Radboud University

Concealing KETJE

A Lightweight PUF-based Privacy Preserving Authentication Protocol

Master's Thesis

J.G. (Gerben) Geltink

Final version

Supervisor
Dr. Lejla Batina
Second Assessor
Prof. dr. Joan Daemen
Daily Supervisors
Dr. Gergely Alpár
Dr. Antonio de la Piedra

Master of Computing Science in Software Science

Radboud University Nijmegen
Institute for Computing and Information Sciences
Digital Security

Nijmegen, July 2016

Abstract

We enroll more and more personal pervasive devices because these simplify our everyday lives. In order to verify the identity of these devices we use authentication protocols. Although simple authentication often suffices, users would like to remain anonymous during these authentications. Many privacy-preserving authentication protocols have been proposed that claim security and privacy. However, most of them are vulnerable in either their design or their proof of concept.

In this research, we focus on the design of a novel authentication protocol that preserves the privacy of embedded devices. A Physically Unclonable Function (PUF) generates challenge-response pairs that form the source of authenticity between a server and multiple devices. We rely on Authenticated Encryption (AE) for confidentiality, integrity and authenticity of the messages. A challenge updating mechanism combined with an authenticate-before-identify strategy is used to provide privacy. The major advantage of the proposed method is that no shared secrets need to be stored into the device's non-volatile memory. We design a protocol that supports server authenticity, device authenticity, device privacy, and memory disclosure. Following, we prove that the protocol is secure, and forward and backward privacy-preserving via game transformations. Moreover, a proof of concept is presented that uses a 3-1 Double Arbiter PUF, a concatenation of repetition and BCH error-correcting codes, and the AE-scheme KETJE. We show that our device implementation utilizes 8,305 LUTs on a 28 nm Xilinx Zynq XC7Z020 System on Chip (SoC) and takes only 0.63 ms to perform an authentication operation.

ACKNOWLEDGEMENTS

Before you lies the hard work of not only one person but many. As you read it, you will find a wide range of topics, techniques and evaluation methods all boiling down to one authentication protocol. I am content with the way it turned out, I had a steep learning curve tackling all the problems. I would like to take this opportunity now to thank everyone involved.

Firstly, I would like to thank all my supervisors, Lejla Batina, Joan Daemen, Gergely Alpár and Antonio de la Piedra. Lejla, you gave me the ideas for the topic and helped me walk along its paths. You also motivated me to submit the work to LightSec. Joan, you helped me a lot with the theoretical parts, your AE-scheme Ketje and the practical views. Gergely, your knowledge on Hardware Security lies at the basis of this work. Antonio, you helped me solving the issues I had with the hardware implementation and the Xilinx tools. You have all helped me a lot with reviewing my intermediate work and the discussions every Friday afternoon.

Secondly, I would like to thank the experts with whom I had contact with during this study. Roel Maes helped me understand how different PUFs work, and which would be best for my implementation. Takanori Machida provided me with the hardware implementation of the PUF we use. Guido Bertoni provided me with the hardware implementation of Ketje and Stjepan Picek provided me with reading material on PUF attacks.

Thirdly, I would like to thank my employer for giving me the opportunity to develop myself as an academic. I hope I can use my knowledge for the better of our organization.

Furthermore, the support of friends and family helped me to stay motivated. I realize that I explained things too technical and complicated at times. I am grateful for the attention you gave me.

Finally, I would like to thank my partner Jonneke for her continuous support and encouragement. It has been a rough time for us combining my study with the care for our newborn daughter Elin. I think we did a pretty good job.

Thank you all,

J.G. (Gerben) Geltink
January-July 2016

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Introduction

Nowadays, RFID-technology and the Internet of Things (IoT) are hot topics due to the increasing desire to simplify our everyday lives via pervasive devices. Hence, we see a shift from simple identification of devices towards complex authentication protocols, in which a challenging feature to implement is the protection of the entity's privacy. Because these entities belong to individuals who may want to preserve their privacy, we notice a shift on focusing more on privacy-preserving authentication protocols [17]. With the use of state-of-the-art cryptographic techniques, device-to-server authentication can be implemented while protecting the privacy with respect to outsiders.

One solution is to use Symmetric Key Cryptography (SKC), with a pre-shared key (PSK) and a key-updating mechanism in order to randomize device credentials at each successful authentication [33]. However, storing these PSKs requires Non-Volatile Memory (NVM) which, in the field of Hardware Security, is considered to be easily compromised by an attacker. Another option is to use Physically Unclonable Functions (PUFs), physical entities that are similar to algorithmic one-way functions. PUFs act on challenges, returning noisy PUF responses that are close enough between equal PUF instances, but far enough between different PUF instances. Using PUFs, one can refrain from storing a PSK in the device's NVM. Instead, one only needs to store a challenge which, similar to the aforementioned construction, is updated on a successful authentication. The strength of this construction is that these challenges are not secret and can safely be stored in NVM. By using a PUF, one needs to implement a Fuzzy Extractor (FE) that can produce an unpredictable key from the non-uniform and noisy PUF responses. On top of that, the FE provides for the recovery of old PUF responses from fresh PUF responses using error-correcting codes.

In order to cover the need for anonymous authentication in the IoT, research has to be done into lightweight privacy-preserving authentication protocols. A PUF-based privacy-preserving authentication protocol might be the solution. However, no such protocol exists yet that both claims security and privacy, and presents a secure proof of concept.

1.2 Related Work

PUFs were first introduced as *physical random functions* by Pappu et al. [52]. Since then, many PUF constructions have been proposed [27, 30, 36, 22, 62]. Maes researched PUFs for his PhD thesis, in which he gives a thorough explanation of PUF constructions, properties and applications [42]. More recent, Machida et al. proposed a 3-1 Double Arbiter PUF (DAPUF) which substantially decreases the prediction rate of delay-based Strong[1] PUF responses [39].

[1] In contrast to a Weak PUF that can only generate a limited amount of responses, a Strong PUF can generate 2^l Challenge-Response Pairs (CRPs), where l is the number of bits in the challenge.

Many PUF based protocols have been proposed [44, 29, 48, 3]. Majzoobi et al. propose the Slender PUF protocol, an efficient and secure method to authenticate the responses generated from a Strong PUF [44]. Their protocol does not rely on FEs and error-correcting codes because response fragments are authenticated using statistical methods. However, as Delvaux et al. pointed out, the implementation of the Slender protocol is subjected to Pseudo-Random Number Generator (PRNG) exploitation [17]. Herrewege et al. propose a reversed FE, putting the computationally less complex generation procedure in the device, and the more complex reproduction procedure on the server [29]. However less severe than the exploit in the Slender protocol, their proof of concept is also subjected to a PRNG issue [17]. Moriyama et al. propose a provably secure privacy-preserving authentication protocol that uses a different PUF response at every authentication, and thus changing the device credential after every successful authentication [48]. Aysu et al. [3] propose a provably secure protocol based on the protocols by Herrewege et al. and Moriyama et al.. Their protocol is optimized for resource-constrained platforms like Radio-Frequency Identification (RFID) devices. The authors evaluate the design using a PUF and True Random Number Generator (TRNG) based on Static Random-Access Memory (SRAM), a Pseudo-Random Function (PRF) using the SIMON block-cipher and a Reverse FE (RFE) based on Bose-Chaudhuri-Hocquenghem (BCH) codes. While this is the first effort to describe an end-to-end design and evaluation of a provable secure privacy-preserving PUF-based authentication protocol, their interleaved FE construction is vulnerable to linear equation analysis [3, p. 12]. Moreover, the authors use an additional PSK that does not increase the entropy of the communicated messages. Thus, this additional PSK can be considered overhead.

1.3 Scope and Contributions

This research focusses on improving the results of the most recent, aforementioned PUF-based privacy-preserving authentication protocol as proposed by Aysu et al. [3]. We do this by integrating a single, compact cryptographic primitive, namely Authenticated Encryption (AE), into a PUF-based privacy-preserving authentication protocol. In contrast to the protocol by Aysu et al., we aim to construct a secure FE and aim to abstain from using a PSK between server and devices. With this, we hope to improve the overall efficiency of the protocol. Therefore, our main research question is:

> *How does the integration of Authenticated Encryption in a PUF-based privacy-preserving authentication protocol affect its performance in relation to other similar, existing authentication protocols?*

For this, we design, prove and implement a novel PUF-based privacy-preserving authentication protocol using AE. We summarize our contributions as follows:

- We present the theoretical design of a novel PUF-based privacy-preserving authentication protocol using AE. By doing this we present a generic approach to create any implementation of the protocol provided the quality of PUF responses.

- We prove that the proposed protocol is mathematically secure, and forward and backward privacy-preserving, under the condition on the security of the AE-scheme and the quality of the PUF responses. For this we define a new type of Strong Extractor (SE), the Entropy Accumulator (EA), which is part of the FE.
- We present a proof of concept of the device on a development board and the server on a PC such that we can elaborate on the performance of the end-to-end design of the protocol. By doing this we present one of the first use-cases of the lightweight AEAD-scheme KETJE, which is part of the running Competition for Authenticated Encryption: Security, Applicability and Robustness (CAESAR).
- We make a comparison of the proposed protocol's performance with other similar, existing authentication protocols.
- We argue about the applicability of the proposed protocol in RFID-technology and the IoT.

1.4 Research Methodology

In order to answer our research question, we use the following research methodology:

- *Literature study.* We carry out a literature study towards authentication protocols in general, and the techniques used in the PUF-based privacy-preserving authentication protocol by Aysu et al. [3] in particular. Moreover, we study techniques that might improve the performance and security of the protocol.
- *Theoretical design.* Based on the literature study, we design a PUF-based privacy-preserving authentication protocol using AE. The aim is to replace the Symmetric Key Encryption (SKE) and the PRF from Aysu et al. [3] with a single, compact cryptographic primitive (Authenticated Encryption with Associated Data (AEAD)) that provides for confidentiality of the PUF responses, authenticity of the devices and the server, and integrity of the transmitted data. Moreover, a secure PUF needs to be selected that forms the basis for the design of a FE in particular and the protocol in general.
- *Mathematical proof.* We think that a novel protocol should be provably secure and privacy-preserving. Hence, we give mathematical proof for both the security as well as forward and backward privacy.
- *Proof of concept.* In order to evaluate the performance of the proposed protocol, we physically implement the device on a development board. On top of that, we implement the server on a PC such that we create an end-to-end design.

1.5 Relevance

As mentioned before, RFID-technology and the IoT is emerging. While the technology often is not new, by interconnecting devices and entities in the World Wide Web (WWW), we create a new type of web, the World-Sized Web (WSW) [60]. In this WSW devices and entities are moving through the network, roaming from access point to access point. One might consider that without proper privacy-preserving authentication protocols, data gets leaked about the specific devices and their locations in time. This traceability is a concern that affects everyone. To illustrate this, the Open Web Application Security Project (OWASP) has introduced a Top 10 IoT vulnerabilities, in which in the fifth place, there is "Privacy Concerns" and in the second place, there is "Insufficient Authentication/Authorization" [51]. This shows that the online community realizes effort should be devoted to both aspects of lightweight privacy-preserving authentication protocols, namely privacy and authentication.

On the other hand, one might consider conventional use-cases for RFID-technology, e.g. in supply chains or in access control. In these scenarios, the amount of devices is substantially reduced in comparison with the IoT. However, a company might want to disclose articles that wear RFID tags and their locations to competitors. Also, in access control, organizations might want to disclose to outsiders what key figures entered where at what times. This demands for a lightweight privacy-preserving authentication protocol.

In addition, this thesis is written in the partial fulfillment of the requirements for the degree Master of Computing Science in Software Science. With this work the author shows his skills in writing, reasoning, specifying, building and managing a project.

1.6 External Validity

As mentioned, since the introduction of PUFs, many authentication protocols have been proposed that rely on key generation by PUFs. However, many of them were either not provably secure or were insecure in their proof of concept. Depending on how successful this research proves to be, the protocol can be implemented in nodes in the IoT, or on a smaller scale in RFID devices in conventional use-cases. With the protocol, a generic approach is presented to construct any instance of the protocol provided the quality of the PUF responses, the desired maximum failure rate for the authentications and the desired security level. A designer can choose which PUF to use, which error-correcting codes and which AEAD-scheme. This way, the protocol might prove useful for a variety of applications.

In order to validate this research externally, a 20-page paper is submitted to the Fifth International Workshop on Lightweight Cryptography for Security & Privacy (LightSec 2016, Aksaray University, Cappadocia, Turkey).

1.7 Outline

In Chapter 2 we describe the theoretical foundation for further chapters. This chapter starts by giving notation and preliminaries before describing and defining PUFs, error-correcting codes, FEs and AE. In Chapter 3 we describe the proposed privacy-preserving authentication protocol, which we call the Concealing Ketje Protocol (CKP), named after the lightweight AEAD-scheme Ketje. This chapter starts by defining the security considerations before presenting the CKP with all of its elements. In Chapter 4 we first describe the security model and formal security definitions before presenting the security and privacy proof of the proposed protocol. In Chapter 5 we describe a proof of concept of the CKP. Chapter 6 both describes the results from the protocol as supported by the mathematical foundation as well as the results from the protocol supported by the proof of concept. Finally, in Chapter 7 we present the conclusions, discussion and future work.

CHAPTER

TWO

BACKGROUND

In this chapter we describe the theoretical foundation for further chapters. The information includes notation and preliminaries as well as theoretical background on the topics addressed in this thesis.

We start by describing the notation that is used throughout this thesis in Section 2.1. Section 2.2 describes the preliminaries. Following, in Section 2.3 we describe Physically Unclonable Functions (PUFs), which is the component that forms the basis of our privacy-preserving authentication protocol. Section 2.4 describes repetition codes and Bose-Chaudhuri-Hocquenghem (BCH) codes, two error-correcting codes that are being used in the Fuzzy Extractor (FE), which is one of the main components in our protocol. FEs are being described in Section 2.5. The chapter concludes in Section 2.6 by describing Authenticated Encryption (AE) in general and Authenticated Encryption with Associated Data (AEAD) in particular.

2.1 Notation

In this section we describe the general notation that is used throughout this thesis. For a detailed description of the specific notation used, please consult the Nomenclature starting at page 95. We use notation from Cryptography [33, 56], Coding Theory [47, 63, 41, 28] and Information Theory [61, 55], the three theoretical foundations our protocol is mainly based on. In general, we use the following notation:

- Classes and sets are denoted by calligraphic letters, e.g. $\mathcal{A}, \mathcal{B}, \ldots, \mathcal{Z}$.
- Vectors and (binary) variables/strings are denoted by capitalized roman letters, e.g. $A \in \mathcal{A}, B \in \mathcal{B}, \ldots, Z \in \mathcal{Z}$.
  - Varying instances of variable A are identified using superscript (e.g. A', A^1 or A^{old})
  - $A = [1, 1, 0]$ denotes a binary string with characters $A_0 = 0$, $A_1 = 1$ and $A_3 = 1$ with $\mathbf{a}(x) = x^2 + x$ its polynomial.
  - $B_{2\to 0}$ denotes the substring of B with characters B_2,B_1 and B_0.
  - $E = C \parallel D$ denotes the concatenation of strings C and D.
  - $|F| = n$ denotes the length n, or the amount of bits of F.
  - $I = G \oplus H$ denotes the bitwise exclusive-OR (XOR) of strings G and H.
  - $\langle J, K\rangle$ denotes a tuple of strings J and K.
- Functions are either denoted by $\textbf{function}(\,\cdot\,, \,\ldots\,, \,\cdot\,)$, where $\cdot$ denotes an input to the function, or by calligraphic letters similar to sets[1].

2.2 Preliminaries

In this section we give the preliminary definitions that are being used throughout this thesis. Again, we use definitions from Cryptography [33, 56], Coding Theory [47, 63, 41, 28] and Information Theory [61, 55]. We define the Hamming distance and Hamming weight, Shannon entropy, min-entropy and statistical distance.

2.2.1 Hamming Distance and Hamming Weight

The Hamming distance, introduced by Hamming [28] is defined as follows:

Definition 2.1 (Hamming distance)**.** *The Hamming distance* $\mathbf{HD}(Y, Y')$ *between two binary vectors* $Y, Y' \leftarrow \mathcal{Y}$ *of the same length is the number of positions in both vectors with differing values:*

$$\mathbf{HD}(Y, Y') = |\{i : Y_i \neq Y'_i\}|$$

[1] The context in which calligraphic letters are used clearly reveals the denotation.

The distance metric $\mathbf{dist}(Y, Y')$ over two binary vectors $Y, Y' \leftarrow \mathcal{Y}$ is defined by the Hamming distance. Similarly, the *Hamming weight* is defined as [28]:

Definition 2.2 (Hamming weight)**.** *The Hamming weight* $\mathbf{HW}(Y)$ *of a vector* $Y \leftarrow \mathcal{Y}$ *is the number of positions with non-zero values:*

$$\mathbf{HW}(Y) = |\{i : Y_i \neq 0\}|$$

Note that $\mathbf{HW}(Y \oplus Y') = \mathbf{HD}(Y, Y')$.

2.2.2 Shannon Entropy

The measurement of entropy we use is *Shannon entropy*, introduced by Shannon [61]:

Definition 2.3 (Shannon entropy)**.** *The Shannon entropy* $\mathbf{H}(Y)$ *of a discrete random variable* $Y \leftarrow \mathcal{Y}$ *is defined as:*

$$\mathbf{H}(Y) = -\sum_{Y_i \in \mathcal{Y}} \mathbf{Pr}(Y_i) * \log_2 \mathbf{Pr}(Y_i)$$

The entropy of a binary variable $Y \leftarrow \{0,1\}^l$ with probabilities $\mathbf{Pr}(Y_i = 1) = p$ and $\mathbf{Pr}(Y_i = 0) = 1 - p$ $(0 \leq i < l)$ is defined in the binary entropy function $\mathbf{h}(p)$:

$$\mathbf{h}(p) = -p \log_2(p) - (1-p) \log_2(1-p) \tag{2.2.1}$$

Sometimes we use an approach that expects the worst outcome to Shannon entropy, which is the min-entropy introduced by Rényi [55]. If $Y \in \mathcal{Y}$ is uniformly distributed, the Shannon Entropy and min-entropy are equal. However, if this is not the case, the 'worst-case' scenario is taken for the min-entropy.

We define the min-entropy as follows [42, p. 206]:

Definition 2.4 (Min-entropy)**.** *The min-entropy* $\tilde{\mathbf{H}}_\infty(Y)$ *of a random variable* $Y \in \mathcal{Y}$ *is defined as:*

$$\tilde{\mathbf{H}}_\infty(Y) = -\log_2 \max_{Y_i \in \mathcal{Y}} \mathbf{Pr}(Y_i)$$

2.2.3 Statistical Distance

The statistical distance is a measure of distinguishability between two probability distributions (e.g. random variables/vectors/strings).

We define the statistical distance as follows [19, p. 528]:

Definition 2.5 (Statistical distance)**.** *The statistical distance* $\mathbf{SD}(A, B)$ *between two probability distributions* A *and* B *is:*

$$\mathbf{SD}(A, B) = \frac{1}{2} \sum_{V \in \mathcal{V}} |\mathbf{Pr}(A = V) - \mathbf{Pr}(B = V)|,$$

where $\mathbf{Pr}(x)$ *denotes the probability that* x *occurs and* $\mathcal{V}$ *denotes the set from which the statistical distance is sampled using* V*.*

2.3 Physically Unclonable Functions

PUFs are entities that are intrinsically embodied in physical structures. The main characteristic of a PUF is that it should be easy to evaluate but hard to predict, moreover, it should be practically impossible to duplicate. Because of its equivalence to algorithmic one-way-functions, PUFs might be ideal for cryptographic purposes.

Although the following definition is somewhat decrepit, we can still use it to illustrate the general idea of a PUF [21, p. 2]:

Definition 2.6 (Physical Unclonable Function)**.** *A Physical Unclonable Function is a function that maps challenges to responses and that is embodied in a physical object. It satisfies the following properties:*

1. *Easy to evaluate: the physical object can be evaluated in a short amount of time.*
2. *Hard to characterize: from a number of measurements performed in polynomial time, an attacker who no longer has the device and who only has a limited (polynomial) amount of resources can only obtain a negligible amount of knowledge about the response to a challenge that is chosen uniformly at random.*

This definition was superseded by numerous broader and often more complex definitions of which Armknecht et al. [1] try to unify them all. For this thesis, we follow Maes [42] as he covers the main characteristics we need to describe a PUF.

A PUF is mainly characterized by its reproducibility, uniqueness, identifiability, unclonability and unpredictability, which are defined by the intra- and inter-distance of the PUF responses [42, p. 20-23, 61-64].

Before we give these definitions, we first introduce the notion of a PUF class, denoted as $\mathcal{P}$, which is the set of PUFs that share the same PUF construction type (see Section 2.3.5). The set of all possible challenges X which can be applied to an instance of $\mathcal{P}$ is denoted as $\mathcal{X}_{\mathcal{P}}$.

2.3.1 Intra-distance and Reproducibility

When challenging a single PUF multiple times with the same challenge, there is a chance that the response bits are different in both responses. We call the probability that a single bit is different between measurements the bit-error probability p_e. This is one of the characteristics of a PUF and is defined by its intra-distance, which is defined as follows [42, p. 20]:

Definition 2.7 (Intra-distance)**.** *A PUF response intra-distance is modeled as a random variable describing the distance between two PUF responses from the same PUF instance using the same challenge:*

$$\mathcal{D}^{\text{intra}}_{\mathbf{puf}_i}(X) = \mathbf{dist}(Y^i \leftarrow \mathbf{puf}_i(X), Y^{i'} \leftarrow \mathbf{puf}_i(X)),$$

with Y^i and $Y^{i'}$ two distinct and random evaluations of PUF instance $\mathbf{puf}_i$ on the same challenge X. Additionally, the PUF response intra-distance for a random PUF instance and a random challenge is defined as the random variable:

$$\mathcal{D}^{\text{intra}}_{\mathcal{P}} = \mathcal{D}^{\text{intra}}_{\mathbf{puf} \leftarrow \mathcal{P}}(X \leftarrow \mathcal{X}_{\mathcal{P}})$$

The intra-distance provides reproducibility of any unique PUF instance $\mathbf{puf}_{0 \leq i < n} \in \mathcal{P}$ (where n is the total number of PUFs in the PUF class $\mathcal{P}$), which means that if two measurements are performed on the same PUF, then these responses are with high probability close to each other. More precisely, reproducibility is defined as follows [42, p. 61]:

Definition 2.8 (Reproducibility)**.** *A PUF class $\mathcal{P}$ exhibits reproducibility if:*

$$\mathbf{Pr}(\mathcal{D}^{\text{intra}}_{\mathcal{P}} \textit{ is small}) \textit{ is high}$$

Note that Maes does not introduce a theoretical or experimental bound. For now, an unbounded perspective suffices. We introduce a theoretical bound ϵ in the formal security definitions in Section 4.2 needed for the security and privacy proofs in Section 4.3.

2.3.2 Inter-distance and Uniqueness

The inter-distance is defined as follows [42, p. 22]:

Definition 2.9 (Inter-distance)**.** *A PUF response inter-distance is modeled as a random variable describing the distance between two PUF responses from different PUF instances using the same challenge:*

$$\mathcal{D}^{\text{inter}}_{\mathcal{P}}(X) = \mathbf{dist}(Y \leftarrow \mathbf{puf}(X), Y' \leftarrow \mathbf{puf}'(X)),$$

with Y and Y' evaluations of the same challenge X on two random but distinct PUF instances $\mathbf{puf} \in \mathcal{P}$ and $\mathbf{puf}'$ $(\neq \mathbf{puf}) \in \mathcal{P}$. Additionally, the PUF response inter-distance for a random challenge is defined as the random variable:

$$\mathcal{D}^{\text{inter}}_{\mathcal{P}} = \mathcal{D}^{\text{inter}}_{\mathcal{P}}(X \leftarrow \mathcal{X}_{\mathcal{P}})$$

The inter-distance provides uniqueness of any PUF instance $\mathbf{puf}_{0 \leq i < n}$ in the PUF class $\mathcal{P}$ (where n is the total number of PUFs in the PUF class $\mathcal{P}$), which implies that responses of measurements performed on different PUFs (taking into account that one of the PUFs might be fake) are with high probability far apart. More precisely, uniqueness is defined as follows [42, p. 62]:

Definition 2.10 (Uniqueness)**.** *A PUF class $\mathcal{P}$ exhibits uniqueness if:*

$$\mathbf{Pr}(\mathcal{D}^{\text{inter}}_{\mathcal{P}} \textit{ is large}) \textit{ is high}$$

2.3.3 Identifiability

Reproducibility of a PUF instance $\mathbf{puf}_i \in \mathcal{P}$ and uniqueness between PUF instances $\mathbf{puf}_i, \mathbf{puf}_j \in \mathcal{P}$ (where $i \neq j$) provides identifiability of PUF instance $\mathbf{puf}_i \in \mathcal{P}$. More precisely, identifiability is defined as follows [42, p. 62]:

Definition 2.11 (Identifiability)**.** *A PUF class $\mathcal{P}$ exhibits identifiability if it is reproducible and unique, and in particular if:*

$$\mathbf{Pr}(\mathcal{D}^{\text{intra}}_{\mathcal{P}} < \mathcal{D}^{\text{inter}}_{\mathcal{P}}) \textit{ is high}$$

2.3.4 Unclonability and Unpredictability

For cryptographic applications, unclonability and unpredictability are essential. The characteristic of unclonability assures that physically and technically, a PUF instance $\mathbf{puf}_{i'} \in \mathcal{P}$ is difficult (or even impossible) to create from an other PUF instance $\mathbf{puf}_i \in \mathcal{P}$. More precisely, unclonability is defined as follows [42, p. 63]:

Definition 2.12 (Unclonability)**.** *A PUF class $\mathcal{P}$ exhibits unclonability it is hard to apply and/or influence the creation procedure in such a way as to produce two distinct PUF instances $\mathbf{puf}, \mathbf{puf}' \in \mathcal{P}$ for which it holds that:*

$$\mathbf{Pr}(\mathbf{dist}(Y \leftarrow \mathbf{puf}(X), Y' \leftarrow \mathbf{puf}'(X)) < \mathcal{D}^{\text{inter}}_{\mathcal{P}}(X)) \textit{ is high,}$$

for $X \leftarrow \mathcal{X}_\mathcal{P}$. Ultimately, it should be hard to produce two PUF instances for which it holds that:

$$\mathbf{Pr}(\mathbf{dist}(Y \leftarrow \mathbf{puf}(X), Y' \leftarrow \mathbf{puf}'(X)) > \mathcal{D}^{\text{intra}}_{\mathcal{P}}(X)) \textit{ is low}$$

The characteristic of unpredictability ensures that unobserved responses remain sufficiently random, even after observing responses to other challenges on the same PUF instance. More precisely, unpredictability is defined as follows [42, p. 64]:

Definition 2.13 (Unpredictability)**.** *A PUF class $\mathcal{P}$ exhibits unpredictability if it is hard to win the following game for a random PUF instance* $\mathbf{puf} \in \mathcal{P}$*:*

- *In a learning phase, one is allowed to evaluate* **puf** *on a limited number of challenges and observe the responses. The set of evaluated challenges is $\mathcal{X}'_\mathcal{P}$ and the challenges are either randomly selected (weak unpredictability) or adaptively chosen (strong unpredictability).*
- *In a challenging phase, one is presented with a random challenge $\mathcal{X} \leftarrow \mathcal{X}_\mathcal{P} \setminus \mathcal{X}'_\mathcal{P}$. One is required to make a prediction Y^{pred} for the response to this challenge when evaluated on* **puf***. One does not have access to* **puf***, but the prediction is made by an algorithm* **predict** *which is trained with the knowledge obtained in the learning phase: $Y^{\text{pred}} \leftarrow \mathbf{predict}(X)$.*
- *The game is won if:*

$$\mathbf{Pr}(\mathbf{dist}(Y^{\text{pred}} \leftarrow \mathbf{predict}(X), Y \leftarrow \mathbf{puf}(X)) < \mathcal{D}^{\text{inter}}_{\mathcal{P}}(X)) \textit{ is high}$$

One way of carrying out this experiment is by using Machine Learning (ML) attacks. Recent studies have shown that PUF responses can be predicted in some practical scenarios [58, 57]. It is evident that a PUF should be designed carefully, taking this risk into account.

2.3.5 PUF Construction Types

Various PUF construction types have been proposed, a few of them are being discussed in this section: Static RAM (SRAM) PUFs, Arbiter PUFs, Ring Oscillator PUFs (ROPUFs) and Non-Intrinsic (PUF-like) PUFs [42].

2.3.5.1 SRAM PUFs

SRAM PUFs were first introduced by Guajardo et al. [27] and Holcomb et al. [30]. SRAM PUFs are based on the principle that upon power cycling an SRAM cell, its transient behavior either sets the value of the cell to 0 or 1. This transient behavior is intrinsically introduced due to the random process variations in the production of the cells. Hence, these variations are cell-specific and will (upon power cycling) determine the state of the cell with a high probability to a certain value. This value is called the preferred initial operating point and is introduced by a race condition in the electrical flow upon powering the SRAM. However, once in a while this value will be different from the preferred initial operating point because of the race condition, which makes SRAM usable in a PUF construction. Challenges and responses can be created by adhering to memory addresses as challenges and the state of the SRAM as the PUF response. Before challenging the PUF with a same challenge (memory address), a power cycle needs to be executed. This construction is closely related to a One-Time Pad (OTP) where addresses are mapped to the state of the SRAM [33]. However, with each power cycle there is a chance that the pad changes from the previous measurement.

2.3.5.2 Arbiter PUFs

Figure 2.1 illustrates an Arbiter PUF, a type of delay-based PUF which was first introduced by Lee et al. [36]. Here, the multiplexers that act on the same challenge bit C_i ($0 \leq i < n$, $|C| = n$) are called a switch block, and the two negative-AND (NAND) ports are called an arbiter. An arbiter PUF is based on the idea that there exists a race condition between signals of two digital paths on an Integrated Circuit (IC). This race condition, or the delay of a path, is introduced by the random process variations in the production of the IC. This behavior makes the measurements of the delays usable in a PUF construction. An arbiter (usually a simple latch) either sets the response to 0 or 1, depending on what signal arrived first at the arbiter. Often, the paths are implemented using switch blocks (usually multiplexers) that act on a challenge bit. These switch blocks either let signals switch from digital paths or keep their paths. By concatenating a number of these switch blocks, a challenge can be sent to the PUF. In order for the response to have sufficient length, a number of these race conditions can be measured in parallel. For example, by measuring n arbiter PUFs in parallel using the same challenge, a n-bit response can be obtained. It is evident that this solution substantially increases the area of the IC. Another option is to measure responses sequentially using differing challenges. For example, one could append n bits to the challenge address space to measure 2^n responses, obtaining a 2^n-bit response. It is evident that this solution substantially increases the latency of PUF responses.

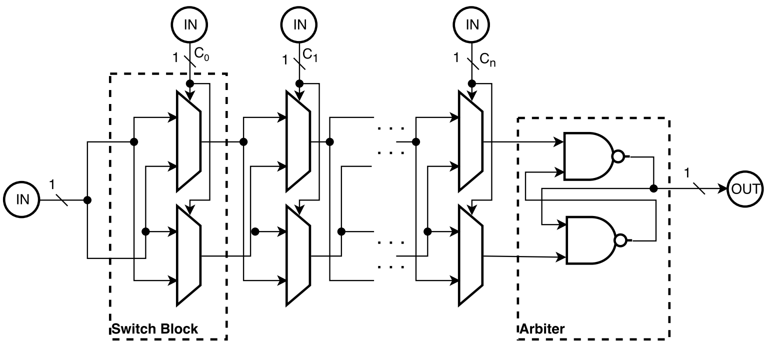


Figure 2.1: An arbiter PUF as introduced by Lee et al. [36]. (IN) denotes the input of the PUF, (OUT) denotes the output of the PUF.

The responses of these PUFs can easily be predicted using ML-attacks. Machida et al. have shown that conventional arbiter PUFs have a prediction rate of 86%[40]. However, arbiter PUFs are used in various constructions that provide for a good prediction rate (i.e. approximating 50%). One example is discussed in the next section, various others by Machida et al. [40].

2.3.5.3 Ring Oscillator PUFs

Figure 2.2 illustrates a ROPUF, another type of delay-based PUF which was first introduced by Gassend et al. [22]. In this figure, the configurable delay can (for example) be the delay introduced in the arbiter PUF from Section 2.3.5.2 or a series of inverters. This PUF is based on the measurements of the frequencies of digital oscillating circuits. Again, the differences in these measurements are introduced by the random process variations in the production of the IC. Usually, a ROPUF consists of a number of ring oscillators and an equal amount of frequency counters [43]. After measuring the frequencies of these oscillators, an ordering of these frequencies, and an encoding of this ordering reveals a PUF response. A challenge can be introduced by adding multiple oscillators in batches. This challenge, fed to a multiplexer can indicate from which oscillator in the batch the frequency needs to be measured.

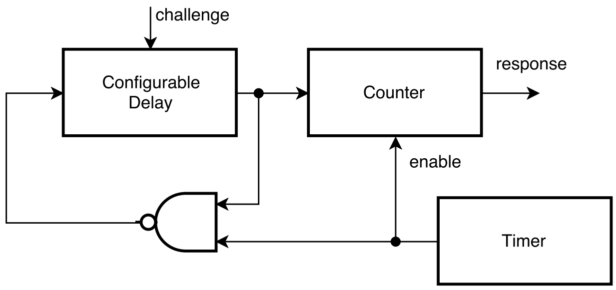


Figure 2.2: A ROPUF as introduced by Gassend et al. [22].

2.3.5.4 Non-Intrinsic (PUF-like) PUFs

Various non-intrinsic (PUF)-like PUFs have been proposed that have randomness that has been explicitly introduced. These PUFs are non-intrinsic because they are not completely integrated in an embedding device and/or they are not produced in the standard manufacturing process of their embedding device. Tuyls and Škorić [62] describe optical, coating and acoustic PUFs.

2.3.5.4.1 Optical PUFs An optical PUF is based on either absorption, transmission, reflection, scattering or a combination thereof, of a microstructural surface. The idea is that this surface has random variations in one of these characteristics introduced in the production of the surface. These random variations in the microstructural surface can be used in a PUF construction by challenging it at different locations on the surface. An example is to shoot a laser through a transparent material (e.g. glass) and observe the speckle pattern using a camera [62].

2.3.5.4.2 Coating PUFs A coating PUF is based on the dielectric variation of the coating of an IC. During manufacturing, the coating is doped with dielectric particles that respond with a different capacitance value on differing voltage inputs with varying frequencies. These differing capacitance values can be used in a PUF construction by challenging the PUF with different voltage inputs.

2.3.5.4.3 Acoustic PUFs An acoustic PUF is a PUF that is based on the response of sending an acoustic wave to an object. The acoustic wave propagates through the object and scatters on randomly distributed inhomogeneities that are introduced during manufacturing of the object. These differing wave responses can be used in a PUF construction by pointing the acoustic wave at different locations on the surface of the object as challenges.

2.4 Error-Correcting Codes

When sending data over a noisy channel, there is a chance that this data might be corrupted. For example, when sending a single bit, there is a probability $\mathbf{Pr}(\text{“bit flipped”}) = p$ that this bit gets flipped. This is due to a Binary Symmetric Channel (BSC) [63, p. 2], as depicted in Figure 2.3.

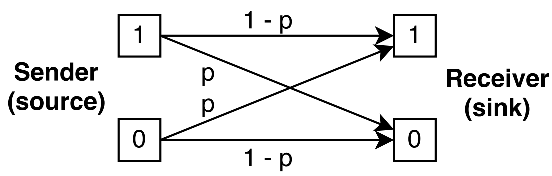


Figure 2.3: The BSC.

In this figure,

$$\mathbf{Pr}(\text{“1 received”} \mid \text{“0 transmitted”}) = \mathbf{Pr}(\text{“0 received”} \mid \text{“1 transmitted”}) = p, \text{ and}$$

$$\mathbf{Pr}(\text{“0 received”} \mid \text{“0 transmitted”}) = \mathbf{Pr}(\text{“1 received”} \mid \text{“1 transmitted”}) = 1 - p.$$

The BSC can be used to model various media, for example the aforementioned noisy channel (tele-communications, satellite communication), a storage medium or PUFs. In this thesis we only consider the BSC in a PUF scenario where multiple measurements on the same challenge return noisy responses. We require that the protocol's channel is ideal (i.e. no errors occur during communication).

In order to recover the original data from the possibly corrupted data, error-correcting codes are used [41, p. 1]. Error-correcting codes have the ability to correct up to t bits of original data from the transmitted information.

Although error-correcting codes are often used for transmitted data, we can also use them for reconstructing PUF-responses.

Example 2.4.1. Take two of the same PUF challenges X on the same PUF $\mathbf{puf}_i$, $Y^i \leftarrow \mathbf{puf}_i(X)$ and $Y^{i'} \leftarrow \mathbf{puf}_i'(X)$, with $Y^i = [0,1,0,1,0,1,1,0]$ and $Y^{i'} = [0,1,1,1,0,0,1,0]$. This can be seen as the transmission of PUF response Y^i using the PUF $\mathbf{puf}_i$ under the influence of a BSC, receiving the PUF response $Y^{i'}$. We say that the error vector, indicating which bits have been 'corrupted', of these two response vectors is $E = Y^i \oplus Y^{i'} = [0,0,1,0,0,1,0,0]$. $\triangleleft$

One way of carrying out error correction to the errors that were introduced in the example is to encode the PUF response Y^i and decode the second PUF response $Y^{i'}$. Specific codes used in an encoding have the ability to carry out error detection to the error(s) introduced by a BSC. Error detection leads to error correction, which helps us obtain the PUF response Y^i in a decoding.

We describe binary repetition codes and binary BCH codes which are used in our proposed protocol to recover PUF responses.

2.4.1 Repetition Codes

One of the most basic error-correcting codes is the class of binary repetition codes, or repetition codes. As the name suggests, the codeword for a bit is the repetition of that bit such that probability indicates the original bit in a decoding.

We define binary repetition codes as follows:

Definition 2.14 (Binary Repetition Code)**.** *A binary linear code* $\mathcal{C}_{REP}(n,1,t)$ *with codewords* $\mathbf{0} = [0,0,\ldots,0]$ *and* $\mathbf{1} = [1,1,\ldots,1]$ *is called a binary repetition code of length* n *and error correcting capability* $t = \lfloor \frac{n-1}{2} \rfloor$*.*

2.4.1.1 Encoding

As we have mentioned before, the most basic form of encoding a string is by repeating its characters for a number of n times. There are more advanced techniques of using repetition codes, of which one will be described in Section 3.5.3.1.

Example 2.4.2. Take the binary repetition code $\mathcal{C}_{\mathrm{REP}}(n,1,t)$ with $n = 3$ and $t = \lfloor \frac{n-1}{2} \rfloor = 1$. An encoding of the message $M = [0,1,1,0]$ of length $l_M = 4$ results in the codeword $W = [0,0,0] \parallel [1,1,1] \parallel [1,1,1] \parallel [0,0,0]$ of length $l_W = 4 * n = 12$. $\triangleleft$

2.4.1.2 Decoding

Decoding of a received codeword $W' = W \oplus E$ of length $l_{W'}$ and error vector E is performed by measuring the Hamming weight of the n-bit substrings of W'.

Take the binary repetition code $\mathcal{C}_{\text{REP}}(n, 1, t)$. Let us say a message M of length l_M was encoded using this code. We can decode the received codeword W' as follows (for $0 \leq i < l_M$):

$$\text{If } \mathbf{HW}(W'_{i \cdot n+(n-1) \to i \cdot n}) > t \text{ then we take } M_i = 1 \text{ and } M_i = 0 \text{ otherwise.} \tag{2.4.1}$$

This gives the highest probability that the decoding of the substrings represents the original message. There is a chance that the decoding will be faulty, as more than t bits in the n-bit string might be corrupted in the channel. The probabilistic foundation of repetition codes does not support detection of this faulty decoding. For this, the more complex BCH codes can be used which will be described in Section 2.4.2.

Example 2.4.3. Take the message $M = [0, 1, 1, 0]$ and the codeword $W = [0, 0, 0] \parallel [1, 1, 1] \parallel [1, 1, 1] \parallel [0, 0, 0]$ from Example 2.4.2 and an error vector $E = [0, 0, 0] \parallel [0, 0, 1] \parallel [0, 1, 0] \parallel [1, 0, 1]$. The received codeword results in $W' = W \oplus E = [0, 0, 0] \parallel [1, 1, 0] \parallel [1, 0, 1] \parallel [1, 0, 1]$. Using Formula 2.4.1, a decoding of the received codeword W' results in the received message $M' = [0, 1, 1, 1]$. ◁

As we can see $M'_0 \neq M_0$ because $\mathbf{HW}(W'_{2 \to 0}) > t$, thus $M_0 = 1$ whereas the original bit was 0. In this specific case, the number of errors in the error vector was higher than the error correcting capability t. In order to prevent this behavior, one can use a better coding technique, a different code with a higher error correcting capability (increasing the overhead on the channel), or one can use a concatenation of error-correcting codes, which is being described in Section 3.5.3.

2.4.2 BCH Codes

BCH codes are a class of cyclic error-correcting codes that are constructed using finite fields. The algebraic foundation of BCH codes makes them ideal for error correction.

For any positive integer $m \geq 3$ and $t < 2^m - 1$, there exists a binary BCH code with the following parameters [45, 47, p. 99]:

- The block-length of a code is the amount of bits that the code acts on. When encoding a message larger than the block-length, the code is applied to a multiple of the block-length. The block-length of the code is given by:

$$n = 2^m - 1 \tag{2.4.2}$$

- The number of parity-check bits is the number of bits that are used to detect and correct errors in the code. The number of parity-check bits is given by:

$$n - k \leq m \cdot t \tag{2.4.3}$$

- The minimum distance is the minimum of $\mathbf{HD}(U, V)$ over all distinct codewords U and V. If the following condition holds, the decoder will always decode correctly when there are t or fewer errors.

$$d_{\min} \geq 2t + 1 \tag{2.4.4}$$

We define a binary BCH code as follows:

Definition 2.15 (Binary BCH Code)**.** *A binary BCH code $\mathcal{C}_{BCH}(n, k, t)$ is called a binary BCH code of order m, length $n = 2^m - 1$, distance d, error correcting capability $t = \lfloor \frac{d-1}{2} \rfloor$ and primitive element $\alpha \in \mathbf{GF}(2^m)$:*

$$\begin{aligned}\mathcal{C}_{BCH}(n, k, t) = \{(W_0, \ldots, W_{n-1}) \in \mathbf{GF}(2^m) \mid \mathbf{w}(x) = W_{n-1}x^{n-1} + \cdots + W_1 x + W_0 \\ \textit{satisfies } \forall\, 1 \leq i \leq 2t : \mathbf{w}(\alpha^i) = \mathbf{w}(\alpha^{2i}) = \cdots = \mathbf{w}(\alpha^{(n-1)i}) = 0\}\end{aligned}$$

2.4.2.1 Encoding

In order to encode a message M of length k we first need to construct the generator polynomial.

Let $\Phi_i(x)$ be the minimal polynomial of α^i, the primitive element. Then the generator polynomial $\mathbf{g}(x)$ must be the least common multiple of $\Phi_1(x)$, $\Phi_2(x)$, $\cdots$, $\Phi_{d-1}(x)$, i.e.,

$$\mathbf{g}(x) = \mathbf{LCM}(\Phi_1(x), \Phi_2(x), \ \cdots\ , \Phi_{d-1}(x)) \tag{2.4.5}$$

The degree of the generator polynomial $\mathbf{g}(x)$ is at most $m \cdot t$, hence, the number of parity-check bits $(n - k)$ is at most $m \cdot t$ (Formula 2.4.3).

Example 2.4.4. Let α be a primitive element of $\mathbf{GF}(2^4)$ generated by the primitive polynomial $\mathbf{p}(x) = x^4 + x + 1$ [14]. The finite field table is given in Appendix B.1. The minimal polynomials $\Phi_i(x)$ $(1 \leq i \leq n)$ of α are[2]:

$$\begin{aligned}
\Phi_1(x) &= (x + \alpha)(x + \alpha^2)(x + \alpha^4)(x + \alpha^8) \\
&= x^4 + x + 1 \\
\Phi_3(x) &= (x + \alpha^3)(x + \alpha^6)(x + \alpha^{12})(x + \alpha^9) \\
&= x^4 + x^3 + x^2 + x + 1 \\
\Phi_5(x) &= (x + \alpha^5)(x + \alpha^{10}) \\
&= x^2 + x + 1 \\
\Phi_7(x) &= (x + \alpha^7)(x + \alpha^{14})(x + \alpha^{13})(x + \alpha^{11}) \\
&= x^4 + x^3 + 1
\end{aligned}$$

[2]Note that conjugates are omitted. A conjugate is a minimal polynomial $\Phi_j(x)$ that is equal to $\Phi_i(x)$ $(i < j)$. For example, in Example 2.4.4, $\Phi_1(x) = \Phi_2(x) = \Phi_4(x) = \Phi_8(x)$.

Then, using Formula 2.4.5, the double-error-correcting ($t = 2$, $d = 5$) BCH code of length $n = 2^4 - 1 = 15$ (Formula 2.4.2) is generated by:

$$\begin{aligned}\mathbf{g}(x) &= \mathbf{LCM}(\Phi_1(x), \Phi_3(x)) \\ &= (x^4 + x + 1)(x^4 + x^3 + x^2 + x + 1) \\ &= x^8 + x^7 + x^6 + x^4 + 1\end{aligned}$$

Hence, $n - k = 8$ such that this is a $\mathcal{C}_{\text{BCH}}(n, k, t) = \mathcal{C}_{BCH}(15, 7, 2)$ BCH code. $\lhd$

The remainder polynomial $\mathbf{r}(x)$ that contains the parity-check bits is obtained by:

$$\mathbf{r}(x) = x^{n-k} \cdot \mathbf{m}(x) \mod \mathbf{g}(x) \tag{2.4.6}$$

Example 2.4.5. Take the message $M = [1, 1, 0, 0, 1, 1, 0]$ with message polynomial $\mathbf{m}(x) = x^6 + x^5 + x^2 + x$ and generator polynomial $\mathbf{g}(x) = x^8 + x^7 + x^6 + x^4 + 1$ as calculated in Example 2.4.4, the remainder polynomial $\mathbf{r}(x)$ is obtained by using Formula 2.4.6:

$$\begin{aligned}\mathbf{r}(x) &= x^8(x^6 + x^5 + x^2 + x) \mod x^8 + x^7 + x^6 + x^4 + 1 \\ &= x^{14} + x^{13} + x^{10} + x^9 \mod x^8 + x^7 + x^6 + x^4 + 1 \\ &= x^3 + 1\end{aligned}$$

$\lhd$

Finally, the code polynomial $\mathbf{w}(x)$ is obtained by:

$$\mathbf{w}(x) = x^{n-k}\mathbf{m}(x) + \mathbf{r}(x) \tag{2.4.7}$$

Example 2.4.6. Take the message polynomial $\mathbf{m}(x) = x^6 + x^5 + x^2 + x$ and the remainder polynomial $\mathbf{r}(x) = x^3 + 1$ as calculated in Example 2.4.5, the code polynomial $\mathbf{w}(x)$ is obtained by using Formula 2.4.7:

$$\begin{aligned}\mathbf{w}(x) &= x^8(x^6 + x^5 + x^2 + x) + x^3 + 1 \\ &= x^{14} + x^{13} + x^{10} + x^9 + x^3 + 1\end{aligned}$$

As a result, the codeword for the message $M = [1, 1, 0, 0, 1, 1, 0]$ is given by
$W = [1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1]$. $\lhd$

2.4.2.2 Decoding

As per Definition 2.15, the decoding is based on the following algebraic characteristic:

$$\forall\ 1 \leq i \leq 2t : \mathbf{w}(\alpha^i) = W_{n-1}\alpha^{(n-1)i} + \cdots + W_2\alpha^{2i} + W_1\alpha^i + W_0 = 0 \tag{2.4.8}$$

The decoding of a received codeword $W' = W \oplus E$ with codeword polynomial $\mathbf{w}'(x) = W'_{n-1}x^{n-1} + \cdots + W'_1 x + W'_0$ and error polynomial $\mathbf{e}(x) = E_{n-1}x^{n-1} + \cdots + E_1 x + E_0$ is performed in four steps:

1. Compute the syndromes $\mathcal{S}_i$ $(1 \leq i \leq 2t)$.
2. Determine the error-locator polynomial $\sigma(x)$.
3. Find the error-locator polynomial coefficients $\sigma_1, \sigma_0, \ldots, \sigma_\tau$ $(\tau \leq t)$.
4. Carry out the error correction using the error correction polynomial $\mathbf{e}'(x)$.

Here, τ is the actual number of errors in the code. We describe these four steps in the following sections.

2.4.2.2.1 Syndrome Computation We can compute the syndromes $\mathcal{S}_i$ $(1 \leq i \leq 2t)$ using:

$$\begin{aligned} \mathcal{S}_i &= \mathbf{w}'(\alpha^i) \\ &= W'_{n-1}\alpha^{(n-1)i} + \cdots + W'_2\alpha^{2i} + W'_1\alpha^i + W'_0 \end{aligned} \tag{2.4.9}$$

Example 2.4.7. Take the message $M = [1, 1, 0, 0, 1, 1, 0]$ and its codeword $W = [1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1]$ as calculated in Section 2.4.2.1. By applying the error vector $E = [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]$ on this message, we obtain the received codeword $W' = [1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1]$. Hence, the codeword polynomial is given by: $\mathbf{w}'(x) = x^{14} + x^{13} + x^9 + x^3 + x^2 + 1$.

The four syndromes are computed using Formula 2.4.9 and Table B.1 from Appendix B:

$$\begin{aligned} \mathcal{S}_1 &= \alpha^{14} + \alpha^{13} + \alpha^9 + \alpha^3 + \alpha^2 + 1 \\ &= \alpha + 1 = \alpha^4 \\ \mathcal{S}_2 &= \alpha^{28} + \alpha^{26} + \alpha^{18} + \alpha^6 + \alpha^4 + 1 \\ &= \alpha^2 + 1 = \alpha^8 \\ \mathcal{S}_3 &= \alpha^{42} + \alpha^{39} + \alpha^{27} + \alpha^9 + \alpha^6 + 1 \\ &= \alpha^3 + \alpha^2 + 1 = \alpha^{13} \\ \mathcal{S}_4 &= \alpha^{56} + \alpha^{52} + \alpha^{36} + \alpha^{12} + \alpha^8 + 1 \\ &= \alpha \end{aligned}$$

$\triangleleft$

2.4.2.2.2 Error Locator Polynomial Coefficients The error locator polynomial can be expressed as follows:

$$\begin{aligned} \sigma(x) &= (1 - \beta_\tau x) \ldots (1 - \beta_2 x)(1 - \beta_1 x) \\ &= \sigma_\tau x^\tau + \cdots + \sigma_1 x + \sigma_0, \end{aligned} \tag{2.4.10}$$

where $\tau \leq t$ is the number of errors in the code.

In the case $\tau \leq t$, the number of roots (of which the calculation is given in Section 2.4.2.2.3) is equal to the degree of the error locator polynomial. If we find a higher degree of the error locator polynomial $\sigma(x)$, we can conclude that there were more errors in the code than its error correcting capability t (i.e. $\tau > t$). In this case, no errors can be located. In order to prevent this behavior, one can use a better coding technique, a different code with a higher error correcting capability (increasing the overhead on the channel), or one can use a concatenation of error-correcting codes, which is being described in Section 3.5.3.

The coefficients of $\sigma(x)$ are:

$$\begin{aligned}
\sigma_0 &= 1 \\
\sigma_1 &= \beta_\tau + \cdots + \beta_2 + \beta_1 \\
\sigma_2 &= \beta_{\tau-1}\beta_\tau + \cdots + \beta_1\beta_3 + \beta_1\beta_2 \\
\sigma_2 &= \beta_{\tau-2}\beta_{\tau-1}\beta_\tau + \cdots + \beta_1\beta_2\beta_4 + \beta_1\beta_2\beta_3 \\
&\ \vdots \\
\sigma_\tau &= \beta_1\beta_2\ldots\beta_\tau
\end{aligned} \tag{2.4.11}$$

In order to solve the coefficients of the error locator polynomial, one has to solve the Newton's identities [47, p. 130]:

$$\begin{aligned}
\mathcal{S}_1 + \sigma_1 &= 0 \\
\mathcal{S}_2 + \sigma_1\mathcal{S}_1 &= 0 \\
\mathcal{S}_3 + \sigma_1\mathcal{S}_2 + \sigma_2\mathcal{S}_1 + \sigma_3 &= 0 \\
&\ \vdots \\
\mathcal{S}_\tau + \sigma_1\mathcal{S}_{\tau-1} + \cdots + \sigma_{\tau-1}\mathcal{S}_2 + \sigma_\tau\mathcal{S}_1 &= 0
\end{aligned} \tag{2.4.12}$$

The objective is to find the minimum degree polynomial $\sigma(x)$ whose coefficients satisfy these Newton identities.

Various algorithms have been proposed to find these coefficients: the Peterson-Gorenstein-Zierler algorithm [24], the Berlekamp-Massey algorithm [6] and Euclid's algorithm [54]. For the purpose of this thesis and because of the complexity of these algorithms, we stop the description for finding the error-locator polynomial coefficients here. For a detailed description, we encourage the reader to consult Moreira and Farrell [47] or any of the papers these algorithms were introduced in [47, 24, 6, 54].

Example 2.4.8. Take the received codeword $W' = [1,1,0,0,0,1,0,0,0,0,0,1,1,0,1]$ and its syndromes $\mathcal{S}_1 = \alpha^4$, $\mathcal{S}_2 = \alpha^8$, $\mathcal{S}_3 = \alpha^{13}$ and $\mathcal{S}_4 = \alpha$ from Example 2.4.7. Using the Berlekamp-Massey algorithm we obtain the error locator polynomial $\sigma(x) = \alpha^{12}x^2 + \alpha^4 x + 1$ with $\sigma_0 = 1$, $\sigma_1 = \alpha + 1 = \alpha^4$ and $\sigma_2 = \alpha^3 + \alpha^2 + \alpha + 1 = \alpha^{12}$. $\triangleleft$

2.4.2.2.3 Finding Roots of Error Locator Polynomial The roots of the error locator polynomial $\sigma(x)$ are $\beta_\tau^{-1}, \ldots, \beta_2^{-1}, \beta_1^{-1}$, the inverse of the error location numbers.

One way of solving the roots for $\sigma(x)$ is by brute-forcing the finite field elements in the error locator polynomial and check whether the following condition holds:

$$\sigma(\alpha^i) = 0 \tag{2.4.13}$$

This is called the Chien search, as introduced by Chien [15].

- If this condition holds, then there was an error at the inverse position of i, i.e. in position $n - i$.
- If this condition does not hold, there was no error.

Example 2.4.9. Take the error locator polynomial $\sigma(x) = \alpha^{12}x^2 + \alpha^4 x + 1$ with $\sigma_0 = 1$, $\sigma_1 = \alpha^4$ and $\sigma_2 = \alpha^{12}$ as computed in Example 2.4.8. Evaluating $\sigma(x)$ for $x = \alpha$, $x = \alpha^2$, $\ldots$, $x = \alpha^n$ (where $n = 2^m - 1$, Formula 2.4.2) gives the following set of equations:

$$\begin{aligned}
\sigma(\alpha) &= \alpha^{12}(\alpha)^2 + \alpha^4(\alpha) + 1 \\
&= (\alpha^3 + 1) + (\alpha^2 + \alpha) + 1 \\
&= \alpha^3 + \alpha^2 + \alpha + 2 \\
&= \alpha^3 + \alpha^2 + \alpha \\
&\neq 0 \\
\sigma(\alpha^2) &= \alpha^{12}(\alpha^2)^2 + \alpha^4(\alpha^2) + 1 \\
&= \alpha^3 + \alpha^2 + \alpha + 1 \\
&\neq 0 \\
&\vdots \\
\sigma(\alpha^{15}) &= \alpha^{12}(\alpha^{15})^2 + \alpha^4(\alpha^{15}) + 1 \\
&= \alpha^3 + \alpha^2 + 1 \\
&\neq 0
\end{aligned}$$

Solving these equations we find $\sigma(\alpha^5) = 0$ and $\sigma(\alpha^{13}) = 0$. Hence, the roots of the error locator polynomial $\sigma(x) = \alpha^{12}x^2 + \alpha^4 x + 1$ are $\beta_2^{-1} = 5$ and $\beta_1^{-1} = 13$ for which Formula 2.4.13 holds. $\triangleleft$

2.4.2.2.4 Error Correction Once the roots $\beta_\tau^{-1}, \ldots, \beta_2^{-1}, \beta_1^{-1}$ of the error locator polynomial $\sigma(x) = \sigma_\tau x^\tau + \cdots + \sigma_1 x + \sigma_0$ are found, we can obtain the error correction polynomial:

$$\mathbf{e}'(x) = x^{n-\beta_\tau^{-1}} + \cdots + x^{n-\beta_2^{-1}} + x^{n-\beta_1^{-1}} \tag{2.4.14}$$

Finally, we can obtain the recovered codeword polynomial $\mathbf{w}''(x)$:

$$\mathbf{w}''(x) = \mathbf{w}'(x) + \mathbf{e}'(x) \tag{2.4.15}$$

Example 2.4.10. Take the received codeword $W' = [1,1,0,0,0,1,0,0,0,0,0,1,1,0,1]$ ($\mathbf{w}'(x) = x^{14}+x^{13}+x^9+x^3+x^2+1$) and the roots $\beta_2^{-1} = 5$ and $\beta_1^{-1} = 13$ of the error locator polynomial $\sigma(x) = \alpha^{12}x^2 + \alpha^4 x + 1$ from the previous exercises. The error correction polynomial is given by using Formula 2.4.14:

$$\begin{aligned}\mathbf{e}'(x) &= x^{15-5} + x^{15-13} \\ &= x^{10} + x^2\end{aligned}$$

This gives us the error correction vector $E' = [0,0,0,0,1,0,0,0,0,0,0,0,1,0,0]$. We can see that $E' = E$ (Exercise 2.4.7). Finally, we can obtain the original codeword polynomial using Formula 2.4.15:

$$\begin{aligned}\mathbf{w}''(x) &= (x^{14}+x^{13}+x^9+x^3+x^2+1) + (x^{10}+x^2) \\ &= x^{14}+x^{13}+x^{10}+x^9+x^3+1\end{aligned}$$

This gives us the recovered codeword $W'' = [1,1,0,0,1,1,0,0,0,0,0,1,0,0,1]$. We can see that $W'' = W$ (Exercise 2.4.7). Hence, we recovered the original message $M = [1,1,0,0,1,1,0]$. $\triangleleft$

2.5 Fuzzy Extractors

Fuzzy Extractors (FEs) were first introduced to turn biometric information into keys usable for cryptographic applications [19]. This biometric data, for example iris scans or fingerprints can be used as a key, where the key must be derived from measurements that are slightly different. With the use of a FE, this data is turned into a key with nearly uniform randomness and helper data to recover this key using another measurement of the biometrics. Even though FEs were first introduced for biometric information, a FE can be used to produce cryptographic keys from any key that is not precisely reproducible and not distributed uniformly given that these keys are 'close enough' to each other. For the purpose of this thesis, FEs are used to correct noisy PUF responses into unpredictable keys.

Using a generation procedure, the FE can convert the biometric input into an key that is unpredictable. Moreover, using a generation procedure the FE can produce public information about the biometric input such that this input can exactly be recovered using error-correction codes. These two steps are performed in a Strong Extractor (SE) and a (), respectively. A reproduction procedure can recover the original biometric input from the public information produced by the generation procedure of the FE. Using the same Strong Extractor (SE) as was used in the generation procedure, this biometric input is converted into the same key.

2.5.1 Strong Extractor

The Strong Extractor (SE) converts the noisy data that is not distributed uniformly into a key that is unpredictable. We define a SE as follows [19, p. 528]:

Definition 2.16 (Strong Extractor)**.** *An efficient* (n, m', k, ϵ)*-strong extractor is a polynomial time probabilistic function* $\mathbf{Ext}(W, X) : \{0,1\}^n \to \{0,1\}^n$ *such that for all min-entropy* m' *distributions* W *we have:*

$$\mathbf{SD}(\langle \mathbf{Ext}(W, X), X \rangle, \langle U_l, X \rangle) \leq \epsilon, \textit{ where}$$

U_l *denotes the uniform distribution on an* l*-bit binary string and* $\mathbf{Ext}(W, X)$ *stands for applying* $\mathbf{Ext}$ *to* W *using uniformly distributed randomness* X*.*

In other words, a SE can extract a uniformly distributed key from non uniformly distributed input W and uniformly distributed randomness X.

Practically, to obtain a high security level, strong assumptions about the min-entropy of the randomness source have to be made. This is often impossible [43, p. 306] and multiple practical solutions have been proposed. Moreover, because we need to use uniformly distributed randomness X which has to be shared between generation and reproductions, large entropy losses need to be taken into account. This makes the overall key generation as defined in Definition 2.16 impractical.

Some solutions are to use a cryptographic hash function [34] or a Pseudo-Random Function (PRF). For example, one can append a random variable to the noisy data and hash this into a key (salting). However still, one has to make strong assumptions about the min-entropy of the noisy data.

We call these constructions Average-case Extractors (AcEs), which we define as follows [20, p. 10]:

Definition 2.17 (Average-case Extractor)**.** *Let* $\mathbf{Ext}(W, X) : \{0,1\}^n \to \{0,1\}^l$ *be a polynomial time probabilistic function which uses* r *bits of randomness. We say that* $\mathbf{Ext}(W, X)$ *is an efficient average-case* (n, m, l, ϵ)*-strong extractor if for all pairs of random variables* (W, I) *such that* W *is an* n*-bit string satisfying* $\tilde{\mathbf{H}}_\infty = (W \mid I) \geq m$*, we have*

$$\mathbf{SD}(\langle \mathbf{Ext}(W, X), X, I \rangle, \langle U_l, X, I \rangle) \leq \epsilon,$$

where X *is uniform on* $\{0,1\}^r$*.*

In other words, if there is enough entropy in W (taking into account the entropy loss introduced by I) and there is enough entropy in X, $\mathbf{Ext}(W, X)$ is indistinguishable from random (U_l).

2.5.2 Secure Sketch

The Secure Sketch (SS) converts the noisy data that is not distributed uniformly into helper data that can be used to recover that same noisy data. Almost always, SSes use encoding of (a composition of) error-correcting codes as described in Section 2.4.

Let $\mathcal{W}$ be the set of all possible noisy and non uniformly distributed vectors with distance function $\mathbf{dist}(W, W')$ ($W, W' \in \mathcal{W}$) as described in Section 2.2.1. We define a SS as follows [19, p. 529]:

Definition 2.18 (). *An $(\mathcal{W}, m, m', t)$-secure sketch is a randomized map $\mathbf{SS}(X) : \mathcal{W} \to \{0,1\}^*$ with the following properties:*

1. *There exists a deterministic recovery function $\mathbf{Rec}(W', H)$ allowing to recover vector W from its sketch $H = \mathbf{SS}(W)$ and any vector W' close to W: for all $W, W' \in \mathcal{M}$ satisfying $\mathbf{dist}(W, W') \leq t$, we have $\mathbf{Rec}(W', \mathbf{SS}(W)) = W$.*

2. *For all random variables $W \in \mathcal{W}$ with min-entropy m, the average min-entropy of W given $\mathbf{SS}(W)$ is at least m'. That is $\tilde{\mathbf{H}}_\infty = (W \mid \mathbf{SS}(W)) \geq m'$.*

In other words, we are able to recover a vector W using another vector W' close to W and the helper data H generated from the secure sketch $H = \mathbf{SS}(W)$. Moreover, the entropy loss during the construction of $H = \mathbf{SS}(W)$ is $m - m'$.

When using an error-correcting code $\mathcal{C}(n, k, t)$ for the SS, knowledge of the helper data H does not fully disclose the entropy of W, only $n - k$ bits of this helper data. Thus, we can use, store and communicate H publicly where it still has $\mathbf{H}(W) - (n - k)$ bits of entropy left in H [43, p. 305]. The bit error probability p_e and entropy ρ of the PUF determine the (composition of) error-correcting codes used in the design of the SS. This should be optimized such that no information is leaked about the key.

2.5.3 Fuzzy Extractor

Now that we have described a SE and a SS, we can start defining a FE. As mentioned before the FE composes a generation procedure and a reproduction procedure.

We define a FE as follows [19, p. 530]:

Definition 2.19 (Fuzzy Extractor). *A $(\mathcal{W}, m, l, t, \epsilon)$-fuzzy extractor is a given by two procedures $(\mathbf{Gen}, \mathbf{Rep})$.*

1. *$\mathbf{Gen}(W)$ is a probabilistic generation procedure, which on input $W \in \mathcal{W}$ outputs an "extracted" string $R \in \{0,1\}^l$ and a public string H. We require that for any distribution W on $\mathcal{W}$ of min-entropy m, if $\langle R, H \rangle \leftarrow \mathrm{Gen}(W)$, then we have $\mathbf{SD}(\langle R, H \rangle, \langle U_l, H \rangle) \leq \epsilon$.*

2. **Rep**(W', H) *is a deterministic reproduction procedure allowing to recover* R *from the corresponding public string* H *and any vector* W' *close to* W*: for all* $W, W' \in \mathcal{W}$ *satisfying* $\mathbf{dist}(W, W') \leq t$*, if* $\langle R, H \rangle \leftarrow \mathbf{Gen}(W)$*, then we have* $\mathbf{Rep}(W', H) = R$*.*

In other words, a random binary variable R can be constructed using noisy and non-random input W using a generation procedure. Moreover, using a reproduction procedure, this randomness R can be recovered, given a second noisy and non-random input W' and the helper data H as constructed by the generation procedure.

We can use Definition 2.17 (Average-case Extractor), Definition 2.18 () and Definition 2.19 (Fuzzy Extractor) to prove that we can construct a FE from SS [20, p. 13]:

Lemma 2.1 (Fuzzy Extractor from)**.** *Assume* $\mathbf{SS}(X)$ *is a* $(\mathcal{W}, m, m', t)$*-secure sketch with recovery procedure* $\mathbf{Rec}(W', H)$*, and let* $\mathbf{Ext}(W, X)$ *be an average case* (n, m', k, ϵ)*-strong extractor. Then the following* $(\mathbf{Gen}, \mathbf{Rep})$ *is a* $(\mathcal{W}, m, l, t, \epsilon)$*-fuzzy extractor:*

1. $\mathbf{Gen}(W)$*: set* $H = \langle \mathbf{SS}(W), X \rangle$*,* $R = \mathbf{Ext}(W, X)$*, output* $\langle R, H \rangle$
2. $\mathbf{Rep}(W', \langle H, X \rangle)$*: recover* $W = \mathbf{Rec}(W', H)$ *and output* $R = \mathbf{Ext}(W, X)$*.*

Proof. From Definition 2.18 ()

$$H_\infty(W|SS(W)) \geq m'$$

And since $\mathbf{Ext}(W, X)$ is an average-case (n, m', k, ϵ)-strong extractor (Definition 2.17), from Definition 2.19 (Fuzzy Extractors) we get:

$$\mathbf{SD}(\langle \mathbf{Ext}(W, X), \mathbf{SS}(W), X \rangle, \langle U_l, \mathbf{SS}(W), X \rangle) = \mathbf{SD}(\langle R, H \rangle, \langle U_l, H \rangle) \leq \epsilon$$

□

Corollary 2.1 (Fuzzy Extractor from)**.** *If* **Rec** *is an* $(\mathcal{W}, m, m', t)$*-secure sketch and* **Ext** *is an* $(n, m' - \log_2(\frac{1}{\delta}), l, \epsilon)$*-strong extractor, then the above construction from Lemma 2.1* $(\mathbf{Gen}, \mathbf{Rep})$ *is a* $(\mathcal{W}, m, l, t, \epsilon + \delta)$*-fuzzy extractor.*

2.6 Authenticated Encryption

Before AE, protocol designers used a generic composition paradigm for which they concatenate a privacy-only encryption scheme with a Message Authentication Code (MAC) [5, 33]. However, this naive approach demands multiple procedures which substantially reduces efficiency of the protocols. AE improves efficiency by providing confidentiality, integrity and authenticity into a single, compact mode [56]. However, there was still a need to efficiently authenticate a message header belonging to the plaintext or cipher-text. Authenticated Encryption with Associated Data (AEAD)[3] provides for this by additional authentication of data other than the plaintext. We define an AEAD-scheme as follows [56, p. 4]:

Definition 2.20 (AEAD-scheme)**.** *An authenticated-encryption scheme with associated data (AEAD-scheme) is a three-tuple*

$$\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D}).$$

Associated to Π *are sets of strings* $\mathcal{N} \subseteq \{0,1\}^*$ *indicating the nonce,* $\mathcal{M} \subseteq \{0,1\}^*$ *indicating the message and* $\mathcal{A}^{\mathcal{D}} \subseteq \{0,1\}^*$ *indicating the associated data, for example a header.*

- *The key space* $\mathcal{K}$ *is a finite nonempty set of strings.*
- *The encryption algorithm* $\mathcal{E}$ *is a deterministic algorithm that takes strings* $K \in \mathcal{K}$*,* $N \in \mathcal{N}$*,* $M \in \mathcal{M}$ *and* $A \in \mathcal{A}^{\mathcal{D}}$ *and returns string* $\langle C, T\rangle = \mathcal{E}_K^{N,A}(M) = \mathcal{E}_K(N, A, M)$.
- *The decryption algorithm* $\mathcal{D}$ *is a deterministic algorithm that takes strings* $K \in \mathcal{K}$*,* $N \in \mathcal{N}$*,* $A \in \mathcal{A}^{\mathcal{D}}$*,* $C \in \{0,1\}^*$ *and* $T \in \{0,1\}^*$ *and returns* $\mathcal{D}_K^{N,A}(\langle C, T\rangle)$*, which is either a string in* $\mathcal{M}$ *or the distinguished symbol* INVALID.

We require that $\mathcal{D}_K^{N,A}(\mathcal{E}_K^{N,A}(M)) = M$ *for all* $K \in \mathcal{K}$*,* $N \in \mathcal{N}$*,* $M \in \mathcal{M}$ *and* $A \in \mathcal{A}^{\mathcal{D}}$.

[3] In this thesis, we refer to AEAD by referring to AE.

CHAPTER

THREE

PROTOCOL DESIGN

In this chapter we describe the proposed privacy-preserving authentication protocol, the Concealing KETJE Protocol (CKP). In this protocol a trusted server and a set of deployed devices will authenticate each other where devices require anonymous authentication such that they are untraceable. As a starting point we take the PUF-based privacy-preserving authentication protocol as proposed by Aysu et al. [3]. However, we want to correct a few design-flaws in the protocol and improve its overall performance. First, the protocol proposed by Aysu et al. makes use of a Fuzzy Extractor (FE) that can be broken using linear equation analysis as pointed out by Becker [3, p. 12]. Moreover, the FE makes use of a pre-shared key (PSK), which increases the overhead of the protocol. We design a new FE that has enough entropy in its output and that does not use a PSK. Second, the protocol uses two cryptographic primitives, namely Symmetric Key Encryption (SKE) and Pseudo-Random Function (PRF), of which for the SKE the SIMON block-cipher is used, an encryption scheme introduced by the untrustworthy National Security Agency (NSA) [18, p. 113] (Snowden revelations [25]). We replace these primitives with a single, compact cryptographic primitive, namely Authenticated Encryption with Associated Data (AEAD).

In Section 3.1 we present the security considerations that play a role in the design of the proposed protocol, which will be presented and elaborated in Section 3.3. In Section 3.2 we present the considerations that we need to take due to the available hardware. Section 3.4 describes the 3-1 Double Arbiter PUF (DAPUF) as is used in our protocol. The extraction of the device credentials together with the construction of the helper data is described in the Reverse FE (RFE), Section 3.5. Finally, our choice for the lightweight AEAD-scheme, KETJE, will be described in Section 3.6.

3.1 Security Considerations

In this section we present the security considerations that play a role in the design of the proposed protocol. We identify the operational and cryptographic properties that a device to be authenticated should adhere to, we present the trust- and attacker model and we present the considerations we take providing the available hardware.

3.1.1 Operational and Cryptographic Properties

For the security considerations of the operational and cryptographic properties we follow Lee et al. [38]. These properties describe what considerations need to be made when designing a privacy-preserving authentication protocol in general and an untraceable device in particular.

3.1.1.1 Scalability

Many protocols face the pitfall that they are not scalable because of the computational workload on the server that increases linearly with the number of devices. Considering that these authentication protocols often work for a large amount of devices, a thoughtful design is necessary. Our protocol is also subject to this risk, which means that we elaborate on the design rationale to minimize this risk. More about this in Section 6.3.

3.1.1.2 Anti-cloning

It should not be possible to clone a device. One property that a device needs to have is that it should have a key that is unique in the sense that it all the bits of the credential should be unpredictable. This way, if an attacker succeeds to crack one of the devices, he/she cannot use this secret to clone any of the other devices. We base our countermeasure mainly on the use of a Physically Unclonable Function (PUF). At every authentication the device credentials are freshly generated to an unpredictable value.

3.1.1.3 Security Against the Replay Attack

This property implies that an attacker should not be able to authenticate a device using a replayed message (i.e. he/she should not be able to successfully carry out Man-in-the-Middle (MitM) attacks). This implies that all communication over the channel should have enough entropy. We base our countermeasure against this attack mainly on the use of a FE. The FE provides for a fresh key at every authentication-try which has enough entropy considering the transmitted messages. Moreover, the AEAD-scheme provides for confidentiality, integrity and authenticity of the messages in the communication channel.

3.1.1.4 Security Against the Tracking Attack

It should not be possible for an attacker to trace a device over multiple authentications. Moreover, it should not be possible to identify a device by probing the device with challenges. This property ensures that our protocol is privacy-preserving with respect to outsiders, which means that a device (and its owner) remain anonymous. We base our countermeasure against this attack on a True Random Number Generator (TRNG), by using the unpredictable PUF responses both as input (seed) and as output of the TRNG.

3.1.1.5 Backward/Forward Un-traceability

This property, that is stronger than the un-traceability property, implies that it should not be possible to track a device in past or future communications, provided that an attacker has cracked a device. If an attacker manages to recover a key from a device, he should not be able to identify a particular device in the past or in the future. This property ensures that a device (an its owner) remain anonymous always. We base our countermeasure on the authenticate-before-identify strategy we adopted. Devices do not carry, store or communicate device specific identification numbers (IDs), which is not needed because of the use of a PUF that ensures authenticity of the device. Moreover, all communication appears random to an attacker.

3.1.2 Trust Model

In order to roll out devices, we present a trust model. Our trust model is mainly based on Aysu et al. [3], the starting point of our proposed protocol. We identified the following trust bases:

- Devices are enrolled in a secure environment using a one-time interface.
- A trusted server and a number of devices will authenticate each other while devices need to remain anonymous.
- Our channel is ideal, i.e. no errors will occur due in the Binary Symmetric Channel (BSC) as described in Section 2.4.
- After enrollment, the server remains trusted but devices are subjected to an attacker.
- The attacker may not know the identity of a device such that the device cannot be tracked.

3.1.3 Attacker Model

In order to prove the security of the proposed protocol, an attacker model needs to be constructed. Our attacker model is mainly based on Aysu et al. [3], the starting point of our proposed protocol.

We identified that the attacker may have two goals:

1. The attacker may want to impersonate a device which will result in a violation of the security.
2. The attacker may want to trace devices in between authentications which will result in a violation of the privacy.

These two main goals will be the subjects of the proofs as described in Chapter 4.

We identified the following permissions and constraints for the attacker:

- An attacker can modify all communication between the server and devices.
- An attacker can know the result of the authentication.
- An attacker can access the Non-Volatile Memory (NVM) of the devices.
- An attacker cannot modify data stored in the NVM of the devices.
- An attacker cannot perform implementation attacks on the device and the server.
- An attacker cannot reverse engineer the PUF such that he can predict PUF responses.
- An attacker does not have access to intermediate values on the device (i.e. the registers on the device).
- An attacker cannot physically trace every device in between authentications.
- An attacker cannot use other (non-cryptographic) mechanisms to identify a device (e.g. the one proposed by Lee et al. [37]).

3.2 Available Hardware

In order to give performance results of the proposed protocol, we implement the device on a Zedboard by Avnet Inc. [2]. The basic specifications of the Zedboard are given in Appendix A. The main operating chip on the Zedboard is the Zynq®-7000 All Programmable System on Chip (SoC) by Xilinx Inc. [64]. This SoC is composed of a Processing System (PS) with two Advanced RISC Machine (ARM) cores and 28 nm Programmable Logic (PL) that is equivalent to the Xilinx 7-series Field Programmable Gate Arrays (FPGAs).

The Zedboard does not have Static RAM (SRAM) that can be power-cycled, which means that we are restricted in the use of SRAM PUFs. We propose to use an existing and recently proposed PUF by Machida et al. [39], which has promising results for the design rationale of our proposed protocol.

3.3 Protocol

The proposed authentication protocol is illustrated in Figure 3.2, its setup phase is illustrated in Figure 3.1. The protocol is based on a PUF that produces noisy, but recoverable, responses on equal challenges due to the unique physical characteristics of the Integrated Circuit (IC) [42]. Because of this behavior, the PUF is identifiable from other PUFs. A FE can extract a key from this noisy data produced by the PUF using helper data generated from a previous key-extraction [19]. However, the recovery procedure is of a higher complexity than the generation of the helper data that is used for this reconstruction. A reverse FE reverses this behavior by placing the helper data generation in the device and the more complex reconstruction in the server [29]. In order to preserve privacy, the device credential is updated at each successful authentication, which results in fresh PUF responses, and thus fresh keys.

The setup procedure is used to synchronize the PUF response of the the device with the server. The responses in the server database will be used to exhaustively search for a matching device. The setup procedure is illustrated in Figure 3.1 and works as follows. In a trusted environment, the server produces a random challenge X^1. The device uses this challenge to produce a PUF response Y^1 which is being sent to the server. The challenge is being stored in the device non-volatile memory. The server stores the response in a database on index n, indicating the number of the device. Notice that the response is stored at Y and Y^{old} in order to prevent desynchronization.

Server $\mathcal{S}(\{\langle Y, Y^{old}\rangle\}_n)$		**Device** $\mathbf{Dev}_i(\mathbf{puf}_i(\,\cdot\,), X)$
$X^1 \leftarrow \mathbf{TRNG}$		
	$\xrightarrow{X^1}$	$Y^1 \leftarrow \mathbf{puf}_i(X^1)$ $X := X^1$
$\langle Y, Y^{old}\rangle_n := \langle Y^1, Y^1\rangle$ $n := n + 1$	$\xleftarrow{Y^1}$	

Figure 3.1: Setup phase.

The authentication phase as illustrated in Figure 3.2 works as follows. First, the server generates an unpredictable challenge A and sends this to the device. The device uses the challenge X stored in its non-volatile memory to produce a PUF response $Y^{1'}$. From this PUF response, helper data H and an unpredictable key R is generated using the FE's generation procedure **FE.Gen**. Consecutively, a new challenge X^2 is randomly generated by the device such that it can be updated on a successful authentication. This challenge is fed to the device's PUF in order to receive a new PUF response Y^2. Following, a nonce N is randomly generated such that the PUF response can be encrypted using the AEAD-scheme. The resulting cipher-text C^1, its tag T^1 and the nonce N will be sent to the server. The server performs an exhaustive search over the database, recovering a key for each index. These keys are used to try to decrypt the cipher-text C^1 using the tag T^1, challenge

$$
\begin{array}{lcl}
\textbf{Server } \mathcal{S}(\{\langle Y, Y^{old}\rangle\}_n) & & \textbf{Device } \mathbf{Dev}_i(\mathbf{puf}_i(\,\cdot\,), X) \\
\hline
A \leftarrow \mathbf{TRNG} & & \\
 & \xrightarrow{A} & Y^{1'} \leftarrow \mathbf{puf}_i(X) \\
 & & \langle R, H\rangle \leftarrow \mathbf{FE.Gen}(Y^{1'}) \\
 & & X^2 \leftarrow \mathbf{TRNG} \\
 & & Y^2 \leftarrow \mathbf{puf}_i(X^2) \\
 & & N \leftarrow \mathbf{TRNG} \\
 & & \langle C^1, T^1\rangle \leftarrow \mathcal{E}_R^{N\|0,A}(Y^2) \\
T^2 \leftarrow \mathbf{TRNG} & \xleftarrow{\langle H, N, C^1, T^1\rangle} & \\
\text{for } 0 \leq i < n : & & \\
\quad R = \mathbf{FE.Rec}(Y, H) & & \\
\quad \text{if } Y^2 \leftarrow \mathcal{D}_R^{N\|0,A}(\langle C^1, T^1\rangle) : & & \\
\quad\quad \langle\,\cdot\,, T^2\rangle \leftarrow \mathcal{E}_R^{N\|1,A}(\,\cdot\,) & & \\
\quad\quad \langle Y, Y^{old}\rangle_i := \langle Y^2, Y\rangle & & \\
\text{"if no device was authenticated" :} & & \\
\quad \text{"repeat search with old values"} & & \\
 & \xrightarrow{T^2} & \langle\,\cdot\,, T^{2'}\rangle \leftarrow \mathcal{E}_R^{N\|1,A}(\,\cdot\,) \\
 & & \text{if } T^{2'} == T^2 : \\
 & & \quad X := X^2 \\
\hline
\end{array}
$$

Figure 3.2: Authentication phase. $|A|, |H|, |N|, |C^1|, |T^1|, |T^2| \geq k$ and PUF responses Y should contain enough entropy w.r.t. H s.t. $|R| \geq k$.

A and nonce N. If there is a successful authentication, the server produces another tag T^2 using $\mathcal{E}$, but with nonce $N^2 \parallel 1$ instead of $N^2 \parallel 0$ in order to create another instance of $\mathcal{E}$. This tag is sent to the device. Moreover, the server updates the old PUF-response Y with the new PUF response Y^2. If there were no successful authentications, the server repeats the procedure over the previous PUF responses in the database. If after this there were still no successful authentications, the server responds with a random value for T^2. Finally, the device checks the tag T^2 with its own produced tag in order to reveal whether the authentication succeeded. If the authentication succeeded, the device updates the old challenge X with the new challenge X^2.

The remainder of this chapter is dedicated to present a generic approach to create any instance of CKP. This generic approach requires the quality of PUF responses $\langle p_e, \rho\rangle$, the desired maximum for the failure rate of the authentications p_{fail} and the desired security level k in order to design the instance. We use the presented examples to construct an instance that forms the basis of our proof of concept in Chapter 5.

3.4 3-1 Double Arbiter PUF

The type of PUF used in the protocol motivates most of the other design parameters for the rest of the protocol. For example, depending on the bit-error-probability p_e of a PUF response-bit, the inter- and intra-distances of the PUF responses, the entropy of the PUF responses ρ and the desired maximum for the failure rate of the authentications p_{fail}, both the number of PUF responses as well as the type and size of error-correcting codes is motivated.

We implement the DAPUF as proposed by Machida et al. [39] because its characteristics are promising for the parameters of our protocol. We use 1275 PUF responses on 64-bit challenges, of which 40 bits are used for the challenge, 12 bits are used to obtain the 1275 unique PUF responses and 12 bits are used to produce random numbers, including a seed for the TRNG that is updated at the beginning of every authentication. More about this design rationale in Section 3.5.1.

This section first presents the design of the DAPUF before elaborating on the main characteristics of the PUF as were described in Section 2.3.

3.4.1 DAPUF Design

In Section 2.3.5.2 we have discussed the arbiter PUF which was illustrated in Figure 2.1. Recent studies have shown that Machine Learning (ML) attacks can predict future PUF responses [58, 57], violating the unpredictability characteristic of a PUF as is described in Definition 2.13. Hence, Machida et al. proposed to alter the design of arbiter PUFs in order to prevent ML attacks having effect.

In Figure 2.1 from Section 2.3.5.2, we call the collection of switch blocks a selector chain. Figure 3.3 illustrates the DAPUF as proposed by Machida et al.. The DAPUF acts on 64-bit challenges, which means that a selector chain contains 64 switch blocks. The DAPUF is composed of three of these selector chains all acting on the same challenge X. Using an 'enable' signal E (E_L and E_R), the competition is started between the left signals E_L and the right signals E_R. For each of the combination of left- or right signals an arbiter is used to measure which signal arrived first at the arbiter. After measuring these race conditions, the results are exclusive-OR (XOR)'ed to collect the 1-bit PUF response Y. By challenging the DAPUF with 1275 different challenges, we obtain a 1275-bit PUF response.

We call the PUF class of the proposed DAPUF $\mathcal{P}_{3\text{-}1}$.

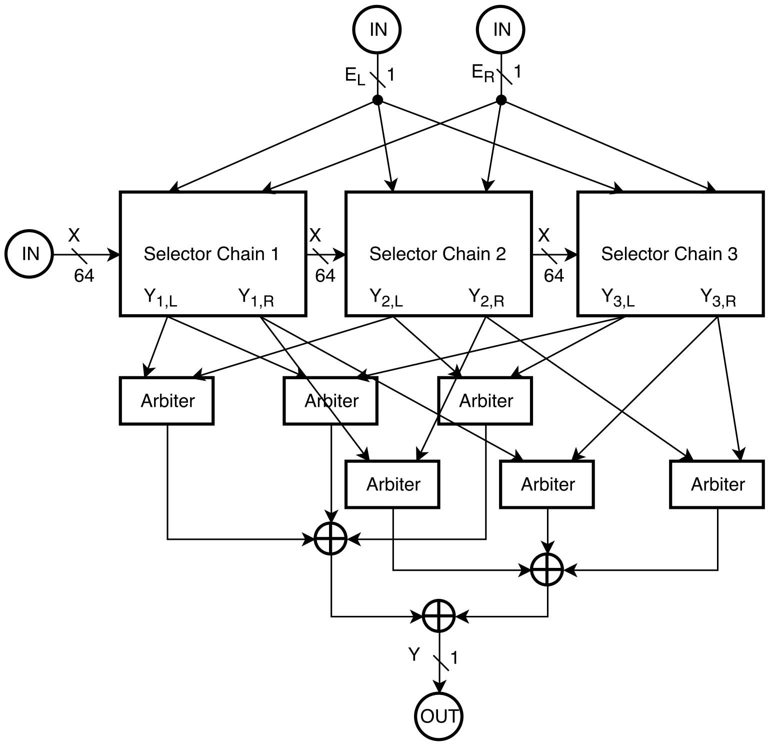


Figure 3.3: DAPUF as proposed by Machida et al. [39]. $\oplus$ denotes a bitwise XOR, (IN) denotes the input of the DAPUF and (OUT) denotes the output of the DAPUF.

3.4.2 Intra-distance and Reproducibility

As described in Section 2.3.1, the intra-distance provides reproducibility of any unique DAPUF instance $\mathbf{puf}_{0\leq i<n} \in \mathcal{P}_{3\text{-}1}$ (where n is the total number of DAPUFs in the DAPUF class $\mathcal{P}_{3\text{-}1}$), which means that if two measurements are performed on the same DAPUF, then these responses are with high probability very close to each other.

The reproducibility results (its inverse is called 'steadiness') are given in Table 3.1 [40]. In the table, steadiness is calculated by challenging the DAPUF instance **puf** a number of m times with a set of n equal challenges X. Of the m n-bit responses, the Hamming distances $\mathbf{HD}(Y, Y')$ between two arbitrary PUF responses $Y, Y' \leftarrow \mathcal{Y}$ is calculated (thus a total of $\binom{m}{2}$ combinations) and averaged. These distances are equivalent to the intra-distance as is described in Definition 2.7. Finally, the calculated average is divided by the bit-length n of the responses. This way, we immediately find the bit-error-probability p_e of DAPUF class $\mathcal{P}_{3\text{-}1}$. This is the average probability that a response bit is different between challenges. Ideally, the average steadiness is 0%. From the table we see that the average steadiness is approximately 12%, which means that the bit-error-probability p_e is 0.12. These results have been achieved by taking $m = 128$ and $n = 128$. The reproducibility is $100\% - \text{steadiness} = 88\%$.

Table 3.1: Results of the overall evaluation of the 3-1 Double Arbiter PUF [40].

Metric	FPGA	Results
Prediction rate [%] (with 1,000 training data)	A	56.47
	B	57.45
	C	56.75
Steadiness [%]	A	14.11
	B	10.93
	C	10.35
Uniqueness [%]	A with B	50.60
	B with C	51.34
	C with A	48.78
Randomness [%]	A	55.68
	B	52.54
	C	53.59

3.4.3 Inter-distance and Uniqueness

As described in Section 2.3.2, the inter-distance provides uniqueness of any DAPUF instance $\mathbf{puf}_{0 \leq i < n}$ in the DAPUF collection $\mathcal{P}_{3\text{-}1}$ (where n is the total number of DAPUFs in the DAPUF class $\mathcal{P}_{3\text{-}1}$), which implies that responses of measurements performed on different DAPUFs are with high probability far apart.

The uniqueness results are given in Table 3.1 [40]. In the table, uniqueness is calculated by challenging two DAPUF instances $\mathbf{puf}_i$ and $\mathbf{puf}_j$ $(i \neq j)$ on two distinct FPGAs a number of n times using randomly chosen challenges X. Of these two n-bit responses $Y, Y' \leftarrow \mathcal{Y}$, the the Hamming distances $\mathbf{HD}(Y, Y')$ is calculated. This distance is equivalent to the inter-distance as is described in Definition 2.9. Finally, the distance metric is divided by the bit-length n of the two responses. Ideally, the average uniqueness between DAPUF instances is 50%. From the table we see that the average uniqueness is approximately 50%, which is close to ideal. These results have been achieved by taking $n = 5,000$ measurements.

3.4.4 Unclonability and Unpredictability

As described in Section 2.3.4, unclonability assures that physically and technically, a DAPUF instance $\mathbf{puf}_{i'} \in \mathcal{P}_{3\text{-}1}$ is difficult (or even impossible) to create from an other DAPUF instance $\mathbf{puf}_i \in \mathcal{P}_{3\text{-}1}$.

Because of the characteristics explained in Section 2.3.5.2, arbiter PUFs and thus also DAPUFs are hard to clone. There are several un-plausible techniques an attacker might try [36].

- An attacker can try to clone the PUF by remembering all Challenge-Response Pairs (CRPs). However, this is implausible because this requires applying an exponential amount of challenges.

- An attacker can try to reproduce the PUF such that the behavior is equal to the original PUF. However, this is implausible because of the random variations that are intrinsically introduced in the manufacturing process.
- An attacker can probe the PUF physically such that delays can be measured and a timing model can be constructed to predict future PUF responses. However, probing with sufficient precision is likely to be very difficult, and will likely cause the delays to be influenced by the probe.
- An attacker could build a non-invasive model, a so called "virtual counterfeit". However, until now, no such model has been proposed.

As described in Section 2.3.4, unpredictability ensures that unobserved responses remain sufficiently random, even after observing responses to other challenges on the same DAPUF instance. This characteristic applies both to the prediction rate of the DAPUF class $\mathcal{P}_{3\text{-}1}$, as well as the randomness of the DAPUF class $\mathcal{P}_{3\text{-}1}$.

First, from Table 3.1 we see that the prediction rate is approximately 57%, which approximates a random guess (i.e. 50%). This is a considerable improvement for arbiter PUF constructions because the prediction rate of conventional arbiter PUFs is 86% [40, p. 8].

Second, from the table we see that the randomness is approximately 54%. This randomness is calculated by challenging a DAPUF instance **puf** a number of n times with randomly chosen challenges X. Then, the Hamming weight $\mathbf{HW}(Y)$ of the result is calculated, giving the number of ones in the PUF response Y. Finally, this number is divided by the number of challenges n, giving the probability that a PUF response-bit is $Y_i = 1$. The randomness of 54% ($\mathbf{Pr}(Y_i = 1) = 0.54$) has been achieved by taking $n = 2^{16}$ measurements. From this result, we can calculate the entropy of the PUF responses ρ using the binary entropy function $\mathbf{h}(p)$ from Formula 2.2.1, Section 2.2.2:

$$\begin{aligned}\rho = &= -\mathbf{Pr}(Y_i = 1)\log_2(\mathbf{Pr}(Y_i = 1)) - \mathbf{Pr}(Y_i = 0)\log_2(\mathbf{Pr}(Y_i = 0)) \\ &= -0.54\log_2(0.54) - 0.46\log_2(0.46) \\ &= 0.9954\end{aligned} \tag{3.4.1}$$

The entropy of the PUF responses $\rho = 0.9954$ is an extremely good result considering true randomness is $\rho = 1$. From this entropy and the bit-error-probability $p_e = 0.12$ as calculated in Section 3.4.2, we conclude that the quality of the PUF responses is $\langle p_e = 0.12, \rho = 0.9954\rangle$-$\mathcal{P}_{3\text{-}1}$. These findings will be used to motivate the design of the RFE.

3.5 Reverse Fuzzy Extractor

As mentioned in Section 3.3, we use a reverse FE construction with the computationally lighter generation procedure in the device and the computationally heavier reproduction procedure on the server. In order to be able to recover the PUF responses, we use a concatenation of error-correcting codes as introduced by Bösch et al. [13], which is a technique to increase the correction rate while minimizing the computational overhead. Our proposed RFE uses a concatenation code of a repetition code $\mathcal{C}_{\text{REP}}(n, 1, t) = C_{\text{REP}}(5, 1, 2)$ and a Bose-Chaudhuri-Hocquenghem (BCH) code $\mathcal{C}_{\text{BCH}}(n, k, t) = C_{\text{BCH}}(255, 139, 15)$. The 1275-bit PUF response is cut into words of 5 bits, which are encoded using the repetition code. The first bit of each word (total of 255 bits) is used to extract a key of 128 bits by using the AEAD-scheme with an empty message. The other four bits of each word (total of 1020 bits) are used to produce helper data that can recover the value of the first bit of each word. In order to further decrease the failure rate of the authentications, for these 255 recovered bits a BCH code with error correcting capability of 15 bits is used.

3.5.1 RFE Design Rationale

In this section, we describe the design rationale of the parameters of the concatenation code. Our goal is to construct a 128-bit key from the DAPUF responses with quality $\langle p_e = 0.12, \rho = 0.9954\rangle$. In this rationale, we assume that all bits of the PUF response are independent.

3.5.1.1 Fail rate

We aim for a fail rate of $p_{\text{fail}} = 10^{-6}$, which is considered an acceptable fail rate for standard performance levels [42].

The probability that a received codeword of n bits has more than t errors is given by [26, 13]:

$$\begin{aligned}\mathbf{Pr}(\text{“}{>}t \text{ errors”}) &= \sum_{i=t+1}^{n} \binom{n}{i} p_e^i (1-p_e)^{n-i} \\ &= 1 - \sum_{i=0}^{t} \binom{n}{i} p_e^i (1-p_e)^{n-i},\end{aligned} \tag{3.5.1}$$

where p_e is the bit-error-probability.

When using a $C_{\text{REP}}(5, 1, 2)$ repetition code, we can decrease the bit-error-probability $p_e = 0.12$ to:

$$\begin{aligned}p_{e,\text{REP}} &= 1 - \sum_{i=0}^{2} \binom{5}{i} 0.12^i (1-0.12)^{5-i} \\ &= 0.01432\end{aligned}$$

Using a $C_{\text{BCH}}(255, 139, 15)$ BCH code on top of that further decreases the bit-error-probability $p_{e,\text{REP}} = 0.01432$ to a fail rate p_{fail} of:

$$\begin{aligned}p_{\text{fail}} &= 1 - \sum_{i=0}^{15} \binom{255}{i} 0.01432^i (1-0.01432)^{255-i} \\ &= 1.176 \cdot 10^{-6}\end{aligned}$$

As a result, using a concatenation code of a $C_{\text{REP}}(5,1,2)$ repetition code and a $C_{\text{BCH}}(255,139,15)$ BCH code achieves a fail rate that is acceptable for standard performance levels [42].

3.5.1.2 Entropy

When implementing a concatenation code, careful considerations of the input entropy is necessary, otherwise the output might yield zero leftover entropy [35]. When using a $C_{\text{REP}}(5,1,2)$ repetition code on 5-bit words of the 1275-bit PUF response, 4-bits per word are disclosed as helper data. Hence, we have an entropy loss of:

$$\mathbf{H}_{\text{REP loss}} = 4 \cdot 255 = 1020 \text{ bits}$$

The entropy loss of the $C_{\text{BCH}}(255,139,15)$ BCH code is introduced by the random string that is needed to construct the code. Hence, we have an entropy loss of:

$$\mathbf{H}_{\text{BCH loss}} = n - k = 255 - 139 = 116 \text{ bits}$$

As a result, the total entropy loss of the 1275-bit PUF response by disclosing the helper data is:

$$\mathbf{H}_{\text{loss}} = \mathbf{H}_{\text{REP loss}} + \mathbf{H}_{\text{BCH loss}} = 1020 + 116 = 1136 \text{ bits}$$

This leaves $(1275 - 1136) \cdot \rho = 139 \cdot 0.9958 = 138$ bits of entropy left in the 255 bits of the BCH codeword. These 255 bits will be compressed in a 128-bit key using using the AEAD-scheme.

3.5.2 Extraction

As mentioned in the previous section, the 255 bits (R_{REP}) will be compressed into a 128-bit key. This method is similar to the construction of Maes et al. [43] and Kelsey et al. [34]. An advantage is that the AEAD-scheme can be used for this construction, minimizing the number of primitives that need to be implemented on the device. Moreover, not using additional randomness decreases the amount of data that needs to be communicated between server and device. However, the amount of entropy in the input needs to be carefully considered.

For this construction we introduce the novel definition of a new type of Extractor, the Entropy Accumulator (EA):

Definition 3.1 (Entropy Accumulator). *Let* $\mathbf{Acc}(W) : \{0,1\}^n \to \{0,1\}^l$ $(n < l)$ *be a polynomial time probabilistic function. We say that* $\mathbf{Acc}(W)$ *is an efficient* (n, m, l, ϵ)*-entropy accumulator if for all pairs of random variables* (W, I) *such that* W *is an* n*-bit string satisfying* $\tilde{\mathbf{H}}_\infty = (W \mid I) \geq m$*, we have*

$$\mathbf{SD}(\langle \mathbf{Acc}(W), I \rangle, \langle U_l, I \rangle) \leq \epsilon.$$

We prove that we can construct a FE from s using Definition 3.1 (Entropy Accumulator), Definition 2.18 () and Definition 2.19 (Fuzzy Extractor):

Lemma 3.1 (Fuzzy Extractor from II)**.** *Assume* $\mathbf{SS}(X)$ *is a* $(\mathcal{W}, m, m', t)$*-secure sketch with recovery procedure* $\mathbf{Rec}(W', H)$*, and let* $\mathbf{Acc}(W)$ *be an* (n, m', k, ϵ)*-entropy accumulator. Then the following* $(\mathbf{Gen}, \mathbf{Rep})$ *is a* $(\mathcal{W}, m, l, t, \epsilon)$*-fuzzy extractor:*

1. $\mathbf{Gen}(W)$*: set* $H = \mathbf{SS}(W)$*,* $R = \mathbf{Acc}(W)$*, output* $\langle R, H \rangle$
2. $\mathbf{Rep}(W', H)$*: recover* $W = \mathbf{Rec}(W', H)$ *and output* $R = \mathbf{Acc}(W)$*.*

Proof. From Definition 2.18 (Secure Sketch)

$$H_\infty(W|SS(W)) \geq m'$$

And since $\mathbf{Acc}(W)$ is a (n, m', k, ϵ)-entropy accumulator (Definition 3.1), from Definition 2.19 (Fuzzy Extractors) we get:

$$\mathbf{SD}(\langle \mathbf{Acc}(W), \mathbf{SS}(W) \rangle, \langle U_l, \mathbf{SS}(W) \rangle) = \mathbf{SD}(\langle R, H \rangle, \langle U_l, H \rangle) \leq \epsilon$$

□

Corollary 3.1 (Fuzzy Extractor from II)**.** *If* $\mathbf{Rec}$ *is an* $(\mathcal{W}, m, m', t)$*-secure sketch and* $\mathbf{Acc}$ *is a* $(n, m' - \log_2(\frac{1}{\delta}), l, \epsilon)$*-entropy accumulator, then the above construction from Lemma 3.1* $(\mathbf{Gen}, \mathbf{Rep})$ *is a* $(\mathcal{W}, m, l, t, \epsilon + \delta)$*-fuzzy extractor.*

3.5.3 Secure Sketch

We use a concatenation code of a $C_{\text{REP}}(5, 1, 2)$ repetition code and a $C_{\text{BCH}}(255, 139, 15)$ BCH code for the Secure Sketch (SS) [13]. This section will describe both code constructions in more detail.

3.5.3.1 Repetition Code

As mentioned in Section 2.4.1.1, there are more advanced techniques of using repetition codes. Simply repeating bits of the PUF response will disclose information in the helper data about this response.

3.5.3.1.1 Encoding Figure 3.4 illustrates the repetition code encoding construction. In this figure, Y denotes the 5-bit word of the original PUF response and H_{REP} denotes the 4-bit helper data that is used by the decoder to retrieve the first bit of Y, R_{REP}.

Example 3.5.1. See Figure 3.4. As an example we take $Y = [1, 0, 1, 1, 0]$. The repetition code encoding construction gives the helper data $H_{\text{REP}} = [1, 0, 0, 1]$ and secret value $R_{\text{REP}} = [1]$. ◁

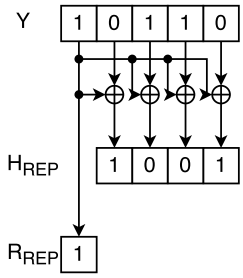


Figure 3.4: Repetition code encoding construction. $\bigoplus$ denotes an XOR.

Note that, as we have assumed in the calculation of the leftover entropy (Section 3.5.1.2), the 4-bit helper data H_{REP} does not disclose any information about the first bit of the 5-bit PUF response word Y.

The first bit of the 5-bit word Y, R_{REP}, will be encoded using the BCH code.

3.5.3.1.2 Decoding Figure 3.5 illustrates the repetition code decoding construction. In this figure, Y' denotes the 5-bit word of the stored PUF response, H_{REP} denotes the 4-bit received helper data and S denotes the resulting syndrome vector. If the Hamming weight **HW** of the syndrome vector S is larger than t, chances are that the first bit of Y' was faulty. Note that this construction will wrongly correct a bit that was assumed faulty if the number of errors $e > t$. Hence, we need the BCH code construction on top of the repetition code construction.

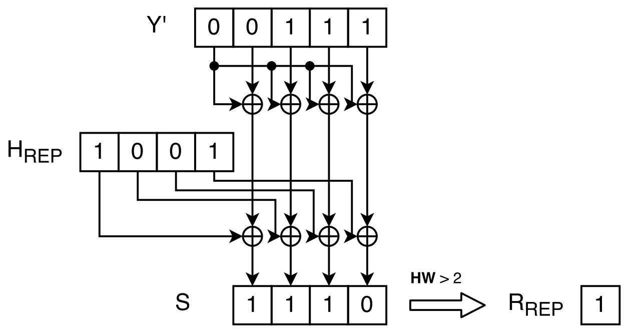


Figure 3.5: Repetition code decoding construction. $\bigoplus$ denotes a bitwise XOR.

Example 3.5.2. See Figure 3.5. As an example we take $Y = [1, 0, 1, 1, 0]$ from Example 3.5.1 and $Y' = [0, 0, 1, 1, 1]$. The repetition code decoding construction gives the syndrome vector $S = [1, 1, 1, 0]$. The Hamming weight **HW** of the syndrome vector S is $\mathbf{HW} = 3 > t$, thus, highly likely the secret value R_{REP} equals $R_{\text{REP}} = [1]$. $\lhd$

3.5.3.2 BCH Code

In order to further decrease the fail rate p_{fail}, a BCH code with error correcting capability $t = 15$ is applied over the binary string that contains the first bit of the 5-bit PUF response words. Then using Formulas 2.4.2 to 2.4.4 we take a BCH code $\mathcal{C}_{\text{BCH}}(n, k, t)$ with order $m = 8$, block-length $n = 2^m - 1 = 255$, error correcting capability $t = 15$, message length $k = 139$, $n - k = 116$ parity-check bits and design distance $d = 2t + 1 = 31$.

3.5.3.2.1 Encoding As mentioned in Section 2.4.2.1, for BCH encoding we have to construct the generator polynomial $\mathbf{g}(x)$. This generator polynomial forms the basis of the Linear Feedback Shift Register (LFSR) design in the hardware of the device, more about this in Section 5.2.2.

Let α be a primitive element of $\mathbf{GF}(2^8)$ generated by the primitive polynomial $\mathbf{p}(x) = x^4 + x^3 + x^2 + 1$. The finite field table is partially given in Appendix B.2. Then, using Formula 2.4.5, we find the generator polynomial $\mathbf{g}(x)$:

$$\begin{aligned}
\mathbf{g}(x) = x^{116} &+ x^{114} + x^{111} + x^{108} + x^{107} + x^{106} + x^{105} + x^{103} + x^{102} + x^{101} \\
&+ x^{99} + x^{98} + x^{97} + x^{95} + x^{93} + x^{90} + x^{89} + x^{88} + x^{87} + x^{84} + x^{82} \\
&+ x^{77} + x^{76} + x^{74} + x^{73} + x^{72} + x^{70} + x^{67} + x^{65} + x^{64} + x^{63} + x^{59} \\
&+ x^{58} + x^{57} + x^{56} + x^{54} + x^{53} + x^{49} + x^{48} + x^{47} + x^{45} + x^{44} + x^{42} \\
&+ x^{41} + x^{40} + x^{39} + x^{38} + x^{31} + x^{30} + x^{25} + x^{23} + x^{22} + x^{20} + x^{19} \\
&+ x^{18} + x^{17} + x^{9} + x^{8} + x^{4} + x^{3} + 1
\end{aligned}$$

Figure 3.6 illustrates the BCH code encoding construction that we use. Note that we use a random message polynomial $\mathbf{m}(x)$ of degree 139 to construct the remainder polynomial $\mathbf{r}(x)$ (and thus the code polynomial $\mathbf{w}(x)$) using the previously calculated generator polynomial $\mathbf{g}(x)$. This codeword W (the code polynomial $\mathbf{w}(x)$) is then XOR'ed with the binary string that contains the first bit of the all the 5-bit PUF response words R_{REP}. The resulting string is the helper data H_{BCH}. Using this construction, we can encode a 255-bit string instead of a 139-bit string as described in Section 2.4.2.

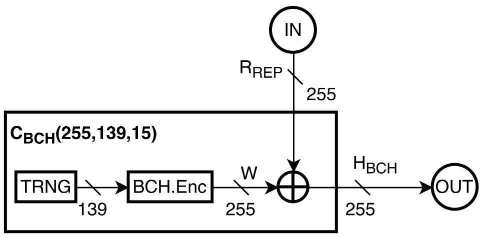


Figure 3.6: BCH code encoding construction. $\bigoplus$ denotes a bitwise XOR, (IN) denotes the input string and (OUT) denotes the output string.

3.5.3.2.2 Decoding Decoding is executed on the server. Figure 3.7 illustrates the BCH code decoding construction that we use. First the helper data H_{BCH} that is received from the device is XOR'ed with the string that contains the recovered first bit of the all the 5-bit PUF response words R'_{REP}, this results in the received codeword W'. Note that this string might still contain faulty bits. Then, this received codeword W' is decoded into the recovered codeword W'', which is XORed with H_{BCH} to produce the recovered PUF response bits R''_{REP}.

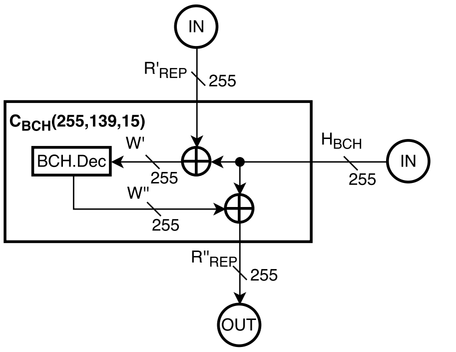


Figure 3.7: BCH code decoding construction. ⊕ denotes a bitwise XOR, (IN) denotes the input strings and (OUT) denotes the output string.

3.6 Ketje

In our implementation of the protocol we use the AEAD-scheme Ketje, one of the 56 candidates in the Competition for Authenticated Encryption: Security, Applicability and Robustness (CAESAR) [7] which was announced in 2013 at the Early Symmetric Crypto workshop in Mondorf-les-Bains, Luxembourg. Similarly to the Advanced Encryption Standard (AES) [16], the European Network of Excellence in Cryptology (ECRYPT) Stream Cipher Project (eSTREAM, [4]), SHA-3 (Keccak [8]) and the Password Hashing Competition (PHC) (Argon2 [12]), CAESAR seeks to select a portfolio of algorithms that enhances AEAD applicability.

We use the AEAD-scheme Ketje for the EA in the RFE, the encryption and decryption of the second PUF response Y^2 and the computation of the authenticator T^2.

Ketje is an AEAD-scheme that is aimed at constrained devices such as Radio-Frequency Identification (RFID) tags and nodes in the Internet of Things (IoT) [11]. The scheme is composed of Ketje Jr and Ketje Sr, of which Ketje Jr is an even more lightweight variant with a security level of 96 bits. However, for the security level of our protocol (128 bits), we use Ketje Sr. As a result, we describe all metrics of Ketje Sr. Also, when we mention Ketje we refer to the specific instance of Ketje Sr.

As with all AEAD schemes, Ketje relies on nonce uniqueness in order for the crypto-system to be semantically secure[1]. For KETJE this is very important, as it can be broken when messages are encrypted with the same nonce. For the permutations, KETJE uses KECCAK-**p**, a permutation that relies on a round reduced version KECCAK-**f** [8]. The construction that calls these permutations is called MONKEYDUPLEX which is based on the duplex construction that is described by Bertoni et al. [9]. The mode that calls the MONKEYDUPLEX construction is called MONKEYWRAP, which is similar and functionally equivalent to SPONGEWRAP, also described by Bertoni et al. [9].

3.6.1 KECCAK-p

The KECCAK cryptographic primitive is a subset of the SHA-3 cryptographic hash function that has been standardized by the National Institute of Standards and Technology (NIST) [49]. Hence, KECCAK-$\mathbf{p}(b, n_r)$ relies on a round reduced version of KECCAK-$\mathbf{f}(b)$ which is defined by its width $b = 25 \cdot 2^l$, with $b \in \{25, 50, 100, 200, 400, 800, 1600\}$, and its number of rounds n_r [8]. More specifically, KECCAK-$\mathbf{p}(b, n_r)$ consists of the last n_r rounds of KECCAK-$\mathbf{f}(b)$, when $n_r = 12 + 2 \cdot l$, KECCAK-$\mathbf{p}(b, n_r)$ = KECCAK-$\mathbf{f}(b)$.

During the permutations of KECCAK-$\mathbf{p}(b, n_r)$, there are five operations that act on a state $\mathcal{T}(x, y, z)$ that is illustrated in Figure 3.8. The size of this state is $\mathcal{T}(5, 5, w)$, with $w = 2^l$. For $0 \leq a, b < 5$ and $0 \leq c < w$, we call $\mathcal{T}(x, b, z)$ a plane, $\mathcal{T}(x, y, c)$ a slice, $\mathcal{T}(a, y, z)$ a sheet, $\mathcal{T}(x, b, c)$ a row, $\mathcal{T}(a, y, c)$ a column, $\mathcal{T}(a, b, z)$ a lane and $\mathcal{T}(a, b, c)$ a bit. For KETJE, KECCAK-$\mathbf{p}(b, n_r)$ with $b = 400$ and $l = 4$ is used. n_r varies per operation in the MONKEYDUPLEX construction, more about this in Section 3.6.2.

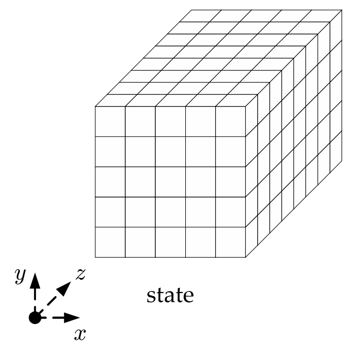


Figure 3.8: State $\mathcal{T}(x, y, z)$ of KECCAK-**p**.

The permutations of one round $\mathbf{R}(\mathcal{T})$ on the state $\mathcal{T}(x, y, z)$ are described by five operations:

$$\mathbf{R}(\mathcal{T}) = \iota(\mathcal{T}) \circ \chi(\mathcal{T}) \circ \pi(\mathcal{T}) \circ \rho(\mathcal{T}) \circ \theta(\mathcal{T}) \tag{3.6.1}$$

The operations in the x and y coordinates are in modulo 5, whereas operations in the z coordinate is in modulo w.

[1]Here a system "[...] is semantically secure if whatever an eavesdropper can compute about the cleartext given the cipher-text, he can also compute without the cipher-text." [23].

3.6.1.1 θ-operation

Figure 3.9 illustrates the θ-operation. The θ-operation is linear and aimed at diffusion of the state. θ is given by the following formula:

$$\theta(\mathcal{T}(x,y,z)) = \mathcal{T}(x,y,z) + \sum_{y'=0}^{4} \mathcal{T}(x-1,y',z) + \sum_{y'=0}^{4} \mathcal{T}(x+1,y',z-1) \quad (3.6.2)$$

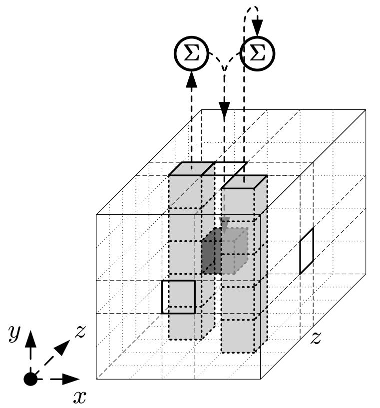


Figure 3.9: θ-operation.

3.6.1.2 ρ-operation

Figure 3.10 illustrates the ρ-operation. The ρ-operation consists of translations within the lanes aimed at providing inter-slice dispersion. ρ is given by the following formula:

$$\rho(\mathcal{T}(x,y,z)) = \mathcal{T}\left(x, y, z - \frac{(t+1)(t+2)}{2}\right), \quad (3.6.3)$$

with t satisfying $0 \leq t < 24$ and $\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}^t \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$ in $\mathbf{GF}(5)^{2\times 2}$, or $t = -1$ if $x = y = 0$.

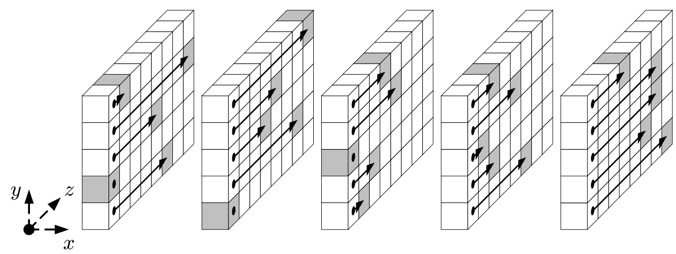


Figure 3.10: ρ-operation.

3.6.1.3 π-operation

Figure 3.11 illustrates the π-operation. The π-operation is based on a transposition of the lanes that provides dispersion aimed at long-term diffusion. π is given by the following formula:

$$\forall\, z : \pi(\mathcal{T}(x, y, z)) = \mathcal{T}(x', y', z), \tag{3.6.4}$$

with $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix}$.

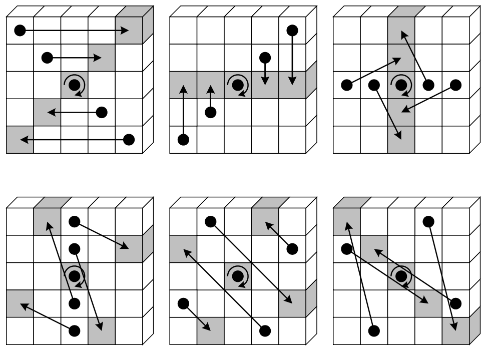


Figure 3.11: π-operation.

3.6.1.4 χ-operation

Figure 3.12 illustrates the χ-operation. The χ-operation is the only nonlinear mapping of KECCAK-**p**. χ is given by the following formula:

$$\forall\, y, z : \chi(\mathcal{T}(x, y, z)) = \mathcal{T}(x, y, z) + (\mathcal{T}(x, y, z) + 1) \cdot \mathcal{T}(x + 2, y, z) \tag{3.6.5}$$

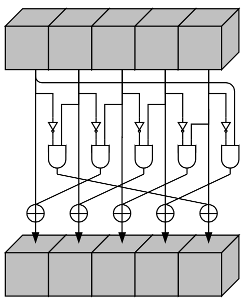


Figure 3.12: χ-operation.

3.6.1.5 ι-operation

The ι-operation consists of applying round-constants in order to disrupt symmetry. ι is given by the following formula:

$$\iota(\mathcal{T}(x,y,z)) = \mathcal{T}(x,y,z) + \mathbf{RC}_i \tag{3.6.6}$$

Here, $\mathbf{RC}_i$ is given by the following formula, indicating the round-constant for round i:

$$\mathbf{RC}_i(0,0,2^j-1) = \mathbf{rc}(j+7i),\ \forall\ 0 \le j \le l,$$

and all other values of $\mathbf{RC}_i(x,y,z)$ are zero. The values $\mathbf{rc}(t) \in \mathbf{GF}(2)$ are given by the LFSR:

$$\mathbf{rc}(t) = (x^t \mod x^8 + x^6 + x^5 + x^4 + 1) \mod x$$

3.6.2 MonkeyDuplex

In this section, we give a brief description of the MonkeyDuplex construction. For a full description we encourage the reader to consult Bertoni et al. [11, 10].

The MonkeyDuplex construction was introduced by Bertoni et al. [10] and then improved by the same authors [11]. The construction is aimed at building stream ciphers and authenticated encryption schemes. MonkeyWrap, which describes the AEAD-mode of Ketje builds on top of MonkeyDuplex, more about this in Section 3.6.3.

Figure 3.13 illustrates the MonkeyDuplex construction. In this figure, we see that the MonkeyDuplex construction is composed of three operations: **start**, **step** and **stride**. These three operations use a permutation function **f** (e.g. Keccak-$\mathbf{p}(b, n_r)$) with different number of rounds n_r.

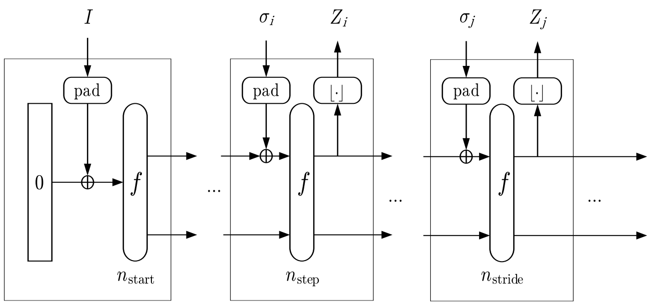


Figure 3.13: The MonkeyDuplex construction.

The MonkeyDuplex($\mathbf{f}, r, n_{\mathbf{start}}, n_{\mathbf{step}}, n_{\mathbf{stride}}$) construction works as follows:

start starts the MonkeyDuplex construction on an empty state of b bits. The operation sets the state with the string I padded up to b bits. Here, in the padding, a single bit 1 is appended to I, followed by the minimum number of bits 0 followed by a single bit 1 such that the length of the result is a multiple of b. Subsequently, the permutation function $\mathbf{f}(n_{\mathbf{start}})$ is applied to the state.

step can process an injection of string σ_i of up to $r-2$ bits, Here, equal to **start**, padding is applied before an XOR with the current state. Subsequently, the permutation function $\mathbf{f}(n_{\mathbf{step}})$ is applied to the state. Finally, the first l bits of the state are extracted (where $l \leq r$), denoted in the figure by $\lfloor \cdot \rfloor$.

stride is similar to **step**, however, **stride** aims at providing resistance to output forgery on top of providing resistance against state retrieval. As a result, we require $n_{\mathbf{step}} < n_{\mathbf{stride}}$.

3.6.3 MONKEYWRAP

In this section, we give a brief description of the MONKEYWRAP construction. For a full description we encourage the reader to consult Bertoni et al. [11].

As mentioned, the MONKEYWRAP construction acts as KETJE's AEAD-mode and builds on top of the MONKEYDUPLEX construction. Equivalent to the encryption-scheme $\mathcal{E}$ and decryption-scheme $\mathcal{D}$ of the AEAD-scheme Π as described in Definition 2.20, Section 2.6, MONKEYWRAP is composed of a construction to wrap ($\mathcal{E}$) and a construction to unwrap ($\mathcal{D}$). The wrapping construction takes as input a message M and associated data A respectively and outputs a cryptogram C and a tag T respectively. The unwrapping construction reverses this by taking the associated data A, a cryptogram C and a tag T as input and returning the message M if the tag T is correct.

Figure 3.13 illustrates the wrapping of a message and associated data using the MONKEYWRAP construction. In this figure, the MONKEYWRAP key should be a Secret Unique Value (SUV), which means that it is composed of the key K and a unique nonce N. An advantage of the construction is that we can produce a tag T without cryptogram C by inputting an empty message M. This is useful for the EA of the FE and the computation of the authenticator T^2 in our protocol.

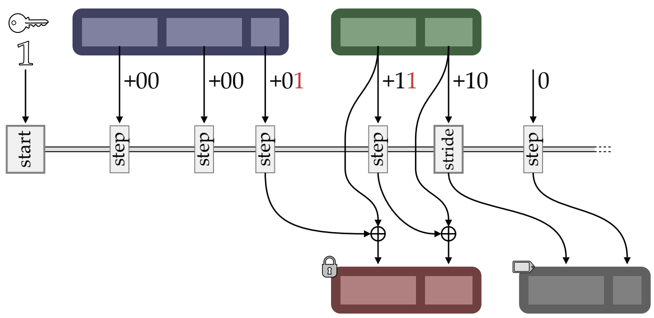


Figure 3.14: The wrapping of a message and authenticated data using the MONKEYWRAP construction.

The **start**, **step** and **stride** operations in the figure are the operations in the MONKEYDUPLEX construction as described in Section 3.6.2. For KETJE the permutation function is $\mathbf{f} = \textsc{Keccak}\text{-}\mathbf{p}(400)$, the input blocks are of size $r = 32$, the number of rounds in **start** is $n_{\mathbf{start}} = 12$, the number of rounds in **step** is $n_{\mathbf{step}} = 1$ and the number of rounds in **stride** is $n_{\mathbf{stride}} = 6$.

CHAPTER

FOUR

SECURITY ANALYSIS

In this chapter we describe the security analysis of the proposed privacy-preserving authentication protocol, the Concealing KETJE Protocol (CKP). We follow the security analyses of the protocols described by Aysu et al. [3] and Moriyama et al. [48] because of the fact that our protocol is based thereon. By doing this, we also base our security analysis on the indistinguishability-based security model of Juels and Weis [32]. Moreover, we consider an active attacker who is able to desynchronize the shared secret between the device and the server [50]. Hence, we assume that the server and the device are able to execute an honest session before and after the challenge phase in the privacy definition.

In Section 4.1 we describe the security model. Section 4.2 describes the formal security definitions. Finally, in Section 4.3 we prove the security and the privacy of the proposed protocol.

4.1 Security Model

In this section we describe the security model, the formal description of the security policy that describes the proposed protocol. We describe the communication model, the theoretical security and the theoretical privacy.

4.1.1 Communication Model

With the security considerations described in Section 3.1 in mind, we take one trusted server $\mathcal{S}(\{\langle Y, Y^{old}\rangle\}_n)$ with n devices $\mathbf{Dev}_i(\mathbf{puf}_i(\,\cdot\,), X)$. Here, the set of n devices is denoted as $\Delta := \{\mathbf{Dev}_0, \mathbf{Dev}_1, \ldots, \mathbf{Dev}_{n-1}\}$. We denote the security parameter as k.

Following Aysu et al. [3] and Moriyama et al. [48], devices will be enrolled in a trusted environment, this happens in a setup phase using a setup algorithm $\mathbf{Setup}(1^k)$ which generates public parameter P and shared-secret Y. Here P denotes all the public parameters available to the environment ($P := X^1 \parallel N^1 \parallel \langle H, N^2, C^1, T^1\rangle \parallel T^2$ in our protocol) and Y denotes the secret Physically Unclonable Function (PUF) response. During the authentication phase, the server $\mathcal{S}$ remains trusted, however, the devices Δ and the communication channel will be subjected to the actions of an attacker. At the end of the authentication phase, both the server and the device will output acceptance ($B_0 = 1$) or rejection ($B_0 = 0$) as result of the authentication.

We call the sequence of communication between the server and the device a session, which is distinguished by a session identifier I. This session identifier I is the transcript of the authentication phase ($I := N^1 \parallel \langle H, N^2, C^1, T^1\rangle \parallel T^2$ in our protocol). Whenever the communication messages generated by the server and the device are honestly transferred until they authenticate each other, we call that a session has a matching session (i.e. I is untampered with). The correctness of the proposed authentication protocol is that the server and the device always accept the session if the session has the matching session.

4.1.2 Security

Following Aysu et al. [3] and Moriyama et al. [48], we consider the canonical security level for authentication protocols, namely the resilience to the Man-in-the-Middle (MitM) attack. This means that the power of an attacker is modeled by letting the attacker control all communication of the protocol. As mentioned earlier, if and only if the communication message is honestly transferred, the authentication results for both the server $\mathcal{S}$ and the device $\mathbf{Dev}_i$ will be $B_0 = 1$. Supplementary to the security requirement of resilience to MitM attacks, we permit the attacker to access the information stored in the non-volatile memory of the device $\mathbf{Dev}_i$ in between sessions (X in our protocol).

Figure 4.1 illustrates the security evaluation on a theoretical level. In this figure, $\mathbf{Exp}^{\mathbf{Sec}}_{\Psi,\mathcal{A}}(k)$ denotes the security experiment between the proposed protocol Ψ and an attacker $\mathcal{A}$ with security parameter k (128-bits in our protocol).

$$
\begin{array}{l}
\mathbf{Exp}^{\mathbf{Sec}}_{\Psi,\mathcal{A}}(k) \\
\hline
\langle P, Y\rangle \leftarrow \mathbf{Setup}(1^k) \\
\langle \mathbf{Dev}_i, I'\rangle \leftarrow \mathcal{A}^{\langle \mathbf{Launch}, \mathbf{SendServer}, \mathbf{SendDev}, \mathbf{Result}, \mathbf{Reveal}\rangle}(P, \mathcal{S}, \Delta) \\
B_0 := \mathbf{Result}(\mathbf{Dev}_i, I') \\
\text{Output} \quad B_0 \\
\hline
\end{array}
$$

Figure 4.1: Security experiment $\mathbf{Exp}^{\mathbf{Sec}}_{\Psi,\mathcal{A}}(k)$.

After the setup phase, and thus after receiving $\langle P, \mathcal{S}, \Delta\rangle$, the attacker $\mathcal{A}$ can query the server $\mathcal{S}$ and the device $\mathbf{Dev}_i$ with the oracle queries
$\mathcal{O} := \langle \mathbf{Launch}, \mathbf{SendServer}, \mathbf{SendDev}, \mathbf{Result}, \mathbf{Reveal}\rangle$:

- $\mathbf{Launch}(1^k)$: Launch the server $\mathcal{S}$ to start a new session with security parameter k.
- $\mathbf{SendServer}(M)$: Send an arbitrary message M to the server $\mathcal{S}$.
- $\mathbf{SendDev}(\mathbf{Dev}_i, M)$: Send an arbitrary message M to device $\mathbf{Dev}_i \in \Delta$
- $\mathbf{Result}(G, I)$: Output whether the session I of G is accepted or not where $G \in \{\mathcal{S}, \Delta\}$.
- $\mathbf{Reveal}(\mathbf{Dev}_i)$: Output all the information stored in the Non-Volatile Memory (NVM) of $\mathbf{Dev}_i$.

The advantage of attacker $\mathcal{A}$ against Ψ is defined as:

$$\mathbf{Adv}^{\mathbf{Sec}}_{\Psi,\mathcal{A}}(k) := \mathbf{Pr}(\mathbf{Exp}^{\mathbf{Sec}}_{\Psi,\mathcal{A}}(k) \to 1 \mid \text{“}I \text{ of } G \text{ has no matching session”}) \quad (4.1.1)$$

We define security of an authentication protocol as follows:

Definition 4.1 (Security)**.** *An authentication protocol Ψ holds the security against MitM attacks with memory compromise if for any probabilistic polynomial time attacker $\mathcal{A}$, $\mathbf{Adv}^{\mathbf{Sec}}_{\Psi,\mathcal{A}}(k)$ is negligible in k (for large enough k).*

In other words, the security of authentication protocol Ψ is based on the fact that the advantage of an attacker is insignificant if k is large enough.

4.1.3 Privacy

Following Aysu et al. [3] and Moriyama et al. [48], we define the privacy definition using indistinguishability between two devices. Here, an attacker selects two devices and tries to distinguish the communication, and thus the identification, between the two devices.

We use the privacy experiment between an attacker $\mathcal{A} := \langle \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3\rangle$ as illustrated in Figure 4.2.

Similar to the security experiment described in Section 4.1.2, the attacker can interact with the devices and the server through the oracle queries
$\mathcal{O} := \langle \mathbf{Launch}, \mathbf{SendServer}, \mathbf{SendDev}, \mathbf{Result}, \mathbf{Reveal}\rangle$.

$\mathbf{Exp}_{\Psi,\mathcal{A}}^{\mathrm{IND}^*-b}(k)$

$\langle P, Y\rangle \leftarrow \mathbf{Setup}(1^k)$
$\langle \mathbf{Dev}_0^*, I^{0'}, \mathbf{Dev}_1^*, I^{1'}\rangle \leftarrow \mathcal{A}_1^{\mathcal{O}}(P, \mathcal{S}, \Delta)$
$b \leftarrow \{0, 1\}$
$\Delta' := \Delta \setminus \langle \mathbf{Dev}_0^*, \mathbf{Dev}_1^*\rangle$
$\psi_0 \leftarrow \mathbf{Execute}(\mathcal{S}, \mathbf{Dev}_0^*)$
$\psi_1 \leftarrow \mathbf{Execute}(\mathcal{S}, \mathbf{Dev}_1^*)$
$\langle I^{0''}, I^{1''}\rangle \leftarrow \mathcal{A}_2^{\mathcal{O}}(\mathcal{S}, \Delta', \mathcal{I}(\mathbf{Dev}_b^*), \psi_0, I^{0'}, \psi_1, I^{1'})$
$\psi_0' \leftarrow \mathbf{Execute}(\mathcal{S}, \mathbf{Dev}_0^*)$
$\psi_1' \leftarrow \mathbf{Execute}(\mathcal{S}, \mathbf{Dev}_1^*)$
$B_0 \leftarrow \mathcal{A}_3^{\mathcal{O}}(\mathcal{S}, \Delta', \psi_0', I^{0''}, \psi_1', I^{1''})$
Output $\quad B_0$

Figure 4.2: Privacy experiment $\mathbf{Exp}_{\Psi,\mathcal{A}}^{\mathrm{IND}^*-b}(k)$ in which it is allowed to communicate with two devices.

After the setup-phase, and similar to the security experiment, the attacker interacts with the server and two randomly chosen devices through the oracle queries $\mathcal{O}$. These two devices $\mathbf{Dev}_0^*, \mathbf{Dev}_1^*$ are being sent to the challenger who flips a coin to choose with which device the attacker will communicate anonymously. This anonymous communication is accomplished by adding a special identity $\mathcal{I}$ which honestly transfers the communication messages between $\mathcal{A}$ and $\mathbf{Dev}_b^*$.

It is trivial that the attacker can trace devices in case the **Reveal** query is issued when there are no successful authentications. Hence, we provide re-synchronization before and after the anonymous access by adding the **Execute** query. This query does a normal protocol execution between the server $\mathcal{S}$ and the device $\mathbf{Dev}_i^*$. During this execution, the attacker can not modify the communications, however the transcript ψ_i is delivered to the attacker.

The advantage of the attacker is defined as:

$$\mathbf{Adv}_{\Psi,\mathcal{A}}^{\mathrm{IND}^*}(k) := |\mathbf{Pr}(\mathbf{Exp}_{\Psi,\mathcal{A}}^{\mathrm{IND}^*-0}(k) \to 1) - \mathbf{Pr}(\mathbf{Exp}_{\Psi,\mathcal{A}}^{\mathrm{IND}^*-1}(k) \to 1)| \qquad (4.1.2)$$

We define privacy of an authentication protocol as follows:

Definition 4.2 (Privacy). *An authentication protocol Ψ holds forward and backward privacy if for any probabilistic polynomial time attacker $\mathcal{A}$, $\boldsymbol{Adv}_{\Psi,\mathcal{A}}^{\mathrm{IND}^*}(k)$ is negligible in k (for large enough k).*

In other words, the privacy preservation of authentication protocol Ψ is based on the fact that the advantage of an attacker is insignificant if k is large enough.

4.2 Formal Security Definitions

In this section, we describe the formal security definitions of the several protocol components by following Aysu et al. [3] and Moriyama et al. [48].

4.2.1 Physical Unclonable Function

We define a PUF using the definitions described in Section 2.3 and the definition described by Aysu et al. [3, p. 24].

For this definition we use $Y \leftarrow \mathbf{puf}_i(X) \in \mathcal{P}$ as a notation for a PUF $\mathbf{puf}_i \in \mathcal{P}$ which takes challenge X and produces response Y. To distinguish between multiple devices, we denote the PUF class $\mathcal{P}$ as $\{\mathbf{puf}_0(\,\cdot\,), \mathbf{puf}_1(\,\cdot\,), \ldots, \mathbf{puf}_{n-1}(\,\cdot\,)\}$, where n is the number of devices. We denote the set of all possible challenges X which can be applied to an instance of $\mathcal{P}$ as $\mathcal{X}_\mathcal{P}$. We say that the PUF class $\mathcal{P}$ is a $\langle n, l, d, h, \epsilon\rangle$-secure PUF class $\mathcal{P}$ if the following conditions hold:

1. For any PUF instance $\mathbf{puf}_i(\,\cdot\,) \leftarrow \mathcal{P}$ and for any input $X \leftarrow \mathcal{X}_\mathcal{P}$,

$$\mathbf{Pr}(\mathbf{dist}(Y \leftarrow \mathbf{puf}_i(X), Y' \leftarrow \mathbf{puf}'_i(X)) < d) = 1 - \epsilon$$

2. For any two PUF instances $\mathbf{puf}_i(\,\cdot\,), \mathbf{puf}_j(\,\cdot\,) \leftarrow \mathcal{P}$, where $i \neq j$ and for any input $X \leftarrow \mathcal{X}_\mathcal{P}$,

$$\mathbf{Pr}(\mathbf{dist}(Y \leftarrow \mathbf{puf}_i(X), Y' \leftarrow \mathbf{puf}_j(X)) > d) = 1 - \epsilon$$

3. For any PUF instance $\mathbf{puf}_i(\,\cdot\,) \leftarrow \mathcal{P}$ and for any two inputs $X^a, X^b \leftarrow \mathcal{X}_\mathcal{P}$, where $a \neq b$,

$$\mathbf{Pr}(\mathbf{dist}(Y \leftarrow \mathbf{puf}_i(X^a), Y' \leftarrow \mathbf{puf}_i(X^b)) > d) = 1 - \epsilon$$

4. For any PUF instance $\mathbf{puf}_i(\,\cdot\,) \leftarrow \mathcal{P}$ and for any input $X^a \leftarrow \mathcal{X}_\mathcal{P}$,

$$\mathbf{Pr}(\tilde{\mathbf{H}}_\infty(Y \leftarrow \mathbf{puf}_i(X^a) \mid \{Y^j \leftarrow \mathbf{puf}_j(X^b)\}_{0 \leq j < n,\ 0 \leq b < l,\ i \neq j,\ a \neq b}) > h) = 1 - \epsilon$$

These conditions provide that the intra-distance $\mathcal{D}^{\text{intra}}_\mathcal{P}$ is smaller than d, the inter-distance $\mathcal{D}^{\text{inter}}_\mathcal{P}$ (from two metrics) is larger than d and the min-entropy of the PUF class $\mathcal{P}$ is always larger than h.

Definition 4.3 ($\langle n, l, d, h, \epsilon\rangle$-secure PUF class $\mathcal{P}$)**.** *A PUF class $\mathcal{P}$ satisfies $\langle n, l, d, h, \epsilon\rangle$-secure PUF class $\mathcal{P}$ if all the above conditions hold.*

4.2.2 Fuzzy Extractor

We define a Fuzzy Extractor (FE) using the definitions described in Section 2.5 and the definition described by Aysu et al. [3, p. 24].

A $\langle d, h, \epsilon \rangle$-FE consists of two algorithms: a key generation algorithm **Gen** and a reconstruction algorithm **Rec**. **Gen** takes as input variable Z and outputs key R and helper data H. For correctness, **Rec** recovers the key R from input variable Z' and helper data H if the distance **dist** between Z and Z' is at most d. The FE provides unpredictable outputs if the min-entropy of input Z is at least h. In that case, R is statistically ϵ-close to a uniformly random variable in $\{0,1\}^k$, even if the helper data H is disclosed.

Definition 4.4 ($\langle d, h, \epsilon \rangle$-secure FE)**.** *A FE satisfies $\langle d, h, \epsilon \rangle$-secure FE if the following conditions hold:*

1. $\mathbf{Pr}(R := \mathbf{Rec}(Z', H) \mid \langle R, H \rangle = \mathbf{Gen}(Z),\ \mathbf{dist}(Z, Z') \leq d) = 1$
2. *If $\tilde{\mathbf{H}}_\infty(Z) \geq h$, $\langle R, H \rangle = \mathbf{Gen}(Z)$ is statistically ϵ-close to $\langle R', H \rangle$ where $R' \leftarrow \{0,1\}^k$ is chosen uniformly at random.*

4.2.3 AEAD-scheme

We take the definition of an AEAD-scheme Π from Definition 2.20, Section 2.6.

The security of the AEAD-scheme Π is defined by the following experiment (Chosen-Plaintext Attack (CPA)) between a challenger and an attacker $\mathcal{A}$:

1. First, the challenger randomly selects coin $b \leftarrow \{0,1\}$ and secret key $K \leftarrow \{0,1\}^k$.
2. The challenger then prepares a truly random function **RF**.
3. The attacker $\mathcal{A}$ can adaptively issue an oracle query to the challenger to obtain a response of a function.
   (a) If $b = 1$ and the attacker $\mathcal{A}$ sends message $M \leftarrow \{0,1\}^*$, challenge $N \leftarrow \{0,1\}^k$ and associated data $A \leftarrow \{0,1\}^*$, the challenger responds with $\langle C, T \rangle = \mathcal{E}_K^{N,A}(M)$.
   (b) On the other hand, if $b = 0$, the challenger inputs the message $M \leftarrow \{0,1\}^*$, challenge $N \leftarrow \{0,1\}^k$ and associated data $A \leftarrow \{0,1\}^*$ to **RF** and responds with its result $\langle C', T' \rangle$.
4. Finally, the attacker outputs a guess b'. If $b' = b$, the attacker wins the experiment.

Similarly, this construction can be applied to test the security of the decryption algorithm $\mathcal{D}_K^{N,A}(\langle C, T \rangle)$.

The advantage of the attacker to win the experiment is defined by $\mathbf{Adv}_{\mathcal{A}}^{\Pi}(k) = |2 \cdot \mathbf{Pr}(b' = b) - 1|$.

Definition 4.5 (ϵ-secure AEAD-scheme)**.** *An AEAD-scheme is an ϵ-secure AEAD-scheme if for any probablistic polynomial time attacker $\mathcal{A}$, $\mathbf{Adv}_{\mathcal{A}}^{\Pi}(k) \leq \epsilon$.*

4.3 Security Proofs

In this section, we give the security proof and privacy proof for the proposed protocol. We follow the proof by game-transformations as described by Aysu et al. [3] and Moriyama et al. [48].

4.3.1 Security

Theorem 4.1 (Security)**.** *Let PUF instance* $\mathbf{puf}^* \leftarrow \mathcal{P}$ *be a* $\langle n, l, d, h, \epsilon_1 \rangle$*-secure PUF, FE be a* $\langle d, h, \epsilon_2 \rangle$*-secure FE and the AEAD-scheme be an* ϵ_3*-secure AEAD-scheme. Then our protocol* Ψ *is secure against MitM attacks with memory compromise. Especially, we have* $\mathbf{Adv}^{\mathbf{Sec}}_{\Psi,\mathcal{A}}(k) \leq l \cdot n \cdot (\epsilon_1 + \epsilon_2 + \epsilon_3)$.

Proof. The aim of the attacker $\mathcal{A}$ is to violate the security experiment which means that either the server or a device accepts a session without it being the matching session. We call S_i the advantage that the attacker wins the game in **Game** i. We consider the following game transformations:

Game 0: This is the original game between the challenger and the attacker.

Game 1: The challenger randomly guesses the device $\mathbf{Dev}^* \leftarrow \Delta$. If the attacker does not impersonate $\mathbf{Dev}^*$ to the server, the challenger aborts the game. Thus, the attacker needs to participate in session I^* and cannot tamper with the communication.

Game 2: Assume that l is the upper bound of the number of sessions that the attacker can establish in the game. For $0 \leq j < l$, we evaluate or change the variables related to the session between the server and $\mathbf{Dev}^*$ up to the l-th session as the following games:

Game 2-j-1: The challenger evaluates the output from the PUF instance $\mathbf{puf}^*$ implemented in $\mathbf{Dev}^*$ at the j-th session. If the intra-distance is larger than d, the inter-distance is smaller than d or the min-entropy of the output is smaller than h, the challenger aborts the game.

Game 2-j-2: The output from the FE H is changed to a random variable.

Game 2-j-3: The output from the encryption algorithm $\mathcal{E}_R^{N\|0,A}(Y)$ of the AEAD-scheme is derived from a truly random function **RF**.

Game 2-j-4: The output from the encryption algorithm $\mathcal{E}_R^{N\|1,A}(\,\cdot\,)$ of the AEAD-scheme is derived from a truly random function **RF**.

The strategy of the security proof is to change the communication messages corresponding to the target device $\mathbf{Dev}^*$ to random variables. However, we must take care of the PUF construction and challenge-update mechanism in our protocol that updates the secret PUF response Y. Hence, we must proceed with the game transformation starting from the first invocation of device $\mathbf{Dev}^*$. The communication messages gradually change from **Game** 2-j-1 to **Game** 2-j-4, and when these are finished, we can move to the next session. This strategy is recursively applied up to the upper bound of l of the sessions that the attacker can establish.

In short, if the implemented PUF instance creates enough entropy, the FE can provide variables that are statistically close to random strings. Then, this output can be applied as a key for the AEAD-scheme which both authenticate the device as well as encrypt the next secret PUF response Y^2. Finally, the server can be authenticated using the AEAD-scheme without encrypting a message.

Lemma 4.1 (Random Guess). $S_0 = n \cdot S_1$ *(where n is the number of devices).*

Subproof. The violation of security means that there is a session which the server or device accepts while the communication is modified by the attacker. Since we assume that the number of devices is at most n, the challenger can correctly guess the related session with a probability of at least $1/n$. $\diamond$

Lemma 4.2 (PUF Response). $|S_1 - S_{2\text{-}1\text{-}1}| \leq \epsilon_1$ *and* $|S_{2\text{-}(j-1)\text{-}4} - S_{2\text{-}j\text{-}1}| \leq \epsilon_1$ *for any* $1 \leq j < l$ *if the PUF instance* $\mathbf{puf}^*$ *is a* $\langle n, l, d, h, \epsilon_1 \rangle$*-secure PUF.*

Subproof. We now assume that the PUF instance $\mathbf{puf}^*$ satisfies a $\langle n, l, d, h, \epsilon_1 \rangle$-secure PUF in advance. This means that the intra-distance $\mathcal{D}_{\mathcal{P}}^{\text{intra}}$ is smaller than d, the inter-distance $\mathcal{D}_{\mathcal{P}}^{\text{inter}}$ is larger than d and the min-entropy of the PUF class $\mathcal{P}$ is always larger than h except the negligible probability ϵ_1. Since S_1 and $S_{2\text{-}(j-1)\text{-}4}$ assume these conditions except the negligible probability ϵ_1 and $S_{2\text{-}1\text{-}1}$ and $S_{2\text{-}j\text{-}1}$ require these conditions with probability 1, respectively, the gap between them is bounded by ϵ_1. $\diamond$

Lemma 4.3 (FE Output). $\forall\ 0 \leq j < l$, $|S_{2\text{-}j\text{-}1} - S_{2\text{-}j\text{-}2}| \leq \epsilon_2$ *if the FE is a* $\langle d, h, \epsilon_2 \rangle$*-secure FE.*

Subproof. From the subproof of Lemma 4.2, we can assume that the PUF instance $\mathbf{puf}^*$ provides enough min-entropy h. Then the property of the $\langle d, h, \epsilon_2 \rangle$-secure FE guarantees that the output for the FE is statistically close to random and no attacker can distinguish the difference between the two games. $\diamond$

Lemma 4.4 (Authenticated Encryption). $\forall\ 0 \leq j < l$, $|S_{2\text{-}j\text{-}2} - S_{2\text{-}j\text{-}3}| \leq \mathbf{Adv}_{\mathcal{A}}^{\Pi}(k)$ *for a probabilistic polynomial time algorithm* $\mathcal{B}$*.*

Subproof. We construct the algorithm $\mathcal{B}$ which breaks the security of our AEAD-scheme Π. $\mathcal{B}$ can access the real encryption algorithm $\mathcal{E}_R^{N\|0,A}(Y)$, the real decryption algorithm $\mathcal{D}_R^{N\|0,A}(\langle C^1, T^1 \rangle)$ or the truly random function $\mathbf{RF}$. $\mathcal{B}$ sets up all secret keys and simulates our protocol except the n-th session (the current session). When the attacker invokes the n-th session $\mathcal{B}$ sends the uniformly random distributed challenge $A \leftarrow \{0,1\}^k$ as the output of the server. When the attacker $\mathcal{A}$ sends the challenge A^* to a device $\mathbf{Dev}_i$, $\mathcal{B}$ randomly selects a nonce N and issues this to the oracle instead of the real computation of $\mathcal{E}_R^{N\|0,A}(Y)$. Upon receiving $\langle C, T \rangle$, $\mathcal{B}$ continues the computation as the protocol specification and outputs $\langle H, N, C^1, T^1 \rangle$ as the device's response. When the attacker sends $\langle H^*, N^*, C^{1^*}, T^{1^*} \rangle$, $\mathcal{B}$ issues challenge A and nonce N^* to the oracle oracle and obtains either Y or the distinguished symbol INVALID.

If $\mathcal{B}$ accesses the real encryption and decryption algorithms $\langle \mathcal{E}, \mathcal{D} \rangle$, this simulation is equivalent to **Game** 2-j-2. Otherwise, the oracle query issued by $\mathcal{B}$ is completely random and this distribution is equivalent to **Game** 2-j-3. Thus we have $|S_{2\text{-}j\text{-}2} - S_{2\text{-}j\text{-}3}| \leq \mathbf{Adv}_{\mathcal{A}}^{\Pi}(k)$. $\diamond$

Lemma 4.5 (Authentication)**.** *$\forall\ 0 \leq j < l$, $|S_{2\text{-}j\text{-}3} - S_{2\text{-}j\text{-}4}| \leq 2 \cdot \mathbf{Adv}_{\mathcal{A}}^{\Pi}(k)$ for a probabilistic polynomial time algorithm $\mathcal{B}$.*

Subproof. Consider an algorithm $\mathcal{B}$ which interacts with the encryption algorithm $\mathcal{E}_R^{N\|1,A}(\,\cdot\,)$ and truly random function **RF**. $\mathcal{B}$ runs the setup procedure and simulates the protocol up to the n-th session. Similarly to the subproof of Lemma 4.4, when the attacker invokes the n-th session $\mathcal{B}$ sends the uniformly random distributed challenge $A \leftarrow \{0,1\}^k$ as the output of the server. $\mathcal{B}$ continues the computation as the protocol specification and outputs $\langle H, N, C^1, T^1 \rangle$ as the device's response. If the attacker $\mathcal{A}$ has sent the challenge A^* to a device $\mathbf{Dev}_i$, $\mathcal{B}$ randomly selects nonce N and issues this to the oracle instead of the real computation $\mathcal{E}_R^{N\|1,A}(\,\cdot\,)$. When the attacker sends $\langle H^*, N^*, C^{1^*}, T^{1^*} \rangle$, $\mathcal{B}$ issues challenge A and nonce N^* to the oracle and obtains T^2.

If $\mathcal{B}$ accesses the real encryption algorithm $\mathcal{E}$, this simulation is equivalent to **Game** 2-j-3. Otherwise, the oracle query issued by $\mathcal{B}$ is completely random and this distribution is equivalent to **Game** 2-j-4. Thus we have $|S_{2\text{-}j\text{-}3} - S_{2\text{-}j\text{-}4}| \leq \mathbf{Adv}_{\mathcal{A}}^{\Pi}(k)$. $\diamond$

When we transform **Game** 0 to **Game** 2-l-4, there is no advantage of the attacker to violate the security. Given the fact that the attacker knows the PUF challenge X from the device's NVM, the attacker cannot produce a valid PUF response. This results in the fact that the attacker cannot produce a key R which matches any of the keys in the server's database. This means that the cryptogram produced by an attacker will never by accepted by the decryption algorithm of the AEAD-scheme in the server. Additionally, changing the authenticator T^2 will only prevent the device from updating its PUF challenge, this is why the server also performs an exhaustive search over the old $(j-1)$ PUF responses.

Therefore, no attacker can successfully mount the MitM attack in our proposed protocol. $\square$

4.3.2 Forward and Backward Privacy

In this section, we give the privacy proof for the proposed protocol.

Theorem 4.2 (Privacy)**.** *Let PUF instance* $\mathbf{puf}^* \leftarrow \mathcal{P}$ *be a* $\langle n, l, d, h, \epsilon_1 \rangle$*-secure PUF, FE be a* $\langle d, h, \epsilon_2 \rangle$*-secure FE and the AEAD-scheme be an* ϵ_3*-secure AEAD-scheme. Then our protocol* Ψ *holds forward and backward privacy.*

Proof. This proof will be similar to the proof of Theorem 4.1. However, we remark that it is important to assume that our protocol satisfies security first for privacy to hold. This is because if security does not hold, a malicious attacker might be able to desynchronize the secret PUF response Y of device $\mathbf{Dev}^*$ to a chosen one. In that case, even if the attacker honestly transfers the communication message between $\mathcal{I}(\mathbf{Dev}^*)$ and the server in the challenging phase the authentication result is always $B_0 = 0$ and the adversary can observe whether device $\mathbf{Dev}^*$ was selected as the challenge device.

Based on the same **Game** transformation that was describes in the proof of Theorem 4.1, we continuously change the communication messages for the device $\mathbf{Dev}^*$, however, we now call this device $\mathbf{Dev}_1^*$. We do a similar game transformation for a second target device $\mathbf{Dev}_2^*$. In **Game** 1, the attacker can guess which device will be chosen by the challenger in the privacy game with probability of at least $1/n^2$. Upon continuing, the game transformation in **Game** 2 is applied to the sessions related to device $\mathbf{Dev}_1^*$ and device $\mathbf{Dev}_2^*$. Then, all the message transcripts of the **Game** transformations are changed to random variables and no biased information which identifies the challenger's coin is leaked. Also here, information stored in the NVM (X in our protocol) of devices $\mathbf{Dev}_1^*$ and $\mathbf{Dev}_2^*$ will not disclose any information because these memories are updated from random sources.

Therefore, no attacker can distinguish any two devices with probability higher than $1/n^2$, hence, the proposed protocol satisfies forward and backward privacy. $\square$

CHAPTER

FIVE

PROOF OF CONCEPT

Following the protocol design as described in Chapter 3, this chapter gives a proof of concept of the proposed protocol. To this end, we implement the device on a Zedboard and the server on a Linux PC.

In Section 5.1 we describe the architecture of the system. Accordingly, in Section 5.2 we describe the architecture of the device for which we describe the 3-1 Double Arbiter PUF (DAPUF), Bose-Chaudhuri-Hocquenghem (BCH) encoder and KETJE in more detail. Finally, in Section 5.3 we describe the server implementation.

5.1 System Architecture

Figure 5.1 illustrates the system architecture of the device and server. The device is implemented on a Zedboard which contains a Xilinx Zynq-7000 All Programmable System on Chip (SoC) XC7Z020-CLG484-1 (see Appendix A.1 for specifications). The server is implemented on a Linux PC. We design the system architecture using Xilinx Vivado and the Xilinx Vivado Software Development Kit (SDK).

The Zynq SoC is composed of 28 nm Programmable Logic (PL) and a Processing System (PS), which can both be programmed through the Universal Serial Bus (USB) Joint Test Action Group (JTAG). Apart from other components, the PS contains two Advanced RISC Machine (ARM)-cores, of which one is used to:

1. control the communication between the device and the server by reading and writing Advanced Extensible Interface (AXI)-addresses from the device and sending and receiving serial data through the Universal Asynchronous Receiver/Transmitter (UART);

2. update the Physically Unclonable Function (PUF) challenge on the device Non-Volatile Memory (NVM) by re-writing to a SD-card pugged into the Zedboard.

The central communication travels through a bus, the Central Interconnect (CI), which is connected with all the components on the Zynq. Communication between the device and the ARM-core is supported with a 32-bit AXI.

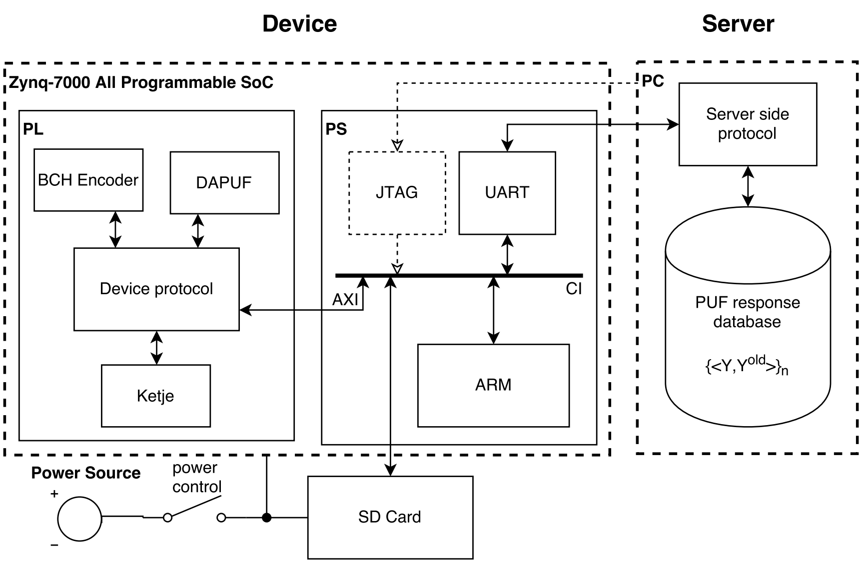


Figure 5.1: System architecture of the Device and Server.

5.2 Device

In this section we describe the architecture of the device. We describe the architectures of the DAPUF, BCH encoder and KETJE in more detail. The source code of the device is given in Appendix C.2.1. We design the device in VHSIC Hardware Description Language (VHDL) using Xilinx Integrated Development Environment (IDE) for High Level Synthesis (HLS) and Xilinx ISE Simulator (iSim) for testing.

Figure 5.2 illustrates the floor-planning of the device as generated by Xilinx Vivado. In this figure, in the top left, the full area of the SoC is illustrated; on the right side the area is illustrated that contains the device logic. The green area represents the PS which uses the yellow, purple and pink logic to set and reset the AXI peripherals. Furthermore, four Physical Blocks (Pblocks) have been defined to constrain the DAPUF component (white logic), the BCH encoder (light blue logic), the KETJE component (orange logic) and the controller (blue logic) to specific areas on the SoC. The three selector chains of the DAPUF are clearly visible (in white).

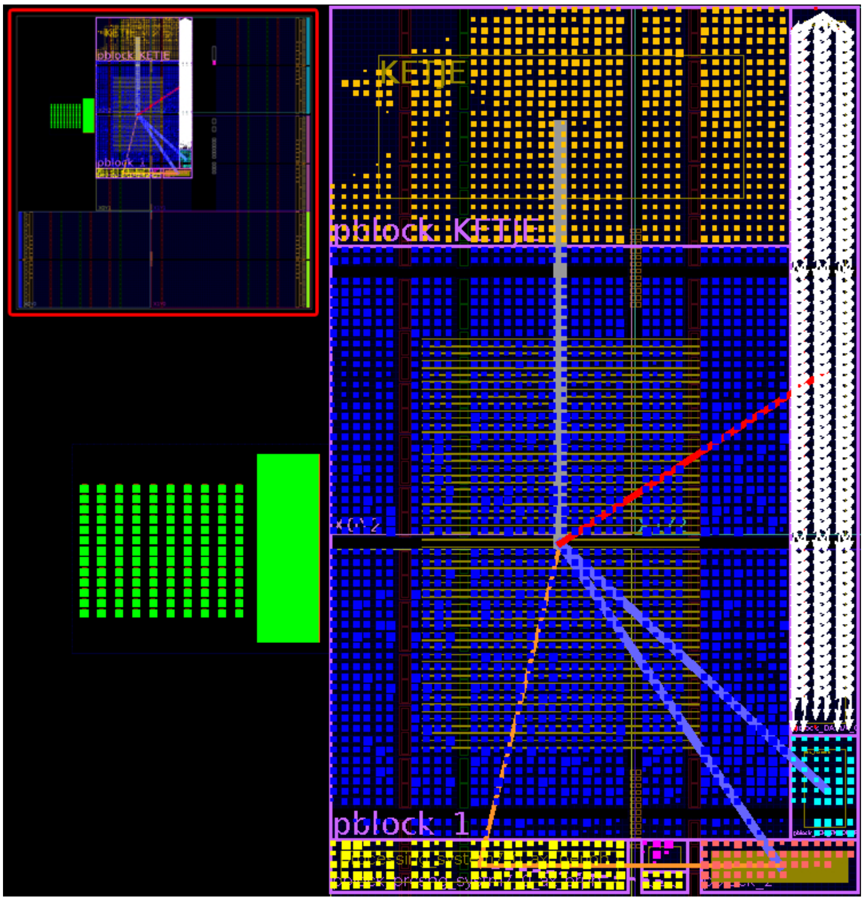


Figure 5.2: Floor-planning of the device.

The controller (blue logic) is composed of seven processes:

Main process : The main process handles all responses for the DAPUF and starts the other processes accordingly. The order of the various subprocesses can be summarized as follows:

1. Update the seed by challenging the DAPUF.
2. Generate the PUF response $Y^{1'}$ by challenging the DAPUF. Subsequently, the repetition encoder process is started.
3. Generate the random value for BCH encoding by challenging the DAPUF. Subsequently, the BCH encoder process is started.
4. Generate the challenge N by challenging the DAPUF. Subsequently, the KETJE mode is set to accumulate the entropy of the PUF response $Y^{1'}$. Finally, the KETJE process is started.
5. Generate the second challenge X^2 by challenging the DAPUF.
6. Generate the second PUF response Y^2 by challenging the DAPUF. Subsequently, the KETJE mode is set to encrypt the second PUF response Y^2. Finally, the KETJE process is started.
7. When the KETJE process is finished the KETJE mode is set to compute the authenticator $T^{2'}$ and the KETJE process is started.
8. Finally, the process waits to receive the authenticator T^2 from the server and compares this with $T^{2'}$. If they are equal, the challenge X is updated with X^2.

DAPUF challenger process: This process challenges the DAPUF by setting an 'enable' signal and the challenge at the first clock cycle. At the third clock cycle, the response is either 0 or 1. This process is repeated every three clock cycles.

Linear Feedback Shift Register (LFSR) process: This process feeds the PUF challenge space. It can operate on two modes, either to feed challenges for the 1275-bit PUF responses, or to feed challenges for the random variables. For both modes, 12 bits have been reserved in the challenge space. As a result the full challenge space of 64 bits is decreased to $64 - 12 - 12 = 40$-bit challenges per authentication. More about this, and about the distribution of the bits, is explained in Section 6.1. The LFSR is either reset with initial value $[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]$, or with the seed as the initial value.

Repetition encoder process: The repetition encoder encodes the DAPUF response and sets the results to the output. The 255-bit DAPUF response is fed to the BCH encoder.

BCH encoder process: The BCH encoder encodes the 255-bit using a random number generated by the DAPUF and sets the results to the output.

KETJE mode process: This process sets KETJE's values according to a specified mode. This mode either signals to accumulate the entropy of the PUF response $Y^{1'}$, to encrypt the second PUF response Y^2, or to compute the authenticator $T^{2'}$.

Ketje process: This process feeds all input data to the Ketje component at the correct clock cycles. Depending on the mode Ketje is running, inputting the message M is skipped or not. As a result, the cipher-text and tag is received or only the tag.

5.2.1 DAPUF

As mentioned in Section 3.4 we implement the DAPUF as proposed by Machida et al. [39]. We obtained the source code designed for a Xilinx Virtex-7 from Machida et al. and adapted this to work for the Xilinx Zynq. This design heavily relies on the constraints that set the specific logic and location of the components in Vivado. These constraints can be set using the following Tool Command Language (Tcl) commands:

```
set_property BEL <logic-type> [get_cells <address>]
set_property LOC <slice> [get_cells <address>]
```

Where `<logic-type>` is replaced with the desirable type of logic, `<slice>` with the desirable slice location and `<address>` with the specific address of the component in Vivado's Netlist. Without these constraints, Vivado will replace the Multiplexers (MUXes) with Look-Up Tables (LUTs) because these require less area and can operate on a higher frequency. The source code of the implemented DAPUF is given in Appendix C.2.3 and the constraints in Appendix C.2.4.

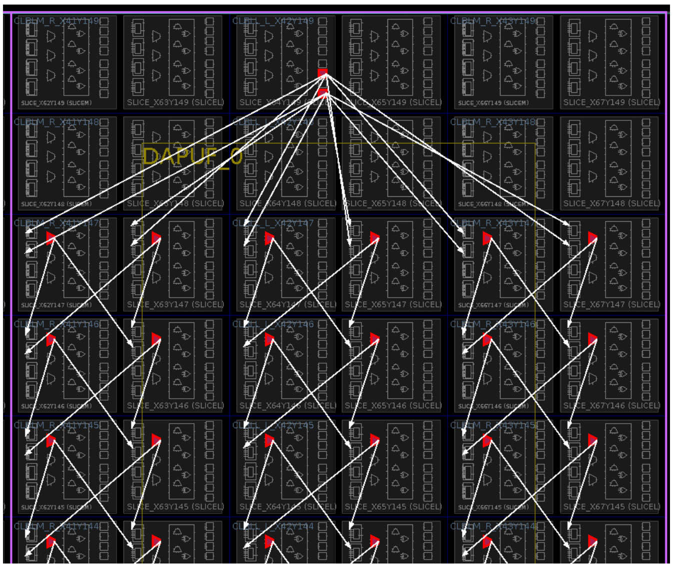
Figure 5.3a illustrates the floor-planning of the first three switch blocks in the three selector chains of the DAPUF as generated by Xilinx Vivado. In the top of the figure, the two registers that contain the 'enable' signals E_L and E_R are illustrated. Then, in three columns all MUXes are constrained to their own logic slice such that path-lengths are equal for equivalent paths.

Figure 5.3b illustrates the floor-planning of the last switch blocks and the arbiters of the DAPUF as generated by Xilinx Vivado. Also here, all MUXes and negative-ANDs (NANDs) are constrained to their own logic slice such that path-lengths are equal for equivalent paths.

Because of the fact that race conditions are undesirable in conventional Field Programmable Gate Array (FPGA) designs, the following Tcl command should be used as a pre-script to Vivado's bitstream generation:

```
set_property SEVERITY {Warning} [get_drc_checks LUTLP-1]
```

Without this command, Vivado will not allow to generate the bitstream that is needed to program the PL through the USB JTAG.



(a) Floorplanning of the start.

(b) Floorplanning of the finish.

Figure 5.3: Floorplanning of 3-1 Double Arbiter PUF.

5.2.2 BCH Encoder

The implemented BCH encoder is designed using a paper by Mathew et al. [46]. The source code of the implemented BCH encoder is given in Appendix C.2.5.

Figure 5.4 illustrates the LFSR that is constructed using the generator polynomial $\mathbf{g}(x)$ as was calculated in Section 3.5.3.2. We omitted the exponents 9 to 107 in the formula and the figure for clarity:

$$\mathbf{g}(x) = x^{116} + x^{114} + x^{111} + x^{108} + \cdots + x^8 + x^4 + x^3 + 1$$

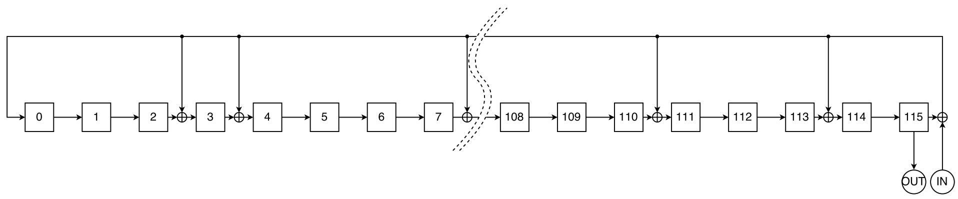


Figure 5.4: Block diagram of the BCH encoder. $\bigoplus$ denotes an exclusive-OR (XOR), IN denotes the input bit, OUT denotes the output bit. The indices correspond to the bit locations and the exponents of the generator polynomial $\mathbf{g}(x)$.

Using this architecture, in 139 clock cycles, the coefficients of the random BCH input polynomial $\mathbf{m}(x)$ are fed to the LFSR. Once this is finished, in 116 clock cycles, the coefficients of the redundancy polynomial $\mathbf{r}(x)$ are obtained. Finally, in the controller these two can be concatenated to obtain the codeword polynomial $\mathbf{w}(x)$.

5.2.3 Ketje

We obtained the source code of the Ketje hardware implementation from Bertoni et al.. The received implementation is designed for the Competition for Authenticated Encryption: Security, Applicability and Robustness (CAESAR) and thus follows the George Mason University (GMU) hardware Application Program Interface (API) for authenticated ciphers [31]. Because this Authenticated Encryption with Associated Data (AEAD)-core is excessively large, we only use the Ketje cipher-core. This cipher-core can perform encryption with message $\langle C, T\rangle = \mathcal{E}_K^{N,A}(M)$, encryption without message $\langle\ \cdot\ , T\rangle = \mathcal{E}_K^{N,A}(\ \cdot\ )$ and decryption $\mathcal{D}_K^{N,A}(\langle C, T\rangle)$. Moreover, it supports sessions such that session based encryption and decryption is possible. We need the Ketje hardware implementation for encryption (for which we have fixed input sizes) but not for decryption. Although this cipher-core is still too large for our purposes, we accept the overhead.

5.3 Server

The server is implemented on a Linux PC using the Python programming language version 2.7.10 [53]. The source code is given in Appendix C.2.5.

Algorithm 1 gives the server-side setup procedure, which is performed in a trusted environment. We rely on serial communication between the PC and the Zedboard through the USB UART.

Algorithm 1 Server-side setup procedure

```
procedure MAIN
    X^1 ← TRNG(40)          ▷ generate random bit-string of length 40
    DEVICE ← X^1            ▷ send serial data to device
    Y^1 ← DEVICE            ▷ receive serial data from device
    𝒴[n] ← Y^1              ▷ n: number of devices
    𝒴^old[n] ← Y^1
    n := n + 1
    return 1
```

Algorithm 2 gives the server-side authentication procedure. Note that because of the conditional branches, this implementation does not prevent an attacker from performing Side-Channel Analysis (SCA). However, for simplicity, in Section 3.1 we stated that an attacker can not perform any implementation attacks. In order to prevent an attacker from successfully performing SCA attacks, one should design a leakage resilient implementation using sound cryptographic engineering.

Also note that the authentication procedure both tries to authenticate devices over the most recent PUF responses as well as the older PUF responses. This is needed because in the event of a loss of connection (card tearing) in the last communication (receiving the authenticator T^2 from the server), a desynchronization will take place between the server and the device.

Algorithm 2 Server-side authentication procedure

```
procedure Main
    T^2 ← TRNG(128)                          ▷ generate random bit-string of length 128
    B_0 := 0                                 ▷ set authentication result to 0
    A ← Start
    H, N, C^1, T^1 ← Device                  ▷ receive serial data from device
    for 0 ≤ i < n do                         ▷ n: number of devices
        Y ← 𝒴[i]
        B_0, T^2 ←AuthenticateTry(Y, H, C^1, T^1, A, N, T^2)
        Y ← 𝒴^old[i]
        B_0, T^2 ←AuthenticateTry(Y, H, C^1, T^1, A, N, T^2)
    Device ← T^2                             ▷ send serial data to device
    return B_0
procedure Start
    A ← TRNG(40)                             ▷ generate random bit-string of length 40
    Device ← A                               ▷ send serial data to device
    return A
procedure AuthenticateTry(Y, H, C^1, T^1, A, N, T^2)
    R ← FE.Rec(Y, H)
    if Y^2 ← Ketje.Dec(R, C^1, T^1, N ∥ 0, A) then
        T^2 ← Ketje.Enc(R, [ ], N ∥ 1, A)
        𝒴[i] ← Y^2
        𝒴^old[i] ← Y
        B_0 := 1                             ▷ update the authentication result to 1
    return B_0, T^2
procedure FE.Rec(Y, H)
    R′ := [ ]
    for 0 ≤ i < 255 do
        R′ += [REP.Dec(Y[i · 5 : i · 5 + 5], H[i · 4 : i · 4 + 4])]
    R″ ←BCH.Dec(R′, H)
    return R″
procedure REP.Dec(y, h)
    s := [y[0] xor y[i + 1] xor h[i] for i in range(4)]
    if sum(s) > 2 then                       ▷ Hamming weight of the syndrome vector
        e := 1
    else
        e := 0
    return e xor y[0]
```

CHAPTER

SIX

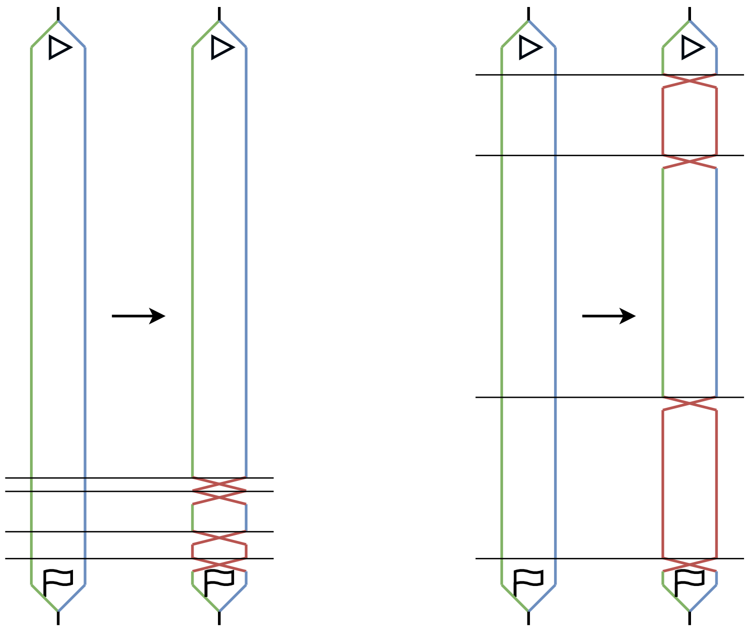
RESULTS

This chapter describes the research results of the protocol. We both describe results from the protocol as supported by the mathematical foundation as well as results from the protocol supported by the proof of concept.

In Section 6.1 we calculate the quality of the Physically Unclonable Function (PUF) responses and compare these to the specified quality by Machida et al. [39]. Section 6.2 summarizes the hardware performance in terms of timing and utilization. Following, in Section 6.3 we discuss the software performance. In Section 6.4 we analyze our protocol using a benchmark for PUF-based authentication protocols [17]. Finally, in Section 6.5 we compare our protocol with the protocol proposed by Moriyama et al. [48] and Aysu et al. [3].

6.1 PUF Response Analysis

Although in Section 3.5.1 we assumed that all the PUF response bits are independent, we found out this is not the case. This is best illustrated with Figure 6.1a. In this figure we see two PUF selector chains each having two data paths. These selector chains have been initialized with two different challenges that only differ at the Least Significant Bits (LSBs). We can see that by doing this, the length of the path fragments that differ in both selector chains is very small. As a result, the probability that the results of the arbiters are different is small. A possible reason why this is not reflected in the results by Machida et al. [39] is that they challenge the PUF instances with random challenges (weak unpredictability, see Definition 2.13, Section 2.3.4). Moreover, the Machine Learning (ML) algorithm is trained with only 1,000 training samples, which means that the probability of having two challenges with low Hamming distance is small.



(a) Differentiation at LSBs.

(b) Diffused differentiation.

Figure 6.1: Illustration of dependency in PUF response bits.

This characteristic means that the 12 bits that are used to retrieve the PUF responses in the protocol need to be diffused throughout the challenge space resulting in the highest probability of having different data paths. This construction is illustrated in Figure 6.1b. Same holds for the 12 bits that are used to retrieve the random variables in the protocol. Moreover, we found out that by feeding these 12 bits using a Linear Feedback Shift Register (LFSR) instead of a counter, more diffusion is created in the switch blocks.

Table 6.1 summarizes the quality of the PUF responses that have been obtained by challenging three PUF instances using the construction that is used in our proof of concept. Because of the limited amount of Zedboards available, we implemented these three PUFs on the same System on Chip (SoC) at different locations. This gives us a good approximation of the PUF response quality on distinct SoCs. The metrics are calculated similarly as Machida et al. did in their paper [39]. However, our results have been achieved by challenging three PUF instances with 40-bit challenges multiple times, obtaining multiple 1275-bit responses. More specifically, steadiness is calculated by challenging the PUF a number of $m = 1275$ times with a set of $n = 128$ equal challenges. Of the 128 1275-bit responses, the Hamming distances between two arbitrary PUF responses is calculated and averaged. Uniqueness is calculated by challenging two PUF instances a number of $m = 1275$ times with a set of $n = 500$ randomly chosen challenges. Of each of these $\binom{500}{2}$ pairs of 1275-bit responses, the the Hamming distances are calculated and averaged. Finally, randomness is calculated by challenging a PUF instance a number of $m = 1275$ times with a set of $n = 500$ randomly chosen challenges. Then, the Hamming weight of these 500 1275-bit results is calculated and averaged.

Table 6.1: Quality of the PUF responses.

Metric	DAPUF	Results
Steadiness [%]	A	5.51
	B	4.03
	C	7.38
Uniqueness [%]	A with B	45.54
	B with C	46.47
	C with A	43.56
Randomness [%]	A	65.56
	B	62.92
	C	70.74

From this table, and Table 3.1 we can see that in the 3-1 Double Arbiter PUF (DAPUF) in our SoC the measure for steadiness is lower (6% versus 12%), which means that our implementation has a higher reproducibility. Moreover, the randomness of our implementation is higher (66% versus 54%), meaning that the probability of a response bit being ‘1’ is higher. To illustrate the effect of the measured quality, we calculate whether enough entropy is left for the Entropy Accumulator (EA) (Definition 3.1) to construct a 128-bit key. Moreover, a recalculation of the fail rate p_{fail} will indicate the performance of the authentications.

We recalculate the entropy of the PUF responses ρ using the binary entropy function $\mathbf{h}(p)$ from Formula 2.2.1, Section 2.2.2:

$$\begin{aligned} \rho &= -\mathbf{Pr}(Y_i = 1)\log_2(\mathbf{Pr}(Y_i = 1)) - \mathbf{Pr}(Y_i = 0)\log_2(\mathbf{Pr}(Y_i = 0)) \\ &= -0.66\log_2(0.66) - 0.34\log_2(0.34) \\ &= 0.9248 \end{aligned} \tag{6.1.1}$$

Recall from Section 3.5.1.2 that $139 \cdot \rho$ bits of entropy is left in the 255 bits of the Bose-Chaudhuri-Hocquenghem (BCH) codeword due to entropy losses through the communicated helper data. Thus, $139 \cdot 0.9248 = 128$ bits of entropy is left to accumulate the 255 bits BCH codeword, which is just enough to construct a 128-bit key. As a result, no information about the key will be leaked through the helper data.

Next, we recalculate the fail rate p_{fail} using Formula 3.5.1, Section 3.5.1.1. When using the implemented $C_{\text{REP}}(5, 1, 2)$ repetition code, we decrease the bit-error-probability $p_e = 0.06$ to:

$$\begin{aligned} p_{e,\text{REP}} &= 1 - \sum_{i=0}^{2} \binom{5}{i} 0.06^i (1 - 0.06)^{5-i} \\ &= 0.001970 \end{aligned}$$

Using the implemented $C_{\text{BCH}}(255, 139, 15)$ BCH code on top of that further decreases the bit-error-probability $p_{e,\text{REP}} = 0.001970$ to a fail rate p_{fail} of:

$$\begin{aligned} p_{\text{fail}} &= 1 - \sum_{i=0}^{15} \binom{255}{i} 0.001970^i (1 - 0.001970)^{255-i} \\ &= 8.438 \cdot 10^{-15} \end{aligned}$$

This is a considerable improvement because we aimed for a fail rate of $p_{\text{fail}} = 10^{-6}$.

6.2 Hardware Performance

This section elaborates on the hardware performance of the proof of concept. The results have been generated by Vivado without the use of Block RAM (BRAM) or Digital Signal Processors (DSPs) and without optimization of the DAPUF design. Synthesis settings are set at `Default` and optimization options are set at `Area`. Furthermore, we allow race conditions to occur due to the nature of the DAPUF.

6.2.1 Timing

By using this specific DAPUF, timing results are suboptimal. Because of the long paths the signals have to travel through the DAPUF, the path delay is high. In the worst case scenario, the data path delay is 76.509 ns which means that the maximum frequency of the SoC is 12 MHz. Considering that some Symmetric Key Encryption (SKE) hardware implementations can run in the magnitude of GHz's, the achieved result is suboptimal. However, the authentication phase of the device takes 8205 clock cycles, which on the frequency of 12 MHz takes 0.63 ms. As a result, our proof of concept might not be applicable to devices in the Internet of Things (IoT) but only to conventional use in Radio-Frequency Identification (RFID) systems.

6.2.2 Utilization

Table 6.2 summarizes the number of Look-Up Tables (LUTs) per component that have been generated by Vivado without optimization in the implementation such that the DAPUF's design is kept. In total, our proof of concept utilizes 8,305 LUTs. Similar to the timing results, these utilization results are suboptimal. In this case the registers take a lot of area because of the long variables in the protocol. We explicitly did not replace these registers with BRAM and DSPs because we want to mimic a passive RFID device which normally does not have these kinds of components.

Table 6.2: Number of LUTs per component.

Component	LUTs
Controller	5,464
Ketje	2,630
DAPUF	195
BCH encoder	16
Total	8,305

6.3 Software Performance

The computation time of the server-side protocol increases linearly in the number of devices in the database. In our implementation the execution time of the server-side protocol is $0.05 \cdot n$ seconds. This means that our implementation might not be applicable to devices in the IoT but only to conventional use in RFID systems.

In a real world scenario, the server would be implemented in hardware which could substantially decrease the execution time. We can not say anything about the performance of a hardware implementation, but it is promising for IoT applications. However, in RFID applications this protocol is more suitable because the number of devices is lower and the maximum execution time is often larger.

6.4 Benchmark Analysis

We analyze our protocol using the recently proposed benchmark for PUF-based authentication protocols [17]. The benchmark results can be summarized as follows:

Resources: Our device uses a PUF, True Random Number Generator (TRNG), Fuzzy Extractor (FE) **Gen** procedure, cryptographic primitive (AEAD-scheme) and a one-time interface.

PUF type: Our PUF is a so-called Strong PUF, indicating that the number of Challenge-Response Pairs (CRPs) is at most 2^l, where l is the number of bits in the challenge.

#CRPs: The amount of CRPs for n authentications is $n + 1$ because we use a one-time interface for the setup.

Claims: The protocol supports server authenticity, device authenticity, device privacy, and memory disclosure.

#Authentications: The protocol can support d-authentications for a perfect privacy use-case and ∞-authentications without token anonymity.

Robustness: Our PUF is noise-robust because of the error correction and modeling-robust because of the EA.

Authenticity: Mutual authentication provides for both server and device authenticity.

Denial-of-Service (DoS) prevention: There is no internal synchronization which means that our implementation is not susceptible to DoS attacks.

Scalability: The execution time of the server per authentication is linear in the amount of devices.

6.5 Protocol Comparison

Table 6.3 summarizes the comparison between the proposed protocol (Concealing Ketje Protocol (CKP)) and the protocols by Moriyama et al. [48] and Aysu et al. [3]. The characteristic that all these protocols have in common is that they are all provably secure PUF-based privacy-preserving protocols. However, the paper by Moriyama et al. only provides a theoretical basis for their proposed protocol, instead of also giving a proof of concept. As a result, no sensible answer can be given to the question whether their protocol is practical or not. On the other hand, the protocol by Aysu et al. uses the paper by Moriyama et al. as a basis. As mentioned, this protocol is vulnerable to linear equation analysis of the FE output [3, p. 12]. Their performance results would highly likely be worsened because their FE needs to be redesigned. To this end, most likely they need more PUF response bits to meet the failure rate requirements. As a consequence, their hardware implementation needs more LUTs and might run slower. Moreover, their implementation stores a key in Non-Volatile Memory (NVM) that does not increase the unpredictability of the communication messages. This overhead is eliminated in our protocol.

Table 6.3: Comparison with previous work.

Reference	Moriyama [48]	Aysu [3]	CKP
Proofs for security and privacy	✓	✓	✓
Implemented parties	✗	device, server	device, server
Security flaws	✗	✓[1]	✗
Reconfiguration method	✗	modify SW, update microcode	follow generic approach, modify HW and SW
Demonstrator	✗	FPGA, PC	SoC, PC
Security-level	k	64-bit/128-bit	128-bit
NVM	PUF challenge & key	PUF challenge & key	PUF challenge
Device FE procedure	**Rec**	**Gen**	**Gen**
PUF type	✗	Weak PUF	Strong PUF
PUF instance	✗	SRAM	DAPUF
Hardware platform	✗	XC5VLX30	XC7Z020
Communication interface	✗	bus, UART	bus, UART
Execution time (clock cycles)	✗	18,597	8,205
Logic cost (w/o PUF)	✗	1,221 LUTs	6,579 LUTs

[1]Due to a vulnerability in their implemented FE.

CHAPTER

SEVEN

CONCLUSIONS

7.1 Conclusions

In this research we have proposed a novel PUF-based privacy-preserving authentication protocol. We base the authenticity of a device on the Challenge-Response Pairs (CRPs) of the Physically Unclonable Function (PUF). Because the responses on equal challenges are not equal, error-correcting codes have to be applied to recover previous PUF responses. Moreover, because the PUF responses are not uniformly distributed, an Entropy Accumulator (EA) is proposed to 'compress' the response into a key. Additionally, confidentiality, authenticity and integrity is supported by an AEAD-scheme. Privacy is preserved by using a challenge updating mechanism in the device's Non-Volatile Memory (NVM). We have evaluated the protocol both by providing a mathematical proof as well as providing a proof of concept.

We evaluated the security and privacy of the protocol using a mathematical proof. We defined a communication model where we assume full control of the attacker over the communication channel as well as read permissions of the device's NVM. We defined security with a security experiment in which the attacker can perform unlimited oracle queries to a device and server that have been setup already. The correctness of the protocol is that the server and the device always accept the session if and only if the session has a matching session. We defined privacy using a similar construction. However in the privacy experiment, the attacker communicates with two devices of which one of the devices honestly transfers the communication messages to the attacker. A re-synchronization step is added in the experiment to make sure successful authentications can occur. The experiment can be won if the attacker can distinguish with which device he has been communicating. We prove security and privacy using a game transformation that shows that all communication in the channel appears random to the attacker always.

We evaluated the applicability and practicality of the protocol by presenting a proof of concept. We have seen that there is a dependency in 3-1 Double Arbiter PUF (DAPUF) response bits when the challenges are close to each other. Also, we have seen that the quality of the DAPUF responses differ on our System on Chip (SoC) with regard to the Field Programmable Gate Array (FPGA) used by Machida et al. [40]. However, these differences are small enough for our implementation to be considered secure and thus privacy-preserving with respect to our security considerations. Because of the use of the DAPUF, timing is suboptimal. However, we still achieve an authentication delay of only 0.63 ms which might make our proof of concept applicable to use for the Internet of Things (IoT) and to conventional use-cases with RFID-technology (e.g. in access control and in supply chains). Also, we have seen that because of the large intermediate registers, utilization is suboptimal.

Concluding, we have seen that in comparison to other similar authentication protocols our implementation does not need a key in NVM and is simpler in its design. Although our implementation is slower and consumes more resources, we claim to have an implementation that is both secure and privacy-preserving with respect to our security considerations.

7.2 Discussion

Although the protocol is mathematically secure and privacy-preserving, we did not achieve a faster and smaller proof of concept in relation to Aysu et al. [3]. This is mainly due to the implemented PUF which defines the design of the Fuzzy Extractor (FE) and the variable sizes in the protocol. Moreover, the authentication time of the server is linear in the number of devices in the database, which makes the protocol impractical with a substantially large number of devices. As a result, our proof of concept might not be practical for use in the IoT but only for conventional use with Radio-Frequency Identification (RFID) technology. There might be various options to design an instance that is applicable to the IoT. We summarize them as follows.

- We can optimize the KETJE scheme which is now a hardware implementation extracted from the George Mason University (GMU) hardware Application Program Interface (API) for Competition for Authenticated Encryption: Security, Applicability and Robustness (CAESAR). Because we use fixed instances of the KETJE scheme, a lot of optimizations are possible.
- Another option is to optimize the controller of our device, which is now implemented using various processes. Sound cryptographic engineering can substantially optimize the area of the SoC.
- With a Strong PUF that has higher quality, following the generic approach, much smaller protocol variables can be achieved, decreasing the area consumption on the Integrated Circuit (IC).
- A different type of PUF can substantially increase the operating frequency of the IC, which decreases the delay of authentication. One solution can be to use a non-intrinsic PUF that are physically embedded in an IC, for example a coating PUF.
- The server could run in parallel, substantially decreasing the time it takes to authenticate a device.

These options can make the protocol more applicable for various use-cases. However, this is not guaranteed because we rely on a Strong PUF which is still in a young research field. If it turns out that practically a Strong PUF cannot be implemented, our PUF-based protocol is only usable with a bounded amount of authentications with respect to a Weak PUF. However, the protocol can be used without without token anonymity and can also easily be adapted to be used with biometric sources like fingerprints and iris-scans.

7.3 Future Work

This research mainly focussed on designing a new type of PUF-based privacy-preserving authentication protocol, namely with the use of an AEAD-scheme. Because our proposal provides a protocol design, a mathematical proof and a proof of concept, three aspects can be further examined in future research.

The design of our protocol might be optimized further. Similar to what this research has achieved with relation to the protocol by Aysu et al. [3]. It would be interesting to see whether we overlooked specific aspects that improve the protocol.

Moreover, our proofs are based on the advantage of a probabilistic polynomial time attacker. Many scientists consider this asymptotic approach outdated and propose a concrete or exact approach specifying precise estimates of the computational complexities of adversarial tasks. It would be interesting to see whether our proof can easily be adapted to this approach.

Our proof of concept might be optimized further. Mainly, future research has to be carried out towards Strong PUF implementations, because these form the basis of our protocol. A Strong PUF that has better quality of PUF responses can substantially reduce the consumption of the device.

7.4 Closing Remarks

Although we have seen that the Concealing KETJE Protocol (CKP) is mathematically secure, it is questionable whether our proof of concept is secure because of the implemented DAPUF. As mentioned, we discovered that the DAPUF response bits are dependent on the input challenge. This makes the DAPUF responses predictable when the challenges are adaptively chosen in a Machine Learning (ML) attack (Definition 2.13, Section 2.3.4). In a personal communication, Maes, who studied PUFs for his PhD [42], even pointed out: "[...] there are not so much arguably secure implementations of Strong PUFs, it is even debatable whether they can be build at all.".

We close this thesis with a recent quote from Bruce Schneier, a renowned specialist in cryptography, security and privacy. This quote matches the conclusions of this thesis and serves as subject for thought.

> "*[...] math has no agency; it can't actually secure anything. For cryptography to work, it needs to be written in software, embedded in a larger software system, managed by an operating system, run on hardware, connected to a network, and configured and operated by users. Each of these steps brings with it difficulties and vulnerabilities.*"
>
> Bruce Schneier, *Cryptography Is Harder than It Looks*, 2016 [59]

BIBLIOGRAPHY

[1] Frederik Armknecht, Daisuke Moriyama, Ahmad-Reza Sadeghi, and Moti Yung. *Topics in Cryptology - CT-RSA 2016: The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings*, chapter Towards a Unified Security Model for Physically Unclonable Functions, pages 271–287. Springer International Publishing, Cham, 2016. ISBN 978-3-319-29485-8. doi: 10.1007/978-3-319-29485-8_16. URL `http://dx.doi.org/10.1007/978-3-319-29485-8_16`. [cited at p. 10]

[2] Avnet Inc. ZedBoard. `http://zedboard.org/product/zedboard`, 2016. Accessed: 2016-06-20. [cited at p. 32, 97, 103, and 104]

[3] Aydin Aysu, Ege Gulcan, Daisuke Moriyama, Patrick Schaumont, and Moti Yung. *Cryptographic Hardware and Embedded Systems – CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, chapter End-To-End Design of a PUF-Based Privacy Preserving Authentication Protocol, pages 556–576. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-48324-4. doi: 10.1007/978-3-662-48324-4_28. URL `http://dx.doi.org/10.1007/978-3-662-48324-4_28`. [cited at p. 3, 4, 29, 31, 32, 51, 52, 53, 55, 57, 71, 76, 77, 81, and 82]

[4] Steve Babbage, C Canniere, Anne Canteaut, Carlos Cid, Henri Gilbert, Thomas Johansson, Matthew Parker, Bart Preneel, Vincent Rijmen, and Matthew Robshaw. The eSTREAM portfolio. *eSTREAM, ECRYPT Stream Cipher Project*, 2008. [cited at p. 44]

[5] Mihir Bellare and Chanathip Namprempre. *Advances in Cryptology — ASIACRYPT 2000: 6th International Conference on the Theory and Application of Cryptology and Information Security Kyoto, Japan, December 3–7, 2000 Proceedings*, chapter Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm, pages 531–545. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-540-44448-0. doi: 10.1007/3-540-44448-3_41. URL `http://dx.doi.org/10.1007/3-540-44448-3_41`. [cited at p. 28]

[6] Elwyn R Berlekamp. *Algebraic Coding Theory: Revised Edition.* World Scientific, 2015. [cited at p. 22]

[7] D.J. Bernstein et al. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. 2016. URL `http://competitions.cr.yp.to/caesar.html`. [cited at p. 44]

[8] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Asche. The KECCAK Reference v3.0. January 2011. URL `http://keccak.noekeon.org/Keccak-reference-3.0.pdf`. [cited at p. 44 and 45]

[9] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Assche. *Selected Areas in Cryptography: 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers*, chapter Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications, pages 320–337. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-28496-0. doi: 10.1007/978-3-642-28496-0_19. URL `http://dx.doi.org/10.1007/978-3-642-28496-0_19`. [cited at p. 45]

[10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Permutation-based Encryption, Authentication and Authenticated encryption. *Directions in Authenticated Ciphers*, July 2012. URL `http://keccak.noekeon.org/KeccakDIAC2012.pdf`. [cited at p. 48]

[11] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Asche, and Ronny Van Keer. CAESAR submission: KETJE v1. March 2014. URL `http://ketje.noekeon.org/Ketje-1.1.pdf`. [cited at p. 44, 48, 49, and 67]

[12] Alex Biryukov, Dumitru-Daniel Dinu, and Dmitry Khovratovich. Argon and Argon2. 2015. [cited at p. 44]

[13] Christoph Bösch, Jorge Guajardo, Ahmad-Reza Sadeghi, Jamshid Shokrollahi, and Pim Tuyls. *Cryptographic Hardware and Embedded Systems – CHES 2008: 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, chapter Efficient Helper Data Key Extractor on FPGAs, pages 181–197. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-85053-3. doi: 10.1007/978-3-540-85053-3_12. URL `http://dx.doi.org/10.1007/978-3-540-85053-3_12`. [cited at p. 38, 39, and 41]

[14] WH Bussey. Galois Field Tables for $p^n \leq 169$. *Bulletin of the American Mathematical Society*, 12(1):22–38, 1905. doi: 10.1090/S0002-9904-1905-01284-2. URL `http://www.ams.org/journals/bull/1905-12-01/S0002-9904-1905-01284-2/`. [cited at p. 19]

[15] R. Chien. Cyclic Decoding Procedures for Bose-Chaudhuri-Hocquenghem Codes. *IEEE Transactions on Information Theory*, 10(4):357–363, Oct 1964. ISSN 0018-9448. doi: 10.1109/TIT.1964.1053699. [cited at p. 23]

[16] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. 1999. [cited at p. 44]

[17] Jeroen Delvaux, Roel Peeters, Dawu Gu, and Ingrid Verbauwhede. A Survey on Lightweight Entity Authentication with Strong PUFs. *ACM Comput. Surv.*, 48(2):26:1–26:42, October 2015. ISSN 0360-0300. doi: 10.1145/2818186. URL `http://doi.acm.org/10.1145/2818186`. [cited at p. 2, 3, 71, and 76]

[18] Yvo Desmedt. *The New Codebreakers: Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*, chapter What is the Future of Cryptography?, pages 109–122. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. ISBN 978-3-662-49301-4. doi: 10.1007/978-3-662-49301-4_7. URL http://dx.doi.org/10.1007/978-3-662-49301-4_7. [cited at p. 29]

[19] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. *Advances in Cryptology - EUROCRYPT 2004: International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004. Proceedings*, chapter Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data, pages 523–540. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-24676-3. doi: 10.1007/978-3-540-24676-3_31. URL http://dx.doi.org/10.1007/978-3-540-24676-3_31. [cited at p. 10, 24, 25, 26, and 33]

[20] Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin, and Adam Smith. Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. *SIAM Journal on Computing*, 38(1):97–139, 2008. doi: 10.1137/060651380. URL http://dx.doi.org/10.1137/060651380. [cited at p. 25 and 27]

[21] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Controlled Physical Random Functions. In *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, pages 149–160, 2002. doi: 10.1109/CSAC.2002.1176287. [cited at p. 10]

[22] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon Physical Random Functions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 148–160, New York, NY, USA, 2002. ACM. ISBN 1-58113-612-9. doi: 10.1145/586110.586132. URL http://doi.acm.org/10.1145/586110.586132. [cited at p. 2, 15, and 97]

[23] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270 – 299, 1984. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/0022-0000(84)90070-9. URL http://www.sciencedirect.com/science/article/pii/0022000084900709. [cited at p. 45]

[24] Daniel Gorenstein and Neal Zierler. A Class of Error-Correcting Codes in p^m Symbols. *Journal of the Society for Industrial and Applied Mathematics*, 9(2):207–214, 1961. doi: 10.1137/0109020. URL http://dx.doi.org/10.1137/0109020. [cited at p. 22]

[25] Glenn Greenwald, Ewen MacAskill, and Laura Poitras. Edward Snowden: the Whistleblower Behind the NSA Surveillance Revelations. *The Guardian*, 9(6), 2013. [cited at p. 29]

[26] J. Guajardo, S. S. Kumar, G. J. Schrijen, and P. Tuyls. Physical Unclonable Functions and Public-Key Crypto for FPGA IP Protection. In *2007 International Conference on Field Programmable Logic and Applications*, pages 189–195, Aug 2007. doi: 10.1109/FPL.2007.4380646. [cited at p. 39]

[27] Jorge Guajardo, Sandeep S. Kumar, Geert-Jan Schrijen, and Pim Tuyls. *Cryptographic Hardware and Embedded Systems - CHES 2007: 9th International Workshop, Vienna, Austria, September 10-13, 2007. Proceedings*, chapter FPGA Intrinsic PUFs and Their Use for IP Protection, pages 63–80. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-74735-2. doi: 10.1007/978-3-540-74735-2_5. URL http://dx.doi.org/10.1007/978-3-540-74735-2_5. [cited at p. 2 and 13]

[28] R. W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 29(2):147–160, 1950. ISSN 1538-7305. doi: 10.1002/j.1538-7305.1950.tb00463.x. URL http://dx.doi.org/10.1002/j.1538-7305.1950.tb00463.x. [cited at p. 8 and 9]

[29] Anthony Herrewege, Stefan Katzenbeisser, Roel Maes, Roel Peeters, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. *Financial Cryptography and Data Security: 16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers*, chapter Reverse Fuzzy Extractors: Enabling Lightweight Mutual Authentication for PUF-Enabled RFIDs, pages 374–389. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-32946-3. doi: 10.1007/978-3-642-32946-3_27. URL http://dx.doi.org/10.1007/978-3-642-32946-3_27. [cited at p. 3 and 33]

[30] Daniel E Holcomb, Wayne P Burleson, Kevin Fu, et al. Initial SRAM State as a Fingerprint and Source of True Random Numbers for RFID Tags. In *Proceedings of the Conference on RFID Security*, volume 7, 2007. [cited at p. 2 and 13]

[31] E. Homsirikamol, W. Diehl, A. Ferozpuri, F. Farahmand, M. U. Sharif, and K. Gaj. A Universal Hardware API for Authenticated Ciphers. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, Dec 2015. doi: 10.1109/ReConFig.2015.7393283. URL http://ece.gmu.edu/~kgaj/publications/conferences/GMU_ReConFig_2015.pdf. [cited at p. 67]

[32] Ari Juels and Stephen A. Weis. Defining Strong Privacy for RFID. *ACM Trans. Inf. Syst. Secur.*, 13(1):7:1–7:23, November 2009. ISSN 1094-9224. doi: 10.1145/1609956.1609963. URL http://doi.acm.org/10.1145/1609956.1609963. [cited at p. 51]

[33] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC Press, 2014. URL http://www.u-cursos.cl/usuario/777719ab2ddbbdb16d99df29431d3036/mi_blog/r/1_book-introduction_to_modern_cryptography.pdf. [cited at p. 2, 8, 14, and 28]

[34] John Kelsey, Bruce Schneier, and Niels Ferguson. *Selected Areas in Cryptography: 6th Annual International Workshop, SAC'99 Kingston, Ontario, Canada, August 9–10, 1999 Proceedings*, chapter Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator, pages 13–33. Springer Berlin Heidelberg, Berlin, Heidelberg,

2000. ISBN 978-3-540-46513-3. doi: 10.1007/3-540-46513-8_2. URL http://dx.doi.org/10.1007/3-540-46513-8_2. [cited at p. 25 and 40]

[35] P. Koeberl, Jiangtao Li, A. Rajan, and Wei Wu. Entropy Loss in PUF-based Key Generation Gchemes: The Repetition Code Pitfall. In *Hardware-Oriented Security and Trust (HOST), 2014 IEEE International Symposium on*, pages 44–49, May 2014. doi: 10.1109/HST.2014.6855566. [cited at p. 40]

[36] J. W. Lee, Daihyun Lim, B. Gassend, G. E. Suh, M. van Dijk, and S. Devadas. A Technique to Build a Secret Key in Integrated Circuits for Identification and Authentication Applications. In *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*, pages 176–179, June 2004. doi: 10.1109/VLSIC.2004.1346548. [cited at p. 2, 14, 37, and 97]

[37] M. Z. Lee, A. M. Dunn, B. Waters, E. Witchel, and J. Katz. Anon-Pass: Practical Anonymous Subscriptions. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 319–333, May 2013. doi: 10.1109/SP.2013.29. [cited at p. 32]

[38] Y. K. Lee, L. Batina, and I. Verbauwhede. Untraceable RFID authentication protocols: Revision of EC-RAC. In *RFID, 2009 IEEE International Conference on*, pages 178–185, April 2009. doi: 10.1109/RFID.2009.4911179. [cited at p. 30]

[39] T. Machida, D. Yamamoto, M. Iwamoto, and K. Sakiyama. A new mode of operation for arbiter PUF to improve uniqueness on FPGA. In *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, pages 871–878, Sept 2014. doi: 10.15439/2014F140. [cited at p. 2, 32, 35, 36, 65, 71, 72, 73, and 97]

[40] T. Machida, D. Yamamoto, M. Iwamoto, and K. Sakiyama. A New Arbiter PUF for Enhancing Unpredictability on FPGA. *The Scientific World Journal*, August 2015. doi: 10.1155/2015/864812. URL http://dx.doi.org/10.1155/2015/864812. [cited at p. 14, 15, 36, 37, 38, 80, and 99]

[41] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error correcting codes*, volume 16. Elsevier, 1977. [cited at p. 8 and 16]

[42] Roel Maes. *Physically Unclonable Functions: Constructions, Properties and Applications*. PhD thesis, Ph. D. thesis, Dissertation, University of KU Leuven, 2012. URL http://lirias.kuleuven.be/bitstream/123456789/353455/1/thesis_online.pdf. [cited at p. 2, 9, 10, 11, 12, 13, 33, 39, 40, and 82]

[43] Roel Maes, Anthony Herrewege, and Ingrid Verbauwhede. *Cryptographic Hardware and Embedded Systems – CHES 2012: 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, chapter PUFKY: A Fully Functional PUF-Based Cryptographic Key Generator, pages 302–319. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-33027-8. doi: 10.1007/978-3-642-33027-8_18. URL http://dx.doi.org/10.1007/978-3-642-33027-8_18. [cited at p. 15, 25, 26, and 40]

[44] M. Majzoobi, M. Rostami, F. Koushanfar, D. S. Wallach, and S. Devadas. Slender PUF Protocol: A Lightweight, Robust, and Secure Authentic-

ation by Substring Matching. In *Security and Privacy Workshops (SPW), 2012 IEEE Symposium on*, pages 33–44, May 2012. doi: 10.1109/SPW.2012.30. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6227682. [cited at p. 3]

[45] J. Massey. Step-by-step Decoding of the Bose-Chaudhuri-Hocquenghem Codes. *IEEE Transactions on Information Theory*, 11(4):580–585, Oct 1965. ISSN 0018-9448. doi: 10.1109/TIT.1965.1053833. [cited at p. 18]

[46] Priya Mathew, Lismi Augustine, Tomson Devis, et al. Hardware Implementation of (63, 51) BCH Encoder and Decoder For WBAN Using LFSR and BMA. *arXiv preprint arXiv:1408.2908*, 2014. URL http://arxiv.org/pdf/1408.2908v2. [cited at p. 67]

[47] Jorge Castiñeira Moreira and Patrick Guy Farrell. *Essentials of Error-Control Coding*. John Wiley & Sons, 2006. [cited at p. 8, 18, and 22]

[48] Daisuke Moriyama, Shin'ichiro Matsuo, and Moti Yung. PUF-Based RFID Authentication Secure and Private under Memory Leakage. Cryptology ePrint Archive, Report 2013/712, 2013. http://eprint.iacr.org/2013/712.pdf. [cited at p. 3, 51, 52, 53, 55, 57, 71, 76, and 77]

[49] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. *Federal Information Processing Standards Publication*, August 2015. doi: 10.6028/NIST.FIPS.202. URL http://dx.doi.org/10.6028/NIST.FIPS.202. [cited at p. 45]

[50] Ching Yu Ng, Willy Susilo, Yi Mu, and Rei Safavi-Naini. *Computer Security – ESORICS 2009: 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, chapter New Privacy Results on Synchronized RFID Authentication Protocols against Tag Tracing, pages 321–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-04444-1. doi: 10.1007/978-3-642-04444-1_20. URL http://dx.doi.org/10.1007/978-3-642-04444-1_20. [cited at p. 51]

[51] OWASP Internet of Things Project. OWASP Top 10 IoT Vulnerabilities. http://www.owasp.org/index.php/OWASP_Internet_of_Things_Project, 2014. Accessed: 2016-06-20. [cited at p. 5]

[52] Ravikanth Pappu, Ben Recht, Jason Taylor, and Neil Gershenfeld. Physical One-Way Functions. *Science*, 297(5589):2026–2030, 2002. ISSN 0036-8075. doi: 10.1126/science.1074376. URL http://science.sciencemag.org/content/297/5589/2026. [cited at p. 2]

[53] Python Software Foundation. Python Release Python 2.7.10. https://www.python.org/downloads/release/python-2710/, 2015. Accessed: 2016-06-20. [cited at p. 68]

[54] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960. doi: 10.1137/0108018. URL http://dx.doi.org/10.1137/0108018. [cited at p. 22]

[55] Alfréd Rényi. On measures of entropy and information. In *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics*, pages 547–561, Berkeley, Calif., 1961. University of California Press. URL http://projecteuclid.org/euclid.bsmsp/1200512181. [cited at p. 8 and 9]

[56] Phillip Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 98–107, New York, NY, USA, 2002. ACM. ISBN 1-58113-612-9. doi: 10.1145/586110.586125. URL http://doi.acm.org/10.1145/586110.586125. [cited at p. 8 and 28]

[57] U. Rührmair, J. Sölter, F. Sehnke, X. Xu, A. Mahmoud, V. Stoyanova, G. Dror, J. Schmidhuber, W. Burleson, and S. Devadas. PUF Modeling Attacks on Simulated and Silicon Data. *IEEE Transactions on Information Forensics and Security*, 8(11):1876–1891, Nov 2013. ISSN 1556-6013. doi: 10.1109/TIFS.2013.2279798. [cited at p. 13 and 35]

[58] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. Modeling Attacks on Physical Unclonable Functions. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 237–249, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0245-6. doi: 10.1145/1866307.1866335. URL http://doi.acm.org/10.1145/1866307.1866335. [cited at p. 13 and 35]

[59] B. Schneier. Cryptography Is Harder than It Looks. *IEEE Security Privacy*, 14(1):87–88, Jan 2016. ISSN 1540-7993. doi: 10.1109/MSP.2016.7. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7397719. [cited at p. 82]

[60] Bruce Schneier. The Internet of Things Will Be the World's Biggest Robot. http://www.schneier.com/blog/archives/2016/02/the_internet_of_1.html, February 2016. Accessed: 2016-06-20. [cited at p. 5]

[61] C. E. Shannon. A Mathematical Theory of Communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, January 2001. ISSN 1559-1662. doi: 10.1145/584091.584093. URL http://doi.acm.org/10.1145/584091.584093. [cited at p. 8 and 9]

[62] Pim Tuyls and Boris Škorić. *AmIware Hardware Technology Drivers of Ambient Intelligence*, chapter Secret Key Generation from Classical Physics: Physical Uncloneable Functions, pages 421–447. Springer Netherlands, Dordrecht, 2006. ISBN 978-1-4020-4198-3. doi: 10.1007/1-4020-4198-5_20. URL http://dx.doi.org/10.1007/1-4020-4198-5_20. [cited at p. 2 and 15]

[63] Henk CA Van Tilborg. *Error-correcting codes: a first course*. Studentlitteratur, 1993. [cited at p. 8 and 16]

[64] Xilinx Inc. Zynq-7000 All Programmable SoC Overview, Product Specification DS190 (v1.9). http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf, January 2016. Accessed: 2016-06-20. [cited at p. 32 and 103]

ABBREVIATIONS

AcE	Average-case Extractor. [used at p. 25, 27, 101]
AE	Authenticated Encryption. [used at p. 3, 4, 6, 7, 28, 58, 102]
AEAD	Authenticated Encryption with Associated Data. [used at p. 4–7, 28, 29, 33, 39, 40, 44, 48, 49, 56–60, 67, 76, 80, 81, 101]
AES	Advanced Encryption Standard. [used at p. 44]
API	Application Program Interface. [used at p. 67, 81]
ARM	Advanced RISC Machine. [used at p. 32, 62]
ASIC	Application Specific IC. [used at p. 103]
AXI	Advanced Extensible Interface. [used at p. 62, 63]
BCH	Bose-Chaudhuri-Hocquenghem. [used at p. 3, 7, 17–20, 39–44, 61, 63, 64, 67, 74, 75, 95, 97]
BRAM	Block RAM. [used at p. 74, 75]
BSC	Binary Symmetric Channel. [used at p. 16, 17, 31, 97]
CAESAR	Competition for Authenticated Encryption: Security, Applicability and Robustness. [used at p. 4, 44, 67, 81]
CI	Central Interconnect. [used at p. 62]
CKP	Concealing KETJE Protocol. [used at p. 6, 29, 34, 51, 76, 77, 82]
CODEC	coder/decoder. [used at p. 104]
CPA	Chosen-Plaintext Attack. [used at p. 56]
CRP	Challenge-Response Pair. [used at p. 2, 37, 76, 80]
DAPUF	3-1 Double Arbiter PUF. [used at p. 2, 29, 35–39, 61, 63–66, 73–75, 77, 80, 82, 96, 97, 99]
DDR3	Double Data Rate Type Three. [used at p. 103]
DoS	Denial-of-Service. [used at p. 76]
DSP	Digital Signal Processor. [used at p. 74, 75, 103]
EA	Entropy Accumulator. [used at p. 4, 40, 41, 44, 49, 73, 76, 80, 101]
ECRYPT	European Network of Excellence in Cryptology. [used at p. 44]

FE	Fuzzy Extractor. [used at p. 2–4, 6, 7, 24, 26, 27, 29, 30, 33, 38, 41, 49, 55–58, 60, 76, 77, 81, 101, 102]
FF	Flip-Flop. [used at p. 103]
FMC	FPGA Mezzanine Card. [used at p. 104]
FPGA	Field Programmable Gate Array. [used at p. 32, 37, 65, 77, 80, 103]
GMU	George Mason University. [used at p. 67, 81]
HDMI	High-Definition Multimedia Interface. [used at p. 104]
HLS	High Level Synthesis. [used at p. 63]
HW	Hardware. [used at p. 77]
I/O	Input/Output. [used at p. 104]
I2C	Inter-IC Sound. [used at p. 104]
IC	Integrated Circuit. [used at p. 14–16, 33, 81]
ID	identification number. [used at p. 31]
IDE	Integrated Development Environment. [used at p. 63]
IoT	Internet of Things. [used at p. 2, 4, 5, 44, 74, 75, 80, 81]
iSim	ISE Simulator. [used at p. 63]
JTAG	Joint Test Action Group. [used at p. 62, 65, 103]
LFSR	Linear Feedback Shift Register. [used at p. 43, 48, 64, 67, 72]
LSB	Least Significant Bit. [used at p. 72]
LUT	Look-Up Table. [used at p. 65, 75–77, 99, 103]
MAC	Message Authentication Code. [used at p. 28]
MACC	Multiply Accumulator. [used at p. 103]
MitM	Man-in-the-Middle. [used at p. 30, 52, 53, 57, 59]
ML	Machine Learning. [used at p. 13, 14, 35, 72, 82]
MUX	Multiplexer. [used at p. 65]
NAND	negative-AND. [used at p. 14, 65]
NIST	National Institute of Standards and Technology. [used at p. 45]
NSA	National Security Agency. [used at p. 29]
NVM	Non-Volatile Memory. [used at p. 2, 32, 53, 59, 60, 62, 76, 77, 80]
OLED	Organic LED. [used at p. 104]
OTG	On-The-Go. [used at p. 103]
OTP	One-Time Pad. [used at p. 14]
OWASP	Open Web Application Security Project. [used at p. 5]
Pblock	Physical Block. [used at p. 63]
PC	Personal Computer. [used at p. 4, 61, 62, 68, 77]
PHC	Password Hashing Competition. [used at p. 44]
PL	Programmable Logic. [used at p. 32, 62, 65, 104]
Pmod	Peripheral Module. [used at p. 104]

PRF Pseudo-Random Function. [used at p. 3, 4, 25, 29]
PRNG Pseudo-Random Number Generator. [used at p. 3]
PS Processing System. [used at p. 32, 62, 63, 104]
PSK pre-shared key. [used at p. 2, 3, 29]
PUF Physically Unclonable Function. [used at p. 2–7, 10–17, 24, 26, 29–44, 52, 55, 57–60, 62, 64, 68, 71–73, 76, 77, 80–82, 95–97, 99, 101, 102]

QSPI Quad-SPI. [used at p. 103]

RAM Random-Access Memory. [used at p. 103]
RFE Reverse FE. [used at p. 3, 29, 38, 39, 44]
RFID Radio-Frequency Identification. [used at p. 2–5, 44, 74, 75, 80, 81]
ROPUF Ring Oscillator PUF. [used at p. 13, 15, 97]

SCA Side-Channel Analysis. [used at p. 68]
SD Secure Digital. [used at p. 62, 103]
SDK Software Development Kit. [used at p. 62]
SE Strong Extractor. [used at p. 4, 24–26, 101]
SKC Symmetric Key Cryptography. [used at p. 2]
SKE Symmetric Key Encryption. [used at p. 4, 29, 74]
SoC System on Chip. [used at p. 32, 62, 63, 73, 74, 77, 80, 81, 103]
SRAM Static RAM. [used at p. 3, 13, 14, 32, 77]
SS Secure Sketch. [used at p. 24, 26, 27, 41, 101]
SUV Secret Unique Value. [used at p. 49]
SW Software. [used at p. 77]

Tcl Tool Command Language. [used at p. 65]
TRNG True Random Number Generator. [used at p. 3, 31, 35, 76]

UART Universal Asynchronous Receiver/Transmitter. [used at p. 62, 68, 77, 103]
USB Universal Serial Bus. [used at p. 62, 65, 68, 103]

VGA Video Graphics Array. [used at p. 104]
VHDL VHSIC Hardware Description Language. [used at p. 63]

WSW World-Sized Web. [used at p. 5]
WWW World Wide Web. [used at p. 5]

XADC Xilinx Analog to Digital Converter. [used at p. 104]
XOR exclusive-OR. [used at p. 8, 35, 36, 42–44, 49, 67]

NOMENCLATURE

$\mathbf{Adv}^{\mathrm{IND}^*}_{\Psi,\mathcal{A}}(k)$	Privacy advantage of attacker $\mathcal{A}$ on authentication protocol Ψ. [used at p. 54]
$\mathbf{Adv}^{\mathbf{Sec}}_{\Psi,\mathcal{A}}(k)$	Security advantage of attacker $\mathcal{A}$ on authentication protocol Ψ, i.e. the probability that $\mathbf{Exp}^{\mathbf{Sec}}_{\Psi,\mathcal{A}}(k)$ outputs $B_0 = 1$ on the condition that I of G has no matching session. [used at p. 53, 57]
$\mathcal{C}_{\mathbf{BCH}}(n,k,t)$	Binary Bose-Chaudhuri-Hocquenghem (BCH) code of order m, length $n = 2^m - 1$, distance d and error correcting capability $t = \lfloor \frac{d-1}{2} \rfloor$, see Definition 2.15. [used at p. 19, 20, 39, 43]
$\mathcal{C}_{\mathbf{REP}}(n,1,t)$	Binary repetition code with length n and error correcting capacity $t = \lfloor \frac{n-1}{2} \rfloor$, see Definition 2.14. [used at p. 17, 18, 39]
$\mathcal{D}^{\mathrm{inter}}_{\mathcal{P}}$	Inter-distance for a random challenge in PUF class $\mathcal{P}$, see Definition 2.9. [used at p. 12]
$\mathcal{D}^{\mathrm{intra}}_{\mathcal{P}}$	Intra-distance for a random PUF instance and a random challenge in PUF class $\mathcal{P}$, see Definition 2.7. [used at p. 11, 12]
$\mathbf{Exp}^{\mathrm{IND}^*-b}_{\Psi,\mathcal{A}}(k)$	Indistinguishability privacy experiment of attacker $\mathcal{A}$ on authentication protocol Ψ with security parameter k. [used at p. 54, 97]
$\mathbf{Exp}^{\mathbf{Sec}}_{\Psi,\mathcal{A}}(k)$	Security experiment of attacker $\mathcal{A}$ on authentication protocol Ψ with security parameter k. [used at p. 52, 53, 95, 97]
$\mathbf{h}(p)$	Binary entropy function of binary random variable $Y \leftarrow \{0,1\}$ with probability p, see Formula 2.2.1. [used at p. 9, 38, 73]
$\mathbf{H}(Y)$	Shannon entropy of binary string $Y \leftarrow \mathcal{Y}$, see Definition 2.3. [used at p. 9]
$\tilde{\mathbf{H}}_\infty(Y)$	Min-entropy, or Rényi entropy of binary string $Y \leftarrow \mathcal{Y}$, see Definition 2.4. [used at p. 9]
$\mathbf{HD}(Y,Y')$	Hamming distance of binary strings $Y, Y' \leftarrow \mathcal{Y}$, see Definition 2.1. [used at p. 8, 9, 36, 37]
$\mathbf{HW}(Y)$	Hamming weight of binary string $Y \leftarrow \mathcal{Y}$, see Definition 2.2. [used at p. 9, 38]

$\mathcal{P}$	PUF class. [used at p. 10–13, 55, 95, 96]
$\mathcal{P}_{\mathbf{3\text{-}1}}$	DAPUF class. [used at p. 35–38]
p_e	Bit error probability. [used at p. 11, 26, 34–36, 38, 39]
$\mathbf{Pr}(x)$	Probability that x occurs. [used at p. 10]
puf	PUF instance $\mathbf{puf} \in \mathcal{P}$. [used at p. 11–13, 17, 36–38, 96]
$\mathbf{SD}(A, B)$	Statistical distance between two probability distributions A and B, see Definition 2.5. [used at p. 10]
X	PUF challenge. [used at p. 11–13, 17, 35–38, 55]
Y	PUF response. [used at p. 11–13, 17, 35–38, 57, 58, 60, 73]

LIST OF FIGURES

LIST OF TABLES

LIST OF THEOREMS, LEMMAS, COROLLARIES AND DEFINITIONS

APPENDIX

A

HARDWARE SPECIFICATIONS

A.1 Zedboard

This appendix contains the basic specifications of the Zedboard, illustrated in Figure A.1 [2]. The Zedboard includes a Zynq®-7000 by Xilinx Inc. [64].

- Zynq®-7000 28 nm All Programmable SoC XC7Z020-CLG484-1
  - Equivalent to Xilinx 7 Series Programmable Logic Artix®-7 FPGA
  - 85K Programmable Logic Cells (∼1.3M Approximate ASIC Gates)
  - 53,200 LUTs
  - 106,400 Flip-Flops (FFs)
  - 560 KB Extensible Block Random-Access Memory (RAM) (140x36 Kb Blocks)
  - 220 Programmable Digital Signal Processor (DSP) Slices (18x25 Multiply Accumulators (MACCs))
- Memory
  - 512 MB Double Data Rate Type Three (DDR3)
  - 256 Mb Quad-SPI (QSPI) Flash
  - 4 GB SD-card
- Onboard Universal Serial Bus (USB)-Joint Test Action Group (JTAG) Programming
- 10/100/1000 Ethernet
- USB On-The-Go (OTG) 2.0 and USB-Universal Asynchronous Receiver/Transmitter (UART)

- Processing System (PS) & Programmable Logic (PL) Input/Output (I/O) expansion (FPGA Mezzanine Card (FMC), Peripheral Module (Pmod) Compatible, Xilinx Analog to Digital Converter (XADC))
- Multiple displays (1080p High-Definition Multimedia Interface (HDMI), 8-bit Video Graphics Array (VGA), 128 x 32 Organic LED (OLED))
- Inter-IC Sound (I2C) Audio coder/decoder (CODEC)

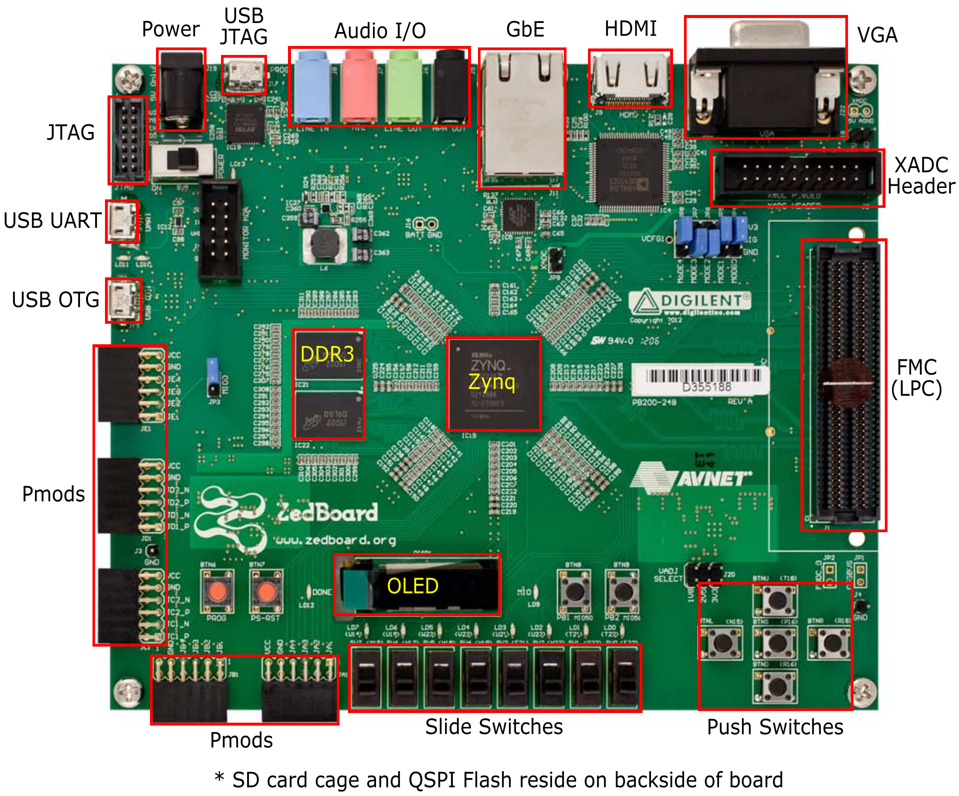


Figure A.1: Functional Overlay of the Zedboard by Avnet Inc. [2].

APPENDIX

B

GALOIS FIELD TABLES

B.1 $\mathbf{GF}(2^4)$ generated by $\mathbf{p}(x) = x^4 + x + 1$

Table B.1: Galois Field table $\mathbf{GF}(2^4)$ as generated by the primitive polynomial $\mathbf{p}(x) = x^4 + x + 1$. REP denotes 'representation'.

Power REP	**Polynomial REP**							**Binary REP**
0							0	$\{0,0,0,0\}$
1							1	$\{0,0,0,1\}$
α					α			$\{0,0,1,0\}$
α^2			α^2					$\{0,1,0,0\}$
α^3	α^3							$\{1,0,0,0\}$
α^4					α	$+$	1	$\{0,0,1,1\}$
α^5			α^2	$+$	α			$\{0,1,1,0\}$
α^6	α^3	$+$	α^2					$\{1,1,0,0\}$
α^7	α^3			$+$	α	$+$	1	$\{1,0,1,1\}$
α^8			α^2			$+$	1	$\{0,1,0,1\}$
α^9	α^3			$+$	α			$\{1,0,1,0\}$
α^{10}			α^2	$+$	α	$+$	1	$\{0,1,1,1\}$
α^{11}	α^3	$+$	α^2	$+$	α			$\{1,1,1,0\}$
α^{12}	α^3	$+$	α^2	$+$	α	$+$	1	$\{1,1,1,1\}$
α^{13}	α^3	$+$	α^2			$+$	1	$\{1,1,0,1\}$
α^{14}	α^3					$+$	1	$\{1,0,0,1\}$

B.2 $\mathbf{GF}(2^8)$ generated by $\mathbf{p}(x) = x^4 + x^3 + x^2 + 1$

Table B.2: Galois Field table $\mathbf{GF}(2^8)$ as generated by the primitive polynomial $\mathbf{p}(x) = x^4 + x^3 + x^2 + 1$. REP denotes 'representation'.

Power REP	**Polynomial REP**	**Binary REP**
0	0	$\{0,0,0,0,0,0,0,0\}$
1	1	$\{0,0,0,0,0,0,0,1\}$
α	α	$\{0,0,0,0,0,0,1,0\}$
α^2	α^2	$\{0,0,0,0,0,1,0,0\}$
α^3	α^3	$\{0,0,0,0,1,0,0,0\}$
α^4	α^4	$\{0,0,0,1,0,0,0,0\}$
α^5	α^5	$\{0,0,1,0,0,0,0,0\}$
α^6	α^6	$\{0,1,0,0,0,0,0,0\}$
α^7	α^7	$\{1,0,0,0,0,0,0,0\}$
α^8	$\alpha^4 + \alpha^3 + \alpha^2 + 1$	$\{0,0,0,1,1,1,0,1\}$
α^9	$\alpha^5 + \alpha^4 + \alpha^3 + \alpha$	$\{0,0,1,1,1,0,1,0\}$
α^{10}	$\alpha^6 + \alpha^5 + \alpha^4 + \alpha^2$	$\{0,1,1,1,0,1,0,0\}$
$\vdots$	$\vdots$	$\vdots$
α^{132}	$\alpha^7 + \alpha^5 + \alpha^4 + \alpha^3$	$\{1,0,1,1,1,0,0,0\}$
α^{133}	$\alpha^6 + \alpha^5 + \alpha^3 + \alpha^2 + 1$	$\{0,1,1,0,1,1,0,1\}$
α^{134}	$\alpha^7 + \alpha^6 + \alpha^4 + \alpha^3 + \alpha$	$\{1,1,0,1,1,0,1,0\}$
α^{135}	$\alpha^7 + \alpha^5 + \alpha^3 + 1$	$\{1,0,1,0,1,0,0,1\}$
α^{136}	$\alpha^6 + \alpha^3 + \alpha^2 + \alpha + 1$	$\{0,1,0,0,1,1,1,1\}$
α^{137}	$\alpha^7 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha$	$\{1,0,0,1,1,1,1,0\}$
α^{138}	$\alpha^5 + 1$	$\{0,0,1,0,0,0,0,1\}$
α^{139}	$\alpha^6 + \alpha$	$\{0,1,0,0,0,0,1,0\}$
α^{140}	$\alpha^7 + \alpha^2$	$\{1,0,0,0,0,1,0,0\}$
α^{141}	$\alpha^4 + \alpha^2 + 1$	$\{0,0,0,1,0,1,0,1\}$
α^{142}	$\alpha^5 + \alpha^3 + \alpha$	$\{0,0,1,0,1,0,1,0\}$
α^{143}	$\alpha^6 + \alpha^4 + \alpha^2$	$\{0,1,0,1,0,1,0,0\}$
α^{144}	$\alpha^7 + \alpha^5 + \alpha^3$	$\{1,0,1,0,1,0,0,0\}$
$\vdots$	$\vdots$	$\vdots$
α^{243}	$\alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + 1$	$\{0,1,1,1,1,1,0,1\}$
α^{244}	$\alpha^7 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha$	$\{1,1,1,1,1,0,1,0\}$
α^{245}	$\alpha^7 + \alpha^6 + \alpha^5 + \alpha^3 + 1$	$\{1,1,1,0,1,0,0,1\}$
α^{246}	$\alpha^7 + \alpha^6 + \alpha^3 + \alpha^2 + \alpha + 1$	$\{1,1,0,0,1,1,1,1\}$
α^{247}	$\alpha^7 + \alpha + 1$	$\{1,0,0,0,0,0,1,1\}$
α^{248}	$\alpha^4 + \alpha^3 + \alpha + 1$	$\{0,0,0,1,1,0,1,1\}$
α^{249}	$\alpha^5 + \alpha^4 + \alpha^2 + \alpha$	$\{0,0,1,1,0,1,1,0\}$
α^{250}	$\alpha^6 + \alpha^5 + \alpha^3 + \alpha^2$	$\{0,1,1,0,1,1,0,0\}$
α^{251}	$\alpha^7 + \alpha^6 + \alpha^4 + \alpha^3$	$\{1,1,0,1,1,0,0,0\}$
α^{252}	$\alpha^7 + \alpha^5 + \alpha^3 + \alpha^2 + 1$	$\{1,0,1,0,1,1,0,1\}$
α^{253}	$\alpha^6 + \alpha^2 + \alpha + 1$	$\{0,1,0,0,0,1,1,1\}$
α^{254}	$\alpha^7 + \alpha^3 + \alpha^2 + \alpha$	$\{1,0,0,0,1,1,1,0\}$

APPENDIX C

SOURCE CODE LISTINGS

This appendix lists the most relevant source code produced in this research. For clarity, similar lines of code have been omitted which is indicated by comments.

C.1 Software

C.1.1 Server

```python
#*******************************************************************************************************#
# Author:   J.G. (Gerben) Geltink # g.geltink@gmail.com
# Module:   Server Side Protocol  # main.py
#   This module describes the server side protocol of the device DAPUF in the Conceiling Ketje
#   Protocol as presented by the guiding MSc thesis: Geltink, J. G. (2016). "Conceiling Ketje: A
#   Lightweight PUF-based Privacy Preserving Authentication Protocol".
#
# Copyright (c) 2016 J.G. Geltink under MIT license
#*******************************************************************************************************#
import commands
import timeit
import serial
import random
import struct

ser = 0

def init_serial():
    global ser          #Must be declared in Each Function
    ser = serial.Serial()
    ser.baudrate = 115200
    ser.port = '/dev/ttyACM0' #If Using Linux
    ser.timeout = 10
    ser.open()          #Opens SerialPort


def xor_bitarray(a, b):
    return map(lambda x: x[0] ^ x[1], zip(a, b))

def int_to_bitlist(n, length):
    return [n >> i & 1 for i in range(length-1,-1,-1)]

def bitlist_to_int(bitlist):
    out = 0
    for bit in bitlist:
        out = (out << 1) | bit
    return out



```

```
def int_to_hexstring(i, n):
    string = hex(i);
    if string[len(string)-1] == "L":
        return string[2:len(string)-1].zfill(2*n)
    else:
        return string[2:].zfill(2*n)

def int_to_bytelist(n, length):
    return [ n >> (8*i) & 0xff for i in range(length-1,-1,-1)]

def bch_dec(a_prime, hd):
    '''Code derived from Robert Morelos-Zaragoza: Encoder/decoder for binary BCH codes in C (Version 3.1, 1997)
    using BCH(255,139,15)
    NOTE: not in constant time!'''
    t = 15
    length = 255
    n = 255
    cw_prime = xor_bitarray(a_prime, hd)
    cw = cw_prime[:]
    # print 'cw_prime\t', cw_prime
        # index->polynomial form: alpha_to[] contains j=alpha^i;
    alpha_to = [1, 2, 4, 8, 16, 32, 64, 128, 113, 226, 181, 27, 54, 108, 216, 193, 243, 151, 95, 190, 13, 26, 52,
          104, 208, 209, 211, 215, 223, 207, 239, 175, 47, 94, 188, 9, 18, 36, 72, 144, 81, 162, 53, 106, 212,
         217, 195, 247, 159, 79, 158, 77, 154, 69, 138, 101, 202, 229, 187, 7, 14, 28, 56, 112, 224, 177, 19,
         38, 76, 152, 65, 130, 117, 234, 165, 59, 118, 236, 169, 35, 70, 140, 105, 210, 213, 219, 199, 255, 143,
          111, 222, 205, 235, 167, 63, 126, 252, 137, 99, 198, 253, 139, 103, 206, 237, 171, 39, 78, 156, 73,
         146, 85, 170, 37, 74, 148, 89, 178, 21, 42, 84, 168, 33, 66, 132, 121, 242, 149, 91, 182, 29, 58, 116,
         232, 161, 51, 102, 204, 233, 163, 55, 110, 220, 201, 227, 183, 31, 62, 124, 248, 129, 115, 230, 189,
         11, 22, 44, 88, 176, 17, 34, 68, 136, 97, 194, 245, 155, 71, 142, 109, 218, 197, 251, 135, 127, 254,
         141, 107, 214, 221, 203, 231, 191, 15, 30, 60, 120, 240, 145, 83, 166, 61, 122, 244, 153, 67, 134, 125,
          250, 133, 123, 246, 157, 75, 150, 93, 186, 5, 10, 20, 40, 80, 160, 49, 98, 196, 249, 131, 119, 238,
         173, 43, 86, 172, 41, 82, 164, 57, 114, 228, 185, 3, 6, 12, 24, 48, 96, 192, 241, 147, 87, 174, 45, 90,
          180, 25, 50, 100, 200, 225, 179, 23, 46, 92, 184, 0]
        # polynomial form -> index form: index_of[j=alpha^i] = i
    index_of = [-1, 0, 1, 231, 2, 207, 232, 59, 3, 35, 208, 154, 233, 20, 60, 183, 4, 159, 36, 66, 209, 118, 155,
          251, 234, 245, 21, 11, 61, 130, 184, 146, 5, 122, 160, 79, 37, 113, 67, 106, 210, 224, 119, 221, 156,
         242, 252, 32, 235, 213, 246, 135, 22, 42, 12, 140, 62, 227, 131, 75, 185, 191, 147, 94, 6, 70, 123,
         195, 161, 53, 80, 167, 38, 109, 114, 203, 68, 51, 107, 49, 211, 40, 225, 189, 120, 111, 222, 240, 157,
         116, 243, 128, 253, 205, 33, 18, 236, 163, 214, 98, 247, 55, 136, 102, 23, 82, 43, 177, 13, 169, 141,
         89, 63, 8, 228, 151, 132, 72, 76, 218, 186, 125, 192, 200, 148, 197, 95, 174, 7, 150, 71, 217, 124,
         199, 196, 173, 162, 97, 54, 101, 81, 176, 168, 88, 39, 188, 110, 239, 115, 127, 204, 17, 69, 194, 52,
         166, 108, 202, 50, 48, 212, 134, 41, 139, 226, 74, 190, 93, 121, 78, 112, 105, 223, 220, 241, 31, 158,
         65, 117, 250, 244, 10, 129, 145, 254, 230, 206, 58, 34, 153, 19, 182, 237, 15, 164, 46, 215, 171, 99,
         86, 248, 143, 56, 180, 137, 91, 103, 29, 24, 25, 83, 26, 44, 84, 178, 27, 14, 45, 170, 85, 142, 179,
         90, 28, 64, 249, 9, 144, 229, 57, 152, 181, 133, 138, 73, 92, 77, 104, 219, 30, 187, 238, 126, 16, 193,
          165, 201, 47, 149, 216, 198, 172, 96, 100, 175, 87]
    t2 = 2 * t; #2* error correcting capability
    # first form the syndromes
    syn_error = False
    s = [0 for i in range(t2+1)]
    for i in range(1, t2+1):
        for j in range(length):
            if (cw_prime[j] != 0):
                s[i] ^= alpha_to[(i * j) % n]
        if (s[i] != 0):
            syn_error = True
        s[i] = index_of[s[i]]
    # print 's\t', s[1:]
    # print 'syn_error\t', syn_error
    if syn_error:
        # /* initialise table entries */
        d = [0 for i in range(t2 + 2)]
        elp = [[0 for i in range(t2 + 2)] for j in range(t2 + 2)]
        l = [0 for i in range(t2 + 2)]
        u_lu = [0 for i in range(t2 + 2)]
        d[0] = 0;     #/* index form */
        d[1] = s[1];    #/* index form */
        elp[0][0] = 0;    #/* index form */
        elp[1][0] = 1;    #/* polynomial form */
        for i in range(1,t2):
            elp[0][i] = -1; #/* index form */
            elp[1][i] = 0;  #/* polynomial form */
        l[0] = 0;
        l[1] = 0;
        u_lu[0] = -1;
        u_lu[1] = 0;
        u = 0;           # 0 -> t2+1
        while True:
            u += 1
            if (d[u] == -1):
                l[u + 1] = l[u]
                for i in range(l[u] + 1):
                    elp[u + 1][i] = elp[u][i]
                    elp[u][i] = index_of[elp[u][i]]
            else:
                # search for words with greatest u_lu[q] for
                # which d[q]!=0
                q = u - 1
                while ((d[q] == -1) and (q > 0)):
                    q -= 1
                # have found first non-zero d[q]
                if (q > 0):
                    j = q
                    while True:
                        j -= 1
                        if ((d[j] != -1) and (u_lu[q] < u_lu[j])):
                            q = j;
```

```python
                        if not (j > 0):
                            break
                # have now found q such that d[u]!=0 and u_lu[q] is maximum
                # store degree of new elp polynomial
                if (l[u] > l[q] + u - q):
                    l[u + 1] = l[u]
                else:
                    l[u + 1] = l[q] + u - q
                # form new elp(x)
                for i in range(t2):
                    elp[u + 1][i] = 0
                for i in range(l[q] + 1):
                    if (elp[q][i] != -1):
                        elp[u + 1][i + u - q] = alpha_to[(d[u] + n - d[q] + elp[q][i]) % n]
                for i in range(l[u] + 1):
                    elp[u + 1][i] ^= elp[u][i]
                    elp[u][i] = index_of[elp[u][i]]
            u_lu[u + 1] = u - l[u + 1]
            # form (u+1)th discrepancy
            if (u < t2):
                # no discrepancy computed on last iteration
                if (s[u + 1] != -1):
                    d[u + 1] = alpha_to[s[u + 1]]
                else:
                    d[u + 1] = 0
                for i in range(1,l[u+1]+1):
                    if ((s[u + 1 - i] != -1) and (elp[u + 1][i] != 0)):
                        d[u + 1] ^= alpha_to[(s[u + 1 - i] + index_of[elp[u + 1][i]]) % n]
                # put d[u+1] into index form
                d[u + 1] = index_of[d[u + 1]]
            if not ((u < t2) and (l[u + 1] <= t)):
                break
        u += 1
        if (l[u] <= t): # Can correct errors
            # put elp into index form
            for i in range(l[u] + 1):
                elp[u][i] = index_of[elp[u][i]]
            sigma = []
            for i in range(l[u] + 1):
                sigma += [elp[u][i]]
            # print 'sigma(x) =\t', sigma
            # Chien search: find roots of the error location polynomial
            reg = [0 for x in range(l[u]+1)]
            root = [0 for x in range(l[u])]
            loc = [0 for x in range(l[u])]
            for i in range(1,l[u]+1):
                reg[i] = elp[u][i]
            count = 0
            for i in range(1,n+1):
                q = 1
                for j in range(1, l[u]+1):
                    if (reg[j] != -1):
                        reg[j] = (reg[j] + j) % n
                        q ^= alpha_to[reg[j]]
                if (q == 0): # store root and error location number indices
                    root[count] = i
                    loc[count] = n - i
                    count += 1
            if (count == l[u]):
                # print 'Roots:\t', loc
                print "\t#errors corrected BCH\t", str(len(loc))
                for i in range(l[u]):
                    cw[loc[i]] ^= 1
            else:
                1 == 1
                print '\tBCH incomplete decoding: errors detected'
    # print 'cw\t', cw
    return xor_bitarray(cw, hd)

def rep_decode(hd_rep, y_1):
    errors = 0
    e = [7, 11, 13, 14, 15] #faulty syndromes
    r_p = 0
    ry_p = 0
    for i in range(255):
        hd = (hd_rep >> 4*i) &  0b1111
        y = (y_1 >> 5*i) &  0b11111
        y_pb = (y_1 >> 5*i) &  0b1

        if y_pb == 1:  #compute syndromes
            s = 0b1111 ^ (y >> 1) ^ hd
        else:
            s = 0 ^ (y >> 1) ^ hd

        if s in e:
            r_pb = y_pb ^ 1
            errors += 1
        else:
            r_pb = y_pb

        r_p += r_pb << i #LSB on right

    print "\t#errors corrected REP\t", str(errors)
    return r_p


x = random.randint(1,2**64)
```

```
x_list = int_to_bytelist(x, 16)
init_serial()
ser.write([0x90])
ser.write(x_list[:4])
ser.write(x_list[4:8])
y_1 = ser.readline()[0:-1]
print "device set up:\t", y_1[-64:], "<Y> (showing rightmost 32 bytes)"
print "-----------------------------\n"
ser.close()

while(1):

    n1 = random.randint(1,2**128)

    init_serial()

    n1_list = int_to_bytelist(n1, 16)
    ser.write([0x90])
    ser.write(n1_list[:4])
    ser.write(n1_list[4:8])
    ser.write(n1_list[8:12])
    ser.write(n1_list[12:])

    print "challenge sent:\t\t", int_to_hexstring(n1,16), "<N1>"

    hd_rep = ser.readline()[0:-1]
    hd_bch = ser.readline()[1:-1]
    c = ser.readline()[1:-1]
    n2 = ser.readline()[1:-1]
    t = ser.readline()[1:-1]

    ser.close()

    print "responses received:\t", t, "<HD, N2, C1, T1> (only showing T1)"

    hd_rep = int(hd_rep, 16)
    y_1 = int(y_1, 16)
    hd_bch = int(hd_bch,16)
    hd = int_to_bitlist(hd_bch, 255)
    c = [c[i:i+8] for i in range(0,len(c),8)]
    c.reverse()
    c = "".join(c) # c is feeded in reverse
    clen = 160
    klen = 16
    nlen= 16
    adlen = 16
    tlen = 16
    auth = False

    start = timeit.default_timer()
    r_p = rep_decode(hd_rep, y_1)
    a_prime = int_to_bitlist(r_p, 255)
    r_pp = bch_dec(a_prime, hd)
    r_pp = bitlist_to_int(r_pp)
    r_pp_str = int_to_hexstring(r_pp, 32)

    key = r_pp_str[:32] # left part of r_pp
    nonce =  n2
    ad = r_pp_str[32:] # right part of r_pp
    ad = ad[24:] + ad[16:24] + ad[8:16] + ad[:8] #ad is feeded in reverse!
    challenge = r"./Ketje_enc " + key + " " + str(klen) + " " + nonce + " " + str(nlen) + " " + ad + " " + str(
            adlen)# + c + str(clen) + t + str(tlen)
    response = commands.getstatusoutput(challenge)
    dev_key = response[1].split()[0]
    # print "dev_key = [" + dev_key + "]"


    nonce = int_to_hexstring(n1,16)
    ad = n2
    ad = ad[24:] + ad[16:24] + ad[8:16] + ad[:8] #ad is feeded in reverse!

    challenge = r"./Ketje_dec " + dev_key + " " + str(klen) + " " + nonce + " " + str(nlen) + " " + ad + " " +
            str(adlen) + " " + c + " " + str(clen) + " " + t + " " + str(tlen)
    response = commands.getstatusoutput(challenge)
    y_2 = response[1].split()[0]
    y_2 = [y_2[i:i+8] for i in range(0,len(y_2),8)]
    y_2.reverse()
    y_2 = "".join(y_2) # y_2 is received in reverse

    if (y_2 == "".zfill(320)):
        t2 = random.randint(1,2**128)
        y_1 = int_to_hexstring(y_1,160)
    else:
        auth = True
        y_1 = y_2
        nonce = int_to_hexstring(n1,16)
        ad = int_to_hexstring(int(n2, 16) + 2**127, 16)
        ad = ad[24:] + ad[16:24] + ad[8:16] + ad[:8] #ad is feeded in reverse!
        challenge = r"./Ketje_enc " + dev_key + " " + str(klen) + " " + nonce + " " + str(nlen) + " " + ad + " "
                + str(adlen)# + c + str(clen) + t + str(tlen)
        response = commands.getstatusoutput(challenge)
        t2 = response[1].split()[0]
        t2 = int(t2,16)
    end = timeit.default_timer()

    if auth:
        print "\033[92mSUCCESFULL AUTHENTICATION\033[0m"
```

```
        print "PUF resp updated\t", y_2[-64:], "<Y> (showing rightmost 32 bytes)"
    else:
        print "\033[91mNO AUTHENTICATION\033[0m"

    t2_list = int_to_bytelist(t2, 16)
    init_serial()
    ser.write([0x90])
    ser.write(t2_list[:4])
    ser.write(t2_list[4:8])
    ser.write(t2_list[8:12])
    ser.write(t2_list[12:])
    ser.close()
    print "authenticator sent:\t", int_to_hexstring(t2,16), "<T2>"

    print "Elapsed time:\t", (end-start)
    print "------------------------------"
    raw_input("\033[95mPress ENTER for a new authentication\033[0m")
    print "------------------------------"
```

C.2 Hardware

C.2.1 CKP Device

```
/**********************************************************************************************************/
// Author:  J.G. (Gerben) Geltink //  g.geltink@gmail.com
// Module:  CKP Device            //  ckp_dev.c
//    This module describes the device in the Conceiling Ketje Protocol as presented by the
//    guiding MSc thesis: Geltink, J. G. (2016). "Conceiling Ketje: A Lightweight PUF-based
//    Privacy Preserving Authentication Protocol".
//
// Copyright (c) 2016 J.G. (Gerben) Geltink under MIT license
/**********************************************************************************************************/

#include "platform.h"
#include "xbasic_types.h"
#include "xparameters.h"
#include <stdio.h>

Xuint32 *baseaddr_p = (Xuint32 *)XPAR_CKP_DEV_2_0_S00_AXI_BASEADDR;

void rcv_int(int a){ //receives a 32-bit word
  int i = 0;
  int val = 0;
  for(i=0;i<4;i++)
    val = val + (inbyte() << (8*(3-i)));
  *(baseaddr_p+a) = val;}

void wait_rcv(){ //waits for a byte indicating the start of a word
  for(;;)
    if (inbyte() == 0x90)
      break;}

int main() {
  init_platform();

  //Setup phase
  wait_rcv(); //wait for starting byte
  rcv_int(1); //receive left word of X1
  rcv_int(0); //receive right word of X1
  *(baseaddr_p+4) = 0xFACADE01;  //set data ready
  for(;;) //wait until data is ready
    if (*(baseaddr_p+93) == 0xACCEDE01)
      break;
  //send Y
  xil_printf("%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x\n\r", *(baseaddr_p+133), *(baseaddr_p+132), *(baseaddr_p+131), *(baseaddr_p+130), *(baseaddr_p+129), *(baseaddr_p+128), *(baseaddr_p+127), *(baseaddr_p+126), *(baseaddr_p+125), *(baseaddr_p+124), *(baseaddr_p+123), *(baseaddr_p+122), *(baseaddr_p+121), *(baseaddr_p+120), *(baseaddr_p+119), *(baseaddr_p+118), *(baseaddr_p+117), *(baseaddr_p+116), *(baseaddr_p+115), *(baseaddr_p+114), *(baseaddr_p+113), *(baseaddr_p+112), *(baseaddr_p+111), *(baseaddr_p+110), *(baseaddr_p+109), *(baseaddr_p+108), *(baseaddr_p+107), *(baseaddr_p+106), *(baseaddr_p+105), *(baseaddr_p+104), *(baseaddr_p+103), *(baseaddr_p+102), *(baseaddr_p+101), *(baseaddr_p+100), *(baseaddr_p+99), *(baseaddr_p+98), *(baseaddr_p+97), *(baseaddr_p+96), *(baseaddr_p+95), *(baseaddr_p+94));

  //Authentication phase (repeat forever)
  while(1){
    wait_rcv(); //wait for starting byte
    rcv_int(3); //receive leftmost word of N1
    rcv_int(2); //receive next word of N1
    rcv_int(1); //receive next word of N1
    rcv_int(0); //receive rightmost word of N1
    *(baseaddr_p+4) = 0xFACADE02;  //set data ready
    for(;;) //wait until data is ready
      if (*(baseaddr_p+93) == 0xACCEDE02)
```

```c
            break;
        //send HD_rep
        xil_printf("%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x\n\r", *(baseaddr_p+36), *(baseaddr_p+35), *(baseaddr_p+34), *(baseaddr_p+33), *(baseaddr_p+32), *(baseaddr_p+31), *(baseaddr_p+30), *(baseaddr_p+29), *(baseaddr_p+28), *(baseaddr_p+27), *(baseaddr_p+26), *(baseaddr_p+25), *(baseaddr_p+24), *(baseaddr_p+23), *(baseaddr_p+22), *(baseaddr_p+21), *(baseaddr_p+20), *(baseaddr_p+19), *(baseaddr_p+18), *(baseaddr_p+17), *(baseaddr_p+16), *(baseaddr_p+15), *(baseaddr_p+14), *(baseaddr_p+13), *(baseaddr_p+12), *(baseaddr_p+11), *(baseaddr_p+10), *(baseaddr_p+9), *(baseaddr_p+8), *(baseaddr_p+7), *(baseaddr_p+6), *(baseaddr_p+5));
        //send HD_bch
        xil_printf("%08x%08x%08x%08x%08x%08x%08x%08x\n\r", *(baseaddr_p+44), *(baseaddr_p+43), *(baseaddr_p+42), *(baseaddr_p+41), *(baseaddr_p+40), *(baseaddr_p+39), *(baseaddr_p+38), *(baseaddr_p+37));
        //send C2
        xil_printf("%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x\n\r", *(baseaddr_p+84), *(baseaddr_p+83), *(baseaddr_p+82), *(baseaddr_p+81), *(baseaddr_p+80), *(baseaddr_p+79), *(baseaddr_p+78), *(baseaddr_p+77), *(baseaddr_p+76), *(baseaddr_p+75), *(baseaddr_p+74), *(baseaddr_p+73), *(baseaddr_p+72), *(baseaddr_p+71), *(baseaddr_p+70), *(baseaddr_p+69), *(baseaddr_p+68), *(baseaddr_p+67), *(baseaddr_p+66), *(baseaddr_p+65), *(baseaddr_p+64), *(baseaddr_p+63), *(baseaddr_p+62), *(baseaddr_p+61), *(baseaddr_p+60), *(baseaddr_p+59), *(baseaddr_p+58), *(baseaddr_p+57), *(baseaddr_p+56), *(baseaddr_p+55), *(baseaddr_p+54), *(baseaddr_p+53), *(baseaddr_p+52), *(baseaddr_p+51), *(baseaddr_p+50), *(baseaddr_p+49), *(baseaddr_p+48), *(baseaddr_p+47), *(baseaddr_p+46), *(baseaddr_p+45));
        //send N2
        xil_printf("%08x%08x%08x%08x\n\r", *(baseaddr_p+88), *(baseaddr_p+87), *(baseaddr_p+86), *(baseaddr_p+85));
        //send T1
        xil_printf("%08x%08x%08x%08x\n\r", *(baseaddr_p+92), *(baseaddr_p+91), *(baseaddr_p+90), *(baseaddr_p+89));
        wait_rcv(); //wait for starting byte
        rcv_int(3); //receive leftmost word of T2
        rcv_int(2); //receive next word of T2
        rcv_int(1); //receive next word of T2
        rcv_int(0); //receive rightmost word of T2
        *(baseaddr_p+4) = 0xACCEDE02;  //set data ready
    } return 0; }
```

C.2.2 CKP Device Core

```vhdl
--------------------------------------------------------------------------------------------------------
-- Author:  J.G. (Gerben) Geltink --  g.geltink@gmail.com
-- Module:  CKP Device Core       --  ckp_dev_core.vhd
--    This module describes the device core of the Conceiling Ketje Protocol as presented by the
--    guiding MSc thesis: Geltink, J. G. (2016). "Conceiling Ketje: A Lightweight PUF-based
--    Privacy Preserving Authentication Protocol".
--
-- Copyright (c) 2016 J.G. Geltink under MIT license
--------------------------------------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY ckp_dev_core IS
  PORT (  clk    : IN  STD_LOGIC;
      pdi1  : IN  STD_LOGIC_VECTOR (31 DOWNTO 0); --public data input
      pdi2  : IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
      pdi3  : IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
      pdi4  : IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
      pdi5  : IN  STD_LOGIC_VECTOR (31 DOWNTO 0); --input instruction
      pdo1  : OUT STD_LOGIC_VECTOR (31 DOWNTO 0); --public data output
      pdo2  : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
      -- pdo2 to pdo87 OMITTED FOR CLARITY
      pdo88 : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
      pdo89 : OUT STD_LOGIC_VECTOR (31 DOWNTO 0); --output instruction
      pdo90 : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
      -- pdo91 to pdo127 OMITTED FOR CLARITY
      pdo128  : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
      pdo129  : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
      );
END ckp_dev_core;

ARCHITECTURE Behavioral OF ckp_dev_core IS
  --COMPONENT DECLARATIONS
  COMPONENT DAPUF
  PORT(
    clk        : IN  STD_LOGIC;
    rst_n      : IN  STD_LOGIC;
    idata      : IN  STD_LOGIC_VECTOR(63 DOWNTO 0);
    esig       : IN  STD_LOGIC;
    Answer       : OUT std_logic
    );
  END COMPONENT;
  COMPONENT bch255_139_31enc
  PORT(
    clk        : IN  STD_LOGIC;
    reset      : IN  STD_LOGIC;
    din        : IN  STD_LOGIC;
    vdin       : OUT STD_LOGIC;
    dout       : OUT std_logic
    );
```

```
END COMPONENT;
COMPONENT ketjeSr --component received from Guido Bertoni
PORT(
  clk         : IN  STD_LOGIC;
  rst         : IN  STD_LOGIC;
  npub        : IN  STD_LOGIC_VECTOR(127 DOWNTO 0);
  nsec        : IN  STD_LOGIC_VECTOR(127 DOWNTO 0);
  key         : IN  STD_LOGIC_VECTOR(127 DOWNTO 0);
  rdkey       : IN  STD_LOGIC_VECTOR(127 DOWNTO 0);
  bdi         : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
  exp_tag       : IN  STD_LOGIC_VECTOR(127 DOWNTO 0);
  len_a       : IN  STD_LOGIC_VECTOR(63 DOWNTO 0);
  len_d       : IN  STD_LOGIC_VECTOR(63 DOWNTO 0);
  key_ready     : IN  STD_LOGIC;
  key_updated     : OUT STD_LOGIC;
  key_needs_update  : IN  STD_LOGIC;
  rdkey_ready     : IN  STD_LOGIC;
  rdkey_read      : OUT STD_LOGIC;
  npub_ready      : IN  STD_LOGIC;
  npub_read     : OUT STD_LOGIC;
  nsec_ready      : IN  STD_LOGIC;
  nsec_read     : OUT STD_LOGIC;
  bdi_ready     : IN  STD_LOGIC;
  bdi_proc      : IN  STD_LOGIC;
  bdi_ad        : IN  STD_LOGIC;
  bdi_nsec      : IN  STD_LOGIC;
  bdi_pad       : IN  STD_LOGIC;
  bdi_decrypt     : IN  STD_LOGIC;
  bdi_eot       : IN  STD_LOGIC;
  bdi_eoi       : IN  STD_LOGIC;
  bdi_read      : OUT STD_LOGIC;
  bdi_size      : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
  bdi_valid_bytes   : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
  bdi_pad_loc     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
  bdi_nodata      : IN  STD_LOGIC;
  exp_tag_ready   : IN  STD_LOGIC;
  bdo_ready     : IN  STD_LOGIC;
  bdo_write     : OUT STD_LOGIC;
  bdo         : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
  bdo_size      : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
  bdo_nsec      : OUT STD_LOGIC;
  tag_ready     : IN  STD_LOGIC;
  tag_write     : OUT STD_LOGIC;
  tag         : OUT STD_LOGIC_VECTOR(127 DOWNTO 0);
  msg_auth_done   : OUT STD_LOGIC;
  msg_auth_valid    : OUT std_logic
  );
END COMPONENT;

--SIGNAL DECLARATIONS
SIGNAL esig_IN      : STD_LOGIC := '0';                   --PUF enable SIGNAL
SIGNAL res        : STD_LOGIC := '0';
SIGNAL challenge      : STD_LOGIC_VECTOR(63 DOWNTO 0);
SIGNAL seed1      : STD_LOGIC_VECTOR(11 DOWNTO 0) := x"001";        --TODO: into NVM
SIGNAL chal1      : STD_LOGIC_VECTOR(39 DOWNTO 0) := (OTHERS => '0');   --TODO: into NVM
SIGNAL setup      : STD_LOGIC := '0';
SIGNAL seed2      : STD_LOGIC_VECTOR(11 DOWNTO 0) := (OTHERS => '0');
SIGNAL chal2      : STD_LOGIC_VECTOR(39 DOWNTO 0) := (OTHERS => '0');
SIGNAL pufresp      : STD_LOGIC_VECTOR(1279 DOWNTO 0) := (OTHERS => '0');
SIGNAL c1       : STD_LOGIC_VECTOR(1279 DOWNTO 0) := (OTHERS => '0');
SIGNAL pdi        : STD_LOGIC_VECTOR(127 DOWNTO 0) := (OTHERS => '0');
SIGNAL nce2       : STD_LOGIC_VECTOR(126 DOWNTO 0) := (OTHERS => '0');
SIGNAL t1       : STD_LOGIC_VECTOR(127 DOWNTO 0) := (OTHERS => '0');
SIGNAL t2       : STD_LOGIC_VECTOR(127 DOWNTO 0) := (OTHERS => '0');
SIGNAL bchrnd     : STD_LOGIC_VECTOR(138 DOWNTO 0) := (OTHERS => '0');
SIGNAL rep_res      : STD_LOGIC_VECTOR(254 DOWNTO 0) := (OTHERS => '0');
SIGNAL hd_rep     : STD_LOGIC_VECTOR(1019 DOWNTO 0) := (OTHERS => '0');
SIGNAL bch_cw     : STD_LOGIC_VECTOR(115 DOWNTO 0) := (OTHERS => '0');
SIGNAL dev_key      : STD_LOGIC_VECTOR(127 DOWNTO 0) := (OTHERS => '0');
SIGNAL ctrl_puf_resp  : STD_LOGIC := '0';
SIGNAL ctrl_rst_puf   : STD_LOGIC := '0';
SIGNAL ctrl_rst_rep   : STD_LOGIC := '0';
SIGNAL ctrl_index   : STD_LOGIC_VECTOR(11 DOWNTO 0) := (OTHERS => '0');
SIGNAL ctrl_index2    : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
SIGNAL ctrl_cnt     : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
SIGNAL ctrl_rst_bch   : STD_LOGIC := '0';
SIGNAL ctrl_rst_ketje : STD_LOGIC := '0';
SIGNAL rst_bch      : STD_LOGIC := '0';
SIGNAL din_bch      : STD_LOGIC := '0';
SIGNAL dout_bch     : STD_LOGIC;
SIGNAL ctrl_rst_lfsr  : STD_LOGIC := '0';
SIGNAL ctrl_lfsr    : STD_LOGIC := '0';
SIGNAL ctrl_comp_t2   : STD_LOGIC := '0';
SIGNAL ctrl_checkandupd : STD_LOGIC := '0';
SIGNAL lfsr       : STD_LOGIC_VECTOR(11 DOWNTO 0);
SIGNAL ctrl_ketje_mode  : STD_LOGIC_VECTOR(1 DOWNTO 0) := (OTHERS => '0');
SIGNAL npub_reg     : STD_LOGIC_VECTOR(127 DOWNTO 0) := (OTHERS => '0');
SIGNAL key_reg      : STD_LOGIC_VECTOR(127 DOWNTO 0) := (OTHERS => '0');
SIGNAL ad_reg     : STD_LOGIC_VECTOR(127 DOWNTO 0) := (OTHERS => '0');
SIGNAL rst_ketje    : STD_LOGIC := '0';
SIGNAL npub       : STD_LOGIC_VECTOR(127 DOWNTO 0) := (OTHERS => '0');
SIGNAL key        : STD_LOGIC_VECTOR(127 DOWNTO 0) := (OTHERS => '0');
SIGNAL bdi        : STD_LOGIC_VECTOR(31 DOWNTO 0) := (OTHERS => '0');
SIGNAL key_ready    : STD_LOGIC := '0';
SIGNAL key_needs_update : STD_LOGIC := '0';
SIGNAL npub_ready   : STD_LOGIC := '0';
SIGNAL bdi_ready    : STD_LOGIC := '0';
```

```
  SIGNAL bdi_ad     : STD_LOGIC := '0';
  SIGNAL bdi_nsec     : STD_LOGIC := '0';
  SIGNAL bdi_eot      : STD_LOGIC := '0';
  SIGNAL bdi_eoi      : STD_LOGIC := '0';
  SIGNAL bdo        : STD_LOGIC_VECTOR(31 DOWNTO 0);
  SIGNAL tag_write    : STD_LOGIC;
  SIGNAL tag        : STD_LOGIC_VECTOR(127 DOWNTO 0);

  --UNUSED SIGNAL DECLARATIONS (will be synthesized out)
  SIGNAL rst_puf      : STD_LOGIC := '0';                    --PUF reset signal
  --OMITTED: vdin_bch_unused, nsec, rdkey, exp_tag, len_a, len_d, rdkey_ready, nsec_ready, bdi_proc, bdi_pad,
        bdi_decrypt, bdi_size, bdi_valid_bytes, bdi_pad_loc, bdi_nodata, exp_tag_ready, bdo_ready, tag_ready,
        key_updated, rdkey_read, npub_read, nsec_read, bdi_read, bdo_write, bdo_size, bdo_nsec, msg_auth_done,
        msg_auth_vali,

  --STATE DECLARATIONS
  TYPE puf_state_type IS (idle,chall,exec,recv);
  SIGNAL puf_state : puf_state_type := idle;
  TYPE rep_state_type IS (idle,exec,set);
  SIGNAL rep_state : rep_state_type := idle;
  TYPE bch_state_type IS (idle,restart,exec,get,set);
  SIGNAL bch_state : bch_state_type := idle;
  TYPE resp_state_type IS (idle,get_seed2,get_pufresp1,get_bchrnd,get_chal2,get_nce2,get_pufresp2,wtaeadenc,
        checkandupd);
  SIGNAL resp_state : resp_state_type := idle;
  TYPE ketje_state_type IS (idle,ldkey,ldnce,initstate,ldad,wtad,comptag,ldmsg,wtmsg,lstmsg);
  SIGNAL ketje_state : ketje_state_type := idle;

BEGIN
  DAPUF_0: DAPUF PORT MAP (
    clk => clk,
    rst_n => rst_puf,
    idata => challenge,
    esig => esig_in,
    Answer => res
    );
  BCH_ENCODER: bch255_139_31enc PORT MAP (
    clk => clk,
    reset => rst_bch,
    din => din_bch,
    vdin => vdin_bch_unused,
    dout => dout_bch
    );
  KETJE: ketjeSr PORT MAP (
    clk => clk,
    rst => rst_ketje,
    npub => npub,
    nsec => nsec,
    key => key,
    rdkey => rdkey,
    bdi => bdi,
    exp_tag => exp_tag,
    len_a => len_a,
    len_d => len_d,
    key_ready => key_ready,
    key_updated => key_updated,
    key_needs_update => key_needs_update,
    rdkey_ready => rdkey_ready,
    rdkey_read => rdkey_read,
    npub_ready => npub_ready,
    npub_read => npub_read,
    nsec_ready => nsec_ready,
    nsec_read => nsec_read,
    bdi_ready => bdi_ready,
    bdi_proc => bdi_proc,
    bdi_ad => bdi_ad,
    bdi_nsec => bdi_nsec,
    bdi_pad => bdi_pad,
    bdi_decrypt => bdi_decrypt,
    bdi_eot => bdi_eot,
    bdi_eoi => bdi_eoi,
    bdi_read => bdi_read,
    bdi_size => bdi_size,
    bdi_valid_bytes => bdi_valid_bytes,
    bdi_pad_loc => bdi_pad_loc,
    bdi_nodata => bdi_nodata,
    exp_tag_ready => exp_tag_ready,
    bdo_ready => bdo_ready,
    bdo_write => bdo_write,
    bdo => bdo,
    bdo_size => bdo_size,
    bdo_nsec => bdo_nsec,
    tag_ready => tag_ready,
    tag_write => tag_write,
    tag => tag,
    msg_auth_done => msg_auth_done,
    msg_auth_valid => msg_auth_valid
    );

pdo81 <= nce2(31 DOWNTO 0);
pdo82 <= nce2(63 DOWNTO 32);
pdo83 <= nce2(95 DOWNTO 64);
pdo84 <= '0' & nce2(126 DOWNTO 96);
pdo85 <= t1(31 DOWNTO 0);
pdo86 <= t1(63 DOWNTO 32);
pdo87 <= t1(95 DOWNTO 64);
pdo88 <= t1(127 DOWNTO 96);
```

```vhdl

pdi <= pdi4 & pdi3 & pdi2 & pdi1;

lfsr_proc : PROCESS(clk, ctrl_rst_lfsr)
BEGIN
  IF (ctrl_rst_lfsr = '1') THEN
    IF (ctrl_lfsr = '1') THEN
      lfsr <= seed1(11 DOWNTO 1) & '1';
    ELSE
      lfsr <= x"001";
    END IF;
  ELSIF RISING_EDGE(clk) THEN
    lfsr(0) <= lfsr(11);
    lfsr(1) <= lfsr(0);
    lfsr(2) <= lfsr(1);
    lfsr(3) <= lfsr(2) XOR lfsr(11);
    lfsr(4) <= lfsr(3) XOR lfsr(11);
    lfsr(5) <= lfsr(4);
    lfsr(6) <= lfsr(5);
    lfsr(7) <= lfsr(6) XOR lfsr(11);
    lfsr(8) <= lfsr(7);
    lfsr(9) <= lfsr(8);
    lfsr(10) <= lfsr(9);
    lfsr(11) <= lfsr(10);
  END IF;
END PROCESS;  --p(x) = 1001100100001

resp_proc : PROCESS(clk)
BEGIN
  IF RISING_EDGE(clk) THEN
    CASE resp_state IS
      WHEN idle =>
      ctrl_rst_ketje <= '0';
        IF (pdi5 = x"facade01") AND (setup = '0') THEN
          chal1 <= pdi1 & pdi2(7 DOWNTO 0);
          resp_state <= get_seed2;
          ctrl_index <= x"000";
          ctrl_rst_puf <= '1';
          pdo89 <= x"facade01"; --data not ready
        ELSIF (pdi5 = x"facade02") THEN
          pdo90 <= (OTHERS => '0');
          pdo91 <= (OTHERS => '0');
          pdo92 <= (OTHERS => '0');
          -- pdo93 to pdo126 OMITTED FOR CLARITY
          pdo127 <= (OTHERS => '0');
          pdo128 <= (OTHERS => '0');
          pdo129 <= (OTHERS => '0');
          resp_state <= get_seed2;
          ctrl_index <= x"000";
          ctrl_rst_puf <= '1';
          pdo89 <= x"facade02"; --data not ready
        END IF;
      WHEN get_seed2 =>
        IF (ctrl_index = x"000") THEN
          ctrl_rst_puf <= '0';
          ctrl_rst_lfsr <= '1';
          ctrl_lfsr <='1';
          --challenge <= chal1(39 DOWNTO 36) [ & chal1(35 DOWNTO 33) & lfsr(11) & '0']* EXAMPLE
          challenge <= chal1(39 DOWNTO 36) & chal1(35 DOWNTO 33) & lfsr(11) & '0' & chal1(32 DOWNTO 30) & lfsr(10) & '0' & chal1(29 DOWNTO 27) & lfsr(9) & '0' & chal1(26 DOWNTO 24) & lfsr(8) & '0' & chal1(23 DOWNTO 21) & lfsr(7) & '0' & chal1(20 DOWNTO 18) & lfsr(6) & '0' & chal1(17 DOWNTO 15) & lfsr(5) & '0' & chal1(14 DOWNTO 12) & lfsr(4) & '0' & chal1(11 DOWNTO 9) & lfsr(3) & '0' & chal1(8 DOWNTO 6) & lfsr(2) & '0' & chal1(5 DOWNTO 3) & lfsr(1) & '0' & chal1(2 DOWNTO 0) & lfsr(0) & '0';
        END IF;
        IF (ctrl_index = x"00c") THEN
          resp_state <= get_pufresp1;
          ctrl_index <= x"000";
          seed1 <= seed2;
          ctrl_lfsr <='0';
        ELSIF (ctrl_puf_resp = '1') THEN
          --challenge <= chal1(39 DOWNTO 36) [ & chal1(35 DOWNTO 33) & lfsr(11) & '0']* OMITTED
          ctrl_rst_lfsr <= '0';
          seed2(TO_INTEGER(UNSIGNED(ctrl_index))) <= res;
          ctrl_index <= STD_LOGIC_VECTOR(UNSIGNED(ctrl_index) + 1);
        END IF;
      WHEN get_pufresp1 =>
        IF (ctrl_index = x"000") THEN
          ctrl_rst_lfsr <= '1';
          --challenge <= chal1(39 DOWNTO 36) [ & chal1(35 DOWNTO 33) & '0' & lfsr(11)]* OMITTED
        END IF;
        IF (ctrl_index = x"4fb") THEN
          IF (pdi5 = x"facade01") AND (setup = '0') THEN
            resp_state <= idle;
            setup <= '1';
            pdo90 <= pufresp(31 DOWNTO 0);
            pdo91 <= pufresp(63 DOWNTO 32);
            pdo92 <= pufresp(95 DOWNTO 64);
            -- pdo93 to pdo126 OMITTED FOR CLARITY
            pdo127 <= pufresp(1215 DOWNTO 1184);
            pdo128 <= pufresp(1247 DOWNTO 1216);
            pdo129 <= pufresp(1279 DOWNTO 1248);
            pdo89 <= x"accede01"; --data ready
          ELSE
            resp_state <= get_bchrnd;
            ctrl_rst_rep <= '1'; --start repetition code in rep_proc
            ctrl_index <= x"000";
```

```
          END IF;
        ELSIF (ctrl_puf_resp = '1') THEN
          --challenge <= chal1(39 DOWNTO 36) [ & chal1(35 DOWNTO 33) & '0' & lfsr(11)]* OMITTED
          ctrl_rst_lfsr <= '0';
          pufresp(TO_INTEGER(UNSIGNED(ctrl_index))) <= res;
          ctrl_index <= STD_LOGIC_VECTOR(UNSIGNED(ctrl_index) + 1);
        END IF;
      WHEN get_bchrnd =>
        IF (ctrl_index = x"000") THEN
          ctrl_rst_lfsr <= '1';
          ctrl_rst_rep <= '0';
          ctrl_lfsr <='1';
          --challenge <= chal1(39 DOWNTO 36) [ & chal1(35 DOWNTO 33) & lfsr(11) & '0']* OMITTED
        END IF;
        IF (ctrl_index = x"08b") THEN
          resp_state <= get_chal2;
          ctrl_rst_bch <= '1'; --start bch code in bch_proc
          ctrl_ketje_mode <= "01";
          ctrl_index <= x"000";
          ctrl_lfsr <='0';
        ELSIF (ctrl_puf_resp = '1') THEN
          --challenge <= chal1(39 DOWNTO 36) [ & chal1(35 DOWNTO 33) & lfsr(11) & '0']* OMITTED
          ctrl_rst_lfsr <= '0';
          bchrnd(TO_INTEGER(UNSIGNED(ctrl_index))) <= res;
          ctrl_index <= STD_LOGIC_VECTOR(UNSIGNED(ctrl_index) + 1);
        END IF;
      WHEN get_chal2 =>
        IF (ctrl_index = x"028") THEN
          resp_state <= get_nce2;
          ctrl_index <= x"000";
          ctrl_lfsr <='0';
        ELSIF (ctrl_puf_resp = '1') THEN
          --challenge <= chal1(39 DOWNTO 36) [ & chal1(35 DOWNTO 33) & lfsr(11) & '0']* OMITTED
          ctrl_rst_bch <= '0';
          chal2(TO_INTEGER(UNSIGNED(ctrl_index))) <= res;
          ctrl_index <= STD_LOGIC_VECTOR(UNSIGNED(ctrl_index) + 1);
        END IF;
      WHEN get_nce2 =>
        IF (ctrl_index = x"07f") THEN
          resp_state <= get_pufresp2;
          ctrl_rst_ketje <= '1'; --start computing dev-key
          ctrl_index <= x"000";
          ctrl_lfsr <='0';
        ELSIF (ctrl_puf_resp = '1') THEN
          --challenge <= chal1(39 DOWNTO 36) [ & chal1(35 DOWNTO 33) & lfsr(11) & '0']* OMITTED
          nce2(TO_INTEGER(UNSIGNED(ctrl_index))) <= res;
          ctrl_index <= STD_LOGIC_VECTOR(UNSIGNED(ctrl_index) + 1);
        END IF;
      WHEN get_pufresp2 =>
        IF (ctrl_index = x"000") THEN
          ctrl_rst_lfsr <= '1';
          --challenge <= chal2(39 DOWNTO 36) [ & chal2(35 DOWNTO 33) & '0' & lfsr(11)]* OMITTED
        END IF;
        IF (ctrl_index = x"4fb") THEN
          resp_state <= wtaeadenc;
          ctrl_index <= x"000";
          ctrl_rst_ketje <= '1'; --start computing C^1,T^1
          ctrl_ketje_mode <= "10";
        ELSIF (ctrl_puf_resp = '1') THEN
          --challenge <= chal2(39 DOWNTO 36) [ & chal2(35 DOWNTO 33) & '0' & lfsr(11)]* OMITTED
          ctrl_rst_lfsr <= '0';
          ctrl_rst_ketje <= '0';
          pufresp(TO_INTEGER(UNSIGNED(ctrl_index))) <= res;
          ctrl_index <= STD_LOGIC_VECTOR(UNSIGNED(ctrl_index) + 1);
        END IF;
      WHEN wtaeadenc =>
        IF (ctrl_comp_t2 = '1') THEN
          ctrl_rst_ketje <= '1'; --start computing T^2
          ctrl_ketje_mode <= "11";
          resp_state <= checkandupd;
          pdo89 <= x"facade09";
        ELSE
          ctrl_rst_ketje <= '0';
        END IF;
      WHEN checkandupd =>
        IF (ctrl_checkandupd = '1') THEN
          pdo89 <= x"accede02"; --data ready
        ELSE
          ctrl_rst_ketje <= '0';
        END IF;
        IF (pdi5 = x"accede02") AND (ctrl_checkandupd = '1') THEN
          IF (pdi = t2) THEN
            chal1 <= chal2;
            pdo89 <= x"accede03"; --finished AND accepted
          END IF;
           resp_state <= idle;
        END IF;
      WHEN OTHERS =>
        ctrl_rst_ketje <= '0';
    END CASE;
  END IF;
END PROCESS;

puf_proc : PROCESS(clk)
BEGIN
  IF (ctrl_rst_puf = '1') THEN
    puf_state <= chal1;
```

```
  ELSIF RISING_EDGE(clk) THEN
    CASE puf_state IS
      WHEN chall =>
        esig_in <= '1';
        puf_state <= exec;
      WHEN exec =>
        esig_in <= '0';
        ctrl_puf_resp <= '1';
        puf_state <= recv;
      WHEN recv =>
        ctrl_puf_resp <= '0';
        puf_state <= chall;
      WHEN OTHERS =>
    END CASE;
  END IF;
END PROCESS;

rep_proc : PROCESS(clk)
BEGIN
  IF (ctrl_rst_rep = '1') THEN
    rep_state <= exec;
  ELSIF RISING_EDGE(clk) THEN
    CASE rep_state IS
      WHEN exec =>
        FOR i IN 0 TO 254 LOOP
          rep_res(i) <= pufresp(i*5);
          hd_rep(i*4) <= pufresp(i*5) XOR pufresp(i*5+1);
          hd_rep(i*4+1) <= pufresp(i*5) XOR pufresp(i*5+2);
          hd_rep(i*4+2) <= pufresp(i*5) XOR pufresp(i*5+3);
          hd_rep(i*4+3) <= pufresp(i*5) XOR pufresp(i*5+4);
        END LOOP;
        rep_state <= set;
      WHEN set =>
        pdo1 <= hd_rep(31 DOWNTO 0);
        pdo2 <= hd_rep(63 DOWNTO 32);
        pdo3 <= hd_rep(95 DOWNTO 64);
        -- pdo4 to pdo29 OMITTED FOR CLARITY
        pdo30 <= hd_rep(959 DOWNTO 928);
        pdo31 <= hd_rep(991 DOWNTO 960);
        pdo32 <= "0000" & hd_rep(1019 DOWNTO 992);
        rep_state <= idle;
      WHEN OTHERS =>
    END CASE;
  END IF;
END PROCESS;

bch_proc : PROCESS(clk)
BEGIN
  IF (ctrl_rst_bch = '1') THEN
    bch_state <= restart;
    ctrl_index2 <= x"00";
  ELSIF RISING_EDGE(clk) THEN
    CASE bch_state IS
      WHEN restart =>
        rst_bch <= '1';
        bch_state <= exec;
      WHEN exec =>
        IF (ctrl_index2 = x"8c") THEN
          bch_state <= get;
          ctrl_index2 <= x"00";
        ELSE
          rst_bch <= '0';
          din_bch <= bchrnd(TO_INTEGER(UNSIGNED(ctrl_index2)));
          ctrl_index2 <= STD_LOGIC_VECTOR(UNSIGNED(ctrl_index2) + 1);
        END IF;
      WHEN get =>
        IF (ctrl_index2 = x"74") THEN
          bch_state <= set;
          ctrl_index2 <= x"00";
        ELSE
          bch_cw(TO_INTEGER(UNSIGNED(ctrl_index2))) <= dout_bch;
          ctrl_index2 <= STD_LOGIC_VECTOR(UNSIGNED(ctrl_index2) + 1);
        END IF;
      WHEN set =>
        pdo33 <= bch_cw(31 DOWNTO 0) XOR rep_res(31 DOWNTO 0);
        pdo34 <= bch_cw(63 DOWNTO 32) XOR rep_res(63 DOWNTO 32);
        pdo35 <= bch_cw(95 DOWNTO 64) XOR rep_res(95 DOWNTO 64);
        pdo36 <= (bchrnd(11 DOWNTO 0) XOR rep_res(127 DOWNTO 116)) & (bch_cw(115 DOWNTO 96) XOR rep_res(115 DOWNTO 96));
        pdo37 <= bchrnd(43 DOWNTO 12) XOR rep_res(159 DOWNTO 128);
        pdo38 <= bchrnd(75 DOWNTO 44) XOR rep_res(191 DOWNTO 160);
        pdo39 <= bchrnd(107 DOWNTO 76) XOR rep_res(223 DOWNTO 192);
        pdo40 <= '0' & (bchrnd(138 DOWNTO 108) XOR rep_res(254 DOWNTO 224));
        bch_state <= idle;
      WHEN OTHERS =>
    END CASE;
  END IF;
END PROCESS;

ketje_mode_proc : PROCESS(ctrl_ketje_mode)
BEGIN
  IF (ctrl_ketje_mode = "01") THEN --keygen
    key_reg <= '0' & rep_res(254 DOWNTO 128);
    npub_reg <= '0' & nce2;
    ad_reg <= rep_res(127 DOWNTO 0);
  ELSIF (ctrl_ketje_mode = "10") THEN --enc
    key_reg <= dev_key;
```

```
    npub_reg <= pdi;
    ad_reg <= ’0’ & nce2;
  ELSIF (ctrl_ketje_mode = "11") THEN --authenticator
    key_reg <= dev_key;
    npub_reg <= pdi;
    ad_reg <= ’1’ & nce2;
  END IF;
END PROCESS;

ketje_proc : PROCESS(clk)
BEGIN
  IF (ctrl_rst_ketje = ’1’) THEN
    ketje_state <= ldkey;
    rst_ketje <= ’1’;
    ctrl_cnt <= x"00";
    ctrl_checkandupd <= ’0’;
    ctrl_comp_t2 <= ’0’;
  ELSIF RISING_EDGE(clk) THEN
    CASE ketje_state IS
      WHEN ldkey =>
        rst_ketje <= ’0’;
        key <= key_reg;
        key_needs_update <=’1’;
        key_ready <=’1’;
        ketje_state <= ldnce;
      WHEN ldnce =>
        key_needs_update <=’0’;
        key_ready <=’0’;
        npub <= npub_reg;
        npub_ready <=’1’;
        ketje_state <= initstate;
      WHEN initstate =>
        npub_ready <=’0’;
        IF (ctrl_cnt = x"0c") THEN --so total of 14 clk_periods
          ctrl_cnt <= x"00";
          ketje_state <= ldad;
        ELSE
          ctrl_cnt <= STD_LOGIC_VECTOR(UNSIGNED(ctrl_cnt) + 1);
        END IF;
      WHEN ldad =>
        bdi <= ad_reg(TO_INTEGER(UNSIGNED(ctrl_cnt))*32+31 DOWNTO TO_INTEGER(UNSIGNED(ctrl_cnt))*32);
        bdi_ready <= ’1’;
        bdi_ad <=’1’;
        IF (ctrl_cnt = x"03") AND (ctrl_ketje_mode = "10") THEN
          bdi_eot <=’1’;
        ELSIF(ctrl_cnt = x"03") THEN
          bdi_eoi <=’1’;
          bdi_eot <=’1’;
        END IF;
        ctrl_cnt <= STD_LOGIC_VECTOR(UNSIGNED(ctrl_cnt) + 1);
        ketje_state <= wtad;
      WHEN wtad =>
        bdi <= x"00000000";
        bdi_ready <= ’0’;
        bdi_ad <= ’0’;
        bdi_eot <=’0’;
        bdi_eoi <=’0’;
        IF (ctrl_cnt = x"04") AND (ctrl_ketje_mode = "10") THEN
          ketje_state <= ldmsg;
          ctrl_cnt <= x"00";
        ELSIF (ctrl_cnt = x"04") THEN
          ketje_state <= comptag;
          ctrl_cnt <= x"00";
        ELSE
          ketje_state <= ldad;
        END IF;
      WHEN ldmsg =>
        bdi <= pufresp(TO_INTEGER(UNSIGNED(ctrl_cnt))*32+31 DOWNTO TO_INTEGER(UNSIGNED(ctrl_cnt))*32);
        bdi_ready <= ’1’;
        IF (ctrl_cnt = x"27") THEN
          bdi_eot <=’1’;
        END IF;
        ctrl_cnt <= STD_LOGIC_VECTOR(UNSIGNED(ctrl_cnt) + 1);
        ketje_state <= wtmsg;
      WHEN wtmsg =>
        bdi <= x"00000000";
        bdi_ready <= ’0’;
        bdi_eot <=’0’;
        bdi_eoi <=’0’;
        IF (ctrl_cnt = x"28") then
          ketje_state <= lstmsg;
          c1(TO_INTEGER(UNSIGNED(ctrl_cnt))*32-33 DOWNTO TO_INTEGER(UNSIGNED(ctrl_cnt))*32-64) <= bdo;
          ctrl_cnt <= x"00";
        ELSIF (ctrl_cnt /= x"00") AND (ctrl_cnt /= x"01") THEN
          c1(TO_INTEGER(UNSIGNED(ctrl_cnt))*32-33 DOWNTO TO_INTEGER(UNSIGNED(ctrl_cnt))*32-64) <= bdo;
          ketje_state <= ldmsg;
        ELSE
          ketje_state <= ldmsg;
        END IF;
      WHEN lstmsg =>
        IF (ctrl_cnt = x"01") THEN
          c1(1279 DOWNTO 1248) <= bdo;
          ketje_state <= comptag;
          ctrl_cnt <= x"00";
          pdo41 <= c1(31 DOWNTO 0);
          pdo42 <= c1(63 DOWNTO 32);
          pdo43 <= c1(95 DOWNTO 64);
```

```
          -- pdo44 to pdo76 OMITTED FOR CLARITY
          pdo77 <= c1(1183 DOWNTO 1152);
          pdo78 <= c1(1215 DOWNTO 1184);
          pdo79 <= c1(1247 DOWNTO 1216);
          pdo80 <= bdo;
        ELSE
          ctrl_cnt <= STD_LOGIC_VECTOR(UNSIGNED(ctrl_cnt) + 1);
        END IF;
      WHEN comptag =>
        IF (tag_write = '1') THEN
          IF (ctrl_ketje_mode = "01") THEN --keygen
            dev_key <= tag;
            ketje_state <= idle;
          ELSIF (ctrl_ketje_mode = "10") THEN --enc
            t1 <= tag;
            ctrl_comp_t2 <= '1';
            ketje_state <= idle;
          ELSIF (ctrl_ketje_mode = "11") THEN --authenticator
            t2 <= tag;
            ketje_state <= idle;
            ctrl_checkandupd <= '1';
          END IF;
        END IF;
      WHEN OTHERS =>
    END CASE;
  END IF;
END PROCESS;

end Behavioral;
```

C.2.3 3-1 DAPUF

```
/*****************************************************************************************************/
// Author:  J.G. (Gerben) Geltink //  g.geltink@gmail.com
// Module:  3-1 DAPUF            //  dapuf.v
//    This module describes the device DAPUF in the Conceiling Ketje Protocol as presented by the
//    guiding MSc thesis: Geltink, J. G. (2016). "Conceiling Ketje: A Lightweight PUF-based
//      Authentication Protocol".
//
// Copyright (c) 2015 Sakiyama, Machida, Iwamoto All Rights Reserved.
// Adapted by J.G. (Gerben) Geltink, Copyright (c) 2016 under MIT license
/*****************************************************************************************************/


/************************************ 3-1 DAPUF Module begin ******************************************/
module DAPUF(clk, rst_n, idata, esig, Answer);

input               clk;   // Clock signal
input               rst_n; // Reset signal
input      [63:0]   idata; // 64-bit challenge
input               esig;  // Input signal
output              Answer; //1-bit response

/************************************ Wire/Register Definition begin **********************************/
//reg       [63:0]    idata;
reg                 Answer_reg;

wire                a0,a1,a2,//// a3 to a64 OMITTED FOR CLARITY
wire                b0,b1,b2,//// b3 to b64 OMITTED FOR CLARITY
wire                c0,c1,c2,//// c3 to c64 OMITTED FOR CLARITY
wire                d0,d1,d2,//// d3 to d64 OMITTED FOR CLARITY
wire                e0,e1,e2,//// e3 to e64 OMITTED FOR CLARITY
wire                f0,f1,f2,//// f3 to f64 OMITTED FOR CLARITY
wire                res_a,res_b,res_c,res_d,res_e,res_f;
/************************************ Wire/Register Definition begin **********************************/

/************************************ Generate Input Signal begin *************************************/
//assign idata = idata;

FDCE #(
  .INIT(1'b0)  // Initial value of register (1'b0 or 1'b1)
) sig_reg_a (
  .Q(a0),      // Data output
  .C(clk),     // Clock input
  .CE(1'b1),   // Clock enable input
  .CLR(1'b0),  // Asynchronous clear input
  .D(esig)     // Data input
);

FDCE #(
  .INIT(1'b0)  // Initial value of register (1'b0 or 1'b1)
) sig_reg_b (
  .Q(c0),      // Data output
  .C(clk),     // Clock input
  .CE(1'b1),   // Clock enable input
  .CLR(1'b0),  // Asynchronous clear input
  .D(esig)     // Data input
);
```

```
/**************************************** Generate Input Signal end ***********************************/

/**************************************** First Selector Chain begin **********************************/
MUXF7 MUX_000(.O(a1), .I0(a0), .I1(a0), .S(idata[0])); // synthesis attribute keep of a1 is true;
MUXF7 MUX_001(.O(b1), .I0(c0), .I1(c0), .S(idata[0])); // synthesis attribute keep of b1 is true;
MUXF7 MUX_002(.O(a2), .I0(a1), .I1(b1), .S(idata[1])); // synthesis attribute keep of a2 is true;
MUXF7 MUX_003(.O(b2), .I0(b1), .I1(a1), .S(idata[1])); // synthesis attribute keep of b2 is true;
//// MUX_005 to MUX_123 OMITTED FOR CLARITY
MUXF7 MUX_124(.O(a63), .I0(a62), .I1(b62), .S(idata[62])); // synthesis attribute keep of a63 is true;
MUXF7 MUX_125(.O(b63), .I0(b62), .I1(a62), .S(idata[62])); // synthesis attribute keep of b63 is true;
MUXF7 MUX_126(.O(a64), .I0(a63), .I1(b63), .S(idata[63])); // synthesis attribute keep of a64 is true;
MUXF7 MUX_127(.O(b64), .I0(b63), .I1(a63), .S(idata[63])); // synthesis attribute keep of b64 is true;
/**************************************** First Selector Chain end ************************************/

/**************************************** Second Selector Chain begin *********************************/
MUXF7 MUX_128(.O(c1), .I0(a0), .I1(a0), .S(idata[0])); // synthesis attribute keep of c1 is true;
MUXF7 MUX_129(.O(d1), .I0(c0), .I1(c0), .S(idata[0])); // synthesis attribute keep of d1 is true;
MUXF7 MUX_130(.O(c2), .I0(c1), .I1(d1), .S(idata[1])); // synthesis attribute keep of c2 is true;
MUXF7 MUX_131(.O(d2), .I0(d1), .I1(c1), .S(idata[1])); // synthesis attribute keep of d2 is true;
//// MUX_132 to MUX_251 OMITTED FOR CLARITY
MUXF7 MUX_252(.O(c63), .I0(c62), .I1(d62), .S(idata[62])); // synthesis attribute keep of c63 is true;
MUXF7 MUX_253(.O(d63), .I0(d62), .I1(c62), .S(idata[62])); // synthesis attribute keep of d63 is true;
MUXF7 MUX_254(.O(c64), .I0(c63), .I1(d63), .S(idata[63])); // synthesis attribute keep of c64 is true;
MUXF7 MUX_255(.O(d64), .I0(d63), .I1(c63), .S(idata[63])); // synthesis attribute keep of d64 is true;
/**************************************** Second Selector Chain end ***********************************/

/**************************************** Third Selector Chain begin **********************************/
MUXF7 MUX_256(.O(e1), .I0(a0), .I1(a0), .S(idata[0])); // synthesis attribute keep of e1 is true;
MUXF7 MUX_257(.O(f1), .I0(c0), .I1(c0), .S(idata[0])); // synthesis attribute keep of f1 is true;
MUXF7 MUX_258(.O(e2), .I0(e1), .I1(f1), .S(idata[1])); // synthesis attribute keep of e2 is true;
MUXF7 MUX_259(.O(f2), .I0(f1), .I1(e1), .S(idata[1])); // synthesis attribute keep of f2 is true;
//// MUX_260 to MUX_379 OMITTED FOR CLARITY
MUXF7 MUX_380(.O(e63), .I0(e62), .I1(f62), .S(idata[62])); // synthesis attribute keep of e63 is true;
MUXF7 MUX_381(.O(f63), .I0(f62), .I1(e62), .S(idata[62])); // synthesis attribute keep of f63 is true;
MUXF7 MUX_382(.O(e64), .I0(e63), .I1(f63), .S(idata[63])); // synthesis attribute keep of e64 is true;
MUXF7 MUX_383(.O(f64), .I0(f63), .I1(e63), .S(idata[63])); // synthesis attribute keep of f64 is true;
/**************************************** Third Selector Chain end ************************************/

/**************************************** Generate Response begin *************************************/
SRL SRL_0(.S(a64), .R(c64), .Q(res_a)); // synthesis attribute keep of res_a is true;
SRL SRL_1(.S(c64), .R(e64), .Q(res_b)); // synthesis attribute keep of res_b is true;
SRL SRL_2(.S(e64), .R(a64), .Q(res_c)); // synthesis attribute keep of res_c is true;
SRL SRL_3(.S(b64), .R(d64), .Q(res_d)); // synthesis attribute keep of res_d is true;
SRL SRL_4(.S(d64), .R(f64), .Q(res_e)); // synthesis attribute keep of res_e is true;
SRL SRL_5(.S(f64), .R(b64), .Q(res_f)); // synthesis attribute keep of res_f is true;

assign Answer = Answer_reg;

always @(posedge clk or negedge rst_n) begin
  if (~rst_n) begin
  //rst_n was unneccesary => changed by JGG
    Answer_reg <= res_a^res_b^res_c^res_d^res_e^res_f;//1'b0;                  // Reset
  end
  else begin
    Answer_reg <= res_a^res_b^res_c^res_d^res_e^res_f;
  end
end
/************************************** Response Generating Part end **********************************/
endmodule
/**************************************** 3-1 DAPUF Module end ****************************************/


/**************************************** SR Latch Module begin ***************************************/
module SRL(S, R, Q);
input S,R;
output Q;
wire QB;

NAND2 NAND_0(.I0(S), .I1(QB), .O(Q)); // synthesis attribute keep of Q is true;
NAND2 NAND_1(.I0(R), .I1(Q), .O(QB)); // synthesis attribute keep of QB is true;
endmodule
/**************************************** SR Latch module end *****************************************/
```

C.2.4 Constraints

```
#*****************************************************************************************************#
# Author:   J.G. (Gerben) Geltink # g.geltink@gmail.com
# Module:   Xilinx Design Constr  # constr.xdc
#   This module describes the design constraints of the device DAPUF in the Conceiling Ketje
#   Protocol as presented by the guiding MSc thesis: Geltink, J. G. (2016). "Conceiling Ketje: A
#   Lightweight PUF-based Authentication Protocol".
#
# Copyright (c) 2016 J.G. Geltink under MIT license
#*****************************************************************************************************#

#***************************************** Start registers begin *************************************#
set_property BEL BFF [get_cells CKP_DEV_CORE_0/DAPUF_0/sig_reg_a]
set_property LOC SLICE_X56Y149 [get_cells CKP_DEV_CORE_0/DAPUF_0/sig_reg_a]
set_property BEL AFF [get_cells CKP_DEV_CORE_0/DAPUF_0/sig_reg_b]
set_property LOC SLICE_X56Y149 [get_cells CKP_DEV_CORE_0/DAPUF_0/sig_reg_b]
#****************************************** Start registers end **************************************#

#************************************** First Selector Chain begin ***********************************#
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_000]
set_property LOC SLICE_X54Y148 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_000]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_001]
set_property LOC SLICE_X55Y148 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_001]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_002]
set_property LOC SLICE_X54Y147 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_002]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_003]
set_property LOC SLICE_X55Y147 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_003]
#### MUX_004 to MUX_123 OMITTED FOR CLARITY
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_124]
set_property LOC SLICE_X54Y86 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_124]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_125]
set_property LOC SLICE_X55Y86 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_125]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_126]
set_property LOC SLICE_X54Y85 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_126]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_127]
set_property LOC SLICE_X55Y85 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_127]
#***************************************** First Selector Chain end **********************************#

#************************************** Second Selector Chain begin **********************************#
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_128]
set_property LOC SLICE_X56Y148 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_128]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_129]
set_property LOC SLICE_X57Y148 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_129]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_130]
set_property LOC SLICE_X56Y147 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_130]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_131]
set_property LOC SLICE_X57Y147 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_131]
#### MUX_132 to MUX_251 OMITTED FOR CLARITY
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_252]
set_property LOC SLICE_X56Y86 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_252]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_253]
set_property LOC SLICE_X57Y86 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_253]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_254]
set_property LOC SLICE_X56Y85 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_254]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_255]
set_property LOC SLICE_X57Y85 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_255]
#**************************************** Second Selector Chain end **********************************#

#************************************** Third Selector Chain begin ***********************************#
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_256]
set_property LOC SLICE_X58Y148 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_256]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_257]
set_property LOC SLICE_X59Y148 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_257]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_258]
set_property LOC SLICE_X58Y147 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_258]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_259]
set_property LOC SLICE_X59Y147 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_259]
#### MUX_260 to MUX_379 OMITTED FOR CLARITY
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_380]
set_property LOC SLICE_X58Y86 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_380]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_381]
set_property LOC SLICE_X59Y86 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_381]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_382]
set_property LOC SLICE_X58Y85 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_382]
set_property BEL F7BMUX [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_383]
set_property LOC SLICE_X59Y85 [get_cells CKP_DEV_CORE_0/DAPUF_0/MUX_383]
#***************************************** Third Selector Chain end **********************************#


#***************************************** Generate Response begin ***********************************#
set_property BEL C6LUT [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_5/NAND_0]
set_property LOC SLICE_X59Y82 [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_5/NAND_0]
set_property BEL C6LUT [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_0/NAND_1]
set_property LOC SLICE_X56Y84 [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_0/NAND_1]
set_property BEL C6LUT [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_1/NAND_1]
set_property LOC SLICE_X58Y83 [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_1/NAND_1]
set_property BEL C6LUT [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_0/NAND_0]
set_property LOC SLICE_X54Y84 [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_0/NAND_0]
set_property BEL C6LUT [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_2/NAND_1]
set_property LOC SLICE_X54Y82 [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_2/NAND_1]
set_property BEL C6LUT [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_1/NAND_0]
set_property LOC SLICE_X56Y83 [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_1/NAND_0]
```

```
set_property BEL C6LUT [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_3/NAND_1]
set_property LOC SLICE_X57Y84 [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_3/NAND_1]
set_property BEL C6LUT [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_2/NAND_0]
set_property LOC SLICE_X58Y82 [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_2/NAND_0]
set_property BEL C6LUT [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_4/NAND_1]
set_property LOC SLICE_X59Y83 [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_4/NAND_1]
set_property BEL C6LUT [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_3/NAND_0]
set_property LOC SLICE_X55Y84 [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_3/NAND_0]
set_property BEL C6LUT [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_5/NAND_1]
set_property LOC SLICE_X55Y82 [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_5/NAND_1]
set_property BEL C6LUT [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_4/NAND_0]
set_property LOC SLICE_X57Y83 [get_cells CKP_DEV_CORE_0/DAPUF_0/SRL_4/NAND_0]
#**************************************** Generate Response end ****************************************#
```

C.2.5 BCH Encoder

```
-----------------------------------------------------------------------------------------------------------
-- Author:  J.G. (Gerben) Geltink --  g.geltink@gmail.com
-- Module:  BCH encodeer          --  bch.vhd
--     This module describes the device BCH encoder of the Conceiling Ketje Protocol as presented by
--     the guiding MSc thesis: Geltink, J. G. (2016). "Conceiling Ketje: A Lightweight PUF-based
--     Authentication Protocol".
--
-- Copyright (c) 2016 J.G. Geltink under MIT license
-----------------------------------------------------------------------------------------------------------

-- ring for encoder
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
  USE WORK.const.ALL;

ENTITY ering IS
PORT (clk, rll, din: IN  std_logic;
  dout : OUT  std_logic); --output serial data
END ering;

ARCHITECTURE eringa OF ering IS
  SIGNAL rin, rout: std_logic_vector(0 TO nk-1) := (others => '0'); -- ring register
  SIGNAL rin0: std_logic;
  BEGIN
  dout<= rout(nk-1);
  rin0 <= (din XOR rout(nk-1)) AND rll;

  rin(0)<= rin0;
  rin(1) <= rout(0);
  rin(2) <= rout(1);
  rin(3) <= rout(2) XOR rin0;
  rin(4) <= rout(3) XOR rin0;
  rin(5) <= rout(4);
  rin(6) <= rout(5);
  rin(7) <= rout(6);
  rin(8) <= rout(7) XOR rin0;
  rin(9) <= rout(8) XOR rin0;
  rin(10) <= rout(9);
  rin(11) <= rout(10);
  rin(12) <= rout(11);
  rin(13) <= rout(12);
  rin(14) <= rout(13);
  rin(15) <= rout(14);
  rin(16) <= rout(15);
  rin(17) <= rout(16) XOR rin0;
  rin(18) <= rout(17) XOR rin0;
  rin(19) <= rout(18) XOR rin0;
  rin(20) <= rout(19) XOR rin0;
  -- rin(21) to rin(99) OMITTED FOR CLARITY
  rin(100) <= rout(99);
  rin(101) <= rout(100) XOR rin0;
  rin(102) <= rout(101) XOR rin0;
  rin(103) <= rout(102) XOR rin0;
  rin(104) <= rout(103);
  rin(105) <= rout(104) XOR rin0;
  rin(106) <= rout(105) XOR rin0;
  rin(107) <= rout(106) XOR rin0;
  rin(108) <= rout(107) XOR rin0;
  rin(109) <= rout(108);
  rin(110) <= rout(109);
  rin(111) <= rout(110) XOR rin0;
  rin(112) <= rout(111);
  rin(113) <= rout(112);
  rin(114) <= rout(113) XOR rin0;
  rin(115) <= rout(114);
  -- Generator polynomial: 100110001100000001111011010000110000001111101101110001101111000111010010111
  --                       0110000101001111001010111011101111100100101
  -- Number of XOR gates= 59

  PROCESS BEGIN
  WAIT UNTIL clk'EVENT AND clk='1';
  rout<= rin;
```

```
  END PROCESS;
END eringa;

------------------------------------------------------------------------
-- COUNTER MODULO n FOR ENCODER BCH CODE (n,k)
--  pe- parallel data in;  rll-ring loop lock
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
  USE WORK.const.ALL;

ENTITY ecount IS
PORT (clk, reset: IN std_logic;
  vdin: OUT std_logic);
END ecount;

ARCHITECTURE ecounta OF ecount IS
  SIGNAL cout: std_logic_vector(0 TO m-1); -- cout in GF(2^m); cout= L^count
  SIGNAL vdinR, vdinS, vdin1: std_logic;
  BEGIN
  vdinR<= cout(0) AND NOT cout(1) AND NOT cout(2) AND NOT cout(3) AND NOT cout(4) AND cout(5) AND NOT cout(6) AND NOT cout(7);
    -- reset vdin if cout==k-1
  vdinS<= ( NOT cout(0) AND cout(1) AND cout(2) AND cout(3) AND NOT cout(4) AND NOT cout(5) AND NOT cout(6) AND cout(7)) OR reset;
    -- vdinS=1 if cout==n-1
  vdin<= vdin1 AND NOT reset;

  PROCESS BEGIN
  WAIT UNTIL clk'EVENT AND clk='1';
  IF vdinR='1' THEN
    vdin1<= '0';
  ELSIF vdinS='1' THEN
    vdin1<= '1';
  END IF;
  END PROCESS;

  PROCESS BEGIN -- increment or reset cout in ring, cout=L^count
  WAIT UNTIL clk'EVENT AND clk='1';
  cout(0)<= cout(m-1) OR reset;
  cout(1)<= cout(0) AND NOT reset;
  cout(2)<= (cout(1) XOR cout(m-1)) AND NOT reset;
  cout(3)<= (cout(2) XOR cout(m-1)) AND NOT reset;
  cout(4)<= (cout(3) XOR cout(m-1)) AND NOT reset;
  cout(5)<= cout(4) AND NOT reset;
  cout(6)<= cout(5) AND NOT reset;
  cout(7)<= cout(6) AND NOT reset;
  END PROCESS; --p(x) = 100011101
END ecounta;

-----------------------------------------------------------------
--    ENCODER
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY bch255_139_31enc IS
PORT (clk, reset, din: IN std_logic;
  vdin, dout: OUT std_logic); --output serial data
END bch255_139_31enc;  -- vdin - valid data in - to enable external data shifting

ARCHITECTURE enca OF bch255_139_31enc IS
  SIGNAL vdin1, rin, rout, rll: std_logic;
    -- rll-ring loop lock, pe-parallel enable din

  COMPONENT ecount --counter encoder
    PORT(clk, reset: IN std_logic; vdin: OUT std_logic);
    END COMPONENT;
    FOR ALL: ecount USE ENTITY WORK.ecount (ecounta);
  COMPONENT ering --ring for encoder
    PORT(clk, rll,  din: IN std_logic; dout: OUT std_logic);
    END COMPONENT;
    FOR ALL: ering USE ENTITY WORK.ering (eringa);
  BEGIN
  c1: ecount
    PORT MAP (clk, reset, vdin1);
  r1: ering
    PORT MAP (clk, rll, rin, rout);
  rin<= din AND NOT reset;
  rll<= vdin1 AND NOT reset;
  vdin<= vdin1;

  PROCESS BEGIN
  WAIT UNTIL clk'EVENT AND clk='1';
  dout<= (NOT vdin1 AND rout) OR (vdin1 AND rin);
  END PROCESS;
END enca;
```