for PUFs requires a comparatively large constraint length, which cannot be handled reasonably by the Viterbi decoding algorithm. In contrast, we verified that this problem can be circumvented by using sequential decoding.

Chapter 6 finally touched the huge field of side-channel attacks on PUFs. In particular, timing attacks on the decoding algorithm were investigated. In order to circumvent such attacks, we studied two solutions, a masking scheme as proposed in the literature and a constant time decoding algorithm. The latter is based on the list decoding paradigm and can be seen as the main contribution of the chapter. In addition to preventing timing-attacks, the proposed algorithm extends the error correction radius when reproducing PUF responses.

Open research problems and links for future studies have been given within the concluding remarks of the corresponding chapters.

A

PUF Characterization

THROUGHOUT this dissertation, we use practical data from ROPUFs, as measured and published in [MCMS10]. Based on these data, we derive a channel model for ROPUFs in Chapter 3.2, which we use to propose soft-decision secure sketches in Chapter 4.3. In addition, we evaluate an error correction scheme based on convolutional codes in Chapter 5.2.4.

The analysis of ring oscillator frequencies in [MCMS10] was performed by using 125 devices, which was the original size of the ROPUF data set. Meanwhile, the data set contains data of 193 devices. In order to derive the ROPUF channel model in Section 3.2 as well as the soft-decision helper data algorithms in Section 4.3, we analyzed the extended data set again within the scope of a master's thesis [Stu17]. Figures A.1–A.8 visualize the results of the analysis of the extended data set and are used with kind permission of Veniamin Stukalov. For comparability, the analyzed quality measures (recall Chapter 2, Definition 2.2) are the same as in the original work.

Table A.1 compares the results obtained from the extended data set with the original results from [MCMS10]. Note that the increase of devices rarely changes the quality parameters. Hence, the ROPUF construction implemented in [MCMS10] can be considered as highly stable, what also emphasizes the robustness of the channel model and the soft-decision helper data algorithms introduced in this dissertation.

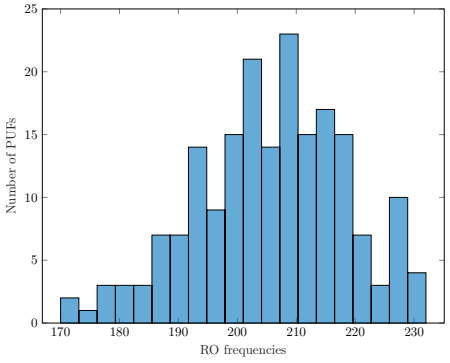


Figure A.1: Average ring oscillator frequencies of the PUFs [Stu17].

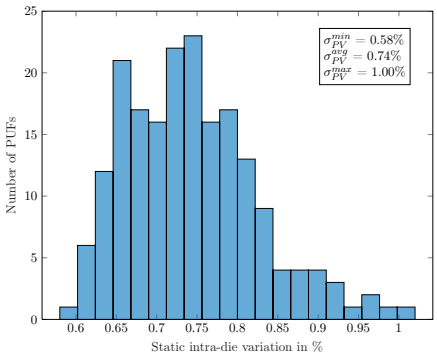


Figure A.2: Static variation of the PUF intra-response distance [Stu17].

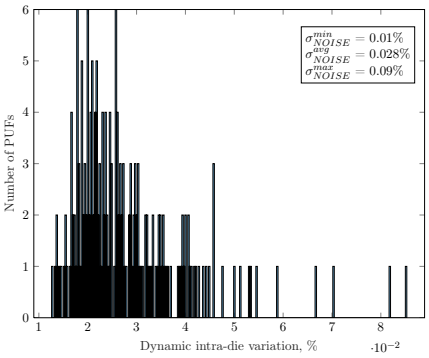


Figure A.3: Dynamic variation of the PUF intra-response Hamming distance [Stu17].

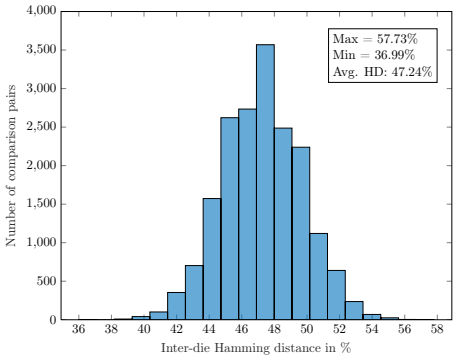


Figure A.4: Distribution of the PUF inter-response distance [Stu17].

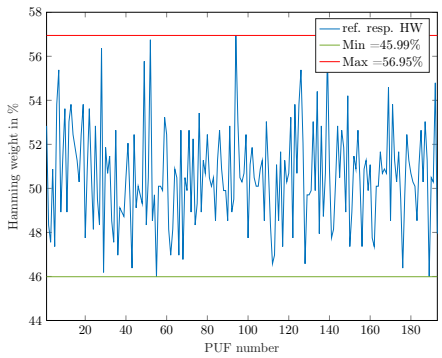


Figure A.5: Percentage Hamming weight of the PUF responses [Stu17].

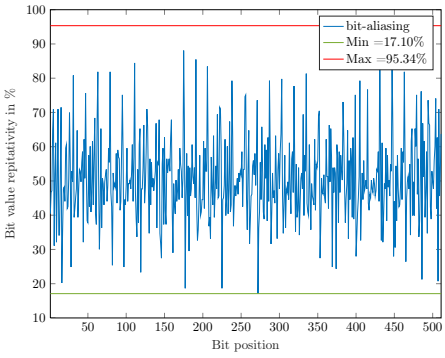


Figure A.6: Bit-aliasing of the response positions [Stu17].

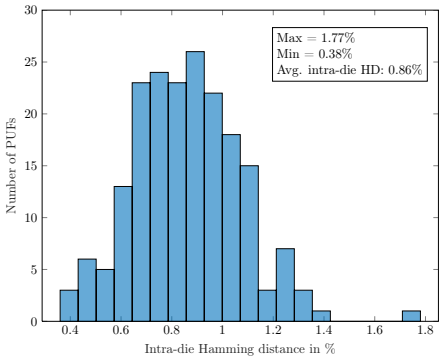


Figure A.7: Distribution of the PUF intra-response distance [Stu17].

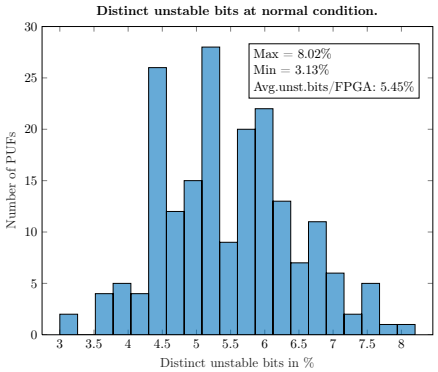


Figure A.8: Distinct unstable bits [Stu17].

Table A.1: Comparison of the results obtained in the analysis based on the extended data set in [Stu17] and the original results published in [MCMS10]. The differences in the results are negligible, what implies the stability of the ROPUF construction from [MCMS10].

		[Stu17] (193 devices)	[MCMS10] (125 devices)
d_{AVG}	Min.	171.66 MHz	171.66 MHz
	Mean	205.7 MHz	205.10 MHz
	Max.	230.24 MHz	230.24 MHz
Δd_{AVG}	σ_F	6.16%	6.61%
	σ_F	12.67 MHz	13.54 MHz
Δd_{PV}	$\sigma_{PV,min}$	0.58%	0.58%
	$\bar{\sigma}_{PV}$	0.74%	0.74%
	$\sigma_{PV,max}$	1.0%	1.0%
Δd_{NOISE}	$\bar{\sigma}_{NOISE}$	0.028%	0.025%
Inter-die HD	Min.	36.99%	38.98%
	Mean	47.24%	47.31%
	Max.	57.73%	53.36%
Hamming weight	Min.	45.99%	45.98%
	Mean	50.56%	50.72%
	Max.	56.95%	56.94%
Bit-aliasing	Mean	50.56%	50.72%
	Max.	95.34%	96.80%
Intra-die HD	Min.	0.38%	0.38%
	Mean	0.86%	0.86%
	Max.	1.77%	1.39%
Unstable bits	Min.	3.13%	3.13%
	Mean	5.45%	5.36%
	Max.	8.02%	7.63%

When PUF constructions are proposed or studies are performed based on real hardware, conclusions are usually drawn from a very small amount of devices. Table A.2 provides a selection of the number of PUFs used in several studies in the literature. Comparing with those numbers, the 193 devices which are used in [MCMS10] can be considered as a large number, what justifies the usage of the available data set within this dissertation.

Table A.2 is structured as follows:

- **Construction:** PUF construction for which the studies in the stated references were conducted.
- **Reference:** Provides the reference in which the studies were performed.
- # **devices (uniqueness):** Number of devices used for evaluating uniqueness. Some of the studies only deal with reproducibility, what is indicated by the entry "not studied" in the column for uniqueness.
- # **devices (reproducibility):** Number of devices used for evaluating reproducibility. In some publications it is not clearly stated, how many PUFs are used in order to evaluate reproducibility. On occasion, it is ambiguous, whether one PUF was selected as an example to present the results, or indeed only one PUF was used in the experiment. Sometimes, it is indirectly implied that the number of PUFs for evaluating reproducibility is the same number than used for uniqueness. In all such cases, the entry "not stated" is used in the table.
- # **responses:** Number of response extractions per PUF, used for evaluating reproducibility. For evaluating uniqueness, only one response per PUF, the so-called reference response, is required. In some publications, the exact number of extracted PUF responses is not precisely stated.

Note that some large numbers of devices in the table actually do not mean physical devices, as indicated by the subscripts. Rather, virtual devices are used, i.e., one physical object is logically separated into several PUFs. For example, the 160 devices used in [HBF07] are located on only 8 physical devices.

Table A.2: Number of PUFs that are used in practical experiments in the literature. *Devices and measurements are the ones that were used to generate the dataset published in [MCMS10]. $^{\diamond}$Virtual devices, different SRAM blocks on a single device. $^{\triangleleft}$Virtual devices, two devices which each implement two PUFs. *Virtual devices located on eight physical devices.

Construction	**Reference**	# **devices** (uniqueness)	# **devices** (reproducibility)	# **responses** per device
Arbiter PUF	[LLG+04]	37	not stated	10.000
DRAM	[FRC+12]	346	346	not stated
	[RFC+13]	346 (266)	346 (266)	not stated
	[KGKF14]	not studied	1	$>$ 100/setting
	[TKXC15]	8	3	10
	[RHHF16]	10	not stated	21
	[SRR16]	5	5	50
ROPUF	[SD07]	15	15	not stated
	[MS09]	5	not stated	not stated
	[YQ10]	9	9	64
	[MKS12]	125*	not stated	100*
	[MVHV12]	10	not stated	not stated
SRAM	[GKST07]	17$^{\diamond}$	4$^{\triangleleft}$	92
	[HBF07]	160*	160*/10/3	100/30/50
	[MTV09a]	not studied	not stated	not stated
Butterfly PUF	[KGM+08]	36	36	200
Latch PUF	[SHO07]	19	19	not stated
Flip-flop PUF	[MTV08]	3	3	101
	[VdLSHT10]	40	40	50

B

Error Correction for PUFs

Table B.1 exemplifies which coding schemes have been proposed in the literature for error correction in the context of PUFs. The table does not claim to be complete, since there exists a multiplicity of publications about that topic. We selected the most prominent publications for the overview. Furthermore, we tried to find publications for different types of PUFs. The table is structured according to different PUF constructions. In the following, we explain the columns of the table:

- **Construction:** PUF construction, for which the code constructions have been developed, e.g., Arbiter PUF, ROPUF, SRAM PUF, etc.
- **Reference:** Provides the reference in which a construction was proposed.
- **Code:** States the used code construction. Note that "OCC" is the abbreviation for "ordinary code concatenation" and the operator $\circ$ represents concatenation.
- **Error model:** States the error model that was assumed in the corresponding work.
- **WEP:** Word error probability that was achieved for a construction. This represents the probability, that a key is erroneously regenerated.
- **Key:** Length of the final key, in bits.
- **Secret:** Dimension of the code. This size is usually larger than the key length.
- # **Bits:** Number of bits that are extracted from the PUF.

Unfortunately, authors often describe these information in a vague manner or even omit to state a part of them. This explains, why the information in the table are not complete for all code constructions. In particular, for DRAM PUFs, no exhaustive studies which focus on error correction exist.

Also, for all kinds of PUFs, there exists a multitude of studies in which error correction is not addressed at all. Often, general statements that results can be improved by applying error-correcting codes are included instead.

Table B.1: Coding schemes for Physical Unclonable Functions proposed in the literature.

Construction	**Reference**	**Code**	**Error model**	**WEP**	**Key**	**Secret**	**# Bits**
Arbiter PUF	[Suh05]	$\mathcal{BCH}$(255,63,61)	BSC($p_b = 0.048$)	$2.4 \cdot 10^{-6}$	63	63	255
DRAM PUF	[KGKF14]	(31,26,3) Hamming code	not stated	not stated	256	256	512 kbit
	[LZLL14]	$\mathcal{BCH}$(255,37,45)	not stated	not stated	128	592	4080
ROPUF	[SD07]	$\mathcal{BCH}$(127,64,21)	not specified	$5 \cdot 10^{-11}$	127	127	127
SRAM PUF	[GKST07]	$\mathcal{BCH}$(511,19,239)	BSC($p_b = 0.15$)	10^{-6}	128	171	4599
		$\mathcal{BCH}$(1023,278,205)	BSC($p_b = 0.06$)	10^{-6}	128	171	1023
	[BGS+08]	Repetition (33,1,33)	BSC($p_b = 0.15$)	1.0010^{-6}	128	171	5643
		$\mathcal{RM}$(256,9,128)	BSC($p_b = 0.15$)	2.0410^{-5}	128	171	4864
		$\mathcal{RM}$(512,10,256)	BSC($p_b = 0.15$)	2.5410^{-9}	128	171	9216
		Golay(23,12,7)	BSC($p_b = 0.15$)	0.4604	128	171	345
		$\mathcal{BCH}$(511,19,239)	BSC($p_b = 0.15$)	$2.97 \cdot 10^{-7}$	128	171	4599
		$\mathcal{BCH}$(1023,46,439)	BSC($p_b = 0.15$)	$1.85 \cdot 10^{-8}$	128	171	4092
		Shortened $\mathcal{BCH}$(1023,43,439)	BSC($p_b = 0.15$)	$1.44 \cdot 10^{-8}$	128	171	4080
		OCC: $\mathcal{BCH}$(127,29,43)$\circ$Rep.(3,1,3)	BSC($p_b = 0.15$)	$8.48 \cdot 10^{-6}$	128	171	2286
		OCC: $\mathcal{RM}$(64,7,32)$\circ$Rep.(3,1,3)	BSC($p_b = 0.15$)	$1.02 \cdot 10^{-6}$	128	171	4800
		OCC: $\mathcal{BCH}$(63,7,31)$\circ$Rep.(3,1,3)	BSC($p_b = 0.15$)	$8.13 \cdot 10^{-7}$	128	171	4725
		OCC: $\mathcal{RM}$(32,6,16)$\circ$Rep.(5,1,5)	BSC($p_b = 0.15$)	$1.49 \cdot 10^{-6}$	128	171	4640
		OCC: $\mathcal{BCH}$(226,86,43)$\circ$Rep.(5,1,5)	BSC($p_b = 0.15$)	$2.28 \cdot 10^{-7}$	128	171	2260
		OCC: Golay(23,12,7)$\circ$Rep.(7,1,7)	BSC($p_b = 0.15$)	$1.58 \cdot 10^{-4}$	128	171	2415
		OCC: Golay(20,9,7)$\circ$Rep.(7,1,7)	BSC($p_b = 0.15$)	$8.89 \cdot 10^{-5}$	128	171	2660
		OCC: $\mathcal{BCH}$(255,171,23)$\circ$Rep.(7,1,7)	BSC($p_b = 0.15$)	$8.0 \cdot 10^{-5}$	128	171	1785
		OCC: $\mathcal{RM}$ (16,5,8)$\circ$Rep.(7,1,7)	BSC($p_b = 0.15$)	$3.47 \cdot 10^{-5}$	128	171	3920
		OCC: $\mathcal{BCH}$(113,57,19)$\circ$Rep.(7,1,7)	BSC($p_b = 0.15$)	$1.34 \cdot 10^{-6}$	128	171	2373
		OCC: $\mathcal{BCH}$(121,86,11)$\circ$Rep.(9,1,9)	BSC($p_b = 0.15$)	$6.84 \cdot 10^{-5}$	128	171	2178
		OCC: Golay(23,12,7)$\circ$Rep.(9,1,9)	BSC($p_b = 0.15$)	$8.0 \cdot 10^{-6}$	128	171	3105
		OCC: $\mathcal{RM}$(16,5,8)$\circ$Rep.(9,1,9)	BSC($p_b = 0.15$)	$1.7 \cdot 10^{-6}$	128	171	5040
		OCC: Golay(24,13,7)$\circ$Rep.(11,1,11)	BSC($p_b = 0.15$)	$5.41 \cdot 10^{-7}$	128	171	3696
		OCC: Golay(23,12,7)$\circ$Rep.(11,1,11)	BSC($p_b = 0.15$)	$4.52 \cdot 10^{-7}$	128	171	3795
	[MTV09a]	$\mathcal{BCH}$(31,6)	[MTV09a]	$10^{-4} \leq 10^{-3}$	128	171	$\approx$ 900

		$\mathcal{BCH}$(1020,43)	[MTV09a]	$10^{-8} \leq 10^{-7}$	128	171	$\approx$ 4100
		Rep.(19,1,19)	[MTV09a]	$10^{-7} \leq 10^{-6}$	128	171	$\approx$ 3250
		OCC: Rep.(3,1)$\circ\mathcal{BCH}$(15,7)	[MTV09a]	$\approx 10^{-4}$	128	171	$\approx$ 1100
		OCC: Rep.(5,1)$\circ\mathcal{BCH}$(226,86)	[MTV09a]	$10^{-7} \leq 10^{-6}$	128	171	$\approx$ 2250
		OCC: Rep.(5,1)$\circ\mathcal{BCH}$(15,5)	[MTV09a]	$\approx 10^{-7}$	128	171	$\approx$ 2600
		OCC: Rep.(4,1)$\circ\mathcal{RM}$(32,16)	[MTV09a]	$10^{-7} \leq 10^{-6}$	128	171	$\approx$ 1400
		OCC: Rep.(6,1)$\circ\mathcal{RM}$(8,4)	[MTV09a]	$10^{-7} \leq 10^{-6}$	128	171	$\approx$ 2100
		OCC: Rep.(3,1)$\circ\mathcal{RM}$(64,22)	[MTV09a]	$\approx 10^{-8}$	128	171	$\approx$ 1600
		OCC: $\mathcal{RM}$(32,6)$\circ\mathcal{RM}$(8,4)	[MTV09a]	$10^{-7} \leq 10^{-6}$	128	171	$\approx$ 2100
		OCC: $\mathcal{RM}$(32,6)$\circ\mathcal{RM}$(64,42)	[MTV09a]	$\approx 10^{-8}$	128	171	$\approx$ 2050
Flip-flop PUF	[MTV08]	$\mathcal{BCH}$(255,47,42)	BSC ($p_b = 0.05$)	10^{-11}	128	141	765

Bibliography

References

[ACLY00] Rudolf Ahlswede, Ning Cai, Shuo-Yen R. Li, and Raymond W. Yeung. Network Information Flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, 2000.

[AGM+15] Aydin Aysu, Ege Gulcan, Daisuke Moriyama, Patrick Schaumont, and Moti Yung. End-to-end Design of a PUF-based Privacy Preserving Authentication Protocol. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 556–576. Springer, 2015.

[Ale02] Michael Alekhnovich. Linear Diophantine Equations over Polynomials and Soft Decoding of Reed-Solomon Codes. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings*, pages 439–448. IEEE, 2002.

[AWSO17] Aydin Aysu, Ye Wang, Patrick Schaumont, and Michael Orshansky. A New Maskless Debiasing Method for Lightweight Physical Unclonable Functions. In *2017 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 134–139. IEEE, 2017.

[Axl15] Sheldon Axler. *Linear Algebra Done Right*. Springer, 2015.

[BB13] Martin Bossert and Sergey Bezzateev. A Unified View on Known Algebraic Decoding Algorithms and New Decoding Concepts. *IEEE Transactions on Information Theory*, 59(11):7320–7336, 2013.

[BBD09] Daniel Bernstein, Johannes Buchmann, and Erik Dahmen. *Post-Quantum Cryptography*. Springer, 2009.

[BDH+10] Ileana Buhan, Jeroen Doumen, Pieter Hartel, Qian Tang, and Raymond Veldhuis. Embedding Renewable Cryptographic Keys into Noisy Data. *International Journal of Information Security*, 9(3):193–208, 2010.

[Ber66] Elwyn R. Berlekamp. Non-binary BCH Decoding. Technical report, North Carolina State University. Dept. of Statistics, 1966.

[BGS+08] Christoph Bösch, Jorge Guajardo, Ahmad-Reza Sadeghi, Jamshid Shokrollahi, and Pim Tuyls. Efficient Helper Data Key Extractor on FPGAs. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 181–197. Springer, 2008.

[BH86] Martin Bossert and Ferdinand Hergert. Hard-and Soft-Decision Decoding Beyond the Half Minimum Distance—An Algorithm for Linear Codes. *IEEE Transactions on Information Theory*, 32(5):709–714, 1986.

[BH12] Christoph Böhm and Maximilian Hofer. *Physical Unclonable Functions in Theory and Practice*. Springer Science & Business Media, 2012.

[BKY03] Daniel Bleichenbacher, Aggelos Kiayias, and Moti Yung. Decoding of Interleaved Reed Solomon Codes over Noisy Data. In *International Colloquium on Automata, Languages, and Programming*, pages 97–108. Springer, 2003.

[Bla03] Richard E. Blahut. *Algebraic Codes for Data Transmission*. Cambridge University Press, 2003.

[BM16] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 602–624. Springer, 2016.

[Bos99] Martin Bossert. *Channel Coding for Telecommunications*. John Wiley & Sons, Inc., 1999.

[Bos12] Martin Bossert. *Einführung in die Nachrichtentechnik*. Walter de Gruyter, 2012.

[Bos13] Martin Bossert. *Kanalcodierung*. Walter de Gruyter, 2013.

[BRC60a] Raj C. Bose and Dwijendra K. Ray-Chaudhuri. Further Results on Error Correcting Binary Group Codes. *Information and Control*, 3(3):279–290, 1960.

[BRC60b] Raj C. Bose and Dwijendra K. Ray-Chaudhuri. On a Class of Error Correcting Binary Group Codes. *Information and Control*, 3(1):68–79, 1960.

[BRS+10] Lawrence E. Bassham, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Elaine B. Barker, Stefan D. Leigh, Mark Levenson, Mark Vangel, David L. Banks, et al. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. *Special Publication (NIST SP) - 800-22 Rev 1a*, 2010.

[BZ74] E. L. Blokh and Victor V. Zyablov. Coding of Generalized Cascade Codes. *Problemy Peredachi Informatsii*, 10(2):45–50, 1974.

[Can15] Canon. C2V Connected Product Description, 2015. Accessed: 2019-04-11. URL: `http://www.canon-its.com.cn/EN/Connected.html`.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private Information Retrieval. In *Annual Symposium on Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[DER17] Iain S. Duff, Albert Maurice Erisman, and John K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 2017.

[DGW+10] Alexandros G. Dimakis, P. Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, 2010.

[Djo12] Ivan Djordjevic. *Quantum Information Processing and Quantum Error Correction: An Engineering Approach*. Elsevier LTD, 2012.

[DLP+01] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert Van Doorn, Sean W. Smith, and Steve Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 34(10):57–66, 2001.

[DORS08] Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin, and Adam Smith. Fuzzy Extractors: How to Generate Strong keys from Biometrics and Other Noisy Data. *SIAM Journal on Computing*, 38(1):97–139, 2008.

[DRS04] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. Fuzzy Extractors: How to Generate Strong Keys from Biometrics and other Noisy Data. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 523–540. Springer, 2004.

[DW09] Jianwei Dai and Lei Wang. A Study of Side-Channel Effects in Reliability-Enhancing Techniques. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'09)*, pages 236–244. IEEE, 2009.

[DXAGL03] Ivana Djurdjevic, Jun Xu, Khaled Abdel-Ghaffar, and Shu Lin. A Class of Low-Density Parity-Check Codes Constructed Based on Reed-Solomon Codes with Two Information Symbols. In *International Symposium on Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes*, pages 98–107. Springer, 2003.

[Eli55] Peter Elias. Coding for Noisy Channels. *IRE Conv. Rec.*, 3:37–46, 1955.

[EO02] Evangelos Eleftheriou and Sedat Olcer. Low-Density Parity-Check Codes for Digital Subscriber Lines. In *IEEE International Conference on Communications*, volume 3, pages 1752–1757. IEEE, 2002.

[ETC17] Charles Eckert, Fatemeh Tehranipoor, and John A. Chandy. DRNG: DRAM-based Random Number Generation using its Startup Value Behavior. In *Proceedings of the 60th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), Boston, MA, USA*, pages 6–9, 2017.

[Fan63] Robert Fano. A Heuristic Discussion of Probabilistic Decoding. *IEEE Transactions on Information Theory*, 9(2):64–74, 1963.

[Fan01] John L. Fan. Array Codes as LDPC Codes. In *Constrained Coding and Soft Iterative Decoding*, pages 195–203. Springer, 2001.

[FH07] Justus Ch. Fricke and Peter A. Hoeher. Word Error Probability Estimation by Means of a Modified Viterbi Decoder. In *IEEE 66th Vehicular Technology Conference, 2007. VTC-2007 Fall*, pages 1113–1116. IEEE, 2007.

[FJ66a] G. D. Forney Jr. Concatenated Codes. *Technical Report 440, Massachusetts Institute of Technology*, 1966.

[FJ66b] G. D. Forney Jr. Generalized Minimum Distance Decoding. *IEEE Trans. on Inf. Theory*, 12(2):125–131, 1966.

[FRC+12] Daniel Fainstein, Sami Rosenblatt, Alberto Cestero, Norman Robson, Toshiaki Kirihata, and Subramanian S. Iyer. Dynamic Intrinsic Chip ID using 32nm High-K/metal Gate SOI Embedded DRAM. In *Symposium on VLSI Circuits (VLSIC)*, pages 146–147. IEEE, 2012.

[Gal63] Robert Gallager. *Low-Density Parity-Check Codes*. PhD thesis, Massachusetts Institute of Technology, 1963.

[Gas03] Blaise Gassend. Physical Random Functions. Master's thesis, Massachusetts Institute of Technology, 2003.

[GCVDD02] Blaise Gassend, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Silicon Physical Random Functions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 148–160. ACM, 2002.

[Geb09] Catherine H. Gebotys. *Security in Embedded Devices*. Springer Science & Business Media, 2009.

[GKKG02] Warren J. Gross, Frank R. Kschischang, Ralf Koetter, and P. Glenn Gulak. A VLSI Architecture for Interpolation in Soft-Decision List Decoding of Reed-Solomon Codes. In *IEEE Workshop on Signal Processing Systems, 2002.(SIPS'02)*, pages 39–44. IEEE, 2002.

[GKST07] Jorge Guajardo, Sandeep S. Kumar, Geert-Jan Schrijen, and Pim Tuyls. FPGA Intrinsic PUFs and their Use for IP Protection. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 63–80. Springer, 2007.

[GM89] Sidney N. Graybeal and Patricia B. McFate. Getting out of the STARTing Block. *Scientific American*, 261(6):61–67, 1989.

[GR05] Venkatesan Guruswami and Atri Rudra. Limits to List Decoding Reed-Solomon Codes. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 602–609. ACM, 2005.

[GS98] Venkatesan Guruswami and Madhu Sudan. Improved Decoding of Reed-Solomon and Algebraic-Geometric Codes. In *39th Annual Symposium on Foundations of Computer Science, 1998. Proceedings*, pages 28–37. IEEE, 1998.

[GZ61] Daniel Gorenstein and Neal Zierler. A Class of Error-Correcting Codes in p^m Symbols. *Journal of the Society for Industrial and Applied Mathematics*, 9(2):207–214, 1961.

[HBF07] Daniel E. Holcomb, Wayne P. Burleson, and Kevin Fu. Initial SRAM State as a Fingerprint and Source of True Random Numbers for RFID Tags. In *Proceedings of the Conference on RFID Security*, volume 7, page 2, 2007.

[HDWH12] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *USENIX Security Symposium*, volume 8, page 1, 2012.

[HH89] Joachim Hagenauer and Peter Hoeher. A Viterbi Algorithm with Soft-Decision Outputs and its Applications. In *Global Telecommunications Conference and Exhibition'Communications Technology for the 1990s and Beyond'(GLOBECOM), 1989. IEEE*, pages 1680–1686. IEEE, 1989.

[HLS14] Matthias Hiller, Leandro Rodrigues Lima, and Georg Sigl. Seesaw: An Area-Optimized FPGA Viterbi Decoder for PUFs. In *17th Euromicro Conference on Digital System Design*, pages 387–393. IEEE, 2014.

[HMSS12] Matthias Hiller, Dominik Merli, Frederic Stumpf, and Georg Sigl. Complementary IBS: Application Specific Error Correction for PUFs. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–6. IEEE, 2012.

[Hoc59] Alexis Hocquenghem. Codes Correcteurs d'Erreurs. *Chiffres*, 2(2):147–56, 1959.

[Hog17] Leslie Hogben. *Handbook of Linear Algebra (Discrete Mathematics and Its Applications)*. Chapman and Hall/CRC, 2017.

[HÖSB16] Matthias Hiller, Aysun Gurur Önalan, Georg Sigl, and Martin Bossert. Online Reliability Testing for PUF Key Derivation. In *Proceedings of the 6th International Workshop on Trustworthy Embedded Devices*, pages 15–22. ACM, 2016.

[HPS15] Matthias Hiller, Michael Pehl, and Georg Sigl. Fehlerkorrekturverfahren zur sicheren Schlüsselerzeugung mit Physical Unclonable Functions. *Datenschutz und Datensicherheit-DuD*, 39(4):229–233, 2015.

[HSW+15] Maryam S. Hashemian, Bhanu Singh, Francis Wolff, Daniel Weyer, Steve Clay, and Christos Papachristou. A Robust Authentication Methodology using Physically Unclonable Functions in DRAM Arrays. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, pages 647–652. IEEE, 2015.

[HWRL+13] Matthias Hiller, Michael Weiner, Leandro Rodrigues Lima, Maximilian Birkner, and Georg Sigl. Breaking Through Fixed PUF Block Limitations with Differential Sequence Coding and Convolutional Codes. In *Int. Workshop on Trustworthy Embedded Devices – TrustED*, pages 43–54. ACM, 2013.

[HYKD14] Charles Herder, Meng-Day Yu, Farinaz Koushanfar, and Srinivas Devadas. Physical Unclonable Functions and Applications: A Tutorial. *Proceedings of the IEEE*, 102(8):1126–1141, 2014.

[Inc18] Intrinsic ID Inc. Intrinsic ID: Internet of Things Security. `https://www.intrinsic-id.com`, 2018. Accessed: 2019-04-11.

[Jel69] Frederick Jelinek. Fast Sequential Decoding Algorithm Using a Stack. *IBM Journal of Research and Development*, 13(6):675–685, 1969.

[JKM00] Hui Jin, Aamod Khandekar, and Robert McEliece. Irregular Repeat-Accumulate Codes. In *International Symposium of Turbo Codes and Related Topics*, pages 1–8. Citeseer, 2000.

[JMR+17] Matthias Jung, Deepak M. Mathew, Carl C. Rheinländer, Christian Weis, and Norbert Wehn. A Platform to Analyze DDR3 DRAM's Power and Retention Time. *IEEE Design & Test*, 34(4):52–59, 2017.

[Joh75] Rolf Johannesson. Robustly Optimal Rate One-half Binary Convolutional Codes. *IEEE Transactions on Information Theory*, 21(4):464–468, 1975.

[JW99] Ari Juels and Martin Wattenberg. A Fuzzy Commitment Scheme. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 28–36. ACM, 1999.

[JW13] Ari Juels and Martin Wattenberg. A Fuzzy Commitment Scheme, 2013. Accessed: 2019-04-11. URL: `http://www.arijuels.com/wp-content/uploads/2013/09/JW99.pdf`.

[JZ15] Rolf Johannesson and Kamil Sh Zigangirov. *Fundamentals of Convolutional Coding*, volume 15. John Wiley & Sons, 2015.

[KDK+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory without Accessing them: An Experimental Study of DRAM Disturbance Errors. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 361–372. IEEE Press, 2014.

[KGKF14] Christoph Keller, Frank Gurkaynak, Hubert Kaeslin, and Norbert Felber. Dynamic Memory-Based Physically Unclonable Function for the Generation of Unique Identifiers and True Random Numbers. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2740–2743. IEEE, 2014.

[KGM+08] Sandeep S. Kumar, Jorge Guajardo, Roel Maes, Geert-Jan Schrijen, and Pim Tuyls. The Butterfly PUF Protecting IP on every FPGA. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 67–70. IEEE, 2008.

[KHS12] Hyunho Kang, Yohei Hori, and Akashi Satoh. Performance Evaluation of the First Commercial PUF-embedded RFID. In *IEEE 1st Global Conference on Consumer Electronics (GCCE)*, pages 5–8. IEEE, 2012.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[KKS05] Grigorii Kabatiansky, Evgenii Krouk, and Sergei Semenov. *Error Correcting Coding and Security for Data Networks*. John Wiley & Sons, 2005.

[KLF00] Yu Kou, Shu Lin, and Marc P.C. Fossorier. Low Density Parity Check Codes: Construction Based on Finite Geometries. In *IEEE Global Telecommunications Conference (GLOBECOM'00)*, volume 2, pages 825–829. IEEE, 2000.

[Kløve81] Torleiv Kløve. *Error Correcting Codes for the Asymmetric Channel*. Department of Pure Mathematics, University of Bergen, 1981.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and other Systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[KPHM18] Jeremie S. Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. The DRAM Latency PUF. In *IEEE International Symposium on High Performance Computer Architeture*, pages 194–207. IEEE, 2018.

[KS10] Deniz Karakoyunlu and Berk Sunar. Differential Template Attacks on PUF Enabled Cryptographic Devices. In *Information Forensics and Security (WIFS), 2010 IEEE International Workshop on*, pages 1–6. IEEE, 2010.

[Kür14] Ludwig Kürzinger. Analysis and Efficient Implementation of GC RM Error Correction Codes for PUFs. Master's thesis, Technical University Munich, 2014.

[KV03] Ralf Koetter and Alexander Vardy. Algebraic Soft-Decision Decoding of Reed-Solomon Codes. *IEEE Transactions on Information Theory*, 49(11):2809–2825, 2003.

[Lar73] Knud Larsen. Short Convolutional Codes with Maximal Free Distance for Rates 1/2, 1/3, and 1/4. *IEEE Transactions on Information Theory*, 19(3):371–372, 1973.

[LB13] Daniel Lidar and Todd Brun. *Quantum Error Correction*. Cambridge University Press, 2013.

[LC04] Shu Lin and Daniel J. Costello. *Error Control Coding*. Pearson Education India, 2004.

[LHA+12] Arjen Lenstra, James P. Hughes, Maxime Augier, Joppe Willem Bos, Thorsten Kleinjung, and Christophe Wachter. Ron was Wrong, Whit is Right. Technical report, IACR, 2012.

[Lim04] Daihyun Lim. Extracting Secret Keys from Integrated Circuits. Master's thesis, Massachusetts Institute of Technology, 2004.

[LJK+13] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 60–71. ACM, 2013.

[LKA15] Ting Lu, Ryan Kenny, and Sean Atsatt. White Paper WP-01252-1.0: Stratix 10 Secure Device Manager Provides Best-in-Class FPGA and SoC Security. *Altera Corporation, San Jose, CA*, 2015.

[LLG+04] Jae W. Lee, Daihyun Lim, Blaise Gassend, G. Edward Suh, Marten Van Dijk, and Srinivas Devadas. A Technique to Build a Secret Key in Integrated Circuits for Identification and Authentication Applications. In *Symposium on VLSI Circuits, 2004. Digest of Technical Papers*, pages 176–179. IEEE, 2004.

[LT03] Jean-Paul Linnartz and Pim Tuyls. New Shielding Functions to Enhance Privacy and Prevent Misuse of Biometric Templates. In *International Conference on Audio-and Video-Based Biometric Person Authentication*, pages 393–402. Springer, 2003.

[LW08] Dejan E. Lazich and Micaela Wuensche. Protection of Sensitive Security Parameters in Integrated Circuits. In *Mathematical Methods in Computer Science*, pages 157–178. Springer, 2008.

[LWK15] Dejan E. Lazich, Micaela Wuensche, and Sebastian Kaluza. Circuit and Method for Generating a True, Circuit-Specific and Time-Invariant Random Number. Patent, US 8,990,276 B2, 03 2015.

[LZLL14] Wenchao Liu, Zhenhua Zhang, Miaoxin Li, and Zhenglin Liu. A Trustworthy Key Generation Prototype Based on DDR3 PUF for Wireless Sensor Networks. *Sensors*, 14(7):11542–11556, 2014.

[Mae13] Roel Maes. *Physically Unclonable Functions: Constructions, Properties and Applications*. Springer Science & Business Media, 2013.

[Man18] Holger Mandry. Entwurf und Implementierung einer vollständig modularisierten PUF-Codierungskette auf einem SoC . Master's thesis, Institute of Microelectronics, Ulm University, 2018.

[Mas92] James L. Massey. Deep-Space Communications and Coding: A Marriage Made in Heaven. In *Advanced Methods for Satellite and Deep Space Communications*, pages 1–17. Springer, 1992.

[MCMS10] Abhranil Maiti, Jeff Casarona, Luke McHale, and Patrick Schaumont. A Large Scale Characterization of RO-PUF. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 94–99. IEEE, 2010.

[Mey00] Carl Meyer. *Matrix Analysis and Applied Linear Algebra*. Society for Industrial and Applied Mathematics, 2000.

[MKS12] Abhranil Maiti, Inyoung Kim, and Patrick Schaumont. A Robust Physical Unclonable Function with Enhanced Challenge-Response Set. *IEEE Transactions on Information Forensics and Security*, 7(1):333–345, 2012.

[MMS11] Abhranil Maiti, Logan McDougall, and Patrick Schaumont. The Impact of Aging on an FPGA-based Physical Unclonable Function. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 151–156. IEEE, 2011.

[MN97] David J.C. MacKay and Radford M. Neal. Near Shannon Limit Performance of Low Density Parity Check Codes. *Electronics letters*, 33(6):457–458, 1997.

[MOP08] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, volume 31. Springer Science & Business Media, 2008.

[MS09] Abhranil Maiti and Patrick Schaumont. Improving the Quality of a Physical Unclonable Function Using Configurable Ring Oscillators. In *International Conference on Field Programmable Logic and Applications*, pages 703–707. IEEE, 2009.

[MS11] Abhranil Maiti and Patrick Schaumont. Improved Ring Oscillator PUF: an FPGA-friendly Secure Primitive. *Journal of Cryptology*, 24(2):375–397, 2011.

[MSS13] Dominik Merli, Frederic Stumpf, and Georg Sigl. Protecting PUF Error Correction by Codeword Masking. *IACR Cryptology ePrint Archive*, 2013:334, 2013.

[MSSS11] Dominik Merli, Dieter Schuster, Frederic Stumpf, and Georg Sigl. Side-channel Analysis of PUFs and Fuzzy Extractors. In *International Conference on Trust and Trustworthy Computing*, pages 33–47. Springer, 2011.

[MTV08] Roel Maes, Pim Tuyls, and Ingrid Verbauwhede. Intrinsic PUFs from Flip-flops on Reconfigurable Devices. In *3rd Benelux Workshop on Information and System Security (WISSec 2008)*, volume 17, 2008.

[MTV09a] Roel Maes, Pim Tuyls, and Ingrid Verbauwhede. A Soft Decision Helper Data Algorithm for SRAM PUFs. In *IEEE International Symposium on Information Theory (ISIT 2009)*, pages 2101–2105. IEEE, 2009.

[MTV09b] Roel Maes, Pim Tuyls, and Ingrid Verbauwhede. Low-overhead Implementation of a Soft Decision Helper Data Algorithm for SRAM PUFs. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 332–347. Springer, 2009.

[Mul54] David E. Muller. Application of Boolean Algebra to Switching Circuit Design and to Error Detection. *Transactions of the IRE Professional Group on Electronic Computers*, EC-3(3):6–12, 1954.

[MvdLvdSW15] Roel Maes, Vincent van der Leest, Erik van der Sluis, and Frans Willems. Secure Key Generation from Biased PUFs. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 517–534. Springer, 2015.

[MVHV12] Roel Maes, Anthony Van Herrewege, and Ingrid Verbauwhede. PUFKY: A Fully Functional PUF-based Cryptographic Key Generator. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 302–319. Springer, 2012.

[Nie15] Johan S. R. Nielsen. Power Decoding Reed–Solomon Codes up to the Johnson Radius. *arXiv preprint arXiv:1505.02111*, 2015.

[Pap01] Ravikanth Pappu. *Physical One-Way Functions*. PhD thesis, Massachusetts Institute of Technology, 2001.

[Pea14] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Elsevier, 2014.

[Pet60] W. Wesley Peterson. Encoding and Error-Correction Procedures for the Bose-Chaudhuri Codes. *IRE Transactions on Information Theory*, 6(4):459–470, 1960.

[Pet18] Ed Peterson. Developing Tamper-Resistant Designs with Zynq UltraScale+ Devices, 2018. Accessed: 2019-04-11. URL: `https://www.xilinx.com/support/documentation/application_notes/xapp1323-zynq-usp-tamper-resistant-designs.pdf`.

[Pis84] Sergio Pissanetzky. *Sparse Matrix Technology*. Academic Press, 1984.

[Plo60] Morris Plotkin. Binary Codes with Specified Minimum Distance. *IRE Transactions on Information Theory*, 6(4):445–450, 1960.

[PRTG02] Ravikanth Pappu, Ben Recht, Jason Taylor, and Neil Gershenfeld. Physical One-Way Functions. *Science*, 297(5589):2026–2030, 2002.

[PT15] Danuta Pamula and Arnaud Tisserand. Fast and Secure Finite Field Multipliers. In *Euromicro Conference on Digital System Design (DSD)*, pages 653–660. IEEE, 2015.

[RB98] Arvind R. Raghavan and Carl W. Baum. A Reliability Output Viterbi Algorithm with Applications to Hybrid ARQ. *IEEE Transactions on Information Theory*, 44(3):1214–1216, 1998.

[RDK12] Ulrich Rührmair, Srinivas Devadas, and Farinaz Koushanfar. Security Based on Physical Unclonability and Disorder. In *Introduction to Hardware Security and Trust*, pages 65–102. Springer, 2012.

[Ree54] Irving S. Reed. A Class of Multiple-Error-Correcting Codes and the Decoding Scheme. *IEEE Transactions on Information Theory*, 1954.

[RFC+13] Sami Rosenblatt, Daniel Fainstein, Alberto Cestero, John Safran, Norman Robson, Toshiaki Kirihata, and Subramanian S. Iyer. Field Tolerant Dynamic Intrinsic Chip ID Using 32 nm High-K/Metal Gate SOI Embedded DRAM. *J. Solid-State Circuits*, 48(4):940–947, 2013.

[RGB+16] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security symposium*, pages 1–18, 2016.

[RHHF16] Amir Rahmati, Matthew Hicks, Daniel E. Holcomb, and Kevin Fu. Probable Cause: The Deanonymizing Effects of Approximate DRAM. *ACM SIGARCH Computer Architecture News*, 43(3):604–615, 2016.

[Rot06] Ron Roth. *Introduction to Coding Theory*. Cambridge University Press, 2006.

[RR00] Ron M. Roth and Gitit Ruckenstein. Efficient Decoding of Reed–Solomon Codes beyond Half the Minimum Distance. *IEEE Transactions on Information Theory*, 46(1):246–257, 2000.

[RS60] Irving S. Reed and Gustave Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[SB94] Gottfried Schnabel and Martin Bossert. Reed Muller Codes as Generalized Multiple Concatenated Codes with Soft-Decision Decoding. *Internal Report, Informationstechnik, University of Ulm, Germany*, 1994.

[Sch02] Walter Schnug. *On Generalized Woven Codes*. PhD thesis, Ulm University, 2002.

[Sch17] Patrick Schaumont. Personal Correspondance, 2017.

[SD07] G. Edward Suh and Srinivas Devadas. Physical Unclonable Functions for Device Authentication and Secret Key Generation. In *Proceedings of the 44th Annual Design Automation Conference*, pages 9–14. ACM, 2007.

[SD15] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. *Black Hat*, 15, 2015.

[Sen11] Christian Senger. *Generalized Minimum Distance Decoding with Arbitrary Error, Erasure Tradeoff*. Der Andere Verlag, 2011.

[SHO07] Ying Su, Jeremy Holleman, and Brian Otis. A 1.6 pJ/bit 96% Stable Chip-ID Generating Circuit using Process Variations. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 406–611. IEEE, 2007.

[Sin13] Kuldeep Singh. *Linear Algebra: Step by Step*. Oxford University Press, 2013.

[SN10] Ahmad-Reza Sadeghi and David Naccache. *Towards Hardware-Intrinsic Security*. Springer, 2010.

[SRK+17] Soubhagya Sutar, Arnab Raha, Devadatta Kulkarni, Rajeev Shorey, Jeffrey Tew, and Vijay Raghunathan. D-PUF: An Intrinsically Reconfigurable DRAM PUF for Device Authentication and Random Number Generation. *ACM Transactions on Embedded Computing Systems*, 17:1–31, 12 2017.

[SRR16] Soubhagya Sutar, Arnab Raha, and Vijay Raghunathan. D-PUF: An Intrinsically Reconfigurable DRAM PUF for Device Authentication in Embedded Systems. In *Compliers, Architectures, and Sythesis of Embedded Systems (CASES), 2016 International Conference on*, pages 1–10. IEEE, 2016.

[SRR17] Soubhagya Sutar, Arnab Raha, and Vijay Raghunathan. Memory-based Combination PUFs for Device Authentication in Embedded Systems. *arXiv preprint:1712.01611*, 2017.

[SS06] Georg Schmidt and Vladimir Sidorenko. Multi-Sequence Linear Shift-Register Synthesis: The Varying Length Case. In *IEEE International Symposium on Information Theory*, pages 1738–1742. IEEE, 2006.

[SSB06a] Georg Schmidt, Vladimir Sidorenko, and Martin Bossert. Decoding Reed-Solomon Codes Beyond Half the Minimum Distance using Shift-Register Synthesis. In *IEEE International Symposium on Information Theory*, pages 459–463. IEEE, 2006.

[SSB06b] Georg Schmidt, Vladimir Sidorenko, and Martin Bossert. Error and Erasure Correction of Interleaved Reed–Solomon Codes. In *Coding and Cryptography*, pages 22–35. Springer, 2006.

[SSO+07] Boris Skoric, Geert-Jan Schrijen, Wil Ophey, Rob Wolters, Nynke Verhaegh, and Jan van Geloven. Experimental Hardware for Coating PUFs and Optical PUFs. In *Security with Noisy Data*, pages 255–268. Springer, 2007.

[SSZB04] Georg Schmidt, Vladimir Sidorenko, Victor V. Zyablov, and Martin Bossert. Finding a List of Best Paths in a Trellis. In *IEEE International Symposium on Information Theory*, page 557. IEEE, 2004.

[Ste19] Sebastian Stern. *Advanced Equalization and Coded-Modulation Strategies for Multiple-Input/Multiple-Output Systems*. PhD thesis, Ulm University, 2019.

[Sud97] Madhu Sudan. Decoding of Reed Solomon Codes Beyond the Error-Correction Bound. *Journal of Complexity*, 13(1):180–193, 1997.

[Sud18] Chirag Sudarshan. Personal Correspondance, 2018. Department of Electrical and Computer Engineering, Microelectronic Systems Design Research Group, Technical University Kaiserslautern, Germany.

[Suh05] G. Edward Suh. *AEGIS: A Single-Chip Secure Processor*. PhD thesis, Massachusetts Institute of Technology, 2005.

[SXA+17] André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. Intrinsic Rowhammer PUFs: Leveraging the Rowhammer Effect for Improved Security. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–7. IEEE, 2017.

[SXA+18] Andre Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Boris Skoric, Stefan Katzenbeisser, and Jakub Szefer. Decay-Based DRAM PUFs in Commodity Devices. *IEEE Transactions on Dependable and Secure Computing*, 2018.

[Tar10] Christopher Tarnovsky. Deconstructing a 'Secure' Processor. *Black Hat DC*, 2010.

[TB06] Pim Tuyls and Lejla Batina. RFID-Tags for Anti-Counterfeiting. In *Cryptographers' Track at the RSA Conference*, pages 115–131. Springer, 2006.

[TJ09] Randy Torrance and Dick James. The State-of-the-art in IC Reverse Engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 363–381. Springer, 2009.

[TKXC15] Fatemeh Tehranipoor, Nima Karimian, Kan Xiao, and John Chandy. DRAM Based Intrinsic Physical Unclonable Functions for System Level Security. In *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, pages 15–20. ACM, 2015.

[TKYC17a] Fatemeh Tehranipoor, Nima Karimian, Wei Yan, and John A. Chandy. DRAM-Based Intrinsic Physically Unclonable Functions for System-Level Security and Authentication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(3):1085–1097, 2017.

[TKYC17b] Fatemeh Tehranipoor, Nima Karimian, Wei Yan, and John A Chandy. Investigation of DRAM PUFs Reliability under Device Accelerated Aging Effects. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE, 2017.

[TRT+18] B. M. S. Bahar Talukder, Biswajit Ray, Mark Tehranipoor, Domenic Forte, and Md Tauhidur Rahman. LDPUF: Exploiting DRAM Latency Variations to Generate Robust Device Signatures. *arXiv preprint arXiv:1808.02584*, 2018.

[TSK07] Pim Tuyls, Boris Skoric, and Tom Kevenaar. *Security With Noisy Data*. Springer, 2007.

[TZC+17] Qianying Tang, Chen Zhou, Woong Choi, Gyuseong Kang, Jongsun Park, Keshab K Parhi, and Chris H Kim. A DRAM based Physical Unclonable Function Capable of Generating $> 10^{32}$ Challenge Response Pairs per 1Kbit Array for Secure Chip Authentication. In *IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4. IEEE, 2017.

[VdLPVdS12] Vincent Van der Leest, Bart Preneel, and Erik Van der Sluis. Soft Decision Error Correction for Compact Memory-based PUFs using a Single Enrollment. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 268–282. Springer, 2012.

[VdLSHT10] Vincent Van der Leest, Geert-Jan Schrijen, Helena Handschuh, and Pim Tuyls. Hardware Intrinsic Security from D flip-flops. In *Proceedings of the fifth ACM Workshop on Scalable Trusted Computing*, pages 53–62. ACM, 2010.

[VDVFL+16] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1675–1689. ACM, 2016.

[Ver10] Ingrid Verbauwhede. *Secure Integrated Circuits and Systems*. Springer, 2010.

[Ver18] Verayo. Verayo Physical Unclonable Function. `http://verayo.com/solutions.php`, 2018. Accessed: 2018-07-27.

[Vit67] Andrew Viterbi. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967.

[VN51] John Von Neumann. Various Techniques used in Connection with Random Digits. *Applied Math Series*, 12(36-38):1, 1951.

[WB86] Lloyd R. Welch and Elwyn R. Berlekamp. Error Correction for Algebraic Block Codes, 1986. US Patent 4,633,470.

[Wil17] Florian Wilde. Large Scale Characterization of SRAM on Infineon XMC Microcontrollers as PUF. In *Proceedings of the Fourth Workshop on Cryptography and Security in Computing Systems*, pages 13–18. ACM, 2017.

[WMG18] Alexander Wild, Amir Moradi, and Tim Güneysu. GliFreD: Glitch-Free Duplication Towards Power-Equalized Circuits on FPGAs. *IEEE Transactions on Computers*, 67(3):375–387, 2018.

[Woz57] John M. Wozencraft. Sequential Decoding for Reliable Communication. *Technical Report 325, Research Laboratory of Electronics, Massachusetts Institute of Technology*, 1957.

[WS14] Christian Wachsmann and Ahmad-Reza Sadeghi. Physically Unclonable Functions (PUFs): Applications, Models, and Future Directions. *Synthesis Lectures on Information Security, Privacy, & Trust*, 5(3):1–91, 2014.

[Wu08] Yingquan Wu. New List Decoding Algorithms for Reed–Solomon and BCH Codes. *IEEE Transactions on Information Theory*, 54(8):3611–3630, 2008.

[Wue08] Micaela Wuensche. Eingebettete Schaltungsspezifische Physikalische Zufallszahlengeneratoren. Master's thesis, Universität Karlsruhe, 2008.

[Xil19] Xilinx. Xilinx Device Reliability Report First Half 2018, 2019. Accessed: 2019-04-11. URL: `https://www.xilinx.com/support/documentation/user_guides/ug116.pdf`.

[XSA+16] Wenjie Xiong, André Schaller, Nikolaos A. Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. Run-time Accessible DRAM PUFs in Commodity Devices. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 432–453. Springer, 2016.

[XZZT16] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security Symposium*, pages 19–35, 2016.

[YD10] Meng-Day Yu and Srinivas Devadas. Secure and Robust Error Correction for Physical Unclonable Functions. *IEEE Design & Test of Computers*, 27(1):48–65, 2010.

[YQ10] Chi-En Daniel Yin and Gang Qu. LISA: Maximizing RO PUF's Secret Extraction. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 100–105. IEEE, 2010.

[ZGA11] Alexander Zeh, Christian Gentner, and Daniel Augot. An Interpolation Procedure for List Decoding Reed–Solomon Codes Based on Generalized Key Equations. *IEEE Transactions on Information Theory*, 57(9):5946–5959, 2011.

[ZGS18] Shaza Zeitouni, David Gens, and Ahmad-Reza Sadeghi. It's Hammer Time: How to Attack (Rowhammer-based) DRAM-PUFs. In *Proceedings of the 55th Annual Design Automation Conference*, page 65. ACM, 2018.

[Zig66] Kamil'Shamil'evich Zigangirov. Some Sequential Decoding Procedures. *Problemy Peredachi Informatsii*, 2(4):13–25, 1966.

[ZSB99] Victor Zyablov, Sergo Shavgulidze, and Martin Bossert. An Introduction to Generalized Concatenated Codes. *European Trans. on Telecommunications*, 10(6):609–622, 1999.

Publications Containing Parts of this Thesis

[HKS+15] Matthias Hiller, Ludwig Kürzinger, Georg Sigl, Sven Müelich, Sven Puchinger, and Martin Bossert. Low-Area Reed Decoding in a Generalized Concatenated Code Construction for PUFs. In *IEEE Computer Society Annual Symposium on VLSI*, 2015.

[MB17a] Sven Müelich and Martin Bossert. A New Error Correction Scheme for Physical Unclonable Functions. In *Proceedings of 11th International ITG Conference on Systems, Communications and Coding (SCC)*. VDE, 2017.

[MB17b] Sven Müelich and Martin Bossert. Applying Convolutional Codes to Key Extraction using Ring Oscillator PUFs. In *International Workshop on Optimal Codes and Related Topics*, 2017.

[MHK+19] Holger Mandry, Andreas Herkle, Ludwig Kürzinger, Joachim Becker, Sven Müelich, Robert Fischer, and Maurits Ortmanns. Modular PUF Coding Chain with High Speed Reed Muller Decoder. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019.

[MPB+14] Sven Müelich, Sven Puchinger, Martin Bossert, Matthias Hiller, and Georg Sigl. Error Correction for Physical Unclonable Functions Using Generalized Concatenated Codes. In *International Workshop on Algebraic and Combinatorical Coding Theory*, 2014.

[MPB18a] Sven Müelich, Sven Puchinger, and Martin Bossert. Constructing an LDPC Code Containing a Given Vector. In *International Workshop on Algebraic and Combinatorical Coding Theory*, 2018.

[MPB18b] Sven Müelich, Sven Puchinger, and Martin Bossert. Using Convolutional Codes for Key Extraction in SRAM Physical Unclonable Functions. *Trustworthy Manufacturing and Utilization of Secure Devices (TRUDEVICE)*, 2018.

[MPSB19] Sven Müelich, Sven Puchinger, Veniamin Stukalov, and Martin Bossert. A Channel Model and Soft-Decision Helper Data Algorithms for ROPUFs. In *Proceedings of 12th International ITG Conference on Systems, Communications and Coding (SCC)*. VDE, 2019.

[PMB+15] Sven Puchinger, Sven Müelich, Martin Bossert, Matthias Hiller, and Georg Sigl. On Error Correction for Physical Unclonable Functions. In *Proceedings of 10th International ITG Conference on Systems, Communications and Coding (SCC)*, 2015.

[PMWZB17] Sven Puchinger, Sven Müelich, Antonia Wachter-Zeh, and Martin Bossert. Timing Attack Resilient Decoding Algorithms for Physical Unclonable Functions. In *Proceedings of 11th International ITG Conference on Systems, Communications and Coding (SCC)*, 2017.

Preprints Containing Parts of this Thesis

[MBS+19] Sven Müelich, Sebastian Bitzer, Chirag Sudarshan, Christian Weis, Norbert Wehn, Martin Bossert, and Robert Fischer. Channel Models for Physical Unclonable Functions based on DRAM Retention Measurements. *Accepted at XVI International Symposium Problems of Redundancy in Information and Control Systems*, 2019.

Other Publications and Preprints by the Author of this Thesis

[GMD13] Oliver Gableske, Sven Müelich, and Daniel Diepold. On the Performance of CDCL-based Message Passing Inspired Decimation using $\rho\sigma$PMPi. In *Pragmatics of SAT Workshop*, 2013.

[MPB17] Sven Müelich, Sven Puchinger, and Martin Bossert. Low-Rank Matrix Recovery using Gabidulin Codes in Characteristic Zero. *Electronic Notes in Discrete Mathematics*, 57:161–166, 2017.

[MPMB16] Sven Müelich, Sven Puchinger, David Mödinger, and Martin Bossert. An Alternative Decoding Method for Gabidulin Codes in Characteristic Zero. In *IEEE International Symposium on Information Theory (ISIT)*, pages 2549–2553. IEEE, 2016.

[PMB17] Sven Puchinger, Sven Müelich, and Martin Bossert. On the Success Probability of Decoding (Partial) Unit Memory Codes. In *International Workshop on Optimal Codes and Related Topics*, 2017.

[PMIB17] Sven Puchinger, Sven Müelich, Karim Ishak, and Martin Bossert. Code-Based Cryptosystems Using Generalized Concatenated Codes. In Ilias S. Kotsireas and Edgar Martínez-Moro, editors, *Springer Proceedings in Mathematics & Statistics: Applications of Computer Algebra: Kalamata, Greece, July 20–23 2015*, volume 198, pages 397–423. Springer International Publishing, 2017.

[PMM+17] Sven Puchinger, Sven Müelich, David Mödinger, Johan Rosenkilde né Nielsen, and Martin Bossert. Decoding Interleaved Gabidulin Codes Using Alekhnovich's Algorithm. *Electronic Notes in Discrete Mathematics*, 57:175–180, 2017.

List of Supervised Theses

[Ahm19] Musab Ahmed Eltayeb Ahmed. Implementation of Gabidulin Codes in Sage (Master's Thesis, jointly supervised with Cornelia Ott), 2019.

[Bal17] Jhoiss Balois. Helper Data Methods for Error Correction in Physical Unclonable Functions (Bachelor's Thesis), 2017.

[Bit18] Sebastian Bitzer. Physical Unclonable Functions based on DRAM (Bachelor's Thesis, jointly supervised with Sven Puchinger and Chirag Sudarshan), 2018.

[Fan17] Liming Fan. Using Sequential Decoding for Key Regeneration in Physical Unclonable Functions (Master's Thesis, jointly supervised with Sven Puchinger), 2017.

[Ish15] Karim Ishak. Analysis of Cryptographic Methods based on Coding Theory (Master's Thesis, jointly supervised with Sven Puchinger), 2015.

[Mar16] Yonatan Marin. Partial Unit Memory Codes based on Reed-Solomon Codes for Streaming (Bachelor's Thesis, jointly supervised with Sven Puchinger), 2016.

[Mar19] Yonatan Marin. Variants of Message-Passing Decoding for Low-Density Parity-Check Codes (Master's Thesis, jointly supervised with George Yammine), 2019.

[Mö15] David Mödinger. Decoding of Gabidulin Codes Using Module Minimization (Master's Thesis, jointly supervised with Sven Puchinger), 2015.

[Raj18] Sushmita Raj. Variants of Message-Passing Decoding for LDPC Codes in Noncoherent Massive MIMO (Master's Thesis, jointly supervised with George Yammine), 2018.

[Sch17] Rebekka Schulz. Code-Based Cryptology Using Moderate Density Parity Check Codes (Bachelor's Thesis), 2017.

[Stu17] Veniamin Stukalov. Error Models in Physical Unclonable Functions (Master's Thesis, jointly supervised with Sven Puchinger), 2017.

[Tsa17] Alexander Tsaregorodtsev. Designing Concatenated BCH Codes for Application in Physical Unclonable Functions (Bachelor's Thesis, jointly supervised with Michael Schelling), 2017.

Curriculum Vitae

For data protection reasons, the curriculum vitæ has been removed from the online version.

ISBN: 978-3-948303-10-5