ALPEN-ADRIA
UNIVERSITÄT
KLAGENFURT

Author
MARTIN DEUTSCHMANN

Cryptographic Applications with Physically Unclonable Functions

Master Thesis

Awarding the Academic Title
Diplom-Ingenieur

Studies
Technical Mathematics and Data Analysis

Alpen-Adria-Universität Klagenfurt
Faculty of Technical Science

Supervisor
Univ.-Prof. Dr. Winfried Müller

Institute of Mathematics

Klagenfurt, 18th November 2010

”This research was conducted within the UNIQUE ”Foundations for Forgery-Resistant Security Hardware” project (ICT-238811) supported by the Seventh Framework Programme of the European Community.”

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende wissenschaftliche Arbeit selbstständig angefertigt und die mit ihr unmittelbar verbundenen Tätigkeiten selbst erbracht habe. Ich erkläre weiters, dass ich keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle aus gedruckten, ungedruckten oder dem Internet im Wortlaut oder im wesentlichen Inhalt übernommenen Formulierungen und Konzepte sind gemäß den Regeln für wissenschaftliche Arbeiten zitiert und durch Fußnoten bzw. durch andere genaue Quellenangaben gekennzeichnet.

Die während des Arbeitsvorganges gewährte Unterstützung einschließlich signifikanter Betreuungshinweise ist vollständig angegeben.

Die wissenschaftliche Arbeit ist noch keiner anderen Prüfungsbehörde vorgelegt worden. Diese Arbeit wurde in gedruckter und elektronischer Form abgegeben. Ich bestätige, dass der Inhalt der digitalen Version vollständig mit dem der gedruckten Version übereinstimmt.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

MARTIN DEUTSCHMANN Klagenfurt, 18th November 2010

Abstract

Every year the IT industry looses billions of dollars due to counterfeited goods and stolen intellectual property. Although this threat today already concerns a wide range of items, such as avionic and automobile parts, software tools and even pharmaceuticals, the technology to prevent such attacks is still in its infancy.

This thesis aims at describing the new technology of Physically Unclonable Functions (PUFs), which are special challenge-response entities that are embedded in a physical device. The unclonability of PUFs is derived by the fact that they contain random components caused by manufacturing variations. The idea is to extract secrets from the unique physical structure of the devices, which can be then used for simple authentication protocols or the generation of secret keys.

One of the big advantages in comparison to existing cryptographic systems is the fact, that at no point secret keys exist in a digital form. The device itself is the key, which makes PUFs secure against any kind of invasive attack. Another benefit is that PUF-based cryptographic primitives promise to be cost efficient, which might make them suitable also for large-scale production.

Acknowledgments

This thesis was written during my employment at TECHNIKON Forschungs- und Planungsgesellschaft in Villach. TECHNIKON is a private research company which was founded in 1999 by Françoise Jouffre and Klaus-Michael Koch. The company's expertise in the area of planning and coordinating research projects is meanwhile recognized all over Europe. TECHNIKON is currently coordinating 9 projects in the area of Information Communication Technologies (ICT) which are funded by the European Commission (EC). Within these projects the company is also actively participating in research activities covering various areas like Cloud Computing, Smart Home Applications or Mobile Network Systems.

One of the most promising projects is called UNIQUE – Foundations for Forgery Resistant Hardware. Key goals are the protection against counterfeiting, reverse engineering and cloning of IT-products. The idea is based on the use of Physically Unclonable Functions, which is the main subject of my thesis.

Working on the UNIQUE project gave me the chance to investigate different mathematical regularities in the context of PUF Technology and it was always a great honor for me to be part of a group of real PUF-experts. I want to take the opportunity to thank the whole UNIQUE consortium for their collaboration and support. Special thanks go to Roel Maes from KU Leuven, Vincent van der Leest from Intrinsic ID and Professor Frederik Armknecht from University Mannheim, who where always open for technical discussions and provided me with a lot of useful information. Not to forget the Scientific Leader Professor Ahmad-Reza Sadeghi from Ruhr University Bochum and the Technical Leader Pim Tuyls, who is the CTO of Intrinsic ID and visiting professor at KU Leuven.

I also greatly appreciate the help of all the colleagues at TECHNIKON, who took their time to reflect on special technical aspects, and asked lots of critical questions. Especially I want to thank Dr. Klaus-Michael Koch, the scientific director of TECHNIKON. He gave me the opportunity to participate in many interesting meetings, provided me with several books in the area of PUFs and always encouraged me to go on with my research.

Of course I want to thank my supervisor Professor Winfried Müller from University Klagenfurt, who greatly supported me during my whole studies and who was always open for new topics like the PUF technology. Last but not least, I want to thank my family and my girlfriend Denise, who always supported me during my work on this thesis and who patiently listened to my technical monologues, although I guess they sometimes hardly understood a single word.

Contents

1 Introduction **1**

2 Introduction to Cryptography **3**

2.1 Basic Ideas 3

2.2 Cryptographic Applications 4

2.3 Symmetric Key Algorithms 5

2.3.1 Cryptographic Hash Functions 6

2.4 Public Key Algorithms 8

2.4.1 One-Way Functions 9

2.5 Hybrid Crypto-Systems 10

2.6 Digital Signatures 11

2.7 Certificates 12

2.8 Challenge Response Authentication 12

2.9 Concluding Remarks 13

2.10 Coding Theory 14

2.10.1 Introduction 14

2.10.2 Error Correcting Codes 16

2.10.3 Linear Codes 17

2.10.4 Hamming Codes 20

2.10.5 Cyclic Codes 21

2.10.6 BCH Codes 23

2.11 Concluding Remarks 26

3 Physically Unclonable Functions **27**

3.1 Biometrics 27

3.1.1 Overview of commonly used Biometrics 27
3.1.2 Biometric System Performance 29
3.2 PUF Properties 31
3.2.1 Definitions 31
3.2.2 Performance 33
3.3 Controlled PUFs 33
3.4 Reconfigurable PUFs 34
3.4.1 Classification of Reconfigurability 35
3.4.2 Implementations 35
3.4.3 Applications 35

4 PUF Instantiations 37

4.1 Non-Electrical Implementations 37
4.1.1 Optical PUFs 37
4.1.2 Acoustical PUFs 38
4.1.3 Coating PUFs 39
4.2 Delay-based Intrinsic PUFs 40
4.2.1 Arbiter PUFs 40
4.2.2 Ring Oscillator PUFs 41
4.3 Memory Based Intrinsic PUFs 42
4.3.1 SRAM PUFs 43
4.3.2 Butterfly PUFs 43
4.3.3 Latch PUFs 44
4.3.4 Flip-Flop PUFs 45
4.4 Special PUF Implementations 45
4.4.1 Reconfigurable Optical PUF 45

5 PUF Applications 47

5.1 Simple Key-card 47
5.2 Secret Key Generation 48
5.2.1 Idea 49
5.2.2 Majority Voting 50

5.3 Key Zeroization . . . 51

5.4 Concluding Remarks . . . 52

6 MATLAB Simulations 53

6.1 Introduction . . . 53

6.2 Application 1 - Secure Key-Card . . . 53

6.2.1 Basic Description . . . 53

6.2.2 Program Structure . . . 54

6.2.3 Simulation Results . . . 54

6.3 Application 2 - Secret Key Generator . . . 57

6.3.1 Basic Description . . . 57

6.3.2 Program Structure . . . 58

6.3.3 Simulation Results . . . 58

6.4 Concluding Remarks . . . 61

7 Conclusion 63

A MATLAB Code 65

A.1 Secure Key-Card . . . 65

A.1.1 Enrollment Phase Secure Key-Card . . . 65

A.1.2 Re-Production Phase Secure Key-Card . . . 67

A.1.3 Subroutines Secure Key-Card . . . 68

A.2 Key Generator . . . 74

A.2.1 Enrollment Phase for Key Generator . . . 74

A.2.2 Re-Production Phase of Key Generator . . . 77

A.2.3 Subroutines for Key Generator . . . 79

List of Figures 87

Abbreviations 89

Bibliography 91

1 Introduction

Counterfeiting, cloning, reverse engineering and the insertion of malicious components are among the most serious challenges facing the information technology industry today [9]. But not only the IT industry suffers from brand damage and the accompanied financial losses, a wide range of items such as pharmaceuticals, avionic and automotive spare parts as well as software tools are affected. It has been estimated, that more than 10 percent of all high tech products sold globally are counterfeits. This corresponds to about US$100 billion that the global IT industry loses to counterfeiters every year [9].
Another problem is the protection of Intellectual Property (IP). When enterprises invent new technologies or components, the design specifications represent their IP. In case these companies do not own production facilities, it is customary to outsource the production to decrease the costs. Although such contracts seem economically attractive, they introduce new security risks. Who guarantees that the terms of the contracts will be fulfilled and that the production operators in the foreign country will not play fast and loose with the intellectual property? A lot of western companies are very concerned about their IP but the technology to protect it is still in the early stages of development.
The current best practice to prevent malicious attacks is to use cryptographic primitives like encryption, digital signatures and authentication codes, which rely on the protection of secret keys. These keys are stored in non-volatile memory such as EEPROMS or fuses, which suffer from a couple of shortcomings. First, safely managing secrets in memory is difficult and expensive. Moreover, storing secrets in non-volatile memory technologies is often subject to invasive attacks, where the devices are opened and sensitive data is read out directly. To counter these attacks, expensive protective coatings are used but still, the devices are vulnerable to sophisticated physical attacks [20].

These problems raise the question of new approaches. An appropriate measure against physical attacks is the Physically Unclonable Function (PUF) technology, which will be described in this thesis in detail. PUFs are challenge-response entities, that are embedded in a physical system, possibly an IC [10]. It can be shown that mass-factored ICs, which offer the same functionality have unique physical characteristics due to manufacturing variations. The idea is to extract a secret key from the random physical structure of the IC, rather than using digitally stored information. A major advantage of PUFs is that ICs implicitly contain random elements which are introduced during the manufacturing process, i.e. no additional IC treatment is necessary.

This makes PUFs feasible for low cost applications like authentication protocols or secret key generation as will be described in Chapter 5.

This thesis is structured as follows. The subsequent Chapter 2 gives an overview of the basic mathematical background including a section about error correcting codes. In Chapter 3 the basic idea of PUFs is described including a section about Biometrics (cf. Section 3.1). Chapter 4 gives an overview of the most common PUF instantiations and Chapter 5 describes a range of PUF-based applications in more detail. To support the presented theory, some PUF application scenarios are simulated with MATLAB in Chapter 6. Finally this thesis closes with some concluding remarks in Chapter 7.

At this point the author would like to substantiate that he is not an expert in the area of semiconductors or electronics. The author does also not have the equipment and knowledge to make proper research in the area of chip technology. Most of the results that are presented in this thesis and which form the foundation for an mathematical investigation, are based on experiments carried out by **Intrinsic ID**, **KU Leuven**, **Ruhr University Bochum**, **Intel** or other partners within the UNIQUE consortium.

2 Introduction to Cryptography

In this chapter the basic fundamental properties of symmetric and asymmetric cryptographic algorithms are explained. Moreover, a brief overview of other cryptographic concepts like hash functions, digital signatures and challenge-response systems is given. At the end of this section, the idea of error correcting codes like the BCH code will be introduced.

To be able to reasonably discuss all the concepts of cryptography, an introduction to the basic mathematical fields *Algebra*, *Number Theory* or the ideas of *Finite Fields* would have to be given, but this would go far beyond the scope of this thesis. For deeper studies, the author recommends the following literature [11, 18, 17, 12] and especially [21], where a beautiful summary of all the major concepts of cryptography and coding theory may be found.

Apart from the above mentioned references, a solid basis of cryptographic mechanisms is given in the lecturer notes of the course **Systemsicherheit** held by Professor Horster and Professor Schartner, which the author attended in the summer term of 2008.

Moreover, the course **Codierungstheorie**, held by Professor Müller in the last term, was greatly beneficial for the progress of this thesis.

2.1 Basic Ideas

Cryptography is the study of methods for sending messages in a *secret*, namely in *enciphered* form, so that only the intended recipient can remove the disguise and read the message. The word Cryptography has its etymology – *kryptos* from the Greek meaning *hidden*, and *graphein*, meaning *to write* [12].

A basic communication scenario is illustrated in Figure 2.1. Two parties, we call them **Alice** and **Bob**, want to communicate with each other in a secure way. A third party, **Eve**, is a potential eavesdropper.

When Alice wants to send a message, called the **plaintext**, to Bob she encrypts it using a prearranged method. What keeps the message secret is the use of an **encryption key**. When Bob receives the encrypted message, called the **ciphertext**, he changes it back to the plaintext using a **decryption key**.

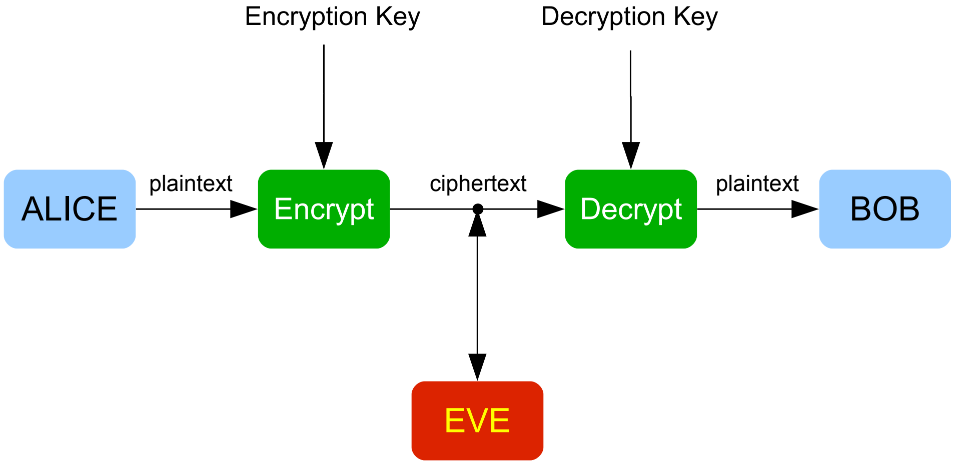


Figure 2.1: Basic Communication Scenario

2.2 Cryptographic Applications

Cryptography is not only about encrypting and decrypting messages, it is also about solving real-world problems that require information security. There are four main requirements that need to be satisfied [21]:

1. **Confidentiality:** Eve should not be able to read Alice's message to Bob. The main tools are encryption and decryption algorithms.
2. **Data Integrity:** Bob wants to be sure, that Alice's message has not been altered. For example, transmission errors might occur. Also an adversary might intercept the transmission and alter it before it reaches the intended recipient. Many cryptographic primitives, such as hash functions, provide methods to ensure data integrity despite malicious or accidental adversaries.
3. **Authentication:** Bob wants to be sure that only Alice could be able to send the message he received. Under this heading we also include identification schemes and password protocols. There are actually two types of authentication that arise in cryptography: **entity authentication** and **data-origin authentication**. Often the term *identification* is used to specify entity authentication, which is concerned with providing the identity of the parties involved in a communication. Data-origin authentication focuses on tying the information about the origin of the data, such as the creator and time of creation, with the data.
4. **Non-repudiation:** Alice cannot claim she did not send the message. Non-repudiation is particularly important in electronic commerce applications, where it is important that a consumer cannot deny the authorization of a purchase.

Possible solutions to obtain these four requirements, are presented in the following sections.

2.3 Symmetric Key Algorithms

In symmetric key algorithms the encryption and decryption keys are known to both, Alice and Bob. For example, the encryption key is shared and the decryption key can be easily calculated from it. In many cases, the encryption key and the decryption key are the same.
The following figure shows the principle of symmetric ciphers.

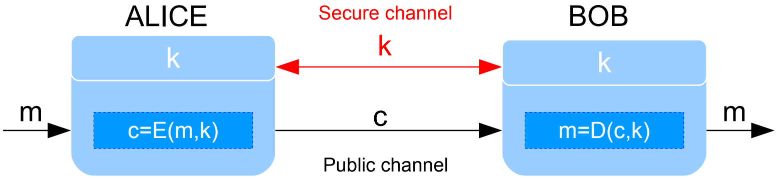


Figure 2.2: Principle of Symmetric Ciphers

Parameters

$m \in M$	...	Plaintext
$c \in C$	...	Ciphertext
$k \in K$	...	Key
E	...	Encryption Function
		$E : M \times K \rightarrow C,\ c = E(m,k)$
D	...	Decryption Function
		$D : C \times K \rightarrow M,\ m = D(c,k)$

Alice encrypts the secret message using the symmetric key k. The ciphertext is transmitted via a public channel to Bob, who is able to transform the cipher back to the plaintext with the help of the symmetric key k, which has to be interchanged in a secure way before the communication starts. The security of the system is based on the confidentiality of the used key. For all $m \in M,\ k \in K$ it has to hold that $D(E(m,k),k) = m$

Examples

- DES
- AES
- RC2, RC4, RC5, RC6
- VERNAM
- SAFER
- FEAL
- Skipjack

Depending on how the input bits are processed, we distinguish between *Block ciphers* and *Stream ciphers*. Like the word might imply, block cipher encryption algorithms take a fixed

length group of bits, *a block* of plaintext, as input, to produce a corresponding block of ciphertext as shown in Figure 2.3.
In contrast to taking a block of input bits, *Stream Ciphers* operate on individual digits at one time and the transformation of successive digits varies during the encryption.

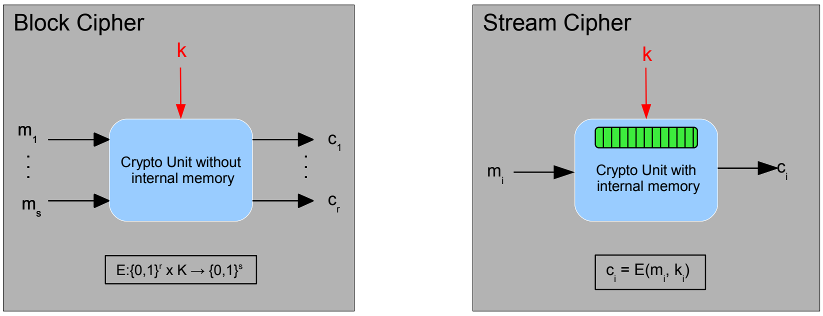

Block Cipher
k
m_1
m_s
Crypto Unit without internal memory
c_1
c_r
E:{0,1}^{r} x K → {0,1}^{s}
Stream Cipher
k
m_i
Crypto Unit with internal memory
c_i
c_i = E(m_i, k_i)

Figure 2.3: Basic Principle of Block -and Stream Ciphers

The best-known example of a stream cipher is the **One-Time Pad** or **VERNAM Cipher**, where the input bits are XOR combined with the key bits. It is the only proven unbreakable cipher if the following conditions are satisfied:

- The key-length equals the message-length,
- the key is only used once and
- is chosen randomly.

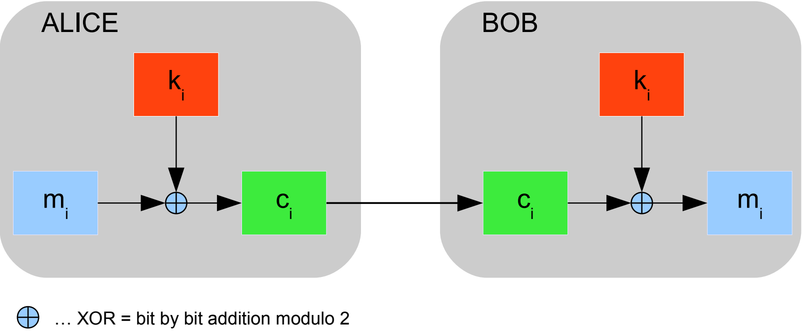


Figure 2.4: One-Time Pad

Figure 2.4 depicts the basic setup of a One-Time Pad. The system is based on the fact that the XOR operation is self-inverse. Therefore, the following equation holds:
$c_i = m_i \oplus k_i, k_i \oplus k_i = 0 \Rightarrow c_i \oplus k_i = m_i$

2.3.1 Cryptographic Hash Functions

A cryptographic hash function H takes an input of arbitrary length and produces a *message digest*, i.e. a fingerprint of the message of a fixed length (e.g. 128 or 160 bits) (see Figure 2.5).

Such a function has to satisfy certain properties:

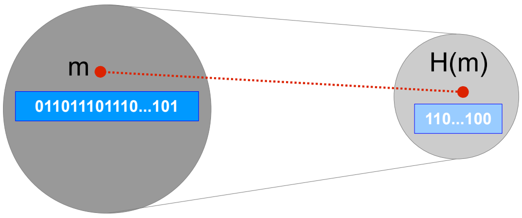


Figure 2.5: Cryptographic Hash Functions

- For a given message m, the message digest $H(m)$ can be calculated very quickly.
- **Weak Collision Resistance**: Given a fingerprint y, it is computationally infeasible to find an m with $H(m) = y$.
- **Strong Collision Resistance**: It is computationally infeasible to find messages m_1 and m_2 with $H(m_1) = H(m_2)$.

Note that, since the set of possible messages is much larger than the set of possible hash functions, there are always many examples of messages m_1 and m_2 with $H(m_1) = H(m_2)$. The last requirement claims that it should be hard to find such examples.

Examples

- MD2, MD4, MD5 (128 bit)
- SHA-1 (160 bit)
- RIPMED-160 (160 bit)
- SHA-256, SHA-384, SHA-512 (256,384 and 512 bit)

If a hash function is conditioned on a secret key k, we call it **Message Authentication Code (MAC)**, or *keyed* hash function. If Bob receives a MAC, he is able to verify its authenticity by using the symmetric key k.

Challenges and State of the Art

The list above of existing hash functions seems quite large and without any comments someone would guess there are many different ways of producing a secure message digest. In reality, only a few of the above mentioned algorithms still offer security on a high level, since there have been successful attacks in the meantime. National authorities like the *Telecommunication Regulation Authority* provide recommendations for secure algorithms and key lengths annually. The following recommendations regarding hash functions were published by the **Federal Agency for Security in Information-Technology Germany** in 2010 [3]:

applicable end 2015	**applicable end 2017**
SHA-224	SHA-256, SHA-384, SHA-512
(SHA-1, RIPEMD-160)*	

* only accepted for verification of accepted certificates, not for their compilation

In recent years, several cryptographic hash functions have been successfully attacked, and serious attacks have been published against SHA-1. In response, the **National Institute of Standards and Technology (NIST)** decided to develop one or more additional hash functions through a public competition, similar to the development process of the Advanced Encryption Standard (AES) [16].

The new hash algorithm will be referred to as "SHA-3" and will complement the SHA-2 hash algorithms currently specified in FIPS 180-3, Secure Hash Standard. In October 2008, 64 candidate algorithms were submitted to NIST for consideration. Among these, 51 met the minimum acceptance criteria and were accepted as First-Round Candidates on December 10, 2008, marking the beginning of the First Round of the SHA-3 cryptographic hash algorithm competition. In July 24 2009, 14 candidate algorithms were announced which moved forward to the second round of the competition. The 14 Second-Round Candidates are BLAKE, BLUE MIDNIGHT WISH, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein [16].

The winner of the competition will be announced by the end of 2012. Further information can be found on the NIST Homepage and in [16].

2.4 Public Key Algorithms

Public Key algorithms have been introduced in the 1970s and revolutionized cryptography. Suppose Alice wants to securely communicate with Bob, but they are hundreds of kilometers apart and have not agreed on a key to use. One solution would be to assign a trusted courier to carry the key from one to the other, but this is simply not always possible. Public key cryptography offers a solution, where the encryption key is made public but it is computationally infeasible to find the decryption key without information known only to Bob. The basic setup of an asymmetric cipher is shown in Figure 2.6.

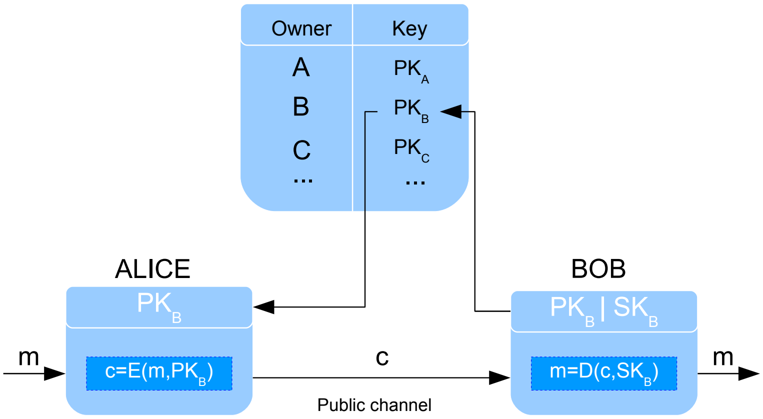


Figure 2.6: Principle of Asymmetric Ciphers

When Alice wants to send a message to Bob she first fetches his public key PK_B. The methods to distribute public keys and to bind them to the users are various and the author refers to the given literature for further studies. For simplicity, we imagine a public database like a telephone book where Alice looks up the public key which she uses to encrypt the message. The ciphertext is again transmitted to Bob via a public channel. It is only Bob who is in possession of the corresponding secret key SK_B, and who is therefore able to decrypt the ciphertext.

Parameters

$m \in M$	...	Plaintext
$c \in C$	...	Ciphertext
$PK_x \in K_{PK}$	...	Public Key of user X
$SK_x \in K_{SK}$	...	Secret Key of user X
E	...	Encryption Function
		$E : M \times K_{PK} \rightarrow C$, $c = E(m, PK)$
D	...	Decryption Function
		$D : C \times K_{SK} \rightarrow M$, $m = D(c, SK)$

2.4.1 One-Way Functions

A *One-Way Function* is a function which is easy to compute, but hard to invert. The terms easy and hard can mean that the computation of the function in the forward direction takes some seconds, but to compute the inverse could take some months, if possible at all. A *trapdoor one-way function* is a one-way function for which the inverse direction is easy, if a certain piece of information is given (the trapdoor), but difficult otherwise. Practice-oriented systems are based on the following one-way functions.

- Multiplication and factoring
  Given two prime numbers p, q, it is relatively easy to compute the product $n = p \cdot q$. On the other hand, it is incomparably harder to resolve the prime factors, when a composed integer n is given.

- Modular squaring and square roots
  Given $n = p \cdot q$ and e with $GCD(e, (p-1) \cdot (q-1)) = 1$, it is relatively easy to compute $y = x^e \ MOD \ n \ \forall x \in \mathbb{I}_{\ltimes}$. However, if n, e and y are given, no efficient algorithm exists to compute the corresponding x.

- Discrete exponential and logarithm
  Given a prime number p and an integer x between 0 and $p-1$ it is relatively easy to compute the discrete exponential $2^x \ MOD \ p$. On the other hand, with a given prime p and an integer y between 0 and $p-1$ there exists no efficient algorithm to find an x such that $2^x = y$.

- Subset Sum Problem
  The input to the *Subset Sum Problem* is a set of numbers $S = \{a_1, a_2, ..., a_n\}$ and a target integer t. The question is whether there is a subset of S that adds up exactly to t. It can be proven that this problem is $NP-$complete.

Examples

- RSA
- El-Gamal
- Chor-Rivest Knapsack
- McEliece
- Elliptic Curve Cryptography

State of the Art

In the same way as public authorities publish recommendations for secure hash algorithms, public key systems are constantly under examination to prevent successful attacks early enough. In August 2010 the latest recommendations for the RSA algorithm were published by the Federal Agency for Security in Information-Technology Germany [3].
According to the document, a key length of $n = 2048$ is advisable to ensure a high level of security until **2017**.
With regards to cryptographic primitives based on the subset sum problem, it has to be pointed out that apart from the Chor-Rivest algorithm, all knapsack-based systems have successfully been broken in the meantime. Since there are already attacks established which have the potential to likewise break the Chor-Rivest system, it is well possible that formerly very popular knapsack systems will not be employed in high level security applications in future anymore.
For further details regarding cryptanalysis of public key algorithms the author refers to [13, 11, 3].

2.5 Hybrid Crypto-Systems

Symmetric and asymmetric key algorithms offer a whole set of advantages, however they do have certain disadvantages. Symmetric ciphers have a very simple design and offer high transfer rates. On the other hand, a secret key has to be exchanged in advance. Asymmetric key algorithms dispose the key exchange problem, but the data rates are much lower.
Therefore, the idea of combining both algorithms is almost self-evident. Hybrid Crypto Systems realize an asymmetric key exchange to create a symmetric session key. The basic build up is depicted in Figure 2.7.

At the beginning of the communication a symmetric session key k is negotiated. A typical strategy is to generate random numbers, which are exchanged via asymmetric communication and which are concatenated, so that every communication partner contributes parts of the key. When this process is finished, a simple symmetric cipher like the AES can be used to exchange the secret information.

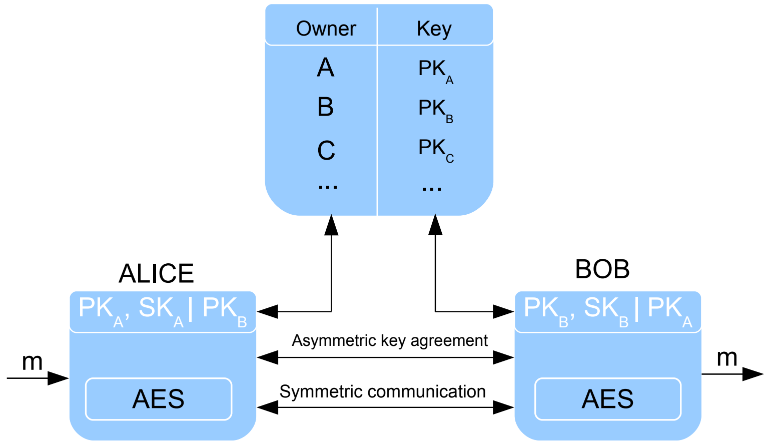


Figure 2.7: Principle of Hybrid Crypto-Systems

2.6 Digital Signatures

When Alice sends a message to Bob using a symmetric cipher, Bob can be sure that it stems from Alice, since she is the only one who is in possession of the secret key. However, Bob cannot prove to a third party that he received the message from Alice - she could simply claim it was Bob on his own who compiled it.

When using asymmetric ciphers, everybody who fetches Bob's public key could send him a message. This way, confidentiality is ensured, while authenticity is not at hand. What is missing is a liaison between the sender and the message, which can be realized using **Digital Signatures**. As equivalent to a personal signature, digital signatures have to satisfy some conditions. Digital Signatures

- have to be dependent on message and sender,
- can only be generated by the signer and
- can be verified by every user of the system.

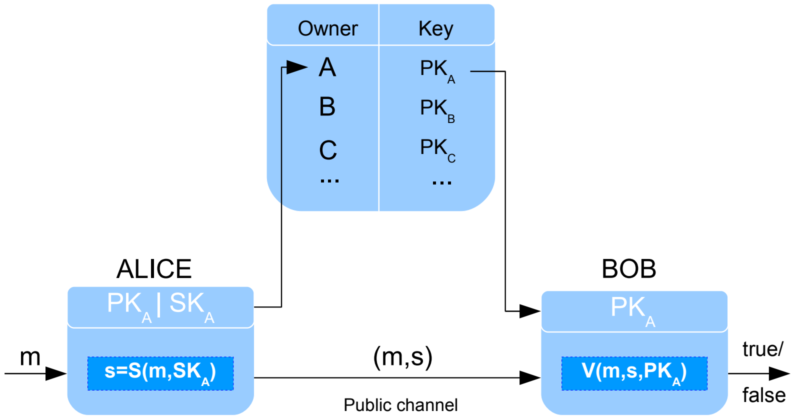


Figure 2.8: Principle of a Digital Signature

Figure 2.8 shows the basic implementation of a digital signature. Unlike the encryption scheme, Alice this times uses her secret key SK_A to sign the message m. When Bob receives the message m and its digital signature s, he is able to verify the authenticity of the message.

Very often cryptographic hash functions are implemented together with digital signatures to counter the threat of replication. Instead of the message, the hash value of the messages gets signed and transfered.

2.7 Certificates

With the help of digital signatures the integrity of the message is ensured. A problem that still occurs is the lack of evidence that the public key PK_x really belongs to user X. To counter this flaw certificates are implemented.

Certificates bind a public key to a particular person. A trusted third party, we call it *Certification Authority CA*, signs the data with its own secret key SK_{CA}. Now only the public key of the CA PK_{CA} has to be authentically established within the system.

Figure 2.9 illustrates which data is commonly included in a certificate.

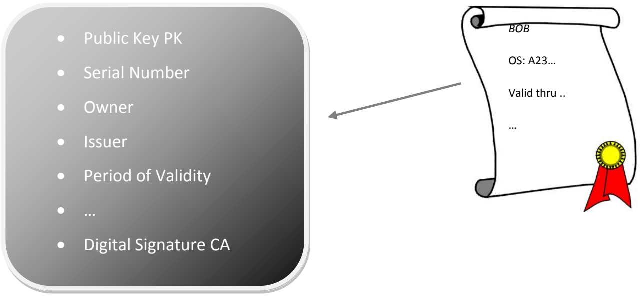


Figure 2.9: Certificate

Certificates are implemented in various areas like:

- Tax Declaration
- Health care
- Electronic Cash
- ePassport
- Remote Access

2.8 Challenge Response Authentication

Challenge-Response protocols serve as secure authentication processes for instances which are based on *knowledge*. A trusted authority sends a random challenge to a user who wants to authenticate himself. He generates a response and sends it back to the authority which verifies if the authentication was successful or not. The big advantage over existing password protocols

is that the challenge is not fixed but varies between different authentication attempts. With this, previously recorded responses will not help a potential attacker.
Such protocols can be implemented with symmetric and asymmetric setups. Since both realizations are quite similar, the author will only describe the symmetric case, for further studies, please again refer to the provided literature list.
A simple symmetric challenge-response implementation is depicted in Figure 2.10.

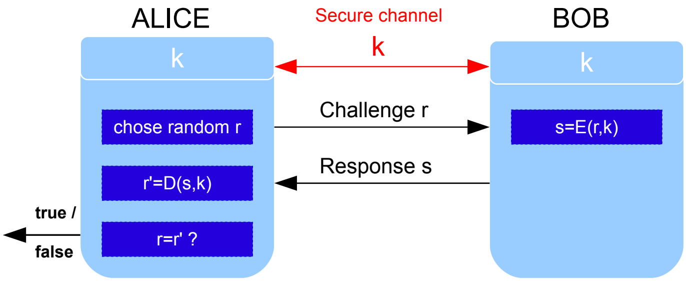


Figure 2.10: Symmetric Challenge Response Setup

Alice generates a random number, the challenge r and transmits it to Bob. He calculates the response $s = E(r, k)$ using the secret key k and sends it back to Alice. Now she verifies if the received response $r' = r = D(s, k)$. If this is the case, Bob successfully authenticated himself to Alice, otherwise the authentication failed.
The above described setup involves some flaws which can easily be fixed using the described cryptographic methods. First of all, the challenge and the response are transmitted as plain text, which invites an attacker Eve to a so-called *Known-Plaintext Attack*. To counter this problem, the challenge can simply be encrypted using the symmetric key before transmission. Another possibility is to use cryptographic hash functions and transmit the message digest of the challenge.
Note that in the described setup only Bob authenticates himself to Alice. If a bilateral authentication is desired, the setup has to be extended in a way that also Bob generates a random number, his challenge, which he sends to Alice awaiting the correct response.

2.9 Concluding Remarks

In addition to the above described cryptographic primitives, various fields exist, based on these algorithms, which could not be included in this thesis:

- Secret Sharing
- E-commerce
- Zero-Knowledge Concepts
- Electronic Cash
- Elliptic Curves
- Quantum Cryptography

2.10 Coding Theory

In this section an overview of the basic ideas of coding theory is given. At the beginning some examples of simple practical codes are presented. Later, the main topic, the BCH code, is described in detail.

2.10.1 Introduction

All communication channels contain some degree of noise, namely interference caused by various sources such as neighboring channels, electric impulses or deterioration of equipment. This noise can interfere with data transmission and make the information unreadable for the receiver. To prevent the loss of information, error correcting codes are implemented, which add redundancy to be able to correct a certain amount of errors.
The following examples describe simple implementations of error correcting codes, which are used very frequently in practice.

Repetition Codes

Consider an alphabet $\{A, B, C, D\}$. We want to send a letter across a noisy channel that has an error probability of $p = 0.1$. If we want to send the message 'B' there is a 90% chance that the symbol received is really 'B'. Since this leaves a quite large chance of errors, we instead repeat the message three times, i.e. we send the word 'BBB'. Suppose an error occurs and the received word is 'BBC'. We take the symbol that occurs most frequently as the message, namely 'B'. The probability of the correct message being found is the probability that all three letters are correct plus the probability that exactly one of the three letters is wrong:

$$(0.9)^3 + 3(0.9)^2(0.1) = 0.972$$

We see that with the principle of repeating the symbols three times the chance of an error can be reduced significantly.
In the following sections, certain conditions for *error detection* and *error correction* will be given. In this example we easily see that we are able to detect errors, and if maximum one letter is wrong, we can also correct it. When for example the word 'BCB' is received, we would suggest the correct word to be 'BBB', but it could also be 'CCC'. What makes us instinctively take the first choice is a statistic principle called *Maximum Likelihood.* The idea is basically that we guess that it is more likely that only one error occurs than two, if the error probability is $p = 0.1$. This concept will be used repeatedly in the following sections.

Parity Check Codes

Suppose we want to send a message of 7 bits. Before we transmit it, we add an eighth bit so that the number of non-zero bits is even. For example, we have the message 0110010. Since the number of non-zero bits is odd, the eighth bit is 1 and we get the new word 01100101. In doing so, we are able to detect an error of one bit immediately, since the message received will have an odd number of bits. However, it is impossible to tell which bit is incorrect, since an error in any bit could have yielded the odd number of non-zero bits. In practice, the message is simply resent when an error occurs.

ISBN Codes

The *International Standard Book Number* (ISBN) provides another example of an error correcting code. The ISBN is a 10-digit codeword that is uniquely assigned to each book when it is published. The ten digits represent the language, the publisher, an identity number and a *check digit* in the following way.

a_1	$a_2a_3a_4$	$a_5a_6a_7a_8a_9$	a_{10}
Language	Publisher	Identity Number	Check Digit

The numbers a_1 to a_9 can take values from 1 to 9 and the tenth digit a_{10} results from the following condition:

$$\sum_{i=1}^{10}(11-i)a_i \equiv 0\ (mod\ 11)$$

With this, it is also possible for a_{10} to take the value 10, which is simply depicted using the character X.

Example: $\boxed{ISBN = 3 - 528 - 08957 - a_{10}}$

To calculate the check digit we assign weights to the according positions and use the above stated formula:

a_1	$a_2a_3a_4$	$a_5a_6a_7a_8a_9$	a_{10}
3	528	08957	a_{10}
10	987	65432	1

$a_{10} \equiv -(3\cdot 10 + 5\cdot 9 + 2\cdot 8 + 8\cdot 7 + 0\cdot 6 + 8\cdot 5 + 9\cdot 4 + 5\cdot 3 + 7\cdot 2)\ mod\ 11$
$\Rightarrow a_{10} = 1$

It can be easily shown that ISBN codes detect single errors, transposition errors and skipped transposition errors, i.e. at least 90.1% of all errors. This only holds with the condition that at maximum one error per word occurred.

EAN Codes

The *European Article Number* is a bar-coding standard which is commonly seen on the package of any kind of grocery goods. This code was invented to replace the fault-prone hand-typing of prices by mechanical reading of 13-digit numbers, which uniquely identify the articles. The EAN code has been implemented in nearly every European country and offers error correction in a similar way as the ISBN code.

The EA-number is built up in the following way:

a_1a_2	$a_3a_4a_5a_6a_7$	$a_8a_9a_{10}a_{11}a_{12}$	a_{13}
Country	Producer	Identity Number	Check Digit

The EAN code operates in the residue ring modulo 10 and so the positions a_1 to a_{13} can take values from 0 to 9. This time, the weights alternate between 1 and 3, so the 13^{th} digit a_{13} results from the following condition:

$$1 \cdot a_1 + 3 \cdot a_2 + 1 \cdot a_3 + \ldots\ + 1 \cdot a_{11} + 3 \cdot a_{12} + 1 \cdot a_{13} \equiv 0\ (mod\ 10)$$

Example: $\boxed{EAN = 90 - 01375 - 00377 - a_{13}}$

Again, we calculate the check digit by assigning the weights to the according positions and inserting the values in the above formula:

a_1a_2	$a_3a_4a_5a_6a_7$	$a_8a_9a_{10}a_{11}a_{12}$	a_{13}
90	01375	00377	a_{13}
13	13131	31313	1

$$a_{13} \equiv -(9 \cdot 1 + 0 \cdot 3 + 0 \cdot 1 + 1 \cdot 3 + 3 \cdot 1 + 7 \cdot 3 + 5 \cdot 1 + 0 \cdot 3 + 0 \cdot 1 + 3 \cdot 3 + 7 \cdot 1 + 7 \cdot 3)\ mod\, 10$$
$$\Rightarrow a_{13} = 2$$

In addition to the correction of single errors, it can be shown that the EAN code is able to correct phonetic errors. Further descriptions and proofs of the stated theorems can be found in [21].

2.10.2 Error Correcting Codes

To be able to detect and correct transmission errors it is necessary to add some degree of redundancy to the actual information bits. The idea is that a sender starts with a message and **encodes** it to obtain the codewords consisting of a sequence of symbols. After transmitting it over a noisy channel, it is very likely that the received codeword includes some errors. Therefore, the receiver has to **decode** the message, meaning to correct the errors in order to change back what is received to codewords and then recover the original message, like depicted in Figure 2.11.

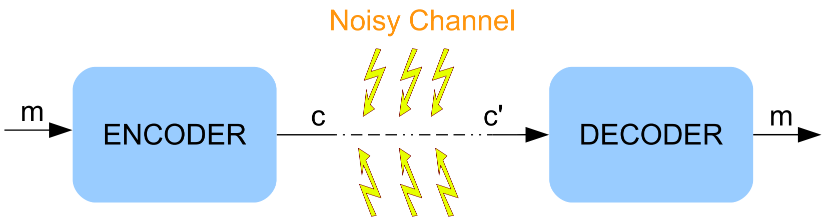


Figure 2.11: Principle of Error Correcting Codes

The symbols used to construct codewords belong to an *alphabet*. The author will only consider an alphabet consisting of binary digits 0 and 1 which is called a *binary code*.
Moreover it is assumed that a *symmetric binary channel* is underlying, i.e. the error likelihood is independent of the instant bit-value, like illustrated in Figure 2.12.

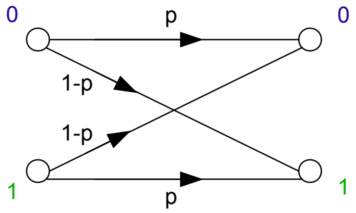


Figure 2.12: Symmetric Binary Channel

The challenge of the decoder is to detect incorrect bits and correct them. The method applied is called *Maximum Likelihood Decoding.* It is assumed that a minimal error number occurs, i.e. a codeword is sought, which differs form the received word in as few positions as possible.
In practice, mostly so-called *Block Codes* are deployed. These codes arrange the binary message in blocks of k digits and append $(n-k)$ so-called check digits, so that we get a codeword with n bits. We call such codes (n,k)-codes.

$$\boxed{\text{Message of length } k} \longrightarrow \boxed{(n,k)\text{-Encoder}} \longrightarrow \boxed{\text{Codeword of length } n}$$

A fundamental instrument to describe the error detection and error correction ability of a code is the so-called *Hamming Distance.*

Definition. *The **Hamming Distance** d of two words $(a_1,...,a_n)$, $(b_1,...,b_n) \in GF(2)^n$ is defined as the number of digits i, $1 \leq i \leq n$, such that $a_i \neq b_i$, i.e. the number of digits where they differentiate.*

Theorem. *(i) A code C can detect up to t errors if the Hamming distance $d(C) \geq t+1$.*

(ii) A code C can correct up to t errors if the Hamming distance $d(C) \geq 2t+1$.

Since these theorems are easy to prove the author again refers to the provided literature for further studies.

2.10.3 Linear Codes

When we use mobile phones our voice is turned into digital data and an error correcting code is applied before a message is sent. When our communication partner receives the data, the errors in the transmission must be detected and corrected. Afterwards, the data is turned into sound again that represents our voice. The amount of time it takes for a packet of data to be decoded is critical in such an application. The question of efficiently decoding a code is therefore of major importance. In order to decode quickly, it is helpful to have some form of structure in the code, rather then taking the code to be a random subset of an alphabet $\mathcal{A}^n$. This is one of the primary reasons for studying linear codes.

Definition. *A (n, k)-Code is called* ***Linear Code*** *if the coding function $f : GF(2)^k \longrightarrow GF(2)^n$ is a linear map.*

(i) $f((a_1, ..., a_k) + (b_1, ..., b_k)) = f((a_1, ..., a_k)) + f((b_1, ..., b_k))$

(ii) $f((0, ..., 0)) = (0, ..., 0)$

For a binary code we may state an even simpler definition:
A binary code of length n and dimension k is a set of 2^k binary n-tuple such that the sum of any two codewords is always a codeword.
Since it is simply not possible to give a full introduction to every aspect of linear codes within this thesis, only the most important characteristics and applications will be stated hereafter.

Linear Codes in Polynomial Representation

A very feasible way to work with linear codes is the representation as polynomials. Every message $(a_0, a_1, ..., a_{n-1}) \in GF(2)^n$ can be written as the following polynomial:

$$a_0 + a_1x + a_2x^2 + ... + a_{n-1}x^{n-1} \in GF(2)[X]$$

Let $p \in GF(2)[X]$ be a polynomial of degree $n - k$. Then we call the code, whose codewords are constructed by all polynomials divisible by p of degree smaller than n, the *polynomial code generated by p.* The following procedure describes the construction of a binary (n, k)-polynomial code:

(i) Represent the message m as a polynomial of degree $\leq k$
$m \mathrel{\widehat{=}} m(x) = a_0 + a_1x + ... + a_{k-1}x^{k-1}$

(ii) Multiply $m(x)$ with x^{n-k}
$m(x) \cdot x^{n-k} = (0, 0, ..., 0, a_0, a_1, ..., a_{k-1})$
In doing so we shift the information bits at the end of the polynomial and produce $(n-k)$ zeros at the beginning.

(iii) Divide $m(x) \cdot x^{n-k}$ by $p(x)$
$m(x) \cdot x^{n-k} = p(x) \cdot q(x) + r(x)$ with $r(x) = 0$ or degree of $r(x) \ < \ degree\ p(x) = n - k$

(iv) Add up r and $m \cdot x^{n-k}$ to get the codeword v
$v = r + m \cdot x^{n-k} = q \cdot p$

Linear Codes in Matrix Representation

One basic rule of Linear Algebra states that for every linear map of a vector space of degree k onto a vector space of degree n we can assign a unique $k \times x$ matrix regarding the basis of the underlying vector spaces. Let $e_1^t, e_2^t, ..., e_h^t$ be the standard basis of $GF(2)^h$ for some $h \in \mathbb{N}$ and $f : GF(2)^k \longrightarrow GF(2)^n$ be a linear map of a (n, k)-code. If we compute $f(e_i^k) = a_{i1}e_1^n + ... + a_{in}e_n^n$

for $1 \leq i \leq k$ so we call the matrix $G = (a_{ij})$, $1 \leq i \leq k$, $1 \leq j \leq n$ the **Generating Matrix** of the (n, k) linear code.

Suppose we have a message $m \in GF(2)^k$, the corresponding codeword v is given by $\boxed{v = m \cdot G}$. Since the information digits are on the last positions of a codeword, the generating matrix has the following representation:

$$G = (A, I_k)$$

where I_k is the $k \times k$ identity matrix and A a $k \times (n-k)$ matrix. If we construct the $n \times (n-k)$ matrix

$$K := \begin{pmatrix} I_{n-k} \\ \hline A \end{pmatrix}$$

it follows that $G \cdot K = 0$, i.e. a vector v is a member of the code C if and only if $v \cdot K = (0, ..., 0)$. We call the matrix K the **Check Matrix** of the underlying (n, k) linear code.

Example

To get a feeling how such an encoding and decoding process works, the single steps will be demonstrated in a simple example.

Wanted are the generating and the check matrix of the $(6, 3)-$code generated by the polynomial $p = 1 + x^2 + x^3$.

The rows of the generating matrix are the codewords that correspond to the messages $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$. Using the procedure described above we get the following codewords:

message	check digits	information digits
100	101	100
010	111	010
001	110	001

From the table above we can derive the generating matrix G and therewith the check matrix K.

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

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

Decoding of Linear Codes

In the descriptions above we have seen how to build up a linear code and how to work with polynomial and matrix representation. A very important question of course is, how to decode a received word. Before the single steps of a decoding process will be described, the author will state some important definitions.

Definition. *The **Hamming weight** of a vector u is defined as the number of nonzero places in u.*

Definition. *Let C be a linear code and let u be an n-dimensional vector. The set $u + C$ given by*

$$u + C = \{u + c | c \in C\}$$

*is called a **coset** of C.*

It is easy to see that if $v \in u + C$, then the sets $v + C$ and $u + C$ are the same.

Definition. *A vector having minimum Hamming weight in a coset is called the **coset leader**.*

Definition. *Let K be the check matrix of a $(n, k)-$linear code and let $v \in GF(2)^n$. Then we call the $(n - k)$-tuple $s := v \cdot K$ the **syndrome** of v.*

Decoding can be achieved by building a syndrome lookup table, which consists of the coset leaders and their corresponding syndromes. With a syndrome lookup table, we can decode following the steps listed hereafter:

1. For a received vector r calculate its syndrome $s(r) = r \cdot K$.
2. Next, find the coset leader with the same syndrome as $s(r)$. Call the coset leader c_0.
3. Decode r as $r - c_0$.

Syndrome decoding requires significantly fewer steps than searching for the nearest codeword to a received vector. However, for large codes it is still to inefficient to be practical.

2.10.4 Hamming Codes

Hamming codes are an important class of single error correcting codes that can easily encode and decode. They were originally used for controlling errors in long-distance telephone calls. Before the characteristics are presented two important definitions will be given, which are in close conjunction with Hamming codes.

Definition. *A linear code is called **perfect**, if the coset leaders in a syndrome lookup table are precisely the words of weight $\leq m$ for an $m \in \mathbb{N}$.*

Definition. *A linear code is called **quasi-perfect**, if the coset leaders in a syndrome lookup table include all words with weight $\leq m$, some words of weight $m+1$ but no words with a higher weight for a certain $m \in \mathbb{N}$.*

Binary Hamming codes form a class of perfect codes and are defined through the following parameters:

1. Code length: $n = 2^m - 1$
2. Dimension: $k = 2^m - m - 1$
3. Minimum distance: $d = 3$

Because of the special characteristics of Hamming codes, the method for decoding can be simplified.

1. Compute the syndrome $s = y \cdot K$ for a received vector y. If $s = 0$ then there are no errors and the received codeword is returned.
2. Otherwise, determine the position j of the row of K that equals the computed syndrome.
3. Change the j-th bit in the received word and output the resulting code.

As long as there is maximum one bit error in the received vector, the result will be the initially sent codeword.

2.10.5 Cyclic Codes

Cyclic codes are a very important class of codes which prove to be convenient regarding error detection and error correction. Cyclic codes are quite easy to realize in technical systems and therefore very feasible for practical applications.
A code is called **cyclic** if

$$(c_1, c_2, ..., c_n) \in C \textit{ implies } (c_n, c_1, c_2, ..., c_{n-1}) \in C.$$

This might seem to be a strange condition for a code to satisfy. After all, it seems to be irrelevant that, for a given codeword, all of its cyclic shifts are still codewords. The point is that cyclic codes have a lot of structure which yield an efficient decoding algorithm (as we will see later in the case of BCH codes).
To reasonably discuss the background of cyclic codes, some knowledge in the Field- and Ring-theory would be necessary. Since such an introduction would go far beyond the scope of this thesis the author here choses to refer to the literature stated at the beginning of this chapter. The following theorem describes the general setup of cyclic codes.

Theorem. *Let C be a cyclic code of length n over $GF(2)^n$. To each codeword $(a_0, ..., a_{n-1}) \in C$, associate the polynomial $a_0 + a_1x + ... + a_{n-1}x^{n-1}$ in $GF(2)[X]$. Among all the nonzero polynomials obtained from C in this way, let $g(x)$ have the smallest degree. The polynomial $g(x)$ is called the **generating polynomial** for C. Then*

1. *$g(x)$ is uniquely determined by C.*
2. *$g(x)$ is a divisor of $x^n - 1$.*
3. *C is exactly the set of coefficients of the polynomials of the form $g(x)f(x)$ with $deg(f) \leq n - 1 - deg(g)$.*
4. *Write $x^n - 1 = g(x)h(x)$. Then $m(x) \in GF(2)^n/(x^n - 1)$ corresponds to an element of C if and only if $h(x)m(x) \equiv 0 \; mod \; (x^n - 1)$.*

From the theorem above it follows that the row vectors of the generating matrix are given with $g(x), xg(x), ..., x^{n-r-1}g(x)$. Moreover, the columns of the check matrix can be derived as $h(x), xh(x), ..., x^{r-1}h(x)$ in reversed order.

Since descriptions of cyclic codes are rather complicated and very theoretical, a concrete example will surely help to bring the topic closer to the reader.

We choose $n = 7$. In $GF(2)$ it holds that

$$x^7 - 1 = (x - 1)(1 + x + x^3)(1 + x^2 + x^3).$$

We choose $g(x) = 1 + x^2 + x^3$ as the generating polynomial and so $h(x)$ arises as $h(x) = (x - 1)(1 + x + x^3) = 1 + x^2 + x^3 + x^4$. In the same way we could also take $g(x) = (x - 1)$ or $g(x) = 1 + x + x^3$ as the generating polynomial but the quality of the code stays always the same. To derive the generating matrix and the check matrix we have to calculate $xg(x)$, $x^2g(x)$, $x^3g(x)$ and $xh(x)$, $x^2h(x)$.

$$\begin{aligned}
xg(x) &= x + x^3 + x^4 & \widehat{=} \; 0101100 \\
x^2g(x) &= x^2 + x^4 + x^5 & \widehat{=} \; 0010110 \\
x^3g(x) &= x^3 + x^5 + x^6 & \widehat{=} \; 0001011 \\
xh(x) &= x + x^3 + x^4 + x^5 & \widehat{=} \; 0101110 \\
x^2h(x) &= x^2 + x^4 + x^5 + x^6 & \widehat{=} \; 0010111
\end{aligned}$$

Applying the instructions above we get the following matrices:

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

$$K^T = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Assume we have the following message $m = (1010)$. The codeword can be simply calculated by multiplying the message with the generating matrix G.

$$(1010)\begin{pmatrix}1&0&1&1&0&0&0\\0&1&0&1&1&0&0\\0&0&1&0&1&1&0\\0&0&0&1&0&1&1\end{pmatrix} = (1001110)$$

Now we check if the received word is a codeword, i.e. no transmission errors occurred:

$$(1001110)\begin{pmatrix}0&0&1&1&1&0&1\\0&1&1&1&0&1&0\\1&1&1&0&1&0&0\end{pmatrix}^T = (000) \checkmark$$

Since we get the desired zero vector, we can assume that no transmission error occurred. The following case shows an error which we correct with the help of the check matrix. Suppose we received the word (1101110), which yields the following syndrome:

$$(1101110)\cdot K = (011)$$

A word of minimum weight over $GF(2)^7$ with the same syndrome is (0100000), so we get the codeword by adding up the error vector to the received word:

$$(1101110) + (0100000) = (1001110) \checkmark$$

The most famous cyclic codes are the BCH codes, which have beneficial attributes regarding error detection and correction and are therefore also implemented in the applications described in the following chapters.

2.10.6 BCH Codes

BCH codes were discovered around 1959 by R. C. Bose and D. K. Ray-Chaudhuri and independently by A. Hocquenghem. One reason for their importance is that they contain good decoding algorithms. BCH codes are implemented in trans-atlantic communications and satellites.
BCH codes are cyclic codes of length $n = 2^m - 1$ ($m \in \mathbb{N}$), which are able to correct any combination of $t < 2^{m-1}$ errors. Before a definition of the generating polynomial will be given, some basic algebraic properties need to be described.
It is easy to show that every element $a \in GF(2)$ is *algebraic* in $GF(2)$, i.e. there exists a polynomial $f \in GF(2)[x]$ such that $f(a) = 0$. As a result, there exists a unique irreducible polynomial $g \in GF(2)[x]$ for each $a \in GF(2^m)$, such that a is a root of g. g is called the *Minimal Polynomial* corresponding to a.

If α is a generating element of the multiplicative group of $GF(2^m)$ and p_i the minimal polynomial corresponding to α^i, then we define:

$$p(x) := LCM(p_1, p_2, ..., p_{2t})$$

Clearly $\alpha, \alpha^2, ..., \alpha^{2t}$ are roots of p. Since it holds that $(p_i(x))^2 = p_i(x^2)$ it follows that simultaneously to α^i also α^{2i} is a root of p_i. So it holds that

$$p(x) := LCM(p_1, p_3, ..., p_{2t-1}).$$

Applying some basic knowledge of linear algebra we can moreover derive that

$$deg(p) \leq (deg(p_1)) + (deg(p_2)) + ... + (deg(p_{2t-1})).$$

To get a feeling how to compute the generating polynomial in practice a simple example will be presented.
Wanted is the generating polynomial for the $3 - error$ correcting BCH code of length $n = 15$; $15 = 2^4 - 1 \ \Rightarrow m = 4$.
The hard part is now to find a generating element in $GF(16)$. In our case such an element is defined with the equation $\alpha^4 + \alpha + 1 = 0$. The wanted generating polynomial arises from $p(x) = LCM(p_1, p_3, p_5)$.
Now we have to calculate the roots of α^3 respectively α^5:

$$(\alpha^3)^2 = \alpha^6,\ (\alpha^6)^2 = \alpha^{12},\ (\alpha^{12})^2 = \alpha^{24} = \alpha^9 \ and\ (\alpha^9)^2 = \alpha^{18} = \alpha^3$$

$$(\alpha^5)^2 = \alpha^{10},\ (\alpha^{10})^2 = \alpha^{20} = \alpha^5$$

With the condition $\alpha^4 + \alpha + 1 = 0$ we get

$$p_3 = (x - \alpha^3)(x - \alpha^6)(x - \alpha^{12})(x - \alpha^9) = x^4 + x^3 + x^2 + x + 1.$$

$$p_5 = (x - \alpha^5)(x - \alpha^{10}) = x^2 + x + 1.$$

So we get the desired generating polynomial $p(x)$ with

$$p(x) = LCM((x^4 + x + 1), (x^4 + x^3 + x^2 + x + 1), (x^2 + x + 1)) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$$

From that point we may proceed in analogy to the last example, i.e. deriving the generating and check matrix and so on.

Decoding of BCH Codes

One of the reasons why BCH codes are so feasible is that there exist good decoding algorithms. Probably the best known was developed by **Berlekamp and Massey**. Hereafter, the author will not present the entire algorithms, but, in order to provide the reader with the basics of some ideas that are involved, the author shows a way to correct one error in a binary BCH code with a designed distance $d \geq 3$.

Let C be a BCH code with a designed distance $d \geq 3$. Then C is a cyclic code with length n and a generating polynomial $g(x)$. There exists a $n-th$ primitive root α such that

$$g(\alpha^{k+1}) = g(\alpha^{k+2}) = 0$$

for some integer k.
Let

$$K^T = \begin{pmatrix} 1 & \alpha^{k+1} & \alpha^{2(k+1)} & ... & \alpha^{(n-1)(k+1)} \\ 1 & \alpha^{k+2} & \alpha^{2(k+2)} & ... & \alpha^{(n-1)(k+2)} \end{pmatrix}.$$

If $c = (c_0, ..., c_{n-1})$ is a codeword, the polynomial $m(x) = c_0 + c_1x + ... + c_{n-1}x^{n-1}$ is a multiple of $g(x)$, so

$$m(\alpha^{k+1}) = m(\alpha^{k+2}) = 0.$$

This may be rewritten in terms of the check matrix K:

$$c \cdot K = (0, 0)$$

Assume the vector $r = c + e$ is received, where c is a codeword and $e = (e_0, ..., e_{n-1})$ is an error vector. We assume that maximum one entry of e is nonzero.
Here is the algorithm for correcting one error.

1. Compute $r \cdot K = (s_1, s_2)$.
2. If $s_1 = 0$, there is no error (or there is more than one error), so we are done.
3. If $s_1 \neq 0$, compute $\frac{s_2}{s_1}$. This will be a power α^{j-1} of α – the error is in the $j-th$ position. Since we are working in $GF(2)$ we are done, otherwise there would be several choices for e_j.
4. Add the error vector e to the received vector r to obtain the correct codeword c.

To complete this section about BCH codes, one last example aims at reassembling the collected ideas.
Let us have a look at the BCH code of length $n = 7$ and a designed distance 7. This is nothing more than the binary repetition code of length $n = 7$ with the two codewords $(0, 0, 0, 0, 0, 0, 0), (1, 1, 1, 1, 1, 1, 1)$.

Assume the received vector is $r = (1, 1, 1, 1, 0, 1, 1)$. Let α be a root of $x^3 + x + 1$. For the generating element α we can derive the following formulas:

$$\alpha^3 = \alpha + 1$$

$$\alpha^4 = \alpha^2 + \alpha$$

$$\alpha^5 = \alpha^2 + \alpha + 1$$

$$\alpha^6 = \alpha^2 + 1$$

Now we can compute

$$r \cdot K = (1, 1, 1, 1, 0, 1, 1) \begin{pmatrix} 1 & 1 \\ \alpha & \alpha^2 \\ \alpha^2 & \alpha^4 \\ \vdots & \vdots \\ \alpha^6 & \alpha^{12} \end{pmatrix} = (\alpha + \alpha^2, \alpha).$$

So, $s_1 = \alpha + \alpha^2$ and $s_2 = \alpha$. Now we need to calculate

$$\frac{s_2}{s_1} = \frac{\alpha}{\alpha + \alpha^2} = \frac{\alpha}{\alpha^4} = \alpha^{-3} = \alpha^4.$$

Therefore $j - 1 = 4$, so the error position $j = 5$. With the calculated error vector $(0, 0, 0, 0, 1, 0, 0)$ we get the corrected message as

$$r + e = (1, 1, 1, 1, 0, 1, 1) + (0, 0, 0, 0, 1, 0, 0) = (1, 1, 1, 1, 1, 1, 1) \quad \checkmark$$

2.11 Concluding Remarks

Prior to the preparation of this chapter, the principle aim was to provide a compressed summary of the most important facts about cryptography and coding theory. Quite soon it turned out that many topics require a certain amount of background knowledge to be able to reasonably discuss them. The anticipated short overview grew more and more detailed although a lot of knowledge about finite fields or linear algebra had been assumed general knowledge already.

As stated already at the beginning of this chapter, a lot of information has been taken from the book “Introduction of Cryptography with Coding Theory” [21] which is considered a very legible assembly of especially coding theory which can be strongly recommended.

3 Physically Unclonable Functions

This chapter will provide a detailed overview of Physically Unclonable Functions (PUFs). The most important definitions will be given, possible applications described and different types of PUFs will be discussed.
However, before going into detail, the author wants to give a short introduction into the very cognate discipline **Biometrics**. This chapter and subsequent paragraphs will show that there is a strong logical connection between the two technologies.

3.1 Biometrics

The main concern of Biometrics is the ability to recognize individuals through physiological or behavioral characteristics or modalities. Biometric recognition systems should provide a reliable personal scheme for either confirming or determining the identity of an individual.
The history of Biometrics can be traced back to early Egyptian times when certain sets of physiological properties were used to distinguish traders. Giving a thorough overview of the historical development of biometric systems would go beyond the scope of this thesis, therefore only some references for intensive studies [26, 2] are provided.
A simple biometric system consists of four basic components

1. *The Sensor Module*: It acquires the biometric data.
2. *The Feature Extraction Module*: This is where the acquired data is processed to extract future feature vectors.
3. *The Matching Module*: Here the feature vectors are compared with those in the template.
4. *The Decision-making Module*: In this module the user's identity is established or a claimed identity is accepted or rejected.

3.1.1 Overview of commonly used Biometrics

In this chapter the author wants to give a brief overview of the most feasible modern biometric systems. For a detailed study, please again refer to [2].

- **Signature**
  A signature is a simple, concrete expression of the unique variations in human hand geometry. Signature verification technologies examine dynamics such as pressure, speed or direction of writing. Personal signatures change over a period of time and are influenced

by physical and emotional conditions, and so the problem of long-term reliability and lack of accuracy occurs.

- **Voice**
  The features of an individual's voice are based on physical characteristics such as vocal tracts, mouth, nasal cavities and lips which are creating the sound. [2]. The speakers have to pronounce special pass-phrases, which are recorded, converted into a digital code and stored in a template. The disadvantages of such a recognition system are that people's voices may change (when we are sick or simply grow older), and that fairly large byte codes are required.
- **Fingerprint**
  We all know the old ink-and-paper method where a user has to place an inked finger on a piece of paper. This personal identification was used over centuries, primarily by law enforcement agencies and accuracy was very high. Today, compact sensors provide digital images of the pattern of ridges and furrows located on the tip of our fingers. In comparison to the Signature verification, the fingerprint offers long-term reliability. Since there has to be a physical contact between the finger and the sensor, additional information such as temperature and pulse may be gathered. However, this contact may also lead to oily and dirty sensors and therefore, reliability and sensitivity of the reader may be reduced.
- **Face**
  Facial images are the most common biometric characteristic used by humans for personal recognition, hence it became obvious to use this biometric in technology. Spatial geometric features like the location and shape of the eyes, eyebrows, nose, lips, or the chin are recorded. Non-cooperative behavior of the user, bad lighting or environmental effects can lead to performance problems. It is questionable if a face itself is a sufficient basis for the recognition of a person given a large number of identities, with a high level of confidence.
- **Infrared Thermogram**
  With the help of infrared cameras, it is possible to capture the pattern of heat radiated by the human body, which is considered to be unique for each person. The advantage of this method is that it is noninvasive; however, there exist certain drawbacks. First of all, image acquisition is rather difficult when other heat emitting surfaces are close to the body. Furthermore, we have to deal with the extra issues of three-dimensional space and orientation of the hand. Finally, the price of infrared sensors is quite high, so this method surely does not represent the best solution for large-scale production.
- **Retina**
  A Retina scan creates an "eye-signature" from the vascular configuration of the retina which is supposed to be a unique characteristic of each individual. Such a scanning system quickly maps the eye's blood vessel pattern into an easily retrievable digitized database. Because of the uniqueness and stability of this characteristic, retina scans are more reliable

than any of the examples discussed so far. However, the fairly close physical contact to the reader and the fear of eye disease reduces the public acceptance of this system.

- **Iris**
  Iris scanners measure the iris pattern in the colored part of the eye. This complex patterns contain many distinctive features such as arching ligaments, furrows, ridges, rings and freckles. Iris scanning is less intrusive than retina scans because the iris is easily visible from several meters away. Another advantage is that the irises of two identical twins are different, what makes measurments totally unique. The disadvantages of this characteristic are that systems are quite expensive, the storage requirements are relatively high and user acceptance is not fully given.
- **DNA**
  Deoxyribonucleic acid is probably the most reliable biometric. It is in fact a one-dimensional code unique for each person, except identical twins. So far, this technology is mainly used in the law enforcement area, as this technology requires complex chemical methods involving experts' skills. Another drawback is that it is easy to steal a piece of DNA from an individual and use it for an ulterior purpose.

Further examples for biometric characteristics are:

- Hand geometry
- Palmprint
- Gait
- Keystroke
- Body Odor
- etc.

For further studies please refer to [2, 23].

3.1.2 Biometric System Performance

It seems evident that measurments of biometric characteristics always include some kind of error. Whether it is the bad user-interaction, imperfect lighting conditions or any other environmental effects, two measurments of the same characteristic of an individual taken at different times will not be exactly the same. Therefore, the system performance plays a major role for practical applications. How long does it take to make an identification decision? How many bytes of storage are required? What about the accuracy?

The following error rates make it possible to characterize the performance of a system, and to assess the reliability of our measurments.

- False Acceptance Rate FAR
  This rate gives us the percentage of impostors accepted, i.e. the probability of invalid inputs which are incorrectly accepted.

- False Rejection Rate FRR
  This rate gives us the percentage of authorized users rejected, i.e. the percentage of valid inputs which are incorrectly rejected.

In a perfect system both rates are zero, but in practice, there has to be a trade-off between both parameters.

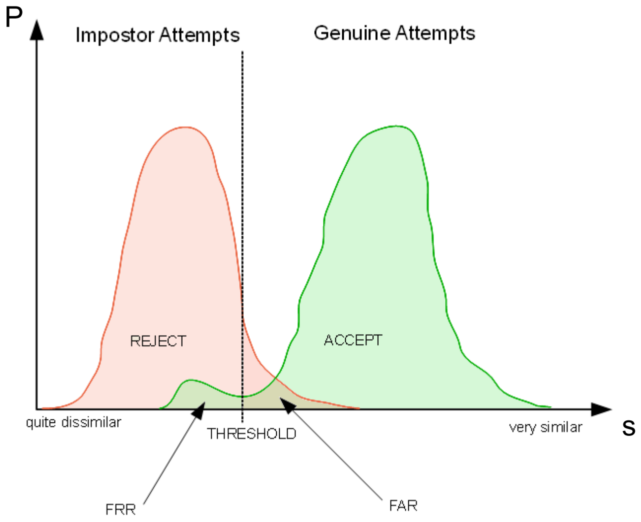


Figure 3.1: FAR and FRR

Figure 3.1 gives a schematic view of the coherence of FAR and FRR. The error rates are basically functions of the threshold t. If we decrease t, we make the system more tolerant to input variations what clearly increases the rate of accepted impostors, i.e. FAR increases. If we raise it in the opposite direction, the system gets more secure, but at the same time the rate of authorized users rejected raises, i.e. FRR increases.
In order to arrive at the best recognition performance, a variety of signal processing steps is performed on the biometric measurement. Next to digital techniques like non-linear filtering or dimension reduction to improve the signal-to-noise ratio, *multimodal biometric systems* are implemented. These are special biometric systems using different applications to capture different types of biometrics [2, 26]. A possible example of a multimodal biometric system is a combination of measuring the fingerprint, recording the voice and performing face recognition. Such systems increase the performance and are expected to be more reliable, due to the presence of multiple independent pieces of evidence.

The principles of biometrics, as described in this section, are applicable not only to the human body but also to physical objects in general. Many objects have features that are unique and difficult to clone and can therefore be used for identification processes and for extracting binary strings.

And here the circle closes in. Just like FAR and FRR are a representations of how reliable biometric systems are, there exist parameters to characterize the performance of PUFs.

3.2 PUF Properties

As described in Chapter 1, PUFs are nothing more than physical objects that contain random components which make them unclonable. The term *function* tells us that for some given input we measure a certain output but in general a PUF is not a function in a *mathematical sense.* For some fixed input there may be several outputs because of physical variations, i.e. the result is not unique.

Typically we call the PUF inputs *challenges* and the measured outputs *responses.* During an *Enrollment Phase* for a given PUF a set of challenge-response pairs (CRP) is measured and stored in a public database. When the PUF is used “in the field”, the *Verification Phase* takes place, where a certain challenge is applied to the PUF and the measured response is compared to the reference response in the database, as illustrated in Figure 3.2.

A typical application using exactly this procedure is the *Secure Key-Card.* A person who is in possession of a PUF-secured Key-Card wants to get access to a terminal, which plays the role of the verifier. It picks a random CRP from the public database and challenges the PUF. Only if the response of the device is close enough to the response in the database, access is granted. To prevent replay attacks every CRP is only used once.

The big advantage in comparison to existing schemes is that the key-card is not clonable. Assume an attacker *Eve* is temporarily in the possession of the card. During a certain period of time, she may record a set of CRPs but she has no idea which CRPs were gathered in the *Enrollment-Phase.* Therefore it is very unlikely that she will be able to build her own card with the same challenge-response behavior.

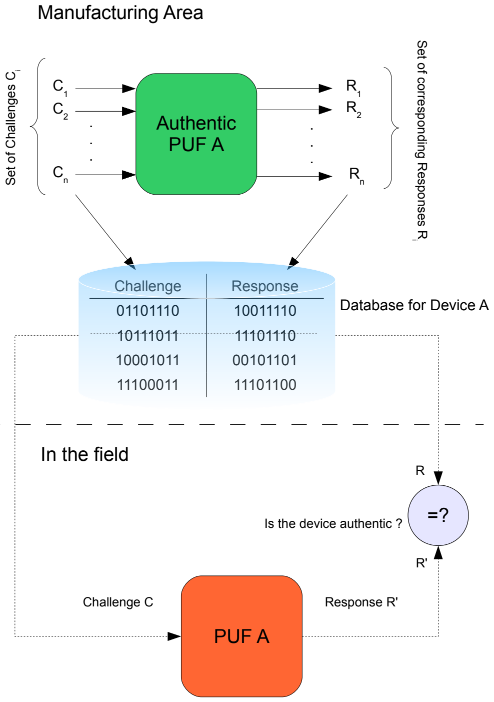
3.2.1 Definitions

Mathematicians always have the intention to formalize given objects and the interrelations they are dealing with. In the case of PUFs this becomes quite hard at some point since we are dealing more with functions in an engineering, than in a mathematical sense. Nevertheless, for the understanding of the thesis, it is necessary to be familiar with some terms and definitions that will accompany us through out the following chapters.

Let us start with a formal definition of Physically Unclonable Functions. There are several ways to characterize the objects; however, the author considers Blaise Gassend’s definition in his thesis Physical Random Functions [4] the completest.

Definition. *A **Physically Unclonable Function (PUF)** is a function that maps challenges to responses, that is embodied by a physical device and that has the following properties:*

1. *Easy to evaluate: The physical device is capable for evaluating the function in a short amount of time.*



Challenge	Response
01101110	10011110
10111011	11101110
10001011	00101101
11100011	11101100

Figure 3.2: PUF-secured Key-Card

2. *Hard to characterize: From a limited number of plausible physical measurments or queries of chosen CRP, an attacker who no longer has the device and who can only use a limited amount of resources (time, money, raw material, etc.) can only extract a negligible amount of information about the response to a randomly chosen challenge.*

In literature the terms *Physical Unclonable Function* and *Physically Unclonable Function* are often used interchangeably although the spelling indicates a different meaning. If we use the term Physical Unclonable Functions, we speak of a physical function that is unclonable in every sense. If we use the term Physically Unclonable Function we speak of a function that is unclonable in a physical sense, which relates better to the concept of PUFs as described hereafter.

Definition. *A type of PUF is said to be* ***Manufacturer Resistant*** *if it is technically infeasible to produce two identical PUFs of this type given only a polynomial amount of resources.*

In other words, the challenge-response behavior of Manufacturer Resistant PUFs only depends on random process variations which are beyond the control of the manufacturer. This greatly increases the level of trust and makes these PUFs even more suitable for authentication applications.

In earlier chapters it has been described that one of the major advantages of PUFs is their ability to offer low cost applications, since the required randomness does not have to be added separately, but is a result of the general manufacturing process, i.e no additional steps are necessary. For the better part of PUF instantiations this is true, but some realizations, as will be described in Chapter 4, additionally insert random components. To distinguish between the different types, the definition of *Intrinsic PUFs* is introduced.

Definition. *We call a PUF* ***Intrinsic*** *if it is embedded on an IC and if the basic building blocks are regular digital primitives for the chosen manufacturing technology.*

One of the key challenges of applications using PUFs is to reduce the error caused by physical process variations. If we think of applications where PUFs are integrated in the generation process of secret keys, as described in Chapter 5, it will be fatal if noisy signals make it impossible to reconstruct a certain secret key. A possible solution to dispose of the noise is the use of error correcting codes as described in Section 2.10.2. We will later discuss the fact that error correcting codes alone often do not suffice to ensure noise free signals, so other procedures like *Majority Voting* (cf. Section 5.2.2) come into play.

3.2.2 Performance

One of the key ideas of applications using PUFs is to capitalize on physical variations that occur randomly and are not predictable. This principle however, only works, if the present noise is within a certain boundary. The following parameters shed light on the randomness of PUF realizations and tell us how feasible certain instantiations are.

Inter Distance

For a particular challenge, the *inter-distance* between two different PUF instantiations is the distance between the two responses, resulting from applying this challenge simultaneously to both PUFs.

Intra Distance

For a particular challenge, the *inter-distance* between two evaluations on one single PUF instantiation is the distance between the two responses, resulting from applying this challenge twice to one PUF.

3.3 Controlled PUFs

A PUF is said to be **controlled** if it can only be accessed via an algorithm that is physically linked to the PUF in an inseperative way. In particular the control algorithm can restrict the challenges that are presented to the PUF, limit the information about responses that is given to the outside world, and implement some functionality for authentication algorithms [4].
This idea allows us in fact, to go beyond the simple key-card application, which offers a wide range of new possibilities.
The following algorithms may be implemented:

- A cryptographic hash function to generate the PUF challenges can prevent chosen-challenge attacks, i.e. make model building more difficult.

- An error correction code, acting on the PUF measurments makes the final responses much more reliable.
- A cryptographic hash function applied on the outputs of the error correcting algorithm effectively breaks the link between the responses and the physical details of the PUF measurement.

It is clear that turning a PUF into a cPUF greatly increases the security. This enhancement strongly depends on the physical linkage of the PUF with the access algorithm [24]. A possible schematic view of a cPUF is given in Figure 3.3 [6].

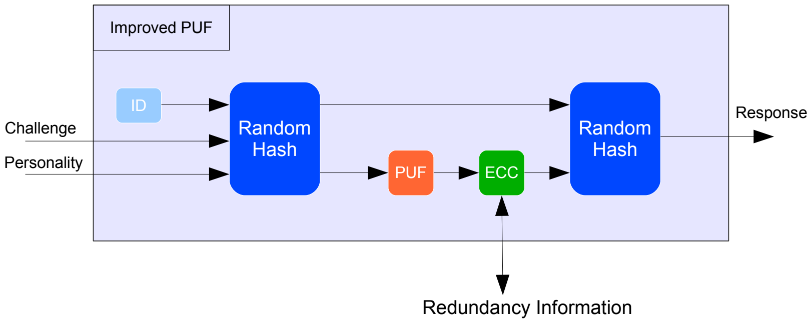


Figure 3.3: Controlled PUF

3.4 Reconfigurable PUFs

In this section a brief introduction to a very interesting type of PUFs is given, which might grow in importance in future. Initially this topic was introduced to the author at a UNIQUE workshop in Bochum in July 2010. Therefore, most of the presented data hereafter was taken from one workshop presentations [7].
Reconfigurable PUFs (rPUFs) offer an additional reconfiguration mechanism, which gives us the opportunity to transform one PUF into a new one, with a completely different unpredictable and uncontrollable challenge response behavior. Figure 3.4 schematically shows how this procedure works.

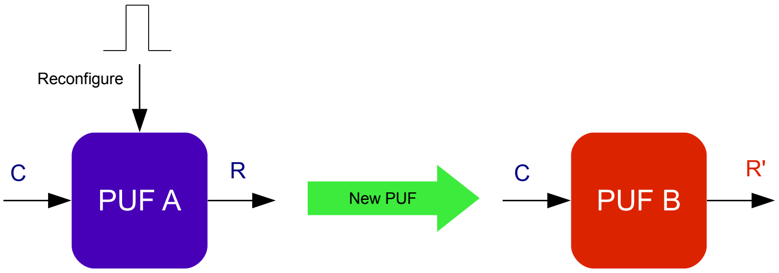


Figure 3.4: Reconfigurable PUF

On the left, we have a rPUF A with a certain CRP-behavior. After reconfiguring the PUF, we transform it to PUF B, which reacts completely different when stimulating the device with a challenge C.

3.4.1 Classification of Reconfigurability

DESCRIPTION	**IDEAL**	**PRACTICAL**
Reconfiguration can be done	**Unlimited**	**N-time**
Way of reconfiguration	**Physical**	**Logical**
Reconfiguration is	**One-way**	**Invertible**

Ideally we would like to have PUFs that can be reconfigured an unlimited amount of times and which then show completely new CRP-behavior. Furthermore physical reconfiguration is sometimes asked, since logical reconfiguration, as described in Section 4.4, offers attackers a bigger target. Very often if is desired that the reconfiguration is one-way, i.e. it is not possible to reset the PUF to previous states. Obviously, this ideal conception is not always possible and in practice there has to be a trade-off between all parameters.

3.4.2 Implementations

- Reconfigurable Optical PUF
- Phase Change Memory
- Reconfigurable Arbiter PUF
- Logical Reconfiguration

Detailed descriptions of the above mentioned implementations of rPUFs are given in Chapter 4.

3.4.3 Applications

- Key Zeroization
- Secure storage with renewable keys
- Prevention of software downgrades

Some of the above mentioned applications will be described at a later stage of this thesis.
In conclusion it needs to be pointed out, that research on rPUFs is still in its early stage and lots of concepts have not been practically verified yet. At this point special thanks go to Gert-Jan Schrijen and his colleagues from Intrinsic ID once again for their superb composition of information and their helpful support.

4 PUF Instantiations

The idea of using physical structures for cryptographic applications is not new at all. Several years ago the *American Lottery* was short before collapse due to counterfeited winning tickets that appeared every Sunday after the drawing. The IT experts solved the problem back then by inserting digital fingerprints on the single tickets for unique identification. This way the notary were able to distinct between the cloned and the original tickets.

Also in Austria companies used physical structures to advance against malicious attacks early on. In 1986 the *VOEST Alpine AG*, an international steel company based in Linz, invented an innovative security concept called *Soft-Seal*. The system protected IP against illegal cloning and counterfeiting and was based on unique physical structures of secure chip cards. One of the pioneer in the development of the system was Prof.(FH) Univ.-Doz. Dipl.-Ing. Dr. Ingrid Schaumüller-Bichl, who was the head of the security department of VOEST from 1984 to 1991.

In the recent past it was Pappu et al. who introduced the concept of Physical One-Way Functions (POWF) [14] in 2001. Tuyls et. al developed the idea of coating PUFs [22], and it was again Pappu et. al who came up with the idea of using optical structures to realize Physical One-Way Functions [15]. In 2002 Gassend et. al introduced silicon PUFs, where the source of randomness are small variations resulting from the manufacturing process of integrated circuits (IC) [5].

In the last years, lists of PUF instantiations grew large and it were Ahmad Sadeghi and David Naccache who published the book "Towards Hardware-Intrinsic Security", which contains a very good chapter about the state of the art of Physically Unclonable Functions [24]. This chapter was compiled by Ingrid Verbauwhede and Roel Maes and it represents probably the best overview of all PUF instantiations that can be found in literature until now. This book also served as the main source for this chapter. Additional references are [1, 10] and [27].

4.1 Non-Electrical Implementations

4.1.1 Optical PUFs

If there is a need to describe the idea of Physically Unclonable Functions to someone not familiar with the matter, the Optical PUF serves as the perfect demonstrative object, although it is unlikely to be used in real word applications, as will be described in detail at a later point.

The core element of the Optical PUF is a token which contains an optical micro-structure

constructed by mixing microscopic refractive glass spheres in a small transparent epoxy plate [24]. When radiating the token with a laser beam, a random speckle pattern arises which is recorded, quantized and encoded, to form the PUF response. The challenge might for example be the angle, focal distance or the wavelength of the laser beam. Experiments have shown, that even slight changes in the angle of the laser beam result in completely different speckle patterns and the respective extracted hash values [1, 27].
Figure 4.1 represents the basic implementation of the Optical PUF. The CRP, consisting of the laser orientation and the resulting hash, is saved in a public database for later use.

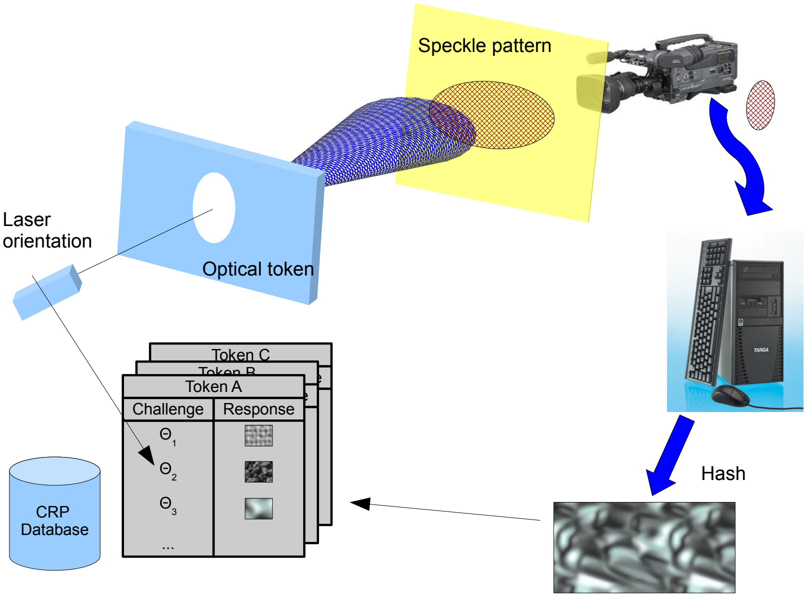


Figure 4.1: Optical PUF

A lot of experiments have been carried out with Optical PUFs to extract some general properties [14], [15]. After having stimulated four different tokens with over 500 distinct challenges, the following parameters unfolded.

Parameters $\boxed{\mu_{inter} = 49.79\%}$ $\boxed{\mu_{intra} = 25.25\%}$

It is clear that Optical PUFs, as described above, are not designed for large scale production. The massive overhead of tools like the laser beam and a mechanical positioning system make Optical PUFs very expensive, complex and therefore suitable rather for laborious use.
Optical PUFs have been implemented already as integrated chips (IC). A detailed description can be found in [4].

4.1.2 Acoustical PUFs

This special type of PUF has been included here to show that there are nearly no limits in designing physical systems, including the measurement of noisy random signals. Acoustical PUFs are based on acoustical delay lines - components to delay electrical signals. They convert an alternating electrical signal into a mechanical vibration and vice versa. Acoustical PUFs are constructed by observing the characteristic frequency spectrum of an acoustical delay line. To observe a bit string, a principle component analysis (PCA) has to be performed. A detailed explanation of Acoustical PUFs can be found in [25].

4.1.3 Coating PUFs

The idea behind a Coating PUF is to insert a protective coating, i.e. a covering that is applied to the surface on the top layer of the device. That means we do not solely rely on random effects of manufacturing variability, but purposely insert random elements into the device. This differentiates Coating PUFs from the group of **Intrinsic PUFs** (cf.Definition 3.2.1) as has been described in the previous chapter.

The opaque coating material is doped with dielectric particles which are sprayed directly on the top of the sensors on the top layer of the device. These elements have random properties in shape, size and location. This way Coating PUFs additionally offer strong protection against physical attacks and we may call them *tamper evident.*

The challenges are specified by voltage of a certain frequency and amplitude, applied to a region of the sensor array [1, 24]. The responses are represented by the measured capacitance as depicted in Figure 4.2.

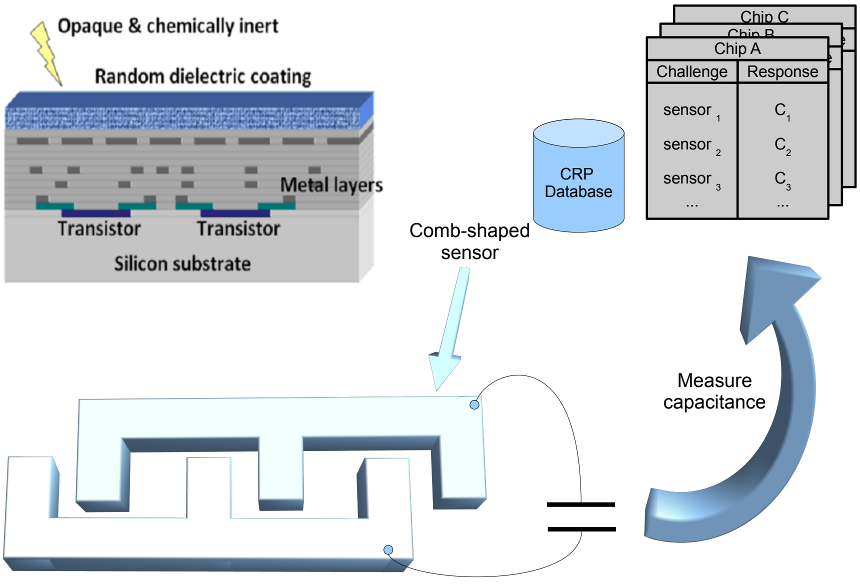


Figure 4.2: Coating PUF

Experiments on 36 chips with 31 sensors revealed the following parameters [28].

Parameters $\boxed{\mu_{inter} \approx 50\%}$ $\boxed{\mu_{intra} < 5\%}$

4.2 Delay-based Intrinsic PUFs

The previous PUF examples basically started with the analogue measurement of a random physical parameter which was processed, and represented an identifier of the underlying system. Now we want to focus on *Intrinsic PUFs* (cf. Definition 3.2.1), which offer the big advantage that no extra manufacturing steps or specialized components are required. We distinguish between two different classes - Intrinsic PUFs based on digital delay measurements, and Intrinsic PUFs based on settling memory elements.

4.2.1 Arbiter PUFs

The basic idea of Arbiter PUFs is to introduce a digital race condition on two paths on a chip and to implement an arbiter to decide which path won the race. Two pulses introduced simultaneously to the inputs of the paths will in general not arrive at the outputs simultaneously, due to small random delay variations [10].

The core elements of Arbiter PUFs are switch blocks which interconnect two input ports to the two output ports. Depending on a control bit, the paths are connected straight or crossed. By connecting a number of switch blocks in series, we create a parameterizable delay line. The setting of the switch blocks represents the challenge and the response is given by the output of the arbiter block.

Figure 4.3 represents the schematic view of the basic build up of an Arbiter PUF.

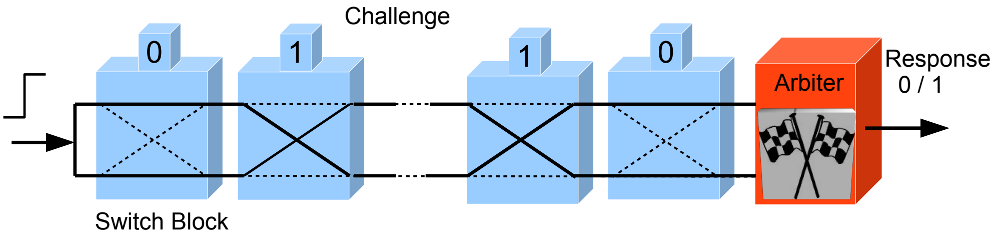


Figure 4.3: Arbiter PUF

The design was implemented on ASIC chaining 64 switch blocks. Experimental validation on 37 chips shows the following parameters.

Parameters $\boxed{\mu_{inter} = 23\%}$ $\boxed{\mu_{intra} < 5\%}$

The problem of the regular Arbiter PUFs is that digital delay is additive by nature. This means that the delay of the chain of switch blocks will be the sum of the delays of all separate blocks. This gives attackers the chance of performing so-called model-building-attacks, where mathematical models of the PUFs are formed out of a number of observed challenge-response queries.

A proper counteraction to prevent model-building-attacks is to introduce non-linearities. *Feed-forward PUFs* are a concrete attempt to introduce non-linearities in the delay lines of Arbiter PUFs. It is an extension to the regular PUF, where some challenge bits are not set by the user but are the outcomes of intermediate arbiters evaluating the race at some point in the delay lines. This increases the metastability and leads to the following parameters.

Parameters $\boxed{\mu_{inter} = 38\%}$ $\boxed{\mu_{intra} = 9.8\%}$

4.2.2 Ring Oscillator PUFs

Ring Oscillator PUFs also belong to the group of delay PUFs, although they use a different approach to measure small random delay variations. The output of a delay line is inverted and fed back to its input to create an oscillating loop, a so-called Ring Oscillator. The frequency of this oscillator is determined by the delay of the delay line, so measuring the frequency is equivalent to measuring the delay. The source of randomness are variations of the delay resulting from the manufacturing process, which go hand in hand with device-dependent variations of the measured frequency.

Figure 4.4 illustrates the general build up of Ring-Oscillator PUFs. They always include an edge detector, which detects the rising edges in the periodical oscillation, and a digital counter to count the number of edges over a fixed interval. Again, the parameterizable delay line represents the challenge of the PUF, whereas the response is given by the value of the counter.

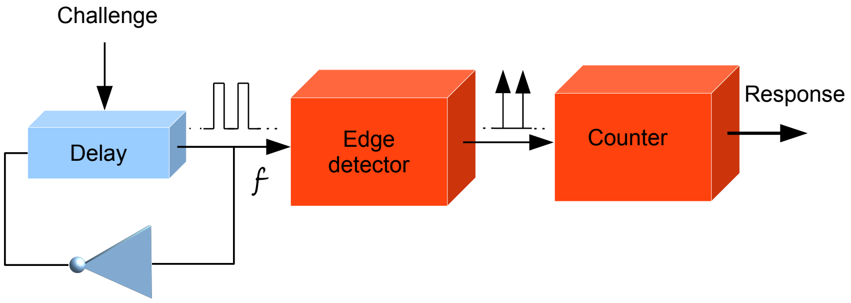


Figure 4.4: Ring-Oscillator PUF

As described in the previous chapter, environmental conditions always affect the measurements of PUF responses. Experiments have shown that in the case of regular Ring-Oscillator realizations, these effects are much greater and special compensation methods are required.

In [4] and [5] the idea of division compensation is described. Figure 4.5 depicts a RO-PUF in division compensation mode, where the counter values of two simultaneously measured oscillations are divided. Experiments on 4 FPGA devices obtained the following parameters.

Parameters $\boxed{\mu_{inter} \approx 10 \times 10^{-3}}$ $\boxed{\mu_{intra} \approx 0.1 \times 10^{-3}}$

In [20] another approach is discussed, where a very simple fixed delay circuit is used. The idea is to implement a number of Ring-Oscillators with the same intended frequency in parallel. Challenging the PUF in this case means nothing more than to select a pair of oscillators – the resulting response is produced by comparing the two obtained counter values as outlined in Figure 4.6.

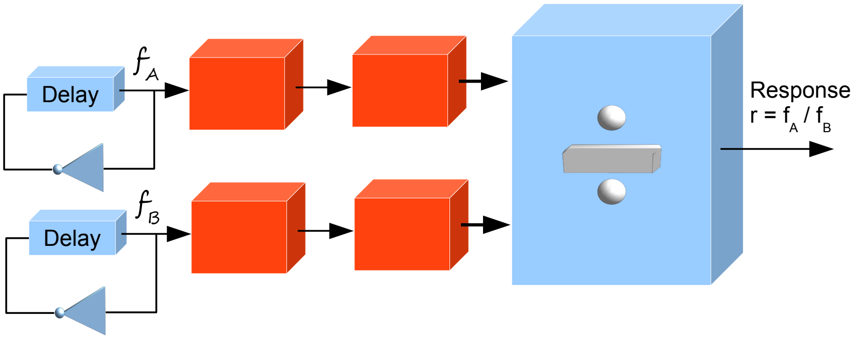


Figure 4.5: Ring-Oscillator PUF in division mode

This RO-PUF realization is easier to implement and evaluate, and offers a higher reliability. On the other hand, such PUF implementations are slower and consume more power [20]. Experiments on 15 FPGAs with 1024 loops per FPGA led to the following parameters [24].

Parameters $\boxed{\mu_{inter} = 46.15\%}$ $\boxed{\mu_{intra} = 0.48\%}$

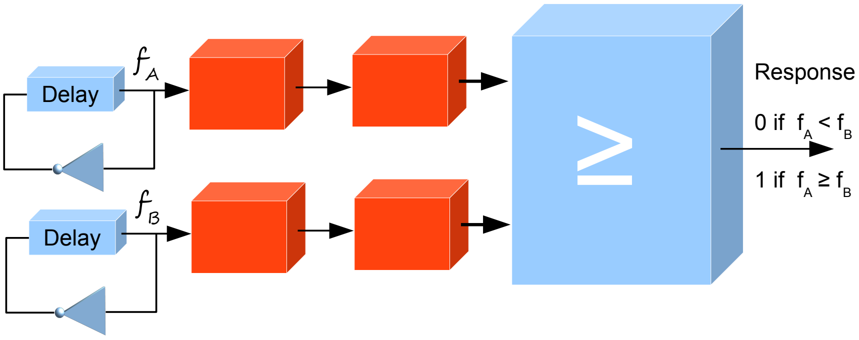


Figure 4.6: Ring-Oscillator PUF in comparator mode

4.3 Memory Based Intrinsic PUFs

The PUFs discussed in the following sections are based on the settling state of digital memory primitives. Memory cells are nothing more than digital circuits with two or more stable states. In the case of two possible states, such a cell stores one binary digit of information, by residing in one of its states. The idea of constructing PUFs is to bring such an element into an unstable state where it is not clear whether it will fall back to one of its stable states or if it will start oscillating between unstable states. Since physical variations during the manufacturing process influence the behavior of destabilized memory cells, the stable settling states of such elements are good candidates for PUF responses. In the following chapters we have a detailed look at concrete memory-based PUF realizations like SRAM cells, data latches or flip-flops.

4.3.1 SRAM PUFs

SRAM - static random access memory is a very common type of volatile storage on an integrated circuit. The actual storage element is realized by two cross-coupled inverters, as displayed in Figure 4.7. It is bistable, meaning it has two stable states and can therefore store one bit of information.

When the cell is powered on, it will quickly converge into one of both stable states – it has been experimentally verified that most cells have an explicit preference of one state over the other. However, which power-on state a cell prefers is random and not known a priori. The reason for this is that every cell contains a certain degree of mismatch between the two halves of the cross-coupled circuit, due to random deviations during manufacturing. This imbalance causes the cell to fall into one state more easily, but which of both states this is, is not known in advance. The power-up state of the SRAM memory will hence be random for a given device and can be used as a PUF response. A challenge is represented by a subset of the memory cells to be read out after power up [24, 10].

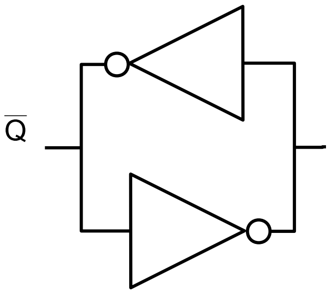


Figure 4.7: SRAM-Cell

After intensive experiments on 8190 bytes SRAM bytes from different memory blocks on different FPGAs, the following parameters have been calculated [10].

Parameters $\boxed{\mu_{inter} = 49.97\%}$ $\boxed{\mu_{intra} = 3.57\%}$

The big advantage of SRAM PUF realizations is that these PUFs produce a binary string as result of the measurement, without any quantization process or other data preparation, which makes them very suitable for use in practice. However, it turns out that in general the use of SRAM PUFs on the most common FPGAs is not possible, since a lot of SRAM cells are hard-reset to zero directly after power-up and hence, the randomness is lost. Another problem that occurs is that for response-generation, a device power-up is required which might not always be possible. To counter these drawbacks Butterfly-PUFs have been introduced [10].

4.3.2 Butterfly PUFs

As mentioned above, these types of PUFs had to be invented to emulate SRAM behavior on FPGAs where all SRAM is reset to power-on. The cross-coupled inverters are replaced by a so-called butterfly-cell, consisting of cross-coupled latches as illustrated in Figure 4.8. Latches are asynchronous memory elements that can be reset to '0' or preset to '1'. Because of the cross-coupling the cell is bistable again. By using the reset/preset functionality, an unstable state can be introduced after which the circuit converges back to one if its two stable states. In comparison to the SRAM-PUF, a power reset is not necessary. Again, the physical mismatch of the two latches determines the behavior of the cell [24, 10].

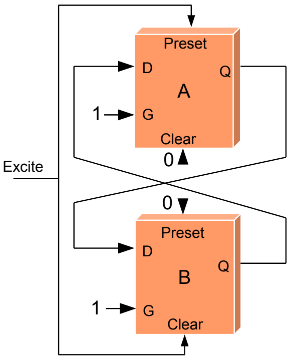


Figure 4.8: Butterfly-Cell

Measurements on 64 Butterfly PUF cells on 36 FPGAs have yielded the following parameters.

Parameters $\boxed{\mu_{inter} \approx 50\%}$ $\boxed{\mu_{intra} < 5\%}$

4.3.3 Latch PUFs

The concept of Latch PUFs is very similar to the SRAM and Butterfly realizations. Instead of cross-coupling two inverters or two latches, two NOR gates are cross-coupled, as shown in Figure 4.9. Again, the internal mismatch of the two symmetric components determines the stable state to which the latch converges after a reset.

Experiments on 128 NOR-latches implemented on 19 ASICs have yielded the following parameters [24, 19].

Parameters $\boxed{\mu_{inter} = 50.55\%}$ $\boxed{\mu_{intra} = 3.04\%}$

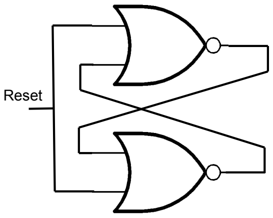


Figure 4.9: Latch-Cell

4.3.4 Flip-Flop PUFs

The last example of Memory-Based Intrinsic PUFs discussed in this thesis are Flip-Flop PUFs. Like in the case of SRAM PUFs the power-up behavior of regular flip-flops is under examination. Experiments on Flip-Flop-PUFs, which have been carried out in [10], produced the following parameters.

Parameters $\boxed{\mu_{inter} \approx 11\%}$ $\boxed{\mu_{intra} < 1\%}$

With very simple post-processing operations, which will be described in detail in Chapter 5, these characteristics improve to $\boxed{\mu_{inter} \approx 50\%}$ $\boxed{\mu_{intra} < 5\%}$

4.4 Special PUF Implementations

In Chapter 3 a short introduction to special types of PUFs like Controlled PUFs or Reconfigurable PUFs has been given. Hereafter, an example of a practical realization will be presented to make theoretical descriptions and definitions more seizable.

4.4.1 Reconfigurable Optical PUF

As described above in Section 4.1.1, the response of Optical PUFs is derived from directing a laser beam onto an epoxy plate and reading out the resulting speckle pattern. To reconfigure an Optical PUF, the polymer is heated up with an intensive laser beam, like illustrated in Figure 4.10.

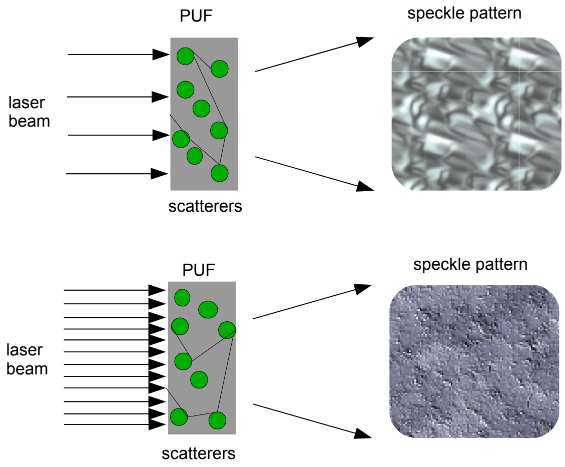


Figure 4.10: Optical rPUF

After reconfiguration, the position of the scatterers changed and so the resulting speckle pattern is completely different. The reconfiguration can be done unlimited of times and is physically one-way.

5 PUF Applications

In this chapter a selection of concrete PUF applications will be described. Since the technology is still in its infancy, the set of products based on PUF technology is still reasonable. However, with the promising research results and the urgent need of unclonable hardware security primitives, it seems likely that the applications described in the following sections are more than simple theoretical models.

5.1 Simple Key-card

This application is probably the simplest one and therefore a perfect starting point for describing the technology. The state-of-the-art solution of *Secure Key-Cards* are little smart cards, furnished with an **RFID** tag to enable contact-free communication. Since this system works well, RFID chips are implemented in various fields like:

- Transportation and Logistics
- Asset management and retail sales
- Product tracking
- Animal identification
- Health care
- Libraries
- Human identification
- etc.

A major drawback of this solution is the lack of hardware security. Keeping in mind the example of a simple key-card, which is used for human access control. The card is equipped with the RFID tag, which contains sensitive information that could be read out by an attacker. It would be conceivable to clone the cards – probably even without the notice of the card owner. This is the point where PUF technology comes into play. Utilizing Physically Unclonable Functions we are able to produce cards which do not need to store a single bit of information locally in the memory. The PUF itself acts as the key and ensures access. Figure 5.1 depicts a simple key-card implementation based on PUF technology.

A user, who is in possession of a PUF-based key-card wants to authenticate himself at a bank terminal. He provides the card to the terminal, which initiates communication with the PUF

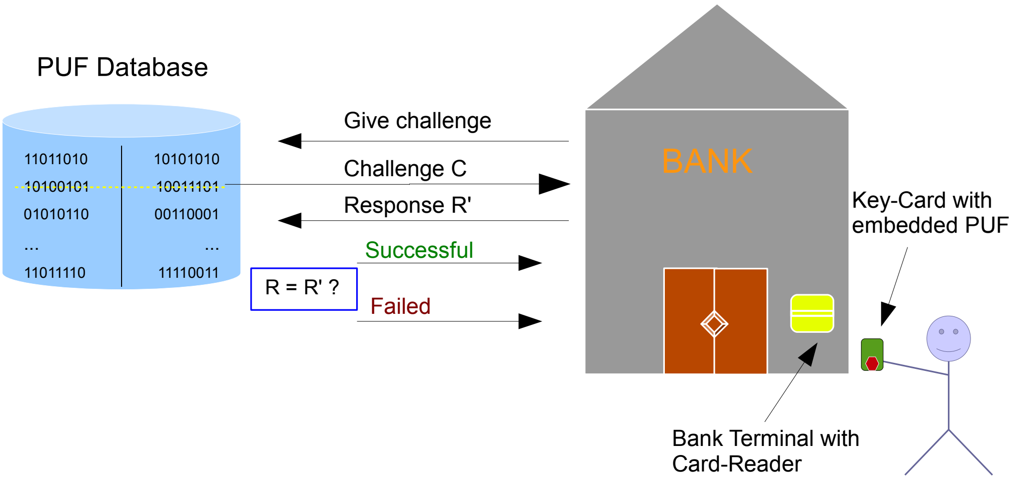


Figure 5.1: Secure Key-Card

database, which in turn contains CRPs that where generated during the manufacturing process. Based on the PUF ID, a certain challenge C is picked from the according PUF database and sent to the terminal. Now the PUF gets challenged and the derived response R' is sent back to the database where it is compared with the reference value R. If the occurred error is below a certain threshold, the authentication was successful and the user is allowed to enter the building. Otherwise, the authentication failed.

Note that at no point during this challenge-response protocol a key was present neither in plain text, nor enciphered. Nevertheless, an authentication protocol could be realized, based on the existing PUF and its CRP behavior.

5.2 Secret Key Generation

In our modern, dynamic world, bunches of sensitive data have to be protected against any kind of malicious attack. The current best practice is to use cryptographic primitives like encryption, digital signatures and authentication codes, which rely on the protection of secret keys. These keys are very often stored in non-volatile memory such as fuses and EEPROMs, which suffer from a couple of shortcomings, as described in Chapter 1.

To counteract these drawbacks, the question of alternative approaches arises. Using PUFs to counter the problem might not sound realistic at first, since any kind of noisy data does not fit with the idea of exact secret keys. We know that slight changes in a key of an AES algorithm result in a completely different output. Moreover, some primitives, such as RSA encryption schemes, require keys to satisfy specific mathematical properties, whereas the PUF outputs are randomly determined by manufacturing variations.

In this section the author will describe how PUFs can be used to generate reliable secret keys.

5.2.1 Idea

As has been discussed in previous chapters, storing keys in non-volatile memory offers attackers the chance to read out sensitive data. When using PUFs to generate secret keys, no explicit key-programming step is required.

In general, such a key generation process has to pass through two separate phases. During the *initialization phase* the PUF is queried and the algorithm produces a secret key, together with some additional information, called *helper data*. This additional information might for example be an error correcting syndrome (cf. Definition 2.10.3), which is used to correct certain bit flips. It is saved in a database, which is only accessible by the verifier.

In the *re-production phase*, the secret key is re-generated. First, the PUF is queried "in the field" i.e. a certain PUF output is measured. Now the beforehand saved helper data is used to re-produce the exact secret key. In practice, this might for example mean nothing more than to execute an error correcting decoding algorithm like the BCH code, as described in Section 2.10.6. The output of the error correction block can now simply be hashed down to the desired length and used as input to a symmetric encryption algorithm such as AES. For cryptographic operations where the keys need to satisfy special properties (e.g. RSA), the hashed PUF output might be used as seed for a key generation algorithm [20]. Figure 5.2 depicts a schematic view of the key generation phases.

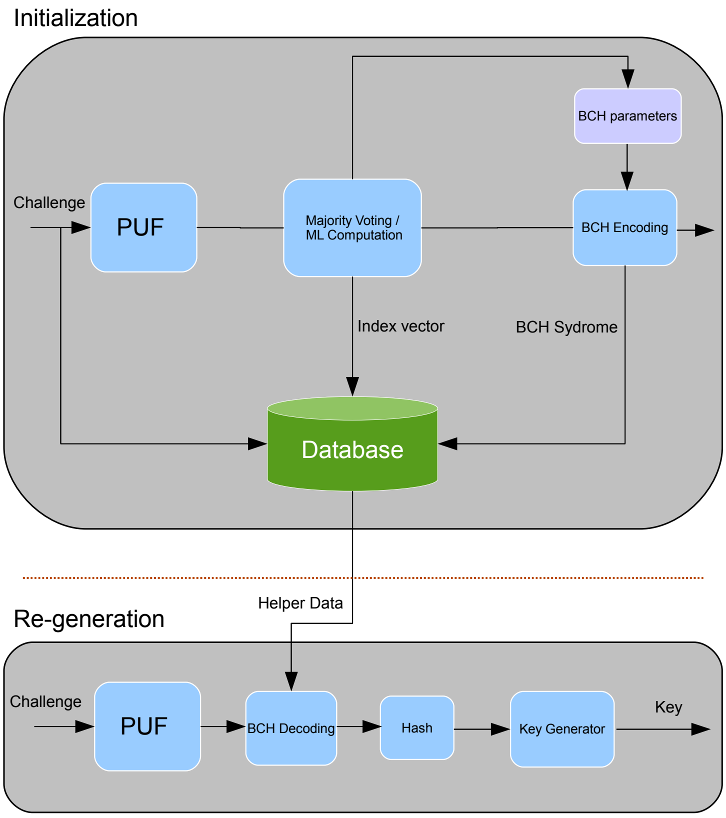


Figure 5.2: Initialization and Reproduction Phase

As illustrated in Figure 5.2 , the key generation process starts with picking a certain challenge, which is applied to the PUF several times. With that, an output matrix is generated, which represents the PUF responses for one specific challenge. This procedure is carried out to detect weak bits, i.e. digits that tend to flip more likely than others. The derived positions in the binary vector are stored in an index vector. Furthermore, a Maximum Likelihood computation is implemented to compute the most probable output. This process is called *Majority Voting.* A detailed description is provided in subsequent paragraphs.

From these compiled outputs some parameters, like the bit length and the error correction ability for the succeeding BCH encoding, are derived. The computed BCH syndrome is stored in a database, together with the chosen challenge and an index vector indicating the so-called weak bits.

The re-generation phase starts with extracting the helper data from the database. The selected challenge is applied to the PUF, but this time only once. With the help of the index vector, the digits previously tagged as weak are deleted. Now the BCH decoding can be employed which generates the reference response of the PUF which has not been stored at any time during the proceedings before.

To increase the security even more, the computed values might be hashed down to the desired length in a further step. It is also possible to use the calculated random string as seed for a key generator, but these additional steps are more a matter of personal taste.

The essential feature of this setup is the error correction processing, which will be described in more detail in the following section.

5.2.2 Majority Voting

As mentioned above, this process is used to decrease the number of errors before an output is fed into the BCH block. With this, the efficiency of the decoder is increased, since it is not necessary to add too much redundancy to be able to correct a certain amount of errors.

As depicted in Figure 5.3, Majority Voting takes place, after challenging the PUF several times with a certain input. Columns where nearly all bit values are equal, represented by column three in Figure 5.3 (all entries are '1'), are considered as stable or strong digits and therefore the index vector is '1'. Columns like column one in the figure, where '0s' and '1s' take turns every now and then, are tagged as weak, i.e. the index vector is '0'. This way, we derive a vector that indicates strong and weak digits, without providing any information about the actual output value. This way, security is increased dramatically, since an attacker is not able to read out any kind of sensitive data.

In this proceeding the *maximum likelihood* values of the single digits are computed implicitly, since there has to be a reference output value to be able to compare the single entries. Majority Voting can be realized with a minimum of computational effort, since every step includes only simple mathematical operations performed on a finite set of elements.

The entire key generation process, including majority voting and error correction, has been simulated in MATLAB and will be presented in the following chapter (6).

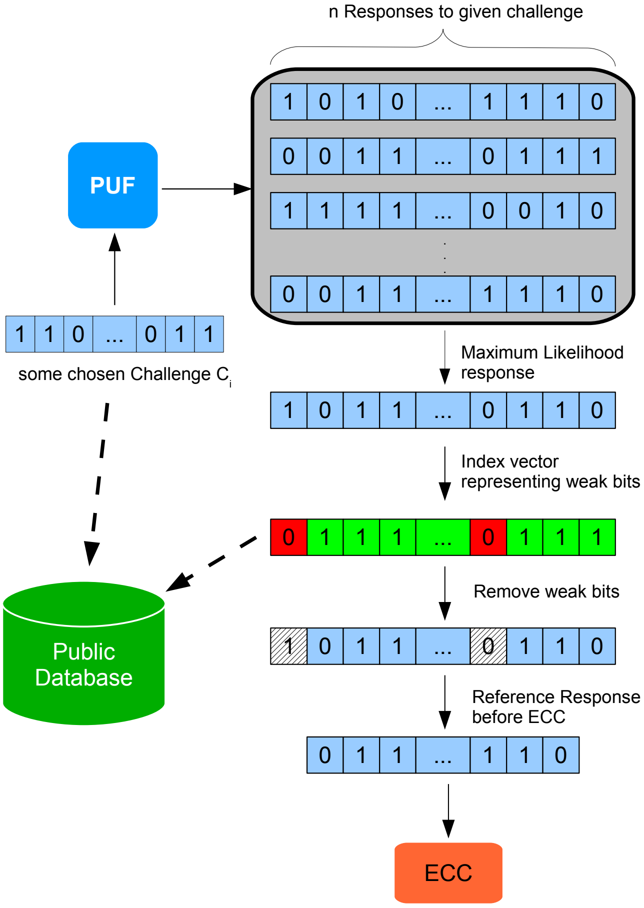


Figure 5.3: Creating index vector

5.3 Key Zeroization

Key Zeroization describes the process of properly deleting certain keys in a system. This might happen by overwriting the entire exiting key value with either zeros or a random bit string. There are various applications where such a key deletion is necessary, for example Pay-TV.

Figure 5.4 illustrates a cryptographic module which is used to administer a set of keys. A user interacts with the the module via an interface, which is connected to a crypto processor. An additional key zeroization mechanism is implemented, to be able to zeroize certain sets of keys. This is exactly the point, where PUFs, more precisely reconfigurable PUFs, come into play.
Reconfigurable PUFs, as they have been described in Section 4.4.1, offer properties which fit perfectly into the idea of key deletion. With an additional reconfiguration mechanism, the special types of PUFs make it possible to transform one PUF into another, which then shows completely new CRP behavior.

Assume the keys we manage were generated with the help of a PUF A, as described in Section 5.2. When we apply the reconfiguration mechanism, we turn the PUF A into PUF B possessing

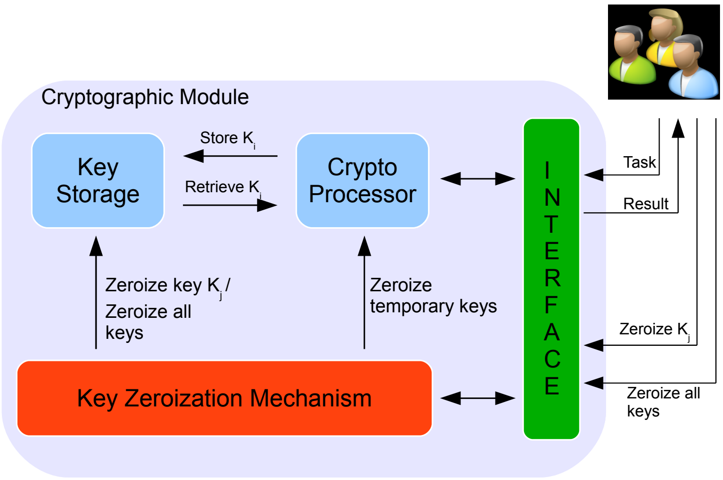


Figure 5.4: Key Zeroization

completely new CRP behavior. In doing so all keys associated with PUF A are corrupt, i.e. they are zeroized. Potential implementations of such a system could make use of logical reconfigurable PUFs (cf. Section 3.4.1). Logical reconfiguration is realized utilizing a normal PUF in combination with non-volatile memory.

With the help of a control unit the original challenge C is turned into a modified challenge C', which is applied to the PUF. The compiled response R' is again transformed into a new response R. This way the CRP behavior has been logically changed.
This PUF use-case is still in a very early stage and there are hardly any practical implementations so far. Within the UNIQUE project a lot has been done in this area, so it is expected that further publications will be released soon.

5.4 Concluding Remarks

Apart from the above described applications, there are various other examples where PUF technology might by used in the future.

- Hardware-Software Binding
- Activation of Hardware Chips
- Remarking of Chips and Devices
- etc.

As already mentioned in the previous sections still a lot has to be done with regards to the practical realization of PUF-based systems. However, there are already concrete implementations available on the market, like *Qiddikey* from Intrinsic ID [8].
To provide the reader with a full picture of the previously described applications, some MATLAB simulations will be provided in Chapter 6 hereafter.

6 MATLAB Simulations

6.1 Introduction

In the previous chapters a lot of information was provided about several types of PUFs, their properties and possible applications. To convey a clearer picture of how such systems work, the author simulated two PUF applications with MATLAB. The reason why MATLAB has been used is easy explained. First of all, the author has some experience with this software, since it offers algorithms for a wide range of mathematical applications. Moreover, it has an integrated BCH module, which makes it very suitable for the purpose of this thesis.

In the following chapters an explanation of the single program parts along with some simulation results is given. The whole program code can be found as appendix attached at the end of this thesis (MATLAB Code A).

6.2 Application 1 - Secure Key-Card

6.2.1 Basic Description

The secure key-card is probably the simplest PUF application but at the same time displays some major advantages of PUF technology.

In the MATLAB simulation, presented hereafter the basic authentication model is illustrated. As described in Section 5.1 such a process has to pass through two phases. The enrollment phase, during which the PUF is challenged several times in a secure environment and the measured CRPs are stored in a private database. In the re-production phase a user wants to authenticate himself at a terminal, which extracts a certain challenge and presents it to the PUF. The calculated response is then compared with the one from the database and if the occurred error is small enough, the authentication was successful.

The MATLAB simulation as well is divided into two program parts, representing the two authentication phases. To simulate the challenge of a PUF, a simple mathematical inversion of the presented challenge is calculated. In reality, this step is of course random and not calculable at all.

At the beginning of the first program part, a table of possible PUF instantiations is given. The user is asked to choose a concrete implementation, like *Optical* or *Coating*, just to show the wide range of possible PUF instantiations, and to integrate particular parameters like the intra-distance of the single realizations. If a user for example chooses an Optical PUF, the calculated response is much noisier since the intra-distance is quite high (see Section 4.1.1). To identify each PUF at a later stage, a unique PUF ID is created. In the next step, the associated response is calculated, afflicted with some noisy bits and the CRP is stored in a private database together with the corresponding ID.

In the second program part, the simulation of the re-production phase, the challenge that corresponds to the PUF ID is loaded from the database and the response is calculated again. To simulate the noisy environment, an error vector is added to the output. Afterwards, the difference between the response in the database and the calculated PUF output is determined. If the number of errors is below a certain threshold, the authentication was successful, otherwise the authentication failed. For security reasons the chosen CRP is deleted after the authentication process.

6.2.2 Program Structure

1. *Enrollment Phase*
   - Choice of the desired PUF
   - Generation of a PUF ID
   - Generation of a random challenge
   - *"Calculation"* of the response
   - Storage of the CRP in a private database
2. *Re-production Phase*
   - Declaration of the PUF type
   - Fetching a random challenge from the PUF database
   - *"Calculation"* of the response for the given challenge
   - Comparing the received response with the reference response from the PUF database
   - Decision whether or not authentication was successful

6.2.3 Simulation Results

To demonstrate the main results, some simulation outputs are presented hereafter.

Enrollment

The following output (6.1) is received during the enrollment phase, when a CRP for a Coating PUF should be stored.

```
>> keycard_enrol
############################################################
Please insert the number of the PUF you would like to use

########################
Nr PUF Type intra[%]
1  Optical  25.25
2  Coating  5
```

```
3  RO       0.48
4  Arbiter  5
5  Latch    3.04
6  Butterfly 5
7  SRAM     3.57
8  Flip-Flop 5
2

########################

Chosen PUF type is Coating

The intra-distance is given as follows: mu_intra=5%

ID of the PUF: C794178104

The CRP was securely stored in the PUF database.

#######################################################
```

Src. 6.1: Enrollment Phase - Secure Key card

Storage

In addition to storing the variables in a database, the calculated values are saved in a text document, which looks as follows.

```
----------------------------------------------------------------------
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
79  56  49  51  53  55  53  52  52  48  32   0   0   0   1   1   1   1
0   1   1   1   0   1   1   1   1   0   0    0   1   0   0   1   1  32
1   1   1   0   0   0   0   1   0   0   0    1   0   0   0   0   1   ...
79  48  54  52  48  51  50  57  54  56  32   1   1   0   1   0   1   1
0   0   0   1   1   0   0   0   1   0   1    1   1   0   0   0   0  32
0   0   1   0   1   0   0   1   1   1   0    0   1   1   1   0   0   ...

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
67  57  49  49  53  57  48  56  48  53  32   0   1   0   1   0   1   1
1   0   0   0   0   0   1   1   0   0   0    1   1   0   1   1   0  32
1   0   1   0   1   0   0   0   1   1   1    1   1   0   0   1   1   ...

RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
82  54  54  49  48  50  49  48  56  49  32   0   1   1   0   0   0   1
1   0   1   0   1   0   1   1   1   1   1    0   1   0   1   1   1  32
1   0   0   1   1   1   0   0   1   0   1    0   1   0   0   0   1   ...
82  50  48  48  52  56  56  48  55  56  32   1   1   1   1   0   1   0
0   1   1   0   0   0   0   1   0   0   0    1   1   1   1   0   1  32
0   0   0   0   1   0   1   1   0   0   1    1   1   1   0   1   1   ...
----------------------------------------------------------------------
```

Src. 6.2: PUF Database

The PUF database always starts with the letter indicating the PUF type, e.g. **O** for **O**ptical or **C** for **C**oating. The entries start with the ID stored as ASCII code, followed by the separator ′32′ which is nothing more than the ASCII code for *blank*. The next 24 bits indicate the challenge, again completed by a blank. The remaining bits are the reference responses, which are not displayed as a whole for obvious reasons. The output displayed above is just a fraction of the entire database, but indicates quite well how the data is stored.

Re-production

Now a user is trying to authenticate himself at a terminal. After presenting the PUF including the ID, the device gets challenged and the calculated response is compared to the one in the database.

```
>> keycard_field
#########################################################
Please insert your ID: A037541578

Chosen PUF type is: Arbiter

The intra-distance is given as follows: mu_intra= 5%

A challenge is fetched from the PUF database

8 Bits flipped

Authentication successful.
#########################################################
```

Src. 6.3: Succesful Authentication

The output above (6.3) shows a situation when a user successfully authenticates himself at a terminal. In this case 8 of 128 bits flipped, which is still beneath the threshold.
The following example displays a situation, when an authentication fails.

```
>> keycard_field
#########################################################
Please insert your ID: 0121621074

Chosen PUF type is: Optical

The intra-distance is given as follows: mu_intra= 2.520000e+001%

A challenge is fetched from the PUF database

86 Bits flipped

Access denied, error rate too high
#########################################################
```

Src. 6.4: Authentication Failed

Since the intra-distance of the Optical PUF is rather high in comparison to other PUF implementations, it is quite likely that the outcome of an authentication process will look like the output shown above (6.4).

6.3 Application 2 - Secret Key Generator

6.3.1 Basic Description

For a detailed description the author again refers to Section 5.2, where a lot of graphical representations and explanations about PUF-based key generators have been provided.

Like the Secure Key-Card, this application passes through an enrollment and a re-production phase, but now the result is a secret key. This time, the enrollment process starts with the definition of an ID, an identification number which is uniquely assigned to each PUF. Similarly to the application above, the PUF gets challenged in the next step, but not only once for a given challenge, but several times. This corresponds to proceedings in practice, where a chip is challenged several times, under different conditions, with the same challenge. This gives us the chance to detect so called *weak bits*, i.e. bits that tend to flip more likely. In the simulation we achieve this behavior by adding different error vectors during the single response calculations. In the next step we investigate the positions, where bit-flips occurred more frequently, and create an index vector that marks those bits.

After deleting the weak bits and shorten the binary output to a narrow sense BCH code length, we are able to perform a BCH encoding. The outcome consists of an information and a syndrome part, which is nothing more than overhead that is essential to re-produce the original message. Together with the ID, the challenge and the index vector, the syndrome is stored in a key database. The author wants to point out that the measured response is not stored anywhere.

In the re-production phase the user has to insert his ID, which is checked first, before the associated helper vector, including challenge, index and syndrome, is loaded. With the help of the challenge, a response is calculated “in the field”, again impinged with some errors. Now the index vector is used to delete the weak bits, which have been specified in the run-up. In the next step, we shorten the output again to a narrow sense BCH code length, and perform the decoding. We extract the information part of the result, which is nothing less than the response and are now in the possession of a secret key.

At this point, some post processing steps like a cryptographic hash function could be performed, but for this simple simulation we take the result of the BCH decoder already as our secret key.

To demonstrate the correctness of the implementation the author additionally created two test functions, *encode* and *decode*, which perform a simple **VERNAM** cipher (cf. 2.4). A possible way to test the extracted key is the following:

(i) Produce a helper vector for a specific device

(ii) Fetch the helper data and extract the secret key for this device

(iii) Encrypt local data with the key

(iv) Repeat (ii) to extract the key again

(v) Decrypt the local data

If the decryption restores the plain data, the correct key has been extracted.

6.3.2 Program Structure

1. *Enrollment Phase*
   - Choice of the desired PUF
   - Generation of random challenge
   - *"Calculation"* of a response matrix
   - Calculation of an index vector representing the weak bits
   - Deleting the weak bits
   - Shortening of the output to a narrow sense bit-length
   - Performing the BCH encoding
   - Storing the helper data in a key database
2. *Re-production Phase*
   - Inserting the PUF ID
   - Checking if the ID is valid
   - Fetching the corresponding helper vector from the key database
   - Extraction of challenge, index vector and syndrome
   - *"Calculation"* of the response for the given challenge
   - Insertion of noisy bits
   - Deleting the weak bits according to the index vector
   - Shortening the output to a narrow sense bit length
   - Performing a BCH decoding with the help of the calculated field-response and the gathered syndrome
   - Encryption of some plain text with the extracted key
   - Decryption of the cipher

6.3.3 Simulation Results

Enrollment

The following output (6.5) has been generated during the enrollment phase, when a PUF gets challenged and the helper data is stored in the key database.

```
>> keygen_enrol
########################################################

ID of the PUF that is challenged: S523886727

Chosen PUF type is SRAM

```

```
The intra-distance is given as follows: mu_intra=3.570000e+000%

With the given choices, we expect there to be around 10 noisy Bits

The following BCH Code parameters are given: m=9 n=511 k=241

The helper data was securely stored in the key database.

#######################################################
```

Src. 6.5: Enrollment Phase - Secret Key Generator

Since an SRAM PUF gets measured, the intra distance is 3.57%. From this it we know that we have to account for around 10 noisy bits altogether.

Key Extraction

Now we want to extract a key from our PUF, with the aid of the previously stored helper data.

```
>> keygen_field
#######################################################
Please insert your ID: S523886727

Chosen PUF type is: SRAM

The intra-distance is given as follows: mu_intra= 3.570000e+000

With the given choices, we expect there to be around 10 noisy Bits

Please specify the desired key length (<241): 200

key length = 200

The helper data is fetched from the PUF database

Now the PUF gets challenged in the field

During the measurement in the Field 14 errors occurred.
#######################################################
```

Src. 6.6: Key Extraction - SRAM PUF

Now we have extracted the key and are able to encrypt our local hard disk.

```
>> encrypt
#######################################################
Please insert message you would like to encrypt:

 My entire hard disk!

Cipher:

1     1     1     0     1     0    0     0     1 ...
#######################################################
```

Src. 6.7: Encryption SRAM PUF

In this example the author encrypted the message *My entire hard disk!* and this results in a binary cipher without specific meaning. After some time the plain data is needed again – in this case the author wants to know the secret message. So the helper data is fetched again to extract the personal PUF-based secret key.

```
>> decrypt
#######################################################
My entire hard disk!
#######################################################
```

Src. 6.8: Decryption SRAM PUF

The result is, as we see above (6.8) *My entire hard disk!*, which is reassuring, but the next MATLAB output shows us that under certain circumstances, this decoding might become difficult.

```
>> keygen_field
#######################################################
Please insert your ID: 0639335329

Chosen PUF type is: Optical

The intra-distance is given as follows: mu_intra= 2.520000e+001%

With the given choices, we expect there to be around 60 noisy Bits

Please specify the desired key length (<241) 200

key length = 200

The helper data is fetched from the PUF database

Now the PUF gets challenged in the field

During the measurement in the Field 56 errors occurred.
#######################################################
```

Src. 6.9: Key Extraction - Optical PUF

In this case an Optical PUF is used to extract a secret key, but as we see, 56 errors occurred which is quite a lot. Let us have a look at what happens, if we encrypt some data.

```
>> encrypt
#######################################################
Please insert message you would like to encrypt:

>> encrypt
Please insert message you would like to encrypt:

Optical PUFs are noisy!

Cipher:

     0     1     1     0     1     1     1     0   ...

#######################################################
```

Src. 6.10: Encryption Optical PUF

In this example the author encrypted the message *Optical PUFs are noisy!*, which again results in a binary cipher with no specific meaning.
After some time we want to recover the plain text and so the helper data is again used to extract the secret key.

```
>> decrypt
#######################################################
 biQ=,#J2Td9)g7_.A.F#'7>
#######################################################
```

Src. 6.11: Decryption

As we can see above (6.11), the result is nothing more than special characters stringed together in a random way - the plain text is lost.
This case has been added to show what happens, if not enough attention has been paid to the error correcting parameters.

6.4 Concluding Remarks

First of all it has to be pointed out, that MATLAB appears to be very suitable to implement error correcting codes, but for the majority of the program parts it seems a bit bold. This may of course also result from personal constituencies since the author's MATLAB skills can be regarded as rather basic. Furthermore, the author just wanted to show the basic functionality and it would have taken too much time to create a complete gap less program code. Nevertheless, this simple simulation demonstrates the big advantages of PUF based schemes, like the reduced need of storing secret keys. It was furthermore interesting to play with the error correcting capability of the BCH code, and see the resulting outcome. With the insertion of several PUF types and with their corresponding error rates, the simulations became even more practical and more realistic. Especially the Secret Key Generator is not too far from a practical case. Using BCH codes to dispose of errors is a widely used technique, and as results show, it is not trivial at all to find the adequate code. All in all, this simulations helped a lot to get a more practical view on this topic and as such will also be helpful to the reader.

7 Conclusion

Cryptography is for sure one of the fastest moving areas of mathematics. If for example an encryption algorithm survives longer than ten years before it is broken, we call it already an old hand in the business. Lots of promising concepts did not push through, although solutions were innovative and unique. Many factors have to collude, to put a good idea into practice.
When the author first came into touch with Physically Unclonable Functions at a meeting in Bochum, an immediate feeling arose that this concept is auspicious. On the one hand, the economic aspect is covered, since PUFs offer low-cost assembly. Moreover, there is an urgent demand which needs to be satisfied, since the IT industry suffers from brand damage and the accompanied financial loss already since years. Research in this special area has only been conducted since less than one decade, so we may call the technology still a very young one. Society has not yet fully taken account of it. The author believes that 99% of the population could not guess the meaning of the mysterious abbreviation **P U F**, although first products have already been launched.
However, the PUF technology will develop, and it is an unbelievable experience for the author, to be involved in the evolution of this great idea. The author is convinced that it might very well be that in future times we will use key-cards containing PUFs, or secret keys produced by a PUF-based generator.

A MATLAB Code

A.1 Secure Key-Card

In this section, all developed program parts of the secure key-card are provided.

A.1.1 Enrollment Phase Secure Key-Card

```
package at.ac.uniklu.simulator.commands.iso7816;

% This programm shows the enrollment phase of a simple authentication
% process. After picking a certain PUF the generated challenge-response
% pair CPR is stored in a private database.

% First of all we may choose a certain type of PUF. Possible choices are
% - Optical PUF
% - Coating PUF
% - RO PUF
% - Latch PUF
% - Arbiter PUF
% - Butterfly PUF
% - Flip-Flop PUF
% Define the length of the challenge
clear;
load ('PUF_database.dat','-mat')

lc=24;
% Define PUF output length
bit_length=128;
% Initialize the variables
global Optical;
global Coating;
global RO;
global Latch;
global Butterfly;
global SRAM;
global FF;
global Arbiter;
global k;

% Coating=Coating(1:153)
% Arbiter=Arbiter(1:153)
% SRAM=SRAM(1:153)
% Butterfly=Butterfly(1:153)
% FF=FF(1:153)
% Latch=Latch(1:153)
% RO=RO(1:153)
% Optical=Optical(1:153)

%A lot of research has been done of the different PUF instantiations and
%the following table of parameters has been developed:

          % Optical Coating RO  Latch   Arbiter Butterfly SRAM  FF
% mu_inter   49.79%   50%   50%  50.55%   23%     50%   49.97%  50%
% mu_intra    25.25%  5%   0.48% 3.04%    5%      5%    3.75%   5%
sprintf('Please insert the number of the PUF you would like to use')
inter_array=['Nr ' 'PUF Type ' 'intra ' ; '1 ' 'Optical ' '25.25 ' ; '2 ' 'Coating ' '5   ' ; '3 ' 'RO
           ' '0.48 ' ; '4 ' 'Arbiter ' '5    ' ; '5 ' 'Latch    ' '3.04 ' ; '6 ' 'Butterfly ' '5   ' ;
     '7 ' 'SRAM     ' '3.57 ' ; '8 ' 'Flip-Flop ' '5   '];
disp(inter_array)
puf_type=input(char);
if ((puf_type > 0) && (puf_type < 9))
        sprintf('Chosen PUF type is %s',inter_array(puf_type+1,4:13))
        mu_intra=str2num(inter_array(puf_type+1,14:17));
        sprintf('The intra-distance is given as follows: mu_intra=%d',mu_intra)
else
    sprintf('Wrong choice')
    break;
end

PUF=inter_array(puf_type+1,4:13);
R1=PUF(1);
R2=randsample('0123456789',9,true);
%ID='0397812';
ID=[R1,R2];
sprintf('ID of the PUF: %s ',ID)


% Now we know the error rates of the choosen PUF and the desired Bitlength.
% To be sure to reach this number of Bits, we have to calculate an overhead
% of PUF-Output Bits, since we know that some of them tend to flip.

% exp_error_nb=ceil(mu_intra*(bit_length/100));
% sprintf('With the given choices, we expect there to be around %d noisy Bits',exp_error_nb)

% Compute the random challenge
 for i=1:lc
        challenge(i)=randint(1,1);
 end

% Now the PUF gets challenged and we get a certain response for our given
% challenge. In practice this process is stochastic and not computable. For
% our simmulation we are using a simple mathematical invertation.
j=1;
for i=(1:bit_length)
    response(i)=1-challenge(j);
    j=j+1;
    if(j==length(challenge)+1)
        j=1;
    end
end


% The calculated CRP is noe stored in a database.
% This is done in the function storage
```

```
storage_new(puf_type,char(ID),challenge,response)

save ('PUF_database.dat', 'Optical', 'Coating', 'RO', 'FF', 'Butterfly', 'SRAM', 'Latch', 'Arbiter', 'k',
      'inter_array','lc','bit_length','ID');
```

Src. A.1: Enrollment Phase of Secure Key-Card

A.1.2 Re-Production Phase Secure Key-Card

```
package at.ac.uniklu.simulator.commands.iso7816;

% This programm simulates the use of a simple keycard in reality. After
% providing the type of PUF, the terminal fetches a challenge from the PUF
% database and challeges the device. The giveb response is compared with
% the one in the database. If it matches close enaugh, then the
% authetication was succefull if not, then the access is denied.
load ('PUF_database.dat','-mat')
% Firstly the user shows his card to the terminal.
global challenge_field;

ID=input('Please insert your ID: ','s');
%If the length is incorrect we may immediately abort.
if(length(ID)~=10)
    sprintf('ID not valid!')
    break;
end
puf_type=find_puf_type(ID);
% Acording the inserted ID, we distinguish which type of PUF we have.

sprintf('Chosen PUF type is: %s',inter_array(puf_type+1,4:13))
mu_intra=str2num(inter_array(puf_type+1,14:17));
sprintf('The intra-distance is given as follows: mu_intra= %d',mu_intra)

pause(2);
sprintf('A challenge is fetched from the PUF database')
pause(2);
% For this action the subroutine get_challenge is performed
get_challenge_new(puf_type,ID,lc);

% If the database is nearly empty, the programm forces the manufacturer to
% start the enrollment phase
if k<=2
    sprintf('Warning, database empty for PUF Type %s',PUF)
    sprintf('################################################################')
    sprintf('Please start enrollment phase')
    keycard_enrol
    sprintf('################################################################')
end

j=1;

% Now the PUF is chalenged with the challenge fetched from the database.
% Again I make a simple mathematical calculation and add up some errors
% later. In reality this is a micro-physical procedure, that is not
% computable.
for i=(1:bit_length)
    response(i)=1-challenge_field(j);
    j=j+1;
    if(j==length(challenge_field)+1)
        j=1;
    end
end
error=0;
% According the given PUF parameter, I calculate the expected number of
% errors. Afterwards I add som Bit flips.
mu_intra=str2num(inter_array(puf_type+1,14:17));
exp_errors=ceil(length(response)*(mu_intra/100));
response_field=response+randerr(1,length(response),1:(3*exp_errors));
for(i=1:length(response_field))
    if(response_field(i) ~= response(i))
        error=error+1;
    end
end
pause(2);
sprintf(’%d Bits flipped’,error)
if((error/length(response_field)*100)>10)
    sprintf(’Access denied, error rate too high’)
    break;
else
    sprintf(’Authentication successful.’)
end
```

Src. A.2: Re-Production Phase of Secure Key-Card

A.1.3 Subroutines Secure Key-Card

Storing CRP

```
package at.ac.uniklu.simulator.commands.iso7816;

function [] = storage(p_type,a,b,c)
global Optical;
global Coating;
global RO;
global Latch;
global Butterfly;
global SRAM;
global FF;
global Arbiter;
switch p_type
    case 1
        [k,n]=size(Optical);
        Optical(k+1,1:length(a))=a;
        Optical(k+1,length(a)+1)=’ ’;
        Optical(k+1,length(a)+2:length(a)+length(b)+1)=b;
        Optical(k+1,length(a)+length(b)+2)=’ ’;
        Optical(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;

   case 2
        [k,n]=size(Coating);
        Coating(k+1,1:length(a))=a;
        Coating(k+1,length(a)+1)=’ ’;
        Coating(k+1,length(a)+2:length(a)+length(b)+1)=b;
        Coating(k+1,length(a)+length(b)+2)=’ ’;
        Coating(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;
   case 3
```

```
       [k,n]=size(RO);
       RO(k+1,1:length(a))=a;
       RO(k+1,length(a)+1)=’ ’;
       RO(k+1,length(a)+2:length(a)+length(b)+1)=b;
       RO(k+1,length(a)+length(b)+2)=’ ’;
       RO(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;
  case 4
       [k,n]=size(Arbiter);
       Arbiter(k+1,1:length(a))=a;
       Arbiter(k+1,length(a)+1)=’ ’;
       Arbiter(k+1,length(a)+2:length(a)+length(b)+1)=b;
       Arbiter(k+1,length(a)+length(b)+2)=’ ’;
       Arbiter(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;
  case 5
       [k,n]=size(Latch);
       Latch(k+1,1:length(a))=a;
       Latch(k+1,length(a)+1)=’ ’;
       Latch(k+1,length(a)+2:length(a)+length(b)+1)=b;
       Latch(k+1,length(a)+length(b)+2)=’ ’;
       Latch(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;
  case 6
       [k,n]=size(Butterfly);
       Butterfly(k+1,1:length(a))=a;
       Butterfly(k+1,length(a)+1)=’ ’;
       Butterfly(k+1,length(a)+2:length(a)+length(b)+1)=b;
       Butterfly(k+1,length(a)+length(b)+2)=’ ’;
       Butterfly(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;
  case 7
       [k,n]=size(SRAM);
       SRAM(k+1,1:length(a))=a;
       SRAM(k+1,length(a)+1)=’ ’;
       SRAM(k+1,length(a)+2:length(a)+length(b)+1)=b;
       SRAM(k+1,length(a)+length(b)+2)=’ ’;
       SRAM(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;
  case 8
       [k,n]=size(FF);
       FF(k+1,1:length(a))=a;
       FF(k+1,length(a)+1)=’ ’;
       FF(k+1,length(a)+2:length(a)+length(b)+1)=b;
       FF(k+1,length(a)+length(b)+2)=’ ’;
       FF(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;
    otherwise
       sprintf(’Error’)
       return;
end
sprintf(’The CRP was securely stored in the PUF database.’)
sprintf(’##########################################################’)

d=length(num2str(Optical));
 %for i=1:(4*n-3)
 for i=1:d
     sepa(i)=’-’;
     O(i)=’O’;
     C(i)=’C’;
     A(i)=’A’;
     R(i)=’R’;
     S(i)=’S’;
     L(i)=’L’;
     F(i)='F';
     B(i)='B';
 end
 datab=[sepa;O;num2str(Optical);C;num2str(Coating);R;num2str(RO);A;num2str(Arbiter);L;num2str(Latch);B;
      num2str(Butterfly);S;num2str(SRAM);F;num2str(FF);sepa];

 save ('PUF_database.dat', 'Optical', 'Coating', 'RO', 'FF', 'Butterfly', 'SRAM', 'Latch', 'Arbiter', 'k'
      );

[m,n]=size(datab);

fid = fopen('PUF_database.txt', 'wt'); % Open for writing
for i=1:m
    fprintf(fid, '%s \n', datab(i,:));
end
fclose(fid);
```

Src. A.3: Storing CRP

Resolving the PUF

```
package at.ac.uniklu.simulator.commands.iso7816;

function [type]=find_puf_type(ID)
global Optical;
global Coating;
global RO;
global Latch;
global Butterfly;
global SRAM;
global FF;
global Arbiter;

switch ID(1)
    case 'O'
        type=1;
    case 'C'
        type=2;
    case 'R'
        type=3;
    case 'A'
        type=4;
    case 'L'
        type=5;
    case 'B'
        type=6;
    case 'S'
        type=7;
    case 'F'
        type=8;
    otherwise
        sprintf('Wrong ID')
        return;
end
```

Src. A.4: Resolving PUF with ID

Fetching the Challenge

```
package at.ac.uniklu.simulator.commands.iso7816;

function [] = get_challenge(p_type,ID,lc)
global Optical;
global Coating;
global RO;
global Latch;
global Butterfly;
global SRAM;
global FF;
global Arbiter;
global k;
global challenge_field;
i=1;
switch p_type
    case 1
        k=length(Optical(:,1));
        while i<=k
            if (strcmp(char(ID),char(Optical(i,1:10)))==0)
                i=i+1;
                challenge_num=0;
            else
                challenge_num=i;
                challenge_field=Optical(challenge_num,1:lc);
                Optical(challenge_num,:)=[];
                return;
            end

        end
        if challenge_num==0
            sprintf('No ID matches');
            return;
        end

   case 2
        k=length(Coating(:,1));
        while i<=k
            if (strcmp(char(ID),char(Coating(i,1:10)))==0)
                i=i+1;
                challenge_num=0;
            else
                challenge_num=i;
                challenge_field=Coating(challenge_num,1:lc);
                Coating(challenge_num,:)=[];
                return;
            end

        end
        if challenge_num==0
            sprintf('No ID matches');
            return;
        end
   case 3
        k=length(RO(:,1));
        while i<=k
            if (strcmp(char(ID),char(RO(i,1:10)))==0)
                i=i+1;
```

```
               challenge_num=0;
           else
               challenge_num=i;
               challenge_field=RO(challenge_num,1:lc);
               RO(challenge_num,:)=[];
               return;
           end

       end
       if challenge_num==0
           sprintf('No ID matches');
           return;
       end
  case 4
       k=length(Arbiter(:,1));
       while i<=k
           if (strcmp(char(ID),char(Arbiter(i,1:10)))==0)
               i=i+1;
               challenge_num=0;
           else
               challenge_num=i;
               challenge_field=Arbiter(challenge_num,1:lc);
               Arbiter(challenge_num,:)=[];
               return;
           end

       end
       if challenge_num==0
           sprintf('No ID matches');
           return;
       end
  case 5
       k=length(Latch(:,1));
       while i<=k
           if (strcmp(char(ID),char(Latch(i,1:10)))==0)
               i=i+1;
               challenge_num=0;
           else
               challenge_num=i;
               challenge_field=Latch(challenge_num,1:lc);
               Latch(challenge_num,:)=[];
               return;
           end

       end
       if challenge_num==0
           sprintf('No ID matches');
           return;
       end
  case 6
       k=length(Butterfly(:,1));
       while i<=k
           if (strcmp(char(ID),char(Butterfly(i,1:10)))==0)
               i=i+1;
               challenge_num=0;
           else
               challenge_num=i;
               challenge_field=Butterfly(challenge_num,1:lc);
               Butterfly(challenge_num,:)=[];
               return;
           end

       end
       if challenge_num==0
           sprintf('No ID matches');
           return;
       end
  case 7
       k=length(SRAM(:,1));
       while i<=k
           if (strcmp(char(ID),char(SRAM(i,1:10)))==0)
               i=i+1;
               challenge_num=0;
           else
               challenge_num=i;
               challenge_field=SRAM(challenge_num,1:lc);
               SRAM(challenge_num,:)=[];
               return;
           end

       end
       if challenge_num==0
           sprintf('No ID matches');
           return;
       end
  case 8
       k=length(FF(:,1));
       while i<=k
           if (strcmp(char(ID),char(FF(i,1:10)))==0)
               i=i+1;
               challenge_num=0;
           else
               challenge_num=i;
               challenge_field=FF(challenge_num,1:lc);
               FF(challenge_num,:)=[];
               return;
           end

       end
       if challenge_num==0
           sprintf('No ID matches');
           return;
       end
    otherwise
       sprintf('Error')
       return;
end

d=length(num2str(Optical));

 for i=1:d
     sepa(i)='-';
     O(i)='O';
     C(i)='C';
     A(i)='A';
     R(i)='R';
     S(i)='S';
     L(i)='L';
     F(i)='F';
     B(i)='B';
 end
 datab=[sepa;O;num2str(Optical);C;num2str(Coating);R;num2str(RO);A;num2str(Arbiter);L;num2str(Latch);B;
      num2str(Butterfly);S;num2str(SRAM);F;num2str(FF);sepa];

 save ('PUF_database.dat', 'Optical', 'Coating', 'RO', 'FF', 'Butterfly', 'SRAM', 'Latch', 'Arbiter', 'k'
      );

[m,n]=size(datab);

fid = fopen('PUF_database.txt', 'wt'); % Open for writing
for i=1:m
    fprintf(fid, '%s \n', datab(i,:));
end
fclose(fid);
```

Src. A.5: Fetching the Challenge

A.2 Key Generator

In this section, all developed program parts for the secure key generator are provided.

A.2.1 Enrollment Phase for Key Generator

```
package at.ac.uniklu.simulator.commands.iso7816;
% - Optical PUF
% - Coating PUF
% - RO PUF
% - Latch PUF
% - Arbiter PUF
% - Butterfly PUF
% - Flip-Flop PUF

%A lot of research has been done of the different PUF instantiations and
%the following table of parameters has been developed:

          % Optical Coating RO   Latch  Arbiter Butterfly SRAM  FF
% mu_inter  49.79%   50%    50%   50.55%  23%     50%    49.97%  50%
% mu_intra   25.25%  5%    0.48% 3.04%   5%      5%     3.75%   5%
% Define length of the challenge
clear;
load ('key_database.dat','-mat')

lc=24;
% Define desired Bit-Length. If a user wished to have a shorter key, the
% unused Bits are simply cutted off from the end.
output_length=280;
% Initialize the variables
global Optical;
global Coating;
global RO;
global Latch;
global Butterfly;
global SRAM;
global FF;
global Arbiter;
global k;

R1=randsample('COASFLBR',1);
R2=randsample('0123456789',9,true);
%ID='0397812';
ID=[R1,R2];
sprintf('ID of the PUF that is challenged: %s ',ID)
puf_type=find_puf_type(ID);

%sprintf('Please insert the number of the PUF you would like to use')
inter_array=['Nr ' 'PUF Type ' 'intra ' ; '1 ' 'Optical ' '10.25 ' ; '2 ' 'Coating ' '5  ' ; '3 ' 'RO
           ' '0.48 ' ; '4 ' 'Arbiter ' '5    ' ; '5 ' 'Latch    ' '3.04 ' ; '6 ' 'Butterfly ' '5   ' ;
     '7 ' 'SRAM     ' '3.57 ' ; '8 ' 'Flip-Flop ' '5   '];
%disp(inter_array)
%puf_type=input(char);
if ((puf_type > 0) && (puf_type < 9))
       sprintf('Chosen PUF type is %s',inter_array(puf_type+1,4:13))
       mu_intra=str2num(inter_array(puf_type+1,14:17));
       sprintf('The intra-distance is given as follows: mu_intra=%d',mu_intra)
else
    sprintf('Wrong choice')
    break;
end

% Compute the random challenge
 for i=1:lc
       challenge(i)=randint(1,1);
 end

% Now we know the error rates of the choosen PUF and the desired Bitlength.
% In the next step we calculate the expected number of Bit-Flips according
% the given parameters.

exp_error_nb=ceil(mu_intra*(output_length/100));
sprintf('With the given choices, we expect there to be around %d noisy Bits',exp_error_nb)

% Now the PUF gets challenged and we get a certain response for our given
% challenge. In practice this process is stochastic and not computable. For
% our simmulation we are using a simple mathematical invertation.

% In practice it bacame common, to challenge the same PUF not only once
% with a single challenge, but to make a set of measurments under different conditions.
% To simulate this situation, we simply generate a matrix consisting of a
% certain number of responses for the same challenge, where some noise is
% added up.

repeat=7;
error=randerr(1,output_length,1:(3*exp_error_nb));

j=1;
for i=(1:output_length)
    puf_output(i)=1-challenge(j);
    j=j+1;
    if(j==length(challenge)+1)
       j=1;
    end
end

for(i=1:repeat)
    output_matrix(i,:)=mod(puf_output(1,:)+randerr(1,output_length,1:((i)*exp_error_nb)),2);
end

s=sum(output_matrix);
maxnoise=2;

% If cartain bits tend to flip very likely, we simply cut them of. This is
% done by implementing an index vactor, which marks strong Bits with '1'
% and weak once with '0'

for i=1:output_length
    if ((s(i)) >= repeat-maxnoise)
        ml_output(i)=1;
        index(i)=1;
    elseif ((s(i)) <= maxnoise)
        index(i)=1;
        ml_output(i)=0;
    else
        index(i)=0;
        ml_output(i)=2; % mark the weak outputs
    end
end

% In the next step we cut off the weak Bits.
i=1;
p=1;
while i<=output_length
    if index(i)==0
        i=i+1;
    else
        trimmed_output(p)=ml_output(i);
        p=p+1;
        i=i+1;
    end
end

i=1;
j=1;

% We need a narrow sense BCH codelength. To make it easier, I set the used
% BCH code. The (511 241 36) BCH code will be used, i.e. the message-length
% is 241 and so 36 errors can be corrected


if (length(trimmed_output)>=241)
    for(i=1:241)
        response(i)=trimmed_output(i);
    end
else
    sprintf('Too many errors. Output not usable')
    break;
    %If too many errors happened during enrollment we abord.
end
```

```
% Finally we found a narrow sense BCH code length
m=9;
n=2^9-1;
k=length(response);
sprintf('The following BCH Code parameters are given: m=%d n=%d k=%d ' ,m,n,k)


% Now the BCH encoding can be performed.
msg = gf(response);
[genpoly,t] = bchgenpoly(n,k);

code= bchenc(msg,n,k);
%disp('Codeword:');
information=code(1,1:k);
syndrome=code(1,k+1:length(code));
syndrome_converted=syndrome.x;

% The calculated data is now stored in a key database.
key_storage(puf_type,ID,challenge,index,syndrome_converted);
save ('key_database.dat', 'Optical', 'Coating', 'RO', 'FF', 'Butterfly', 'SRAM', 'Latch', 'Arbiter', 'k',
     'lc', 'output_length','n')
```

Src. A.6: Enrollment Phase for Key Generator

A.2.2 Re-Production Phase of Key Generator

```
package at.ac.uniklu.simulator.commands.iso7816;

clear
% First of all we load the stored key database
load ('key_database.dat','-mat')
inter_array=['Nr ' 'PUF Type ' 'intra ' ; '1 ' 'Optical ' '10.25 ' ; '2 ' 'Coating ' '5  ' ; '3 ' 'RO
           ' '0.48 ' ; '4 ' 'Arbiter ' '5    ' ; '5 ' 'Latch    ' '3.04 ' ; '6 ' 'Butterfly ' '5   ' ;
     '7 ' 'SRAM     ' '3.57 ' ; '8 ' 'Flip-Flop ' '5   '];
% Then the user inserts his PUF ID.

ID=input('Please insert your ID: ','s');
%If the length is incorrect we may immediately abort.
if(length(ID)~=10)
    sprintf('ID not valid!')
    break;
end
puf_type=find_puf_type(ID);
% Acording the inserted ID, we distinguish which type of PUF we have.

sprintf('Chosen PUF type is: %s',inter_array(puf_type+1,4:13))
mu_intra=str2num(inter_array(puf_type+1,14:17));
sprintf('The intra-distance is given as follows: mu_intra= %d',mu_intra)

pause(2);
exp_error_nb=ceil(mu_intra*(output_length/100));
sprintf('With the given choices, we expect there to be around %d noisy Bits',exp_error_nb)
PUF=inter_array(puf_type+1,4:13);
keylength=str2num(input('Please specify the desired key length (<241)','s'))
if keylength>240
    sprintf('No possible!. Keylength=240');
    keylength=240;
end
```

```
pause(2);
sprintf('The helper data is fetched from the PUF database')
pause(2);
% Now the compatible helper data, that was stored during an enrollment phase is
% fetched.
helper=get_helper(puf_type,ID);


j=1;
l_ID=length(ID);
challenge=helper(12:lc+11);
index=helper(lc+2+11:lc+2+10+output_length);
syndrome_double=helper(lc+2+11+output_length+1:end);
syndrome=gf(syndrome_double);
pause(2);

% With the extracted challenge we are now able to challenge the PUF 'in the
% field'.
sprintf('Now the PUF gets challenged in the field')
j=1;
for i=(1:output_length)
    response(i)=1-challenge(j);
    j=j+1;
    if(j==length(challenge)+1)
        j=1;
    end
end


error=0;

% According the given PUF parameter, I calculate the expected number of
% errors. Afterwards I add som Bit flips so simulate the noisy area that we
% have to deal with in reality.
response_field=mod(response+randerr(1,length(response),1:(5*exp_error_nb)),2);
for(i=1:length(response_field))
    if(response_field(i) ~= response(i))
        error=error+1;
    end
end
pause(2);

sprintf('During the measurement in the Field %d errors occured.',error)


% We use the index vector to delete bits which were previously
% marked as weak bits.
j=1;
i=1;
while (i<=length(response_field))
    if (index(i)==1)
        trimmed_response(j)=response_field(i);
        i=i+1;
        j=j+1;
    else
        i=i+1;
    end
end
```

```
if length(trimmed_response)<241
    sprintf(’Too many errors occurred during the measurement in the field. No key is extractable’)
    break;
else
    for i=1:241
        final_response(i)=trimmed_response(i);
    end
end

% Now we are able to the BCH decryption to reproduce the key i.e the
% response. I again want to point out, that during the whole procedure at
% not time the response was stored.

msg=gf(final_response);
field_code=[msg syndrome];

[newmsg,err] = bchdec(field_code,n,length(final_response));

newmsg;
h=newmsg.x;
key=h(1:keylength);

save (’private_key.dat’, ’key’)
```

Src. A.7: Re-Production Phase of Key Generator

A.2.3 Subroutines for Key Generator

Key Storage

```
package at.ac.uniklu.simulator.commands.iso7816;

function [] = key_storage(p_type,a,b,c,d)
global Optical;
global Coating;
global RO;
global Latch;
global Butterfly;
global SRAM;
global FF;
global Arbiter;


switch p_type
    case 1
        [k,n]=size(Optical);
        Optical(k+1,1:length(a))=a;
        Optical(k+1,length(a)+1)=’ ’;
        Optical(k+1,length(a)+2:length(a)+length(b)+1)=b;
        Optical(k+1,length(a)+length(b)+2)=’ ’;
        Optical(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;
        Optical(k+1,length(a)+length(b)+length(c)+3)=’ ’;
        Optical(k+1,length(a)+length(b)+length(c)+4:length(a)+length(b)+length(c)+length(d)+3)=d;

   case 2
```

```
      [k,n]=size(Coating);
      Coating(k+1,1:length(a))=a;
      Coating(k+1,length(a)+1)=' ';
      Coating(k+1,length(a)+2:length(a)+length(b)+1)=b;
      Coating(k+1,length(a)+length(b)+2)=' ';
      Coating(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;
      Coating(k+1,length(a)+length(b)+length(c)+3)=' ';
      Coating(k+1,length(a)+length(b)+length(c)+4:length(a)+length(b)+length(c)+length(d)+3)=d;

   case 3
      [k,n]=size(RO);
      RO(k+1,1:length(a))=a;
      RO(k+1,length(a)+1)=' ';
      RO(k+1,length(a)+2:length(a)+length(b)+1)=b;
      RO(k+1,length(a)+length(b)+2)=' ';
      RO(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;
      RO(k+1,length(a)+length(b)+length(c)+3)=' ';
      RO(k+1,length(a)+length(b)+length(c)+4:length(a)+length(b)+length(c)+length(d)+3)=d;
  case 4
      [k,n]=size(Arbiter);
      Arbiter(k+1,1:length(a))=a;
      Arbiter(k+1,length(a)+1)=' ';
      Arbiter(k+1,length(a)+2:length(a)+length(b)+1)=b;
      Arbiter(k+1,length(a)+length(b)+2)=' ';
      Arbiter(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;
      Arbiter(k+1,length(a)+length(b)+length(c)+3)=' ';
      Arbiter(k+1,length(a)+length(b)+length(c)+4:length(a)+length(b)+length(c)+length(d)+3)=d;
  case 5
      [k,n]=size(Latch);
      Latch(k+1,1:length(a))=a;
      Latch(k+1,length(a)+1)=' ';
      Latch(k+1,length(a)+2:length(a)+length(b)+1)=b;
      Latch(k+1,length(a)+length(b)+2)=' ';
      Latch(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;
      Latch(k+1,length(a)+length(b)+length(c)+3)=' ';
      Latch(k+1,length(a)+length(b)+length(c)+4:length(a)+length(b)+length(c)+length(d)+3)=d;
  case 6
      [k,n]=size(Butterfly);
      Butterfly(k+1,1:length(a))=a;
      Butterfly(k+1,length(a)+1)=' ';
      Butterfly(k+1,length(a)+2:length(a)+length(b)+1)=b;
      Butterfly(k+1,length(a)+length(b)+2)=' ';
      Butterfly(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;
      Butterfly(k+1,length(a)+length(b)+length(c)+3)=' ';
      Butterfly(k+1,length(a)+length(b)+length(c)+4:length(a)+length(b)+length(c)+length(d)+3)=d;
  case 7
      [k,n]=size(SRAM);
      SRAM(k+1,1:length(a))=a;
      SRAM(k+1,length(a)+1)=' ';
      SRAM(k+1,length(a)+2:length(a)+length(b)+1)=b;
      SRAM(k+1,length(a)+length(b)+2)=' ';
      SRAM(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;
      SRAM(k+1,length(a)+length(b)+length(c)+3)=' ';
      SRAM(k+1,length(a)+length(b)+length(c)+4:length(a)+length(b)+length(c)+length(d)+3)=d;
   case 8
      [k,n]=size(FF);
      FF(k+1,1:length(a))=a;
      FF(k+1,length(a)+1)=' ';
```

```
        FF(k+1,length(a)+2:length(a)+length(b)+1)=b;
        FF(k+1,length(a)+length(b)+2)=' ';
        FF(k+1,length(a)+length(b)+3:length(a)+length(b)+length(c)+2)=c;
        FF(k+1,length(a)+length(b)+length(c)+3)=' ';
        FF(k+1,length(a)+length(b)+length(c)+4:length(a)+length(b)+length(c)+length(d)+3)=d;
    otherwise
        sprintf('Error')
        return;
end
sprintf('The key was securely stored in the key database.')
sprintf('################################################################')


d=length(num2str(Optical));
for i=1:d
     sepa(i)='-';
     O(i)='O';
     C(i)='C';
     A(i)='A';
     R(i)='R';
     S(i)='S';
     L(i)='L';
     F(i)='F';
     B(i)='B';
 end
 Chal(1:length(a))='C';
 Index(1:length(b))='I';
 Synd(1:length(c))='S';
 header=[Chal,' ',Index,' ',Synd];
 %keys=[sepa;O;num2str(Optical);C;num2str(Coating);R;num2str(RO);A;num2str(Arbiter);L;num2str(Latch);B;
      num2str(Butterfly);S;num2str(SRAM);F;num2str(FF);sepa];



%[m,n]=size(keys);

 %fid = fopen('key_database.txt', 'wt'); % Open for writing
 %for i=1:m
 %    fprintf(fid, '%s \n', keys(i,:));
 %end
 %fclose(fid);
```

Src. A.8: Key Storage

Fetching Helper Data

```
package at.ac.uniklu.simulator.commands.iso7816;


function [helper] = get_helper(p_type,ID)
global Optical;
global Coating;
global RO;
global Latch;
global Butterfly;
global SRAM;
global FF;
global Arbiter;
global k;
global challenge_field;
i=1;
switch p_type
    case 1
        k=length(Optical(:,1));
        while i<=k
            if (strcmp(char(ID),char(Optical(i,1:10)))==0)
                i=i+1;
                helper_num=0;
            else
                helper_num=i;
                helper=Optical(helper_num,1:end);
                return;
            end

        end
        if helper_num==0
            sprintf('No ID matches');
            return;
        end

   case 2
        k=length(Coating(:,1));
        while i<=k
            if (strcmp(char(ID),char(Coating(i,1:10)))==0)
                i=i+1;
                helper_num=0;
            else
                helper_num=i;
                helper=Coating(helper_num,1:end);
                return;
            end

        end
        if helper_num==0
            sprintf('No ID matches');
            return;
        end
   case 3
        k=length(RO(:,1));
        while i<=k
            if (strcmp(char(ID),char(RO(i,1:10)))==0)
                i=i+1;
                helper_num=0;
            else
                helper_num=i;
                helper=RO(helper_num,1:end);
                return;
            end

        end
        if helper_num==0
            sprintf('No ID matches');
            return;
        end
  case 4
        k=length(Arbiter(:,1));
```

```
         while i<=k
             if (strcmp(char(ID),char(Arbiter(i,1:10)))==0)
                 i=i+1;
                 helper_num=0;
             else
                 helper_num=i;
                 helper=Arbiter(helper_num,1:end);
                 return;
             end

         end
         if helper_num==0
             sprintf('No ID matches');
             return;
         end
    case 5
         k=length(Latch(:,1));
         while i<=k
             if (strcmp(char(ID),char(Latch(i,1:10)))==0)
                 i=i+1;
                 helper_num=0;
             else
                 helper_num=i;
                 helper=Latch(helper_num,1:end);
                 return;
             end

         end
         if helper_num==0
             sprintf('No ID matches');
             return;
         end
    case 6
         k=length(Butterfly(:,1));
         while i<=k
             if (strcmp(char(ID),char(Butterfly(i,1:10)))==0)
                 i=i+1;
                 helper_num=0;
             else
                 helper_num=i;
                 helper=Butterfly(helper_num,1:end);
                 return;
             end

         end
         if helper_num==0
             sprintf('No ID matches');
             return;
         end
    case 7
         k=length(SRAM(:,1));
         while i<=k
             if (strcmp(char(ID),char(SRAM(i,1:10)))==0)
                 i=i+1;
                 helper_num=0;
             else
                 helper_num=i;
                 helper=SRAM(helper_num,1:end);
```

```
                return;
            end

        end
        if helper_num==0
            sprintf('No ID matches');
            return;
        end
  case 8
        k=length(FF(:,1));
        while i<=k
            if (strcmp(char(ID),char(FF(i,1:10)))==0)
                i=i+1;
                helper_num=0;
            else
                helper_num=i;
                helper=FF(helper_num,1:end);
                return;
            end

        end
        if helper_num==0
            sprintf('No ID matches');
            return;
        end
    otherwise
        sprintf('Error')
        return;
end

d=length(num2str(Coating));
for i=1:d
     sepa(i)='-';
     O(i)='O';
     C(i)='C';
     A(i)='A';
     R(i)='R';
     S(i)='S';
     L(i)='L';
     F(i)='F';
     B(i)='B';
 end
 Chal(1:length(a))='C';
 Index(1:length(b))='I';
 Synd(1:length(c))='S';
 header=[Chal,' ',Index,' ',Synd];
 keys=[sepa;O;num2str(Optical);C;num2str(Coating);R;num2str(RO);A;num2str(Arbiter);L;num2str(Latch);B;
      num2str(Butterfly);S;num2str(SRAM);F;num2str(FF);sepa];
 [m,n]=size(keys);
  fid = fopen('key_database.txt', 'wt'); % Open for writing
 for i=1:m
     fprintf(fid, '%s \n', keys(i,:));
 end
 fclose(fid);
```

Src. A.9: Fetching Helper Data

VERNAM Encryption

```
package at.ac.uniklu.simulator.commands.iso7816;


clear
load (’private_key.dat’,’-mat’)
message=input(’Please insert message you would like to encrypt: \n ’,’s’)
message_bin=dec2bin(message);
[n,k]=size(message_bin);
message_vek=message_bin(:)’;
trimmed_key=key(1:length(message_vek));
for i=1:length(trimmed_key)
    cipher(i)=mod(double(trimmed_key(i))+double(message_vek(i)),2);
end
sprintf(’Encrypted message’);
disp(cipher);
save (’cipher.dat’, ’cipher’,’n’,’k’)
```

Src. A.10: VERNAM Encryption

VERNAM Decryption

```
package at.ac.uniklu.simulator.commands.iso7816;


clear
load (’private_key.dat’,’-mat’)
load (’cipher.dat’,’-mat’)
trimmed_key=key(1:length(cipher));
for i=1:length(trimmed_key)
    message_field(i)=mod(double(trimmed_key(i))+double(cipher(i)),2);
end
message1=reshape(message_field,n,k);
message2=num2str(message1);
message3=bin2dec(message2);

sprintf(’Decrypted message’);
disp(char(message3)’);
```

Src. A.11: VERNAM Decryption

List of Figures

2.1 Basic Communication Scenario 4

2.2 Principle of Symmetric Ciphers 5

2.3 Basic Principle of Block -and Stream Ciphers 6

2.4 One-Time Pad 6

2.5 Cryptographic Hash Functions 7

2.6 Principle of Asymmetric Ciphers 8

2.7 Principle of Hybrid Crypto-Systems 11

2.8 Principle of a Digital Signature 11

2.9 Certificate 12

2.10 Symmetric Challenge Response Setup 13

2.11 Principle of Error Correcting Codes 16

2.12 Symmetric Binary Channel 17

3.1 FAR and FRR 30

3.2 PUF-secured Key-Card 32

3.3 Controlled PUF 34

3.4 Reconfigurable PUF 34

4.1 Optical PUF 38

4.2 Coating PUF 39

4.3 Arbiter PUF 40

4.4 Ring-Oscillator PUF 41

4.5 Ring-Oscillator PUF in division mode 42

4.6 Ring-Oscillator PUF in comparator mode 42

4.7 SRAM-Cell 43

4.8 Butterfly-Cell 44

4.9 Latch-Cell 44

4.10 Optical rPUF 45

5.1 Secure Key-Card 48

5.2 Initialization and Reproduction Phase 49

5.3 Creating index vector 51

5.4 Key Zeroization 52

Abbreviations

AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
BCH	Error Correcting Code by Bose, Chaudhuri and Hocquenghem
CMOS	Complementary Metal Oxide Semiconductor
CRP	Challenge Response Pair
EAN	European Article Number
EEPROM	Electrical Erasable Programmable Read Only Memory
FAR	False Acceptance Rate
FPGA	Field Programmable Gate Array
FRR	False Rejection Rate
GF	Galois Field
IC	Integrated Circuit
IP	Intellectual Property
ISBN	International Standard Book Number
LCM	Least Common Multiple
MAC	Message Authentication Code
PCA	Principle Component Analysis
PIN	Personal Identification Number
POWF	Physical One-Way Functions
PUF	Physically Unclonable Function
RFID	Radio Frequency Identification
rPUF	Reconfigurable PUF
RSA	Asymmetric Cryptographic Protocol by Rivest, Shamir and Adleman
SRAM	Static Random Access Memory

Bibliography

[1] Heike Busch, M. Sotakova, Stefan Katzenbeisser, and R. Sion. The PUF Promise. In *3rd International Conference on Trust and Trustworthy Computing (TRUST 2010)*, page 17. Springer Lecture Notes in Computer Science, 2010.

[2] Kresimir Delac and Miroslav Grgic, editors. *A Survey of Biometrical Recognition Methods*, 2004.

[3] Bundesamt für Sicherheit in der Informationstechnik, August 2010.

[4] Blaise Gassend. Physical random functions. Master's thesis, Massachusetts Institute of Technology, 2003.

[5] Blaise Gassend, Dwaine Clark, Marten van Dijk, and Srinivas Devadas, editors. *Silicon Physical Random Functions.* ACM Conference on Computer and Communications Security-CCS, 2002.

[6] Blaisse Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas, editors. *Controlled Physical Random Functions*, December 2002.

[7] Intrinsic ID Geert Jan Schrijen. Reconfigurable PUFs - Presentation at UNIQUE Workshop in Bochum, July 2010.

[8] Intrinsic ID. Technology/product Backgrounder Qiddikey. Technical report, Intrinsic ID, HighTech Campus 9, 2010.

[9] Software KPMG Electronics and Services. White paper about managing the risks of counterfeiting in the information technology industry. Technical report, KPMG, 2005.

[10] 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)*, page 17, Eindhoven,NL, 2008.

[11] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography.* CRC Press, Inc., Boca Raton, FL, USA, 1996.

[12] Richard A. Mollin. *An Introduction to Cryptography, Second Edition.* Taylor and Francis Group, LLC, 2007.

[13] National Institute of Standards and Technology. *FIPS PUB 140-2: SECURITY REQUIREMENTS FOR CRYPTOGRAPHIC MODULES.* National Institute for Standards and Technology, Gaithersburg, MD, USA, May 2001.

[14] Srinivasa Ravikanth Pappu. *Physical One-Way Functions.* PhD thesis, Massachusetts Institute of Technology, 2001.

[15] Srinivasa Ravikanth Pappu, Ben Recht, and Neil Gershenfeld. Physical one-way functions. Technical report, Science, 2002.

[16] Andrew Regenscheid, Ray Perlner, Shu jen Chang, John Kelsey, Mridul Nandi, and Souradyuti Paul. Status report on the first round of the sha-3 cryptographic hash algorithm competition. Technical report, National Institute of Standards and Technology - NIST, September 2009.

[17] Hans Riesel. *Prime Numbers and Computer Methods for Factorization.* Birkhäuser Boston Inc., 1985.

[18] Bruce Schneier. *Applied Cryptography.* Pearson Studium, 2006.

[19] Yin Su, Jeremy Holleman, and Brian Otis. A Digital 1.6 pJ/bit Chip Identification Circuit using process variations. *Solid State Cicuits Conference*, 2007.

[20] G. Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *Design Automation Conference*, pages 9–14, New York, NY, USA, 2007. ACM Press.

[21] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography with Coding Theory.* Prentice Hall, Upper Saddle River NJ, 2002.

[22] Pim Tuyls and Boris Skoric. *Secret Key Generation from Classical Physics.* Number 6 in Philips Research book series. Springer Netherlands, 2006.

[23] Mayank Vatsa, Richa Singh, P.Gupta, and A.K.Kaushik. Biometric technologies - introduction, background, fingerprint, facial recognition, iris scan, retinal scan, voice recognition, signature verification. 2010.

[24] Ingrid Verbauwhede and Roel Maes. *Towards Hardware-Intrinsic Security*, chapter Physically Unclonable Functions: A Study on the State of the Art and Future Research Directions. Springer, 2010.

[25] Serge Vrijaldenhoven. Acoustical physical unclonable functions. Master's thesis, Department of Mathematics and Computing Science, 2004.

[26] Boris Škorić, Pim Tuyls, and Tom Kevenaar, editors. *Security with Noisy Data.* Springer, 2007.

[27] Boris Škorić, Pim Tuyls, and Wil Ophey. Robust Key Extraction from Physical Uncloneable Functions. In *Applied Cryptography and Network Security (ACNS) 2005*, volume 3531 of *LNCS*, pages 407–422. Springer, 2005.

[28] Boris Škorić, Pim Tuyls, Geert-Jan Schrijen, Jan von Geloven, Nynke Verhaegh, and Rob Wolters. Read-Proof Hardware from Protective Coatings. In *Cryptographic Hardware and Embedded Systems - CHES 2006*, volume 4249 of *LNCS*, pages 369–383. Springer, 2006.