FOM University of Applied Sciences for Economics & Management

University Center Düsseldorf

Bachelor's Thesis

in the degree programme

Wirtschaftsinformatik - Business Information Systems

submitted in partial fulfilment of the requirements for the degree

Bachelor of Science (B.Sc.)

on the subject

An Infrastructure for a Challenge-Response based Authentication System using Physical Unclonable Functions

by

Luis Pflamminger

First Examiner:	Prof. Dr. Bernd Ulmann
Matriculation Number:	538276
Date of Submission:	2022-11-11

Contents

List of Figures

List of Tables

List of Abbreviations

2FA	Two-Factor Authentication
CRP	Challenge-Response Pair
DoS	Denial of Service
DRAM	Dynamic Random-Access Memory
FAR	False Acceptance Rate
FRR	False Rejection Rate
IoT	Internet of Things
IoV	Internet of Vehicles
IBE	Identity Based Encryption
IoMT	Internet of Medical Things
MITM	Man-in-the-Middle
PKI	Public Key Infrastructure
OPUF	One-Time PUF
PUF	Physical Unclonable Function
SRAM	Static Random-Access Memory
V2G	Vehicle to Grid

1 Introduction

1.1 Motivation

This thesis is part of a research project concerned with developing an authentication system that makes use of Physical Unclonable Functions (PUFs) for asserting identities. Authentication is the act of verifying a person's or entity's identity through some identifying feature, which proves, that they are, who they say they are. [1, p. 398]

In today's electronic access control systems, digital keys are used for authentication. They are stored in databases and assigned to a physical entity like a person's fingerprint or a smart card. This approach can be problematic, as the digital key is not directly bound to the physical object. If an attacker were to gain access to the key, they could use it to create a copy of the physical entity and gain access. [2, p. 81]

A solution to this might be PUFs, which are physical devices, instead of digitally stored keys. During manufacturing, certain uncontrollable production tolerances lead to each device differing slightly from every other. This introduces unique characteristics and a certain variability and randomness into every PUF, which can be used to create a unique fingerprint of every device without the need for a digital key. While the variation is measurable, it is considered impractical to create an identical physical copy of a PUF, because it is deemed impossible to gain full control of the manufacturing tolerances to imitate the behavior of another PUF device. [2, p. 81]

The physical characteristics of PUFs have to be measured to be able to infer an identity from them. This is done using a Challenge-Response Pair (CRP). The challenge is some input given to the PUF, which uses it to generate an output (the response), leveraging its unique characteristics. For the same challenge, each individual PUF will therefore generate a different response. By previously generating and recording specific CRPs or using neural networks to evaluate a response to a given challenge, the authenticity of a given PUF can be established. [2, p. 81]

The goal of the research project, that this thesis is part of, is to create an authentication system that can be used as an electronic locking system for physical access protection of office buildings or other secure facilities. The thesis focuses on the infrastructure behind the challenge-response based authentication system. This includes specifying a protocol for communication between the components and developing a backend for generating and sending challenges to the PUF and verifying its responses. The thesis is not concerned with the design of the PUF modules themselves.

1.2 Research Questions

It is not possible to formulte specific research questions about the different components of the infrastructure from the beginning, as it is not yet known at this point, what the infrastructure will look like. However, looking at the overall project, this thesis aims to answer the following question, which is closely related to the problem statement and title:

> *RQ1: What does the infrastructure behind an authentication system using PUFs for physical access protection look like?*

This question is answered by using literature research to understand the fundamentals behind these systems and gaining knowledge about the current state of scientific research as a basis for the design.

As the implementation of a prototype is also part of the thesis, a second overarching question can be asked:

> *RQ2: How can a functional prototype for the proposed infrastructure concept be implemented?*

There is another aspect to this thesis, which is the evaluation of the proposed solution in the context of existing concepts and research on the topic. After answering these questions, the design and implementation should be evaluated to understand possible shortcomings of the proposed solution and solidify the results.

1.3 Methodology

To achieve the research goal of designing and implementing an infrastructure, each of the sections are approached differently.

In the fundamentals section, qualitative literature research is used to gain insight into the current state of scientific research and build a wide basis of knowledge that can be used for the conceptualization and implementation of the infrastructure. As there are is a broad range of topics to cover, no full systematic literature review is conducted in the fundamentals section.

The main sources for literature are be the EBSCO Discovery Service, IEEE Xplore, the ACM Digital Library, Springer Link and Google Scholar. These search engines and databases cover the most important sources for scientific literature in the field. The first step will be to gain a good understanding of PUF related concepts. From there, search

terms are developed dynamically depending on the required knowledge. Titles and abstracts of the most cited and relevant papers are scanned to find the papers and books providing the most relevant information. Additionally, if current review articles are found for a given topic, snowballing might be used to find additional relevant information.

After building the fundamental knowledge basis, a systematic literature review of the most relevant scientific literature about PUF-based authentication protocols is conducted. The methodology for this review is based on the guidelines for conducting literature reviews in the information systems field proposed by Brocke et al. in [3]. The exact approach is outlined in section 4.1.

In the next step, the insights gained from the literature review are used to design and implement the protocol and supporting infrastructure. Next, a concept is created based on the knowledge gained from literature research. Existing solutions for each part of the infrastructure are evaluated based on the requirements of this thesis.

The goal for the implementation section of the thesis is to create a prototype. Therefore, a language capable of rapid prototyping like Python is most likely selected. Some components of the finished system, like the neural networks and PUFs, are not part of this thesis. Because of this, they are not going to be part of the prototype implementation and will be simulated in some way. Additionally, tests are implemented to evaluate the finished prototype and to better understand possible edge cases.

1.4 Structure

In section 2, fundamental concepts of PUF are explained. First, this includes the definition of basic terms and description of basic concepts (2.1). Statistical considerations of PUFs are examined (2.2) and the different types of PUF are classified (2.3). Next some examples of popular PUF implementations are presented (2.4). Concluding the fundamentals section, central concepts of PUF-based authentication are examined (2.5) and additional applications for PUFs are presented (2.6).

In section 4, a literature review of existing PUF-based authentication protocols is conducted. It consists of specifying the methodology of the review process (4.1), presenting, which articles were chosen for review (4.2), analyzing the collected literature (4.3) and a discussion of the results (4.4).

In section 5, a prototype for an authentication protocol is implemented. This includes the selection of a protocol (5.1), the design of the protocol (5.2) and a presentation of the proposed solution (5.3).

Finally, the thesis is concluded (6) with an evaluation and results and an outlook on future research topics.

2 Fundamentals

A basic description of what a PUF is and how it works has been given in section 1.1. In order to conceptualize an authentication infrastructure around PUFs, a deeper understanding of how PUFs work, what types or classes of them exist and how they relate to their environment in terms of inputs and outputs, is needed.

2.1 Definitions and Basic Concepts

In this section, important terms needed for describing PUFs are explained. Then, the term Physical Unclonable Function (PUF) itself and some additional characteristics are defined.

PUFs can be physically constructed in different ways. All PUFs constructed using the same structural design, belong to the same PUF *class* P. [4, p. 14]

One specific device manufactured according to the design of a PUF class is called an *instance* of the class. Each time a new device is manufactured, a new PUF is *instantiated*. The process of creating a new instance of a class is also called the creation procedure $P.Create$, which returns a new instance puf. Each instance has a certain state. This state is usually determined during construction of the PUF, but certain classes of PUFs can also have configurable state. This type of state can be modified even after instantiation using some form of external input x. This input is called *challenge* and is applied to the instance, altering its state. A PUF with state that is configured using challenge x is called $puf(x)$. [4, p. 14]

Each PUF instance has an evaluation procedure called $puf.Eval$ or, if the particular instance has configurable state, $puf(x).Eval$. As PUFs are physical objects, the evaluation procedure is some kind of physical experiment resulting in a measurement, which depends on the internal state of the instance. This measurement is the result of $puf.Eval$ and also called the PUF's *response* [4, p. 14f]. The set consisting of a challenge and its corresponding response is called a CRP.

Many different definitions of the term *PUF* exist in the literature [5] [6] [4]. Pappu et al. first defined the concept now known as PUF as a "Physical One-Way Function", making them the physical equivalent of the mathematical concept of a one-way function. [7]

To summarize the concept of a PUF concisely, Maes defines PUFs in the following way:

"A PUF is an expression of an inherent and unclonable instance-specific feature of a physical object" [4, p. 12]

To explain this definition further, he compares it to a human fingerprint, as it is similar in many regards. Fingerprints can not just be used to confirm that someone is actually a human, but are a feature which can be used to identify a specific individual, just like a PUF can be used to identify a specific instance of a class of PUFs. Therefore PUFs are instance-specific. [4, p. 11f]

Fingerprints are inherent, in that humans are born with them. PUFs are also inherent, because they are a direct result of an object's creation process. Theoretically, every physical object has some kind of property which fits the definition of a PUF, as no two objects are exactly identical. Whether or not these properties are useful or not, depends on their intra- and inter-distance and how easily they can be measured. [4, p. 12]

Lastly, fingerprints are unclonable, as there is no way to create a human clone with exactly the same fingerprint. This is also a central requirement for an object's property or feature to be considered a PUF .[4, p. 12]

In addition to Maes, the definition given by Guajardo et al. in [6] should be mentioned, as it was one of the first attempts at a formalized definition of the term PUF [5, p. 84f]. This definition matches Maes' definition very well, in that it also focuses on the inherency of PUFs stemming from uncontrollable random variations in the manufacturing process. Furthermore, he defines a PUF as a function, which maps a challenge to a response .[6, p. 65f]

Additionally, multiple assumptions about PUFs are made:

- A response R_i received from a PUF instance based on challenge C_i, should not give significant information about another response R_j resulting from challenge C_j. This means, that by knowing one CRP of a PUF instance, no conclusions about other CRPs should be possible.
- Without access to the PUF, one should not be able to work out the correct response R_i to a challenge C_i, except for randomly guessing it.
- He also assumes, that PUFs are "tamper evident". This means, that the process of collecting detailed information about a specific PUF instance to the point of being able to predict CRPs should alter the instances behavior, essentially destroying it.

In a recent review article [8], McGrath et al. extend the definitions mentioned previously with some additional properties:

- *Robustness*: PUFs need to be stable over time. This means, that their behavior doesn't change with increasing age and CRPs stay consistent

- *Hard to replicate*: Creating a physical copy of a PUF instance needs to be difficult in order to preserve the uniqueness and instance-specificity of PUFs. Replicating a PUF would allow an attacker to gain access to a system which employs PUF-based security.
- *Unpredictable*: Predicting how an instance would respond to a given challenge needs to be hard or impossible. This ensures that an attacker would not be able to create a model of an instance which would again allow them to gain access.

For many PUF classes, external physical conditions influence the result of the evaluation procedure. Such conditions could be temperature or voltage level. Using the example of a PUF which is influenced by the environment temperature, an evaluation with $T_{env} = 80°C$ would be written as $puf(x).Eval^{\alpha}$ with $\alpha = T_{env}$. [4, p. 15]

The usability of a PUF class greatly depends on the behavior of the responses given the same or different instances, challenges and conditions.

2.2 Statistical Consideration of PUFs

As PUFs use internal random variations for generating responses, the challenge-response behavior is never completely consistent and always contains variability. For PUFs to be useful for authentication or other use-cases, these variations need to be considered. The statistical knowledge required is presented in this section.

2.2.1 Hamming Distance

After post processing, PUF responses typically consist of bit vectors. A bit vector Y has length $|Y|$. The *Hamming Distance* is a measure which quantifies the distance between two bit vectors of the same length. It is typically used in error correction for measuring the amount of flipped bits in a transmission. Specifically, it is the number of bits with differing values in both vectors. It is denoted as $HD(Y;Y')$, where Y and Y' are two bit vectors of the same length. [4, p. 173f]

The *Fractional Hamming Distance* is the number of bits with differing values divided by $|Y|$. [4, p. 173f]

As an example we define two bit vectors:

$$Y = 01110111$$

$$Y' = 01101011$$

In this example the length of the bit vector is 8 and 3 bits are different between the two. The Hamming Distance and Fractional Hamming Distance are calculated like this:

$$HD(Y;Y') = 3$$

$$FHD(Y;Y') = \frac{HD(Y;Y')}{|Y|} = \frac{3}{8} = 0.375$$

2.2.2 Intra- and Inter-Distance

The *intra-distance* of two PUF responses describes the distance between two responses from the same PUF instance given the same challenge. It is important to understand how the intra-distance behaves for a given class, in order to make assumptions about how reproducible the results for a given instance are [4, p. 16f]. Although any distance metric could be used for quantifying this distance, almost all of the literature considered in this thesis uses the Hamming distance.

External conditions can have a big impact on the typical intra-distance of a class. Typically, two responses generated at different external conditions have a larger intra-distance, than those generated under the same conditions. One has to understand which external conditions influence responses in which way. Often, a reference value α_{ref} for the condition is set before evaluating the responses for varying values experimentally. It is important to understand the worst case intra-distance of a given instance. This is done by defining a range of conditions and measuring responses at both ends of the spectrum. [4, p. 16f]

The *inter-distance* describes the distance between two responses from different PUF instances using the same challenge. A high inter-distance means that it is easier to distinguish two unique instances of a given class and correctly identify them. [4, p. 18ff]

2.2.3 Central Statistical Properties

For a class of PUF to be statistically relevant for security related purposes, it has to have two central statistical properties.

Uniqueness

A PUF class has the property of uniqueness, if there is a high probability, that the inter-distance between two given instances is large. If this probability was relatively small, there

would be a high chance, that two instances have a similar challenge-response behavior and would thus not be considered unique. [4, p. 53]

Identifiability

For instances of a class to be generally identifiable, responses need to be reproducible and instances have to be unique. This ensures that it is more likely for responses from the same instance to be similar, than for responses from two different instances. This means that the probability of the inter-distance being larger than the intra-distance of the PUF needs to be high. [4, p. 53f]

More information about inter- and intra-distance, and the statistical properties of PUF is given in section 2.5.

2.3 Classification of PUFs

PUF constructions can be classified in different ways [8, p. 4ff]. In this section PUFs are classified by their specific properties, as opposed to using e.g. chronological classification.

2.3.1 Electronic vs. Non-Electronic PUFs

Firstly, PUFs can be classified by their electronic characteristics. [4, p. 22]

Non-electronic PUFs are based on technologies or materials which are not based on electronics. This could be the random molecular structure of an object causing light to be scattered in different ways or surfaces to have different characteristics. The non-electronic nature of this class is limited to the random variation itself. Evaluation, including the measurement, post-processing and digitization of a response, can be done electronically [4, p. 22]. PUFs using this form of electronic evaluation of a non-electronic property are also called *Hybrid PUFs* [8, p.]. This is the most common form of non-electric PUFs, as responses typically need to be compared to database entries, which is not practical without electronic data processing.

Electronic PUFs are the counterpart to non-electronic PUFs. They rely on constructions which have electronic characteristics that can be used as PUFs. This could be resistance, capacitance or others. Responses are commonly evaluated using electronic means. Usually electronic variations in these PUFs occur due to non-electronic physical characteristics, such as the length of a wire. There are PUF constructions which can't be strictly

classified as either electronic or non-electronic, as they combine electrical elements with properties of non-electronic materials to introduce randomness. [4, p.22f]

Silicon PUFs are a subclass of electronic PUFs consisting of integrated electronic circuits embedded on a silicon chip [4, p. 23]. Silicon PUFs have the advantage of being contained within the circuit of a chip, which can be directly embedded into an electronic circuit board and interact as part of an embedded system or computer. They can thus be used as cryptographic elements in security systems. [4, p. 23]

2.3.2 Implicit vs. Explicit Random Variations

A classification by implicitness is also sensible. PUFs need some kind of physical property which introduces randomness that makes an instance unique and ensures, that a response can not easily be inferred from a given challenge. Implicitness depends on wether the manufacturing process needs to be specifically adapted by the manufacturer in order to introduce randomness into the object, or if the randomness is inherently present in all objects, without changing the process. The latter would be called an *implicit random variation*, while randomness introduced by adjusting the process (e.g. adding additional components) is called an *explicit random variation*. [4, p. 24] [8, p. 3]

A meaningful benefit to implicit random features exists from a security standpoint. As even manufacturers are not able to control these variations, there is no entity that has to be trusted, as nobody has the ability to remove or control the randomness of the given feature used for PUF evaluation. In contrast, when using a PUF with explicitly introduced random variations, a level of trust towards the manufacturer is required. If the manufacturer were to decide to intentionally manipulate PUF instances and introduce equivalent behavior into several instances, the instance-specificity and unclonability principles of the PUF construction would be violated, which has severe security implications [4, p. 24] [8, p. 3]. Additionally, implicit random variations reduce cost, as no extra steps are needed in manufacturing. [8, p. 3]

2.3.3 Intrinsic vs. Extrinsic PUFs

A PUF has an internal evaluation, if the evaluation function, which is the measurement of the random feature and creation of a response, happens entirely within the PUF device itself, without requiring external measurement equipment or analysis [4, p. 23f]. For a PUF to be considered an *intrinsic* PUF, the random variation has to be implicit and the

evaluation has to be internal. If one of these criteria is not met, the class is classified as a non-intrinsic or *extrinsic* PUF. [8, p. 3]

There are multiple advantages to internal evaluation. If the measurement is taken as part of the PUF construction, there is no variation that can be introduced by varying measurement equipment or execution, leading to more accurate results and less errors. It is also more practical, as no specific equipment is needed to read responses. Lastly there is a security advantage. While a response is kept within an instance and not exposed to the outside, it can be considered secret and is not subject to attacks. This would also be a benefit for an authentication system, as this secret could then be used as part of cryptographic evaluations .[4, p. 23f] [8, p. 3]

Maes describes an advantage of external evaluation procedures to be, that they allow better verification of a successful measurement. As there is no way to inspect an internal evaluation, one has to trust that the PUF evaluation function is working as intended [4, p. 24]. On the other hand, according to McGrath et al. in [8], internal evaluation is more accurate.

2.3.4 Strong vs. Weak PUFs

This classification is based on the number of CRPs that a single PUF instance has [8, p. 2], which correlates with how secure the challenge-response behavior of that PUF class is [4, p.25]. Imagine an attacker having an extended period of time with access to a specific PUF instance. They would certainly use that time to try and find out all possible CRPs of the instance, as this would give them unlimited access to whatever system the PUF is used to protect. [4, p.25]

Strong PUFs have the property, that after this time, there would still be a high probability of finding a challenge, to which the attacker does not know the response. This implies, that the number of CRPs must be large enough, to make brute forcing all possible solutions impossible within the given time. It also implies, that the instance must be entirely unpredictable. Otherwise it would be possible to create a model or simulation of the PUF instance, which would behave exactly like the real instance and be able to deliver the correct response to every given challenge [4, p.25] [6, p. 66]. An additional characteristic of a strong PUF is, that the number of CRPs scales exponentially with the physical devices size. This means, that by increasing the size, of a PUF can be made much more secure [8, p. 2].

The properties of strong PUFs have interesting implications for authentication systems. In a system based on symmetric cryptographic keys or Public Key Infrastructure (PKI), the

secret used for authentication is useless if a third-party had access to the (private) key at some point in the past. In contrast, if strong PUFs are used for encryption, there is no way for an attacker to falsely authenticate, as long as the attacker doesn't have physical access to the device at the time of authentication [8, p. 2]. The large set of CRPs also makes authentication using strong PUFs resistant to eavesdropping, because no challenge ever has to be used twice. This means, that even if an attacker gets hold of a CRP, they are not able to compromise future authentications, as a different challenge would be given every time.

Weak PUFs have a relatively small set of CRPs, which means that an attacker could brute-force or record all possible CRPs in a reasonable amount of time [8, p. 2]. Weak PUFs could also be predictable enough to be modelled [4, p.25] [6, p. 66] and the number of CRPs doesn't typically scale with device size [8, p. 2]. An additional characteristic is, that the size of the set of CRPs does generally not scale exponentially, but linearly or in a polynomial order, when increasing device size [8, p. 2]. This limits the practical applications of weak PUFs. If they were used in an authentication system, an attacker could easily breach security because of the downsides mentioned. A benefit of weak PUFs and the reason for them being preferred for some applications is, that they are typically much easier and cheaper to manufacture. [4, p.25]

2.4 Popular PUF Implementations

Many different concepts for PUF constructions have been explored in the literature. While some, like the Arbiter PUF, are well established and have been widely researched, others, like the MEMS PUF [9], provide a more niche approach [8, p. 2]. In this section, some of the most popular PUF constructions according to [8] from different families are examined to get a sense of how they work internally, what kinds of challenges they take, and what kinds of responses they emit.

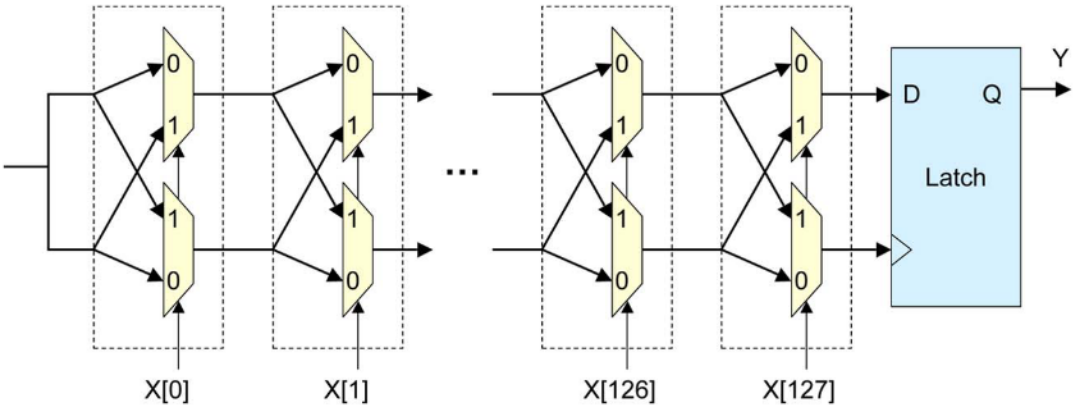
2.4.1 Arbiter PUF

According to [8, p. 6], this PUF concept was first proposed by Lee et al. in 2004 [10]. Arbiter PUFs use statistical variations in the time delay characteristics of electrical components like wires or transistors in integrated circuits as a unique characteristic. [10, p. 176]

A delay line makes it possible to introduce and control a time delay into an electrical signal. These delays are usually in the range of nanoseconds to microseconds [11, p. 1]. In

an arbiter PUF, two electrical signals are simultaneously emitted and sent through two separate delay lines, which consist of a number of switch components and terminate in a latch component. Figure 1 shows the structure of an Arbiter PUF circuit.

Figure 1: Structure of an Arbiter PUF



Source: [12, p. 1130]

The challenge consists of n bits, where n is the number of switch components in the delay line. In figure 1, n equals 128. Each bit configures the corresponding switch to adjust the delay path accordingly. If a switch is set to 0, the signals stay on their current paths, if it is 1, the signals cross. At the end of the line, the latch component measures, which of the two signals arrives first. If the paths are always symmetric regardless of switch position, a response of 0 or 1 is equally likely. This would be ideal, as the variation in speed would only occur based on variations in the manufacturing of the conductor and switches. In practice, constructions can't always achieve exact symmetry because of space constraints. [10, p. 177]

The delay of electrical components is dependant on external factors, namely temperature, ambient noise levels and voltage variations. The arbiter PUF solves this by not using absolute delay values, but relying on relative comparisons between different delay characteristics. [10, p. 176]

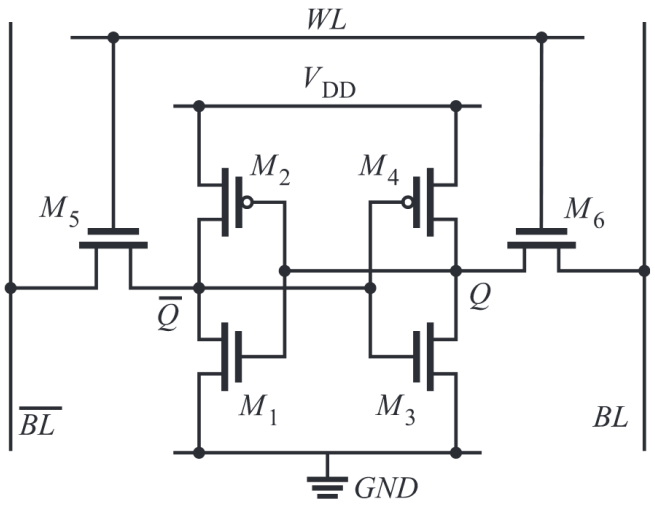
Arbiter PUFs belong to the group of fully electronic PUFs. The source of randomness is implicit, because it relies on variations within the electrical components of the circuit, which can't be controlled in any way. They also have internal response evaluation, which makes them intrinsic, and have a large set of CRPs that increases exponentially while adding more switch components, making them strong PUFs. [10, p. 176]

2.4.2 SRAM PUF

According to [8, p. 6], this PUF concept was first proposed by Guajardo et al. in 2007 [6]. Compared to Dynamic Random-Access Memory (DRAM), which is typically used as the main memory in modern computers, Static Random-Access Memory (SRAM) has higher power consumption and very complex internal circuitry, which makes it larger and more expensive. However, it is still an important part in modern computers, as it allows for much lower access times than DRAM, which makes it viable as a fast caching layer in the memory hierarchy [13, p. 2f]. An SRAM cache consists of peripheral circuitry which handles data input and output and address decoding. They also consist of many SRAM cells, typically consisting of six transistors. Each cell can hold one bit of information [13, p. 5]. The architecture of a typical bit cell is shown in figure 2. It consists of four load transistors $M_1 - M_4$, which store the information, two access transistors M_5 and M_6, a word line WL and the data bit-lines BL and $\overline{BL}$, which carry data to and from the cell [6, p. 73].

The load transistors are connected in such a way, that they create a positive feedback loop. If the voltage at node Q tends towards being high, the feedback loop will amplify this tendency and force the voltage to high. Q and $\overline{Q}$ always have opposite states. The voltage at node Q dictates the bit of information stored within the cell. If Q is high, the cell is 1, if it's low, the cell is 0. [13, p. 31]

Figure 2: Circuit Diagram of SRAM PUF



Source: [8, p. 12]

An SRAM cell can have three possible states:

- *Standby*: Here the word-line WL is low, which disconnects the load transistors from the bit line. This means, that their state will not change as long as the load transistors are powered. [13, p. 31]

- *Reading*: Here, both bit-lines are set high. Then the access transistors are engaged, which connects the bit-lines to the load transistors. If the cell's state is 1, $\overline{BL}$ will drop in voltage. If the state is 0, BL will drop in voltage. A sense amplifier measures, which of the bit-lines had a voltage drop and has thus read the cell. In the end, the word-line is set high again, to disconnect the cell [13, p. 32].

- *Writing*: If a 1 should be written, first BL is set to 1 and $\overline{BL}$ is set to 0. WL is set high to connect the cell to the bit-lines. Because the load transistors are much weaker than the bit-line drivers, they take the value of BL. The word-line is then disengaged[13, p. 36].

As SRAM cells have been scaled down to increase speed and size, electrical deviations on an atomic-level become a challenge [14, p. 658]. The write process requires load-transistors in the cell to be relatively weak, so they can be overwritten by the voltage one the bit-lines. This is why they are particularly vulnerable to fluctuations on an atomic level. These variations can not be controlled during the manufacturing process, which makes them implicit [6, p. 73]. Under normal operation, the cells are manufactured in a way, that these fluctuations don't decrease the stability of the cell during reads, writes and standby, in a meaningful way. However, when powering up the SRAM, for a short period of time, the cell is not exposed to any external electrical signal. During this time, the intrinsic electrical variations tend to a 0 or a 1. This tendency is amplified due to the positive feedback loop of the load-transistor circuit. Although the variations across multiple cells are random, they are consistent over time. This means that a single cell will assume the same state after each power-up with a high probability, independent from neighboring cells. [6, p. 73]

These intrinsic variations across cells, which are stable over time, are used as the PUF. The challenge dictates a range of cells within the SRAM cache to be read. Because of error correction, 4600 SRAM cells are required to extract a stable 128-bit response from the PUF. The response does not need to be converted or quantized further, as it is already in the form of binary data. Using a 512 kbit SRAM module, a total of $512.000/4600 = 111$ CRPs is available. This increases with the number of cells or by decreasing the secret size. [6, p. 73]

2.5 PUF-based Authentication

This section describes the basic principles behind authentication and gives an overview of how PUFs are used for entity authentication.

2.5.1 Fundamentals of Authentication

Authentication is used to establish the identity of an entity, such as a person or object. When a person is trying to authenticate themselves, they have to prove in some way that they are who they say they are. This can be done in three main ways [1, p.398]:

1. By providing a secret that only they know, such as a password.
2. By providing something only they have, such as a physical key.
3. By using some unique physical characteristic, such as a fingerprint.

In addition to proving the entity's identity, the authentication process also needs to reveal that the entity was actively present and involved in the process [4, p. 117f]. As discussed in section 2.1, PUFs can be closely compared to human fingerprints, as both are unclonable, inherent and instance-specific. They therefore provide an entity-specific feature, which can be used for entity authentication.

To put it concisely, Maes presents the following definition for entity authentication:

> *"An entity authentication technique assures one party, through acquisition of corroborative evidence, of both: (i) the identity of a second party involved, and (ii) that the second party was active at the time the evidence was created or acquired."* [4, p. 117]

The two parties involved in the authentication process have specific names. The party, that wants to authenticate itself is called the *prover*, because it is proving its identity. The party it is authenticating to is called the *verifier*, because it verifies the proof of identity provided by the authenticating party. [4, p. 129]

Contrary to authentication, authorization determines whether someone is allowed to access a specific resource. It happens once the identity has been established and is not further considered in this thesis.

2.5.2 Fuzzy Identification

To authenticate an entity, it first needs to be identified. Identification can be seen as a precondition - or a weaker form of - authentication, as the entity merely has to provide their identity, but not prove it in a significant way or provide evidence of its active involvement at that point in time. It is therefore sufficient in cases where security is not important, like tracking a product in shipping. Without clear identification, authentication does not work. [4, p. 118]

Identification using inherent features, like the ones PUFs provide, consists of two phases [4, p. 119f]:

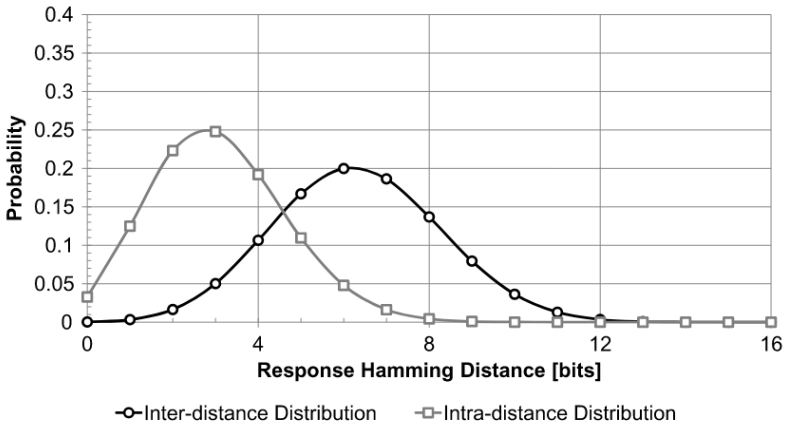
1. **Enrollment Phase**: Collect the inherent identities of every entity that needs to be identified. When using PUFs for identification, this step would involve creating CRPs and storing them in a database [15] or training a model, which is able to imitate the identity of the PUF. [16]

2. **Identification Phase**: Request the entity to provide its identity and compare it to the set of entities collected in phase one to identify the correct entity. In the case of PUF-based identification, this would be requesting a response to a given challenge.

As mentioned in section 2.2.2, the inherent random variations in PUFs cause responses to not be entirely consistent, but to vary across different evaluations of the same challenge. This presents a complication when identifying entities, as the recorded responses during the enrollment phase can be expected to be slightly different from the responses provided during identification, which means that a simple comparison will likely fail to identify a large amount of entities correctly. [4, p. 121f]

The concept of identifiability was discussed in section 2.2.3, where it was defined as a property found in PUFs with high probability of having a larger inter- than intra-distance. Figure 3 shows the distribution of the inter- and intra-distances for a hypothetical PUF, using Hamming distances as a distance metric. [4, p. 121f]

Figure 3: Hypothetical Inter- and Intra-Distance Distributions for PUF Responses



Source: [4, p. 122]

As the inter-distance is generally larger than the intra-distance, this PUF can be considered to be identifiable. However, there are only two assumptions, that can be made with certainty:

1. Two responses with $HD = 0$ definitely stem from the same entity.
2. Two responses with $HD > 8$ definitely stem from different entities.

For all other distances, it is not certain wether the two responses stem from the same or different entities, because the probability curves overlap. If there exists an overlap like this for a given PUF class, a threshold distance t_{id} needs to be defined, if it is to be used for identification. All response pairs with a HD below t_{id} are considered to stem from the same instance. All response pairs with a HD above t_{id} are considered to stem from different instances. [4, p. 121f]

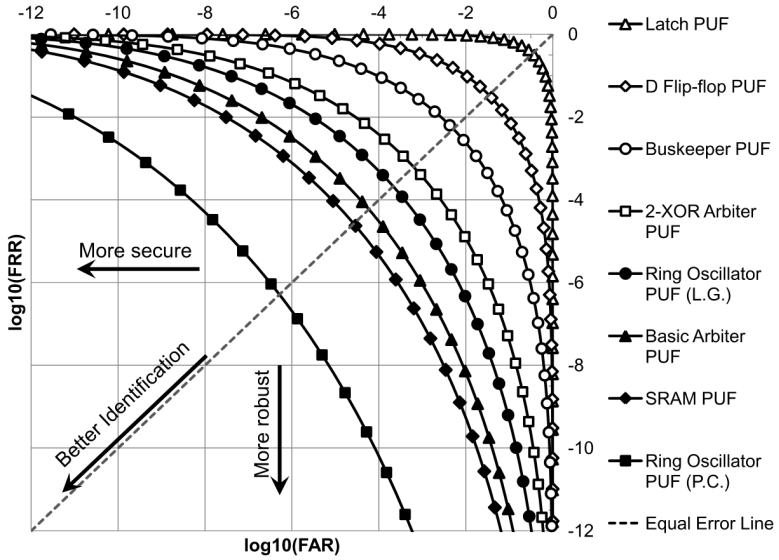
When identifying an entity, a CRP established during the enrollment phase is used and the response generated during enrollment is compared to the response generated during identification. This comparison can yield four different outcomes [4, p. 122f]:

1. *True Acceptance*: The entity is the same one that was enrolled and HD of both responses lies below t_{id}. The entity is correctly identified and accepted.
2. *False Acceptance*: The entity is different from the one that was enrolled and HD of both responses lies below t_{id}. The entity is incorrectly identified and accepted.
3. *True Rejection*: The entity is different from the one that was enrolled and HD of the responses lies above t_{id}. The entity is correctly rejected.

4. *False Rejection*: The entity is the same one that was enrolled, but the HD of both responses lies above t_{id}. The entity is incorrectly rejected.

The False Rejection Rate (FRR) is the probability, that a given identification attempt will be falsely rejected. It is simultaneously the probability, that the inter-distance is smaller than the identification threshold. The False Acceptance Rate (FAR) is the probability, that a given identification attempt will be falsely accepted. It is also the probability, that the intra-distance is larger than the threshold. These metrics can be used to find the ideal threshold for the identification system [4, p. 123]. Figure 4 shows the relationship between FRR and FAR for different PUF implementations. The curves for each type show the FRR as a function of FAR. If the threshold is lowered, the system will become more secure as the FAR decreases and the FFR increases. If the threshold is increased (higher values for HD are accepted), the system becomes more robust at the cost of security.

Figure 4: Relationship between FAR and FRR for different PUF implementations



Source: [4, p. 125]

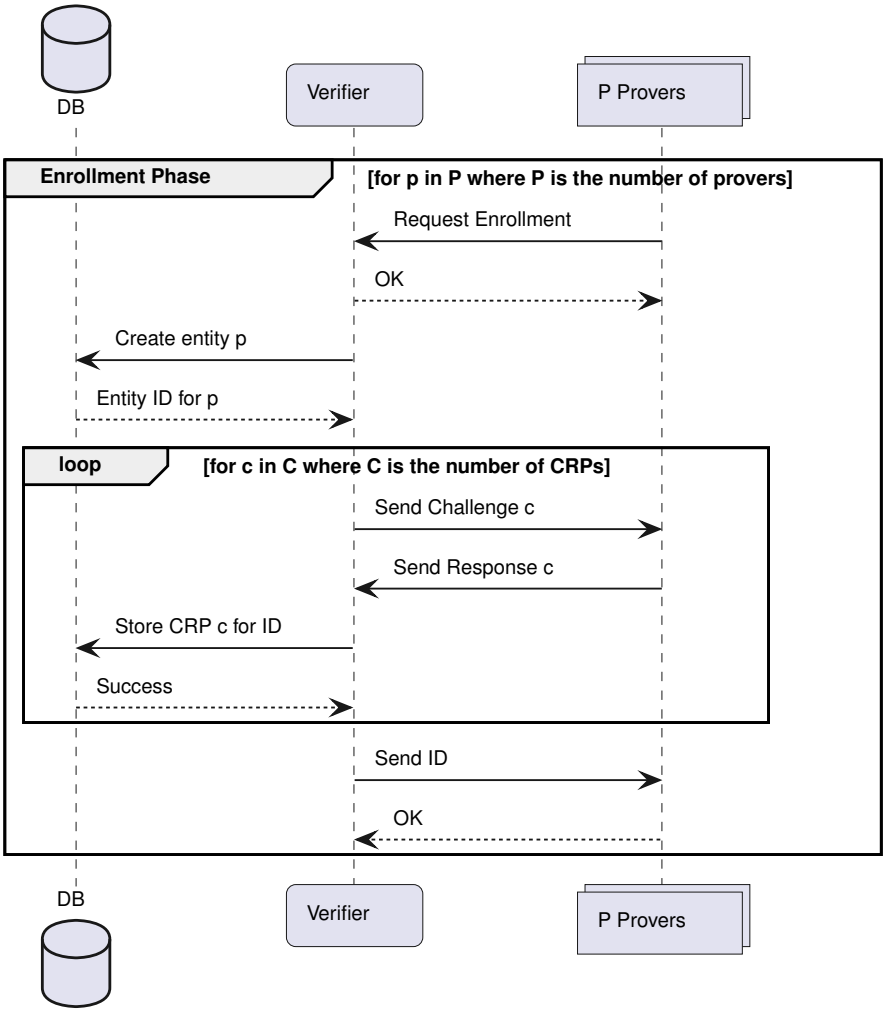
2.5.3 Basic PUF-based Challenge-Response Authentication Protocol

Challenge-Response protocols allow the inherent identifying features of PUFs to be used for authentication. The most basic protocol used for PUF-based authentication is described here.

Like with identification, a challenge-response protocol using PUFs typically consists of two phases. The first phase is *enrollment*, which is identical to the first phase present in identification and involves the verifier recording the identity of every entity by collecting

many different CRPs from each PUF and storing them in a database [4, p. 129f]. The verifier also assigns a unique ID to each entity and provides the entity with their ID.

Figure 5: UML Sequence Diagram showing the Enrollment Phase in the Basic Protocol

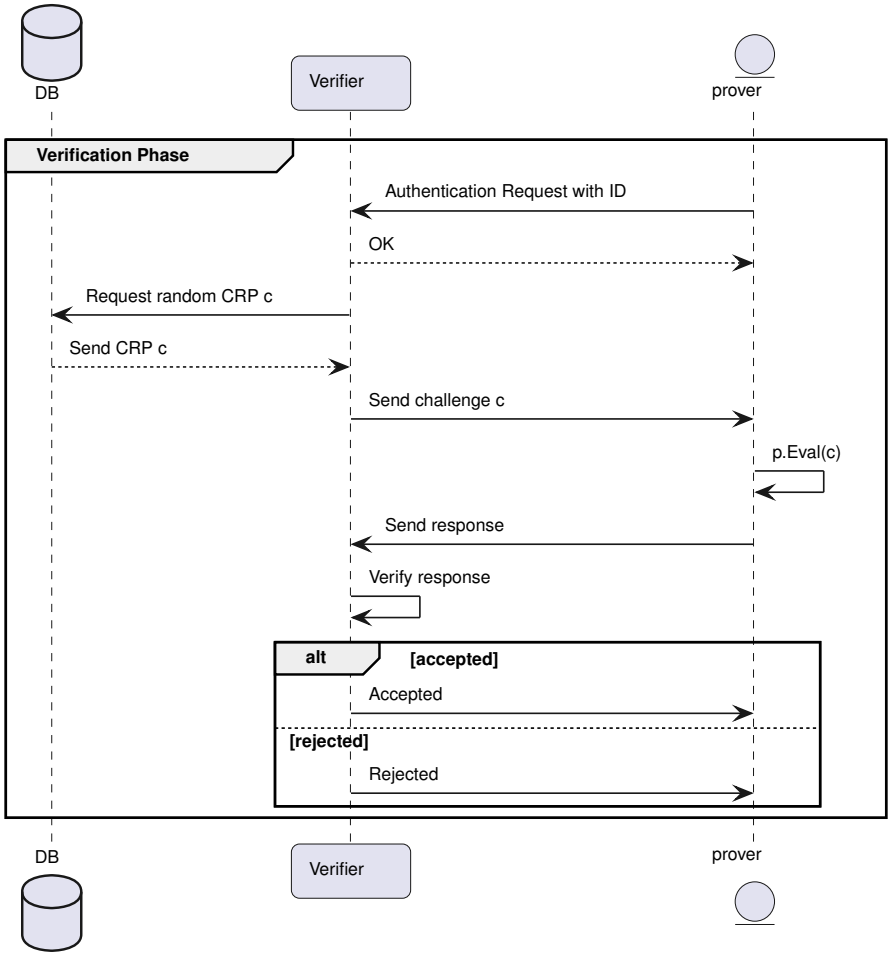


Source: Own design based on protocol proposed in [4, p. 129f]

The second phase is *verification*, shown in figure 6. Here the authenticating entity first provides their unique ID. The verifier looks up the available CRPs for that ID in the database and randomly selects one of them. The respective challenge is then sent to the prover, which uses it to generate the response and sends it to the verifier. The verifier then calculates the distance between the response recorded at enrollment and the response received from the entity and checks, wether it is below a previously defined authentication threshold t_{auth}. If this check is successful, the entity is authenticated. The used CRP is then deleted from the database, so it can not be used again. This protects against man-in-the-middle

replay attacks [4, p. 130]. As previously described, there is room for error in this process, depending on how high or low t_{auth} is set.

Figure 6: UML Sequence Diagram showing the Verification Phase in the Basic Protocol



Source: Own design based on protocol proposed in [4, p. 129f]

2.6 Other Applications for PUFs

In addition to authentication, PUFs can also be applied different scenarios. The following is a short overview of alternative use-cases for PUFs, to get a sense of their versatility.

2.6.1 Storage and Generation of Cryptographic Keys

As explained in section 2.3.4, weak PUFs are not ideal for authentication, but have different uses. One application for weak PUFs is to generate and store cryptographic keys. The

security of cryptographic protocols is entirely reliant on selecting a key with appropriate entropy and keeping that key secret. If an attacker were to extract the key from memory, application code or a database, the security of the given system is compromised. [8, p. 3f]

By using PUFs, keys don't have to be stored permanently, but can instead be generated on demand by the PUF, stored temporarily in memory and deleted after usage. Essentially, the PUF itself acts as a key that only generates a copy, which is kept for a short period of time. A challenge is sent to the PUF to generate a key (response) from it. The response is never entirely precise, so error correction is used to ensure that keys are consistent. Next, the response is converted into a uniform key using a hash function, which can then be used for encryption. [17, p. 884f]

Even though weak PUFs don't have a large set of CRPs, the central properties of PUFs are still essential. The PUF needs to be unpredictable, so an attacker is not able to create a model of the PUF used for key generation. If the attacker were able to do this, they could generate the appropriate key and intercept communications. The PUF also needs to be unclonable, so no third-party is able to have direct access to the exact CRPs. It needs to be robust, so the same key is always generated for a given challenge. [17, p. 885]

2.6.2 Object Identification

This is again a good use-case for weak PUFs, as the burden of proving the identity is eliminated, allowing the use of less secure constructions. In terms of weak PUFs, identification of specific instances of a class could again be done by using a single CRP. After manufacturing, a generic challenge could be posed to each instance and responses could be recorded in a database. If the identity of a device is to be established at a later point, one would simply have to issue the same challenge to the instance and look up the response in the database [4, p. 118]. An example use-case for object identification would be tracking specific products in a logistics system [4, p. 118]. One could also use this method to detect counterfeit products, as the counterfeit would not be able to have the same response behavior as the original, because PUFs are unclonable [17, p. 884].

2.6.3 Internet of Things

The two applications explained above both deliver great value in the Internet of Things (IoT) space. As more devices and appliances in the daily life of consumers become connected and "smart", there is a need to create trust in these devices and ensure security. By having IoT devices generate the keys they use to authenticate to external systems using intrinsic

random variations, their identity can be reliably and consistently verified. Constructions like the SRAM PUF are especially interesting in this regard, as they do not require modifications to hardware and can be applied to all classes of existing electronic devices that use embedded SRAM memory. [2, p. 88]

3 Prototype Implementation of a Protocol

3.1 Selecting a Protocol

The initial plan for this thesis was to take the knowledge gained from the literature review and use it to design a new protocol, which combines the best features of each protocol to perfectly fit the requirements of the research project. However, during the review it became clear, that the design of such a protocol is far out of scope for this thesis, as extensive knowledge of cryptography and extensive security proofs are needed to ensure the security of the protocol. Due to this fact, the methodology is changed and one of the reviewed protocols is selected, based on the requirements. The protocol is then examined closely in order to fully understand it. After that, a prototype of the infrastructure is implemented.

The goal of the research project, that this thesis is a part of, is to design a complete authentication system, that can be used in an electronic locking system for physical access protection. There are several criteria that can be derived from this goal:

- To prevent unauthorized persons from entering protected areas, security is of the highest importance. The protocol should thus not be susceptible to any known attacks that could compromise the integrity of the system.
- The system should be practical for daily use when entering buildings, floors or rooms. Therefore, authentication needs to be fast, and there needs to be an easy and secure way for the authentication device to interface with the system.
- Authentication needs to be possible at multiple physical entry points
- Privacy and anonymity are important, because users would carry these devices with them for long periods of time, potentially allowing attackers to track them.

Looking at the reviewed papers in table 2, [18], [19] and [20] immediately seem like good choices, because they are focused on the use-case of RFID. This would allow the PUF devices to be embedded in RFID tags, which could interface with RFID readers at every required physical entry point to authenticate persons against the system and grant access. In addition, [19] delivers solid proofs for all security related features, does not require any special PUF implementation, is able to handle noisy PUF responses and is proven to be scalable. Therefore, this protocol is selected.

In the next section, the protocol is thoroughly examined.

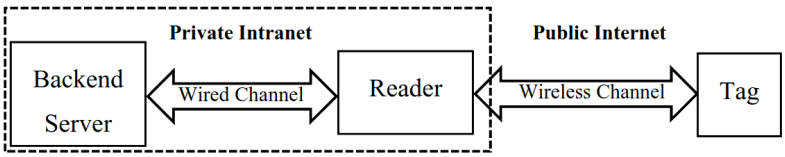
3.2 Design of the Protocol

3.2.1 Assumptions

Two versions of the protocol were proposed by Zhu et al. One version assumes an ideal PUF with intra-distance $HD = 0$, while the other uses fuzzy extractors to deal with noise in PUF responses [19, p. 6, 8]. To keep the complexity manageable for the implementation phase, the simplified version of the protocol is examined in this thesis. This means, that from now on, the PUF is considered to have ideal challenge-response characteristics. In a future work, this could easily be improved upon by implementing the enhanced version of the protocol, which is specified in great detail in the paper [19, p. 8].

Figure 12 shows the layout of the underlying RFID system, including the three parties backend server, reader and tag. The tag is a small device embedded with a PUF module, which has inherently unique characteristics. The backend server and reader are connected through a secure channel, which an attacker does not have access to. A physical attack on the tag is assumed to change the behavior of the PUF and make the tag useless. Both tag and server have access to a hash function, which generates the same output for a given input on both sides. The tag is considered to be resource limited, which means that the protocol does not specify any spatially or computationally intensive tasks on the tag [19, p. 5].

Figure 7: Layout of the RFID infrastructure



Source: [19, p. 5]

In addition to the hash function, each party requires two more operations. One is an XOR operation, denoted as $\oplus$, the other is a concatenation of two bitstrings, denoted as $\|$.

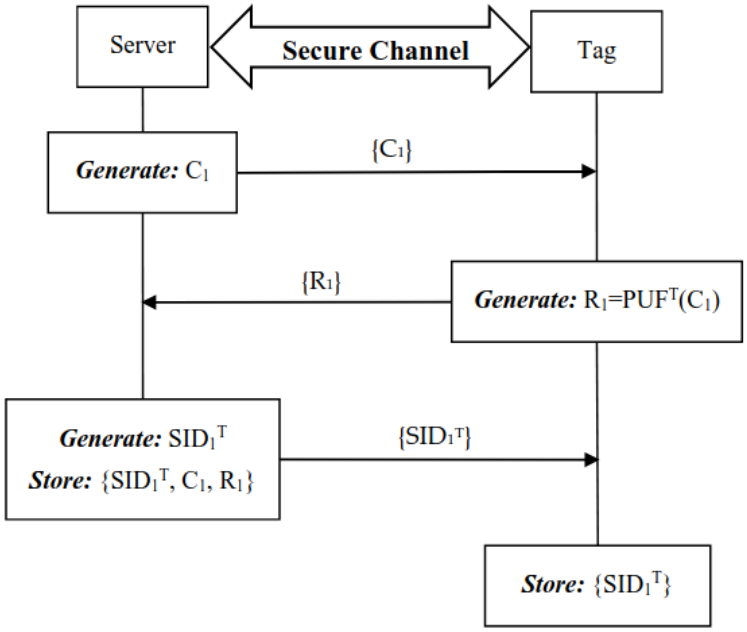
The protocol consists of two phases, the setup phase and the authentication phase. [19, p. 6-8]

3.2.2 Setup Phase

The setup phase only needs to be executed once in the beginning to initially synchronize tag and server over a secure channel. It is comparable to the enrollment phase discussed

for the basic protocol in section 2.5.3. After that, the authentication phase can happen over an insecure channel [19, p. 7]. Figure 13 shows a sequence diagram of the setup phase.

Figure 8: Setup Phase of the Proposed Protocol



Source: [19, p. 6]

In the first step, the server generates a challenge and sends it to the tag. The tag evaluates its internal PUF module using the challenge, producing a response. The response is sent back to the server. Next, the server prepares a unique session identity SID, which is used later in the first round of authentication. The server sends it to the tag, which stores it in memory. The server stores the SID, as well as the CRP for the tag. This concludes the setup phase. The tag is now enrolled in the system an can be used for authentication from now on. [19, p. 7]

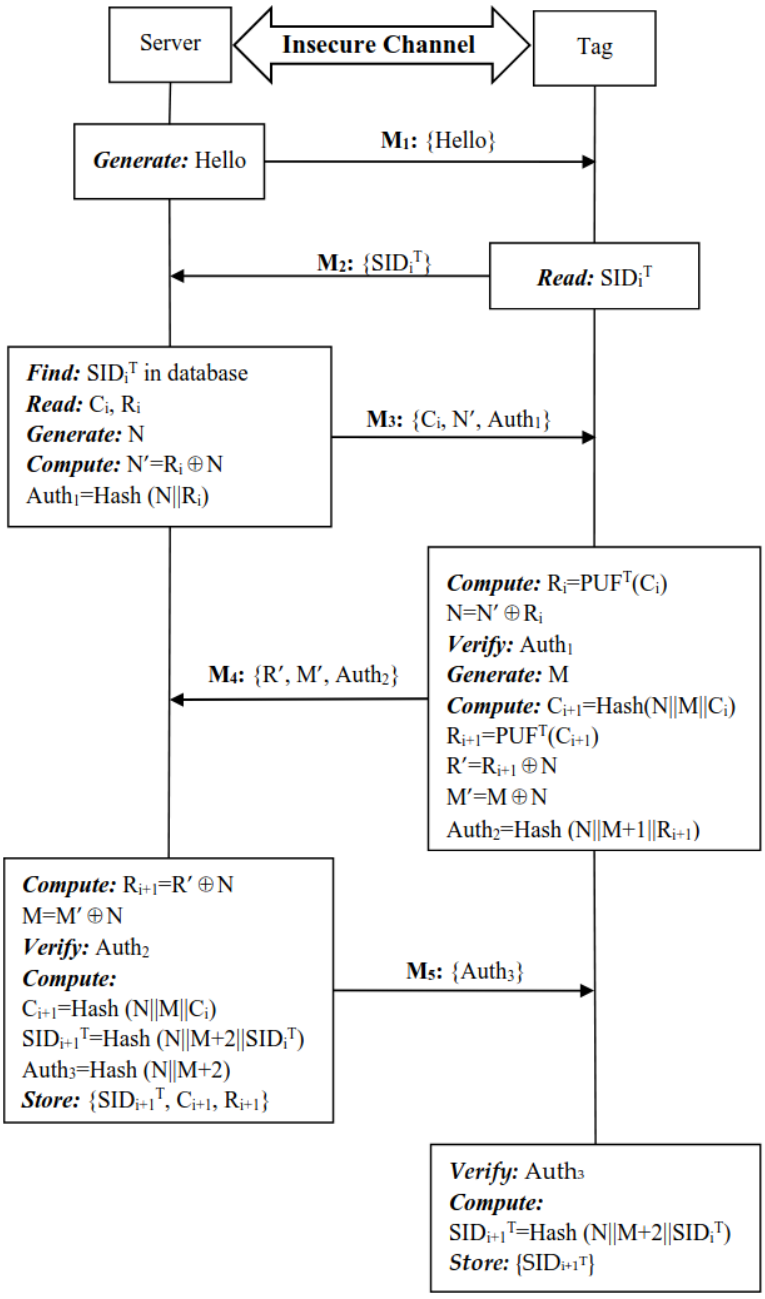
This process can be repeated for an arbitrary number of tags. The server generates a different challenge and SID for each tag and stores all of them in its database. [19, p. 7]

3.2.3 Authentication Phase

The authentication phase is described next. Note that the reader is not shown in figure 14, as it is connected with the server through a secure channel. Therefore, server and reader are seen as a single unit. In reality, the messages from the tag would be received by the reader and then forwarded to the server. Figure 14 shows the i-th round of authentication.

This means that variables like C_i and R_i are used in the current round of authentication, while variables like C_{i+1} are prepared for the next round.

Figure 9: Authentication phase of the proposed protocol



Source: [19, p. 7]

In the first step, the server sends a "Hello" message to initiate the protocol. The tag answers this with the session identifier it received during the setup phase or a previous authentication phase. [19, p. 7]

Next, the server uses the SID to find the previously stored CRP for that tag in the database. If it can't find the SID in the database, the tag is considered to be invalid and the authentication phase is terminated. Next it generates a random number N and performs an XOR operation $R_i \oplus N$ on the random number and the response to produce N'. It hashes a concatenation of N and the response to produce $Auth_1$. The challenge, N' and $Auth_1$ are sent to the tag. [19, p. 7]

The tag uses the challenge C_i sent by the server to evaluate its PUF module and produce the response R_i. Note that in this scenario, the PUF is considered ideal, so the response is equal to the one the server had stored in its database. It uses the response and N' to perform the same XOR operation and compute N. It then produces the same hash from N and the response as the server to verify $Auth_1$, as proof that the server is genuine, because knowledge of R_i is needed to calculate $Auth_1$ in the first place. After verification, it generates a second random number M and uses it to compute the next challenge C_{i+1} which is a hash of the concatenation of N, M and the challenge $C_{i+1} = Hash(N \parallel M \parallel C_i)$. The PUF is evaluated using C_{i+1} to produce the next response R_{i+1}. $R' = R_{i+1} \oplus N$ and $M' = M \oplus N$ are calculated using XOR operations on N. Lastly, the tag calculates $Auth_2 = Hash(N \parallel M + 1 \parallel R_{i+1})$. It sends R', M' and $Auth_2$ back to the server. [19, p. 7]

The server uses R' and M' received from the tag to compute R_{i+1} and M in the same way as the tag. It uses them to verify $Auth_2$, which proves the identity of the tag, as the tag would not be able to generate a correct $Auth_2$ without knowing N, which requires knowledge of the response to C_i, that it can only know using the PUF module. It computes the next challenge C_{i+1}, the next session identity SID_{i+1} and $Auth_3$. The server stores the next sid, and the next CRP in its database and sends $Auth_3$ to the tag. However, the old CRP and SID are still kept in the database, to prevent desynchronization attacks. The tag verifies $Auth_3$ in the same way, computes the next session ID and updates it internally. [19, p. 7]

If during this process, verification of any of the three $Auth$ parameters fails on either party, the authentication procedure should be terminated to prevent attacks. If all validations pass, each party has proven their identity to the other party. They are thus mutually authenticated. [19, p. 7]

3.3 Python Implementation

A basic prototype of the protocol was implemented using Python. Note that the full code is not shown in the text to improve readability and focus on the important details. For the full implementation, refer to Appendix 3.

3.3.1 Simulating the PUF Module

The PUF module was simulated using the *pypuf* Python library [21]. It provides simulations for many of the major strong PUFs, including Arbiter PUFs and Optical PUFs. For this implementation, the *ArbiterPUF* simulation was used. The following code shows how the simulation is used in code:

```
>>> from pypuf.simulation import ArbiterPUF
>>> puf = ArbiterPUF(n=64, seed=1)
>>> from pypuf.io import random_inputs
>>> puf.eval(random_inputs(n=64, N=3, seed=2))
array([ 1,  1, -1], dtype=int8)
```

The constructor takes the number of switch gates, which equals the number of challenge bits, and a seed which defines the random characteristics of the PUF module. The library additionally supports setting a level of noisyness for the PUF, however this was not used here. The PUF instance provides a $puf.eval()$ method, which takes a two dimensional array of shape (N,n), where n is the number of challenge bits and N is the number of evaluations. Each value of the array is an integer of either -1 or 1, which dictates wether that switch is turned on or off. The PUF returns an array of length N, because on each evaluation of the PUF, a single value is returned. The response consists of the results of N evaluations, making it length N.

While testing, it was noticed that for small N, it is easy to have colliding PUF responses for different challenges, which would decrease security of the implementation. In the end, $N = 8; n = 64$ was used, as it proved to have no problems with value collisions, even at a high number of evaluations.

3.3.2 Additional Dependencies

For hashing, the *sha256* hash function from the builtin Python *hashlib* was used. The library provides a number of hash functions, which can easily be imported and used to hash different types of data. There is no particular reasoning behind the choice of *sha256* for hashing. As there are no hardware constraints for this implementation, complexity was not an issue. However on a real RFID tag, one would not be able to implement *sha256*, as the number of logic gates required is far too high. Therefore, in a real-world setting, a different lightweight hash function, like *SPONGENT*, should be used [19, p. 2].

To represent the binary challenges and responses, the Python *bitstring* library was used. It provides a *BitArray* class, which allows storing binary data in a space efficient way, while

also providing methods to convert hex or integer data to binary and vice-versa, which proved useful during the implementation of the protocol steps. The *numpy* library was used to convert the PUF responses from the *ArbiterPUF* modules to and from *BitArrays*.

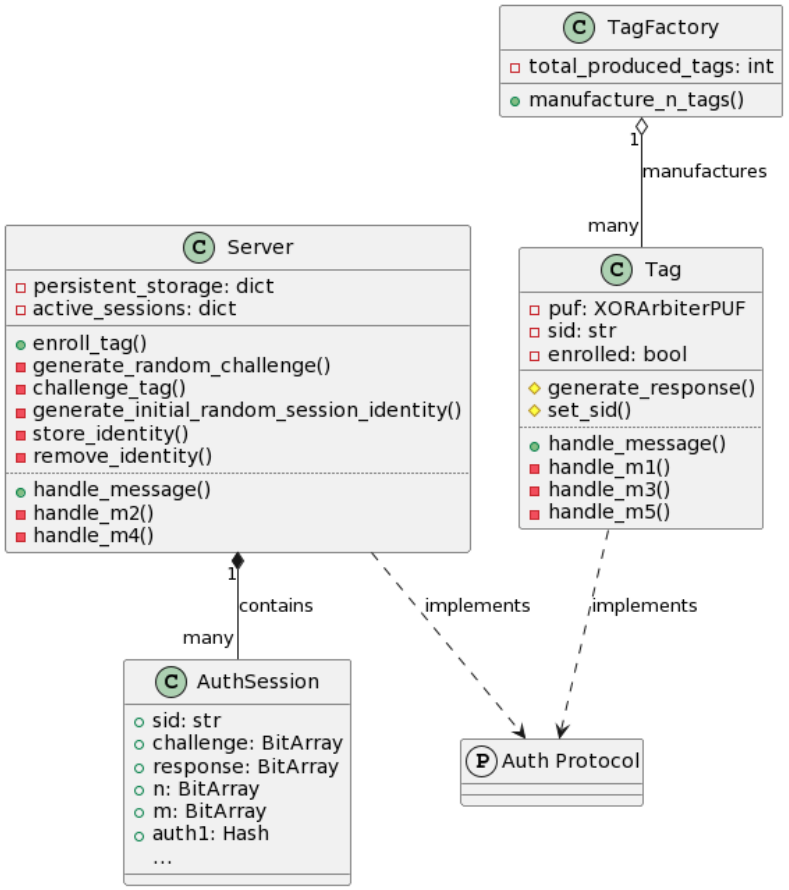
For the creation of SIDs, a source of randomness was needed. For this, the Python builtin *uuid* module, as well as the *random* module were used.

The *pytest* framework was used for testing the implementation.

3.3.3 Structure of the Program

The structure of the main program is shown in figure 15. It consists of four main classes *Server*, *Tag*, *TagFactory* and *AuthSession*.

Figure 10: UML Class Diagram showing Structure of Main Implementation



Server and *TagFactory* are implemented using a singleton pattern to ensure, that only one instance can be present at all times, which helps to keep state consistent. *TagFactory* keeps track of the number of total produced *Tags*, which is important for managing random

variations. As the *ArbiterPUF* takes a seed integer, the factory needs to ensure, that no two tags with the same seed are created, as they would have PUF modules with the same characteristics. Therefore, each produced tag is assigned a different seed based on the total number of tags in existence.

The server has two types of storage, a dictionary storing active sessions of type *AuthSession* and a persistent storage dictionary, which is used for storing SIDs and CPRs for all enrolled tags. The server also provides an *enroll_tag* method, which takes a tag and a challenge seed and executes the setup phase of the protocol with the tag, so it can be used for authentication in the future. *AuthSession* is a type responsible for storing all relevant data needed in the authentication phase. The server supports the use of multiple readers at the same time, which allows multiple *Tags* to be authenticated simultaneously. Each active authentication phase needs its own *AuthSession*. The sessions are destroyed, once the tag has been authenticated and the new SID and CRP have been stored in persistent storage.

A *Tag* consists of a PUF module, a variable for storing the SID and a boolean which keeps track of whether the tag has been enrolled by the server. It also has two methods for generating responses and setting the SID on the tag, which the server can call directly during the setup phase.

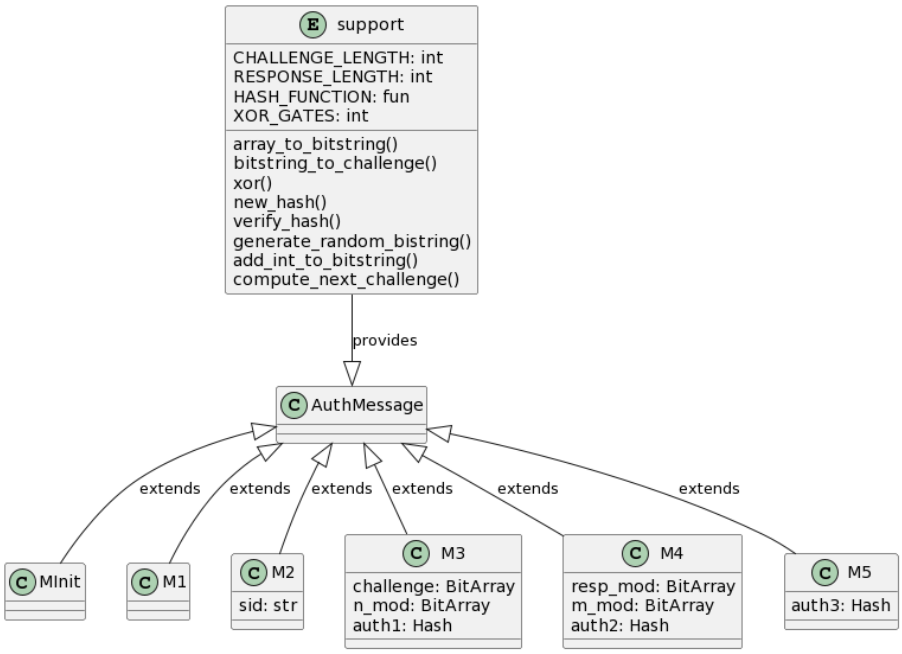
Both *Tag* and *Server* provide a *handle_message* method. It is used for communication between the two entities during the authentication phase. The server *handle_message* takes an *AuthMessage* , and a reader ID, which is needed to keep track of the different simultaneous *AuthSessions*. The tag's *handle_message* only takes messages, as it does not need to keep track of sessions.

Server and Tag both implement the Auth Protocol, but this is only included in the class diagram for the purpose of showing that both entities support the protocol. There is no specific *AuthProtocol* type implemented.

In addition to the main entities described above, there is an external *support* module which provides all necessary supporting functions and constants, that are not defined by the protocol, but both parties need to agree on. The support module structure is shown in figure 16. It includes constant values for the length of challenges and responses, as well as the hash function that should be used by each party to verify responses. It also provides functions for converting numpy arrays to *BitArrays*, as the PUF module takes and returns numpy arrays, but the rest of the program uses *BitArrays* for handling binary data, as they have better compatibility with hash functions and other calculations. The basic operations

required by the protocol are also provided, including creation of a new hash, verifying a hash, XOR and concat, among others.

Figure 11: UML Class Diagram of the Support Module



Lastly the *support* module provides types for each of the messages required for the authentication phase. These classes extend a common class *AuthMessage* and each message holds different values, like *sid*, *challenge*, or *auth* values, depending on what data is being sent in that step of the authentication phase. The *handle_message* endpoint of both tag and server takes the *AuthMessage* type and all of its children and uses the specific type internally, to ensure that communication is synchronized and that each message is handled according to protocol.

3.3.4 Enrollment of Tags

The following code sets up the infrastructure, manufactures $n = 10$ tags with unique PUF modules and enrolls them with the server:

```
factory = TagFactory()
server = Server()
n = 10
tags = factory.manufacture_n_tags(10)
for seed, tag in enumerate(tags):
    server.enroll_tag(tag, seed)
```

First, the factory and server are instantiated. Next the factory's manufacturing method is called with the number of tags that should be created.

The code for tag creation looks like this:

```
def manufacture_n_tags(self, n) -> List['Tag']:
    tags = []
    for i in range(0, n):
        seed = self.total_produced_tags
        puf = ArbiterPUF(n=CHALLENGE_LENGTH, seed=seed)
        tags.append(Tag(puf))
        print(f"Tag with seed {seed} manufactured")
        self.total_produced_tags += 1

    return tags
```

The seed for the PUF module is directly taken from the factories total production count. This makes sure that each tag has a unique challenge response behavior. It also makes the behavior of tags very predictable, but this can be ignored, because in the real world the behavior would stem from uncontrollable intrinsic variations. The PUF is created with the challenge length provided by the support module. A new tag object is instantiated with the PUF module and added to the list of tags. The list is then returned, concluding the production process.

Next, the newly created tags need to be enrolled. Therefore, the server's enroll_tag method is called directly for each tag. The following code shows that method:

```
def enroll_tag(self, tag: tag.Tag, challenge_seed: int) -> None:
    challenge = self.generate_random_challenge(challenge_seed)
    response = self.challenge_tag(tag, challenge)
    sid = self.generate_initial_random_session_identity()
    self.store_identity(sid, challenge, response)
    tag.set_sid(sid)
    print("Enrolled Tag with SID ", sid)
```

It takes the tag and a challenge seed, which it uses to generate a random challenge of the correct length. It then calls the server's challenge_tag method, which triggers the tag's PUF module to generate a response. Next, a session id is generated and SID and the CRP are stored in the server's persistent storage. Lastly, the SID is set in the tag's memory and the enrollment is finished.

3.3.5 Authentication Phase

To demonstrate the authentication phase, the mutual authentication of a single tag is shown here. We assume a fully set up infrastructure with Tag *tag* and Server *server*. The following code executes the authentication phase and returns a success message:

```
tag = tags[0]
reader = 0
m1 = server.handle_message(support.MInit(), reader)
m2 = tag.handle_message(m1)
m3 = server.handle_message(m2, reader)
m4 = tag.handle_message(m3)
m5 = server.handle_message(m4, reader)
success = tag.handle_m5(m5)
if success:
    print("Tag with SID: ", tag.sid, "successfully mutually
    authenticated.")
```

The server's and tag's handle_message methods are called in an alternating way, using the previous message returned by the other party as an argument each time. Note that the server's method takes two arguments, to account for the use of multiple RFID readers at the same time on a single server. The next piece of code shows the server's handle_message method:

```
def handle_message(self, m: AuthMessage, reader_id: int) -> None:
    if reader_id in self.active_sessions:
        session = self.active_sessions[reader_id]
    else:
        session = self.AuthSession()
        self.active_sessions[reader_id] = session
        session.expected_message = MInit
    if type(m) != session.expected_message:
        raise TypeError(
            f"Expected {session.expected_message}, got {type(m)}")
    elif (type(m)) == MInit:
        session.expected_message = M2
        return M1()
    elif (type(m)) == M2:
        session.expected_message = M4
        return self.handle_m2(m, session)
    elif (type(m)) == M4:
        m = self.handle_m4(m, session)
        del self.active_sessions[reader_id]

        print(f"Tag at reader {reader_id} successfully authenticated.")
```

```
22          print("New session identity written to persistent storage\n")
23          return m
24      else:
25          raise TypeError("Unknown Message Type")
```

It first checks if there is an active session for the given reader and creates a new one if there is none. Next it checks which type of message was sent and calls the appropriate internal method to handle that message. To ensure consistency within a session, the last received message is recorded and an expected_message variable is set, which is checked before each handling call. After handling M4, the session is deleted from the session memory, as the tag is now successfully authenticated.

The code of handle_m2 is shown next to explain handling of specific messages:

```
1  def handle_m2(self, m, session: AuthSession) -> M3:
2      session.sid = m.sid
3      try:
4          session.challenge = self.persistent_storage[session.sid]["c"]
5          session.response = self.persistent_storage[session.sid]["r"]
6      except KeyError:
7          raise ValueError(
8              "Invalid SID sent by tag.\nTag might be invalid.
     Authentication terminated.")
9
10     session.n = generate_random_bitstring(RESPONSE_LENGTH)
11     session.n_mod = xor(session.response, session.n)
12     session.auth1 = new_hash(
13         bytes(concat(session.n, session.response).hex, 'utf-8'))
14     return M3(session.challenge, session.n_mod, session.auth1)
```

M2 is sent by the tag after the initial hello message and contains the SID stored in the tag's memory. It is stored in the session, then the persistent_storage of the server is queried, to find the CRP stored during enrollment or a previous authentication phase. If it is not found, the tag is assumed to be invalid and needs to be re-enrolled. The session generates N not as an integer, but in the form of a random BitArray of length *RESPONSE_LENGTH*. This is done so it is possible to easily perform XOR operations on the random number with the challenge. *n_mod* represents N' from the protocol definition. *auth1* is created by hashing the hexadecimal representation of the concatenation of n and the $response$. n, n_mod and *auth1* are all stored in the session and returned in *M3*.

Only a small part of this implementation has been shown in this section. To look at the full code, please reference Appendix 3.

3.3.6 Testing the Implementation

To test the implementation, some rudimentary test scenarios were developed using the *pytest* framework. It provides methods for asserting, wether a variable has taken on the correct value, or if the correct errors were raised during execution. The following tests were implemented:

- Enrolling 100 tags and authenticating them in sequence.
- Enrolling 100 tags and authenticating them in parallel, using 100 different readers.
- Enrolling a single tag and authenticating it 100 times in sequence, to test if generation of new CRPs and SIDs works correctly.
- Enrolling two tags and switching them in the middle of the authentication process on the same reader. This should raise a ValueError, as the hashes should not be able to be verified.
- Enrolling a tag and changing its SID before the authentication phase. This should again lead to a ValueError, as the SID should not be found in the server's database.
- Enrolling a tag and modifying its internal PUF module before authentication. This would be the equivalent of an attacker tampering with the tag and changing the PUFs characteristics. This should raise a ValueError, as the challenge response behavior has changed.

As an example the last test is shown here:

```
1  def test_tag_with_invalid_puf():
2      """An enrolled tag with a PUF module, that was modified by an
       attacker,
3      should not be able to authenticate."""
4      factory = TagFactory()
5      server = Server()
6      tag = factory.manufacture_n_tags(1)[0]
7      server.enroll_tag(tag, 0)
8
9      attacker_puf = ArbiterPUF(n=CHALLENGE_LENGTH, seed=1)
10     tag.puf = attacker_puf
11
12     m1 = server.handle_message(MInit(), 0)
13     m2 = tag.handle_message(m1)
14     m3 = server.handle_message(m2, 0)
15     with pytest.raises(ValueError):
16         tag.handle_message(m3)
```

After enrollment, the puf variable of the tag is reassigned to a newly created ArbiterPUF instance with a different seed. Following this, a ValueError is raised, because the authentication hash can not be verified on the tag's side. This could still be circumvented by an attacker, by disabling the tag-side hash verification, but the tag still would not be able to generate $Auth_2$ correctly, due to the different PUF response.

Note, that these tests do not represent a comprehensive security analysis of the protocol, but give some sense of its capabilities. For the complete set of tests, refer to Appendix 3.

4 Review of Existing PUF-based Authentication Protocols

4.1 Methodology for the Literature Review

To be able to conceptualize a protocol and the surrounding infrastructure, a review of the existing literature on PUF-based challenge-response authentication protocols is conducted. In [3], Brocke et al. propose guidelines for conducting literature reviews in the information systems field. A summary of the five-step framework can be found in Appendix 1.

Steps one and two of the framework are not necessary in the context of a bachelor's thesis, as the scope has been defined in section 1 and fundamentals have been laid out extensively in section 2. During the research of fundamental topics, it became clear, that defining the authentication protocol is central to answering *RQ1*. Therefore, the following search string was developed:

> *(physical unclonable function OR puf OR physically unclonable function) AND (authentication OR challenge response OR challenge-response) AND (protocol)*

The databases all have slightly different syntax requirements for search strings, so the actual strings used differ a bit from this one. The exact strings are shown in Appendix 2.

The search string was applied to IEEEXplore, ACM and Google Scholar and restricted to document titles. IEEExplore yielded 53 results. Most of them are centered around lightweight PUF-based authentication protocols for IoT applications. Results were ordered by number of citations and evaluated by their titles and abstracts. Some articles were deemed irrelevant and thus not considered. An example for this are [22] and [23],as they propose PUF-based authentication for distributed systems using blockchain technology, which is out of scope for this thesis.

Next the same search was conducted in the ACM Digital Library. This yielded only three results, none of which were relevant.

To fill in the gaps, the same search was conducted using Google Scholar. This yielded 104 results, most of which were irrelevant or already contained in the IEEExplore search results. From this search, only articles that were well-cited, very recent or offered interesting new perspectives were considered. This resulted in eight results across different publishers, namely Elsevier, Springer and MDPI. Backward and forward search was not used to further increase the number of papers, as the sample size was already large enough for the purposes of this review.

4.2 Reviewed Articles

Table 1 shows an overview of the papers used, source database, number of citations and year published. As analyzing all articles in depth would be too time-consuming, a subset of the most relevant results was selected beforehand. During this process, articles with a higher number of citations were preferred. Additionally, because this thesis is part of a research project, concerned with creating a physical access protection system using PUF-based authentication, literature proposed for similar purposes was preferred. Lastly, it was important that a wide variety of approaches was included. To ensure this, if two papers had a similar approach, the one with more citations was preferred.

Table 1: Summary of reviewed papers

Article	Citations	Year	Database	Section
Majzoobi et al. [16]	124	2012	IEEExplore	4.3.1
Gope et al. [18]	103	2018	IEEExplore	4.3.2
Chatterjee et al. [24]	95	2019	IEEExplore	4.3.3
Braeken [25]	78	2018	Google Scholar	4.3.4
Bansal et al. [26]	48	2020	IEEExplore	4.3.5
Zhu et al. [19]	19	2019	Google Scholar	4.3.6
Jiang et al. [27]	10	2019	IEEExplore	4.3.7
Gope et al. [28]	9	2022	IEEExplore	4.3.8
Hristea et al. [20]	4	2019	Google Scholar	4.3.9

4.3 Analysis of the collected Literature

Next, each of the papers is analyzed. The focus of analysis was put on a couple of central questions:

- What is the goal of the protocol and what problem does it solve?
- What requirements to the underlying infrastructure (hardware, databases) does the protocol have?
- What kinds of attacks does the protocol aim to prevent?
- Is the protocol a good solution for the research questions posed in section 1.2

4.3.1 Slender PUF Protocol by Majzoobi et al.

This protocol requires the use of a PUF with a specific statistical property called the strict PUF avalanche criterion. It describes PUFs, which behave in a way, that any bit-flip in the challenge will result in half of the response bits to flip. To achieve this, the protocol uses a PUF consisting of eight individual Arbiter PUFs connected through XOR logic gates, which improves the statistical properties to fulfill this criterion. [16]

As a strong PUF, it has an exponentially large set of CRPs, which limits attacks by recording previously used CRPs (replay attacks), because the same challenge does not have to be used multiple times. This method is also used in the basic authentication protocol discussed in section 2.5.3, where CRPs are deleted from the database after usage. This safeguard alone does not protect against machine learning attacks, where an untrusted third-party can use previously recorded CRPs to train a model of the PUF. This model learns the response behavior and potentially allows an attacker to calculate the appropriate response to a future challenge. This protocol aims to make this kind of attack impossible [16, p. 34f].

The protocol assumes that the verifier knows the relationship of challenges and responses for each instance through a compact model of the PUF, which has been trained during the enrollment phase and is stored in a database on the verifiers side. The model can be trained with as many CRPs as necessary, because the PUF can be directly evaluated. After enrollment, the physical access points that allow direct evaluation, need to be disabled to prevent potential attackers with physical access from evaluating the PUF directly and and building a model. This is done by blowing a fuse to disable the external pins used for access to the PUF. After this, the PUF can only be accessed through a controller on the device, which adheres to the protocol specifications and prevents direct readouts. The only other way to obtain knowledge about the challenge-response behavior at this point would be by opening the PUF, which would likely change its characteristics. [16]

In the paper, three types of attacks on this protocol are discussed. The first is a PUF modeling attack, where an attacker trains a model on previously recorded CRPs to try and imitate a PUF instance. The basic PUF authentication protocol does not protect against this kind of attack, as the full challenges and responses are exposed during the authentication process. This protocol mitigates this risk, by not directly exposing responses. Instead, only substrings with a random starting index are exposed. There is still a way for an attacker to create a model by guessing the substring each time, but the amount of CRPs required to successfully build a model increases dramatically which greatly reduces the

risk. Of course, this attack vector would have to be analyzed much more closely before deploying a system using this protocol. [16]

The second is a random guessing attack. The paper gives concrete figures for the probability of such an attack being successful. For example, a protocol implementation with a response length of 1024 bits, substring length of 256 bits and Hamming authentication threshold of 76 would only give a $10^{-}11$ chance of a successful guess. [16]

The paper also discuss attacks that compromise the randomness of the seed by impersonating either verifier or prover, but these are deemed to be fixable depending on the implementation. [16]

The main benefits of this protocol include the resiliency against ML attacks, the robustness against noise in PUF responses without the need for traditional expensive error correction. [16]

4.3.2 Lightweight Anonymous Protocol for RFID Systems by Gope et al.

This protocol was developed for using PUFs as a cryptographic primitive for entity authentication in IoT using RFID technology. A typical RFID system involves an RFID tag, a reader and a backend server. The protocol requires the tag to consist of a micro controller and a tamper-proof PUF with intrinsic random variations. It also assumes a secure connection between the reader and the backend. [18]

There are many security issues and attack vectors with traditional RFID systems. One problem with typical RFID tags is, that they threaten the user's privacy, because identifying information stored on the tag can be retrieved without immediate access to the device and without need for authentication by the other party. This allows tracking the location of users. Attacks against forward secrecy are another problem. As the communication channel in RFID is insecure, an eavesdropper can record all traffic, although it is usually encrypted. If at some point in the future, the encryption is broken, all previously recorded communication can now be accessed by the attacker. Another problem is a desynchronization attack, where a third party blocks messages between reader and tag, to desynchronize communication. In many classical RFID authentication protocols, the secret is exchanged frequently to provide forward secrecy. If an attacker blocks the acknowledgement from the server, that the secret was successfully changed, the tag does not know which secret to use. This will cause a denial of service for that tag. Additional attacks include impersonation attacks, physical attacks, where information is retrieved by reading the physical memory on the tag, or cloning attacks, where a copy of the tag is created. [18]

This protocol aims to protect against these attacks. For example, PUFs can introduce anonymity and protect the privacy of users. Mutual authentication between server and tag is used to protect against impersonation attacks and location tracking. The researchers analyzed many previously proposed authentication schemes and improved upon them. Additionally, the protocol exhibits resilience against Denial of Service (DoS) attacks and forward secrecy. It does not require secret keys to be stored on the tag and does not require computationally expensive operation on the backend side, which makes it scalable. [18]

4.3.3 Protocol without explicit CRPs in Verifier Database by Chatterjee et al.

This protocol focuses on a PUF-based authentication solution for a network of distributed IoT devices, that does not require CRPs to be stored in a database on the verifier's side. This is important, because in all of the previously described protocols, exposure of the database is a high security risk. The use of Identity Based Encryption (IBE) and keyed hash functions allows this protocol to only require the verifier to store a single key in its storage [24, p. 1f]. IBE is a type of public key encryption, where the public key can be any string [29, p. 8]. This differs from regular PKI, where the keys have to have a specific length and format, dictated by the encryption system (e.g. AES). [29, p. 3]

Protecting that single key is much easier than ensuring security for an entire database. The protocol is designed, so that a prover does not have to store a key and does not need to do expensive computational work. The protocol allows for an unlimited number of authentications using the same prover [24, p. 2f]. The paper mathematically proves, that the protocol is resistant against eavesdropping attacks. [24, p. 8]

Requirements include, that each prover has to have a PUF module with intrinsic random variations and some additional mathematical and cryptographic modules for scalar multiplication, pairing operations and cryptographic hashing. The verifier is required to be able to calculate a keyed hash function. [24, p. 1f]

4.3.4 Protocol for IoT by Braeken

This protocol was developed for IoT applications. It aims to solve problems with a previously published protocol and is thus developed to protect against Man-in-the-Middle (MITM), DoS, impersonation and replay attacks. Additionally it offers mutual authentication, which means that both the node and the server prove their identity to each other. The protocol focuses on power efficiency and low computational cost [25, p. 1f, 8]. It relies

on storing a large amount of CRPs in a database on the verifier side and requires a strong PUF [25, p. 4]. In addition to simple authentication, it also allows the communication of two nodes with each other, even though a central verification server is still needed. [25, p. 3f]

During enrollment, CRPs are shared between prover and verifier and stored in the database. After authentication, a public and private key pair are generated. Public keys can be shared with other nodes, with the server acting as a trusted party facilitating the exchange. [25, p. 4]

4.3.5 Lightweight Protocol for V2G by Bansal et al.

This protocol is also developed for IoT applications. More specifically, it is designed to be used in a vehicle smart grid ecosystem, which would allow to tightly integrate electric cars into the power grid and use their batteries for energy trading and load management, enabling a more efficient use of the grid's energy. This approach is also called Vehicle to Grid (V2G) and would help to manage the ever increasing power demand world wide. However, the communication between the grid and vehicles is a privacy and security concern, because a lot of sensitive information is exchanged between the parties, which an attacker could compromise. This could lead to imbalanced energy transactions or privacy violations. Physical access to devices is also easy, because vehicles are usually parked for a long time without supervision. PUFs are used here as a means of tamper-proofing and protection against physical attacks, as no keys need to be stored. Tampering likely changes the characteristic of the PUF and cloning is impossible. [26, p. 7234]

The system consists of three parties, the vehicle, an aggregator (charging station) and the power grid. Both the aggregators and the vehicles have PUF modules, as both entities need to be authenticated with privacy and security. Mutual authentication is required between grid and aggregator, as well as vehicle and aggregator, to ensure a secure exchange of encryption keys, which can be used for further communication. To protect privacy, the keys are generated using the PUF during each authentication run and pseudonyms are used for identifying vehicles to protect their information. [26, p. 7234f] Only one challenge response pair needs to be stored on the grid server for every vehicle. It achieves this with significantly reduced computational overhead compared to similar protocols for V2G in the field. [26, p. 7244]

There are four central security goals of the protocol. Messages need to be confidential, so only authorized entities can access the information. The integrity of messages needs to be ensured, so aggregators and the grid can know, if a message has been altered

or compromised in any way. The identity of vehicle owners has to be kept hidden from an attacker even when eavesdropping. Mutual authentication is also a requirement to prevent impersonation attacks and false energy transfer [26, p. 7237]. The paper formally proves resiliency against man-in-the-middle attacks, impersonation attacks, replay attacks and physical attacks. It is additionally able to provide mutual authentication, protection of identity, message integrity and security of session keys. [26, p. 7243]

4.3.6 Lightweight mutual RFID protocol by Zhu et al.

This protocol is another attempt at creating a mutual authentication protocol for RFID systems. It focuses on the same problems and goals as [18], which were discussed in section 4.3.2, and also makes the same assumptions about the infrastructure [19, p. 1, 5]. Because the paper was published after [18], it identifies some weaknesses of that protocol and many other proposed implementations in the RFID space and tries to improve on them. Specifically, it identifies an issue with the protection against desynchronization attacks, where a tag can run out of pseudo-identities that are used for authentication, if communication is denied by an attacker too many times. This leads to the tag needing to be re-enrolled. This protocol is able to correct that flaw. [19, p. 4] Another drawback to the protocol in [18] is, that it requires active RFID tags, which have an internal power source, whereas this protocol is able to be implemented with passive tags. [19, p. 18]

The paper provides proofs, that the protocol is secure against a large number of attacks, including universal untraceability of tags, ensurance of forward secrecy, secure mutual authentication, resilience against desynchronization attacks, resiliency against physical and cloning attacks and immunity to modeling attacks [19, p. 12-15]. All of these attacks are very relevant in an RFID setting. It is also able to do this more efficiently both in space complexity and bandwidth requirements and comes close to other protocols in computational complexity. The protocol is considered to be scalable. [19, p. 16]

4.3.7 Two-Factor protocol by Jiang et al.

A new approach is introduced here, which has not been discussed in any other paper considered here. This protocol combines PUF-based authentication with a password that the user enters, to create a Two-Factor Authentication (2FA) authentication protocol. It is designed to be used in IoT, specifically Internet of Vehicles (IoV), for managing vehicles and automating intelligent route planing. [27, p. 195]

As with [26], PUFs are a solid foundation for security in vehicles, because they are good at preventing physical attacks, which is a big issue with cars that are left unattended for long periods of time. The combination with a password ensures, that a vehicle can only be authenticated if the owner or a different trusted party is present. [27, p. 195]

The architecture of the protocol involves three parties, a data center which manages the information, users which receive data from sensors and statistical data used for route planning from the data center, and the vehicle sensors which collect real-time traffic data. It involves three phases, including system setup, registration and login/authentication [27, p. 197]. The protocol needs to be able to protect data including the vehicles location or driving data, like speed and fuel consumption, to ensure privacy and traffic safety. It also needs to protect against attackers injecting false information about external conditions, threatening the safety of passengers. [27, p. 195f]

The paper proves that the protocol successfully implements mutual authentication, providing anonymity. Vehicles are not identifiable or traceable, as no easily identifiable information is sent. The protocol is secure against physical attacks because of the PUF-based nature. The paper also claims that the protocol is secure against impersonation and desynchronization attacks, but does not specifically argue that point. Only one CRP per entity has to be stored by the server, which is another security benefit. [27, p. 199f]

4.3.8 Protocol level approach to prevent ML attacks by Gope et al.

This protocol is aimed at IoT systems in the medical field. The main focus is to provide a protocol which protects against machine learning attacks on PUFs. The architecture of the systems includes several medical IoT devices like wearable health sensors or clinical systems, which communicate with a central IoT gateway. This gateway talks to a service provider over the internet, who provides the authentication server, information database and analysis systems. As with many of the other papers, the wireless nature of Internet of Medical Things (IoMT) devices opens the system up to a number of attacks, such as replay, man-in-the-middle, impersonation, physical and guessing attacks. Additionally modeling attacks are a big threat. [28, p. 1971f]

The proposed protocol aims to improve upon several previously proposed protocols including [16], which is part of this review. Gope et al. criticize it, because it does not provide sufficient protection against replay and impersonation attacks [28, p. 1972]. This protocol uses a special type of PUF, called One-Time PUF (OPUF), which changes its behavior after each session, making modelling entirely impractical. Benefits of the protocol offer mutual authentication preserving privacy, prevention of modelling attacks, good scalability,

prevention of man-in-the-middle and replay attacks [28, p. 1972]. It also provides forward secrecy and prevents physical attacks. These security characteristics are argued and verified through security simulation software [28, p. 1979].

4.3.9 Destructive private mutual RFID protocol by Hristea et al.

This protocol is another attempt at a mutual authentication protocol for RFID systems [20, p. 331]. It assumes the same architecture as [19] and [18], consisting of a tag, reader and server component. The main threats this protocol is trying to solve are impersonation and tracking of tags. The main benefit this paper offers, is that it is stateful, meaning that it allows a tag to have persistent state, which can be updated and shared with the reader, while also providing resilience against physical attacks, mutual authentication and forward secrecy. [20, p. 332] It is also scalable and able to prevent desynchronization attacks. The security of the protocol is proven using a privacy model for RFID protocols. [20, p. 340f]

4.4 Discussion of Results

In this section, the results of the literature review are discussed.

The methodology used for collecting the relevant literature to be reviewed is described in detail in section 4.1. The database queries with the respective search strings lead to a set of papers which were mostly very recent. Almost all papers were published between 2018 and 2022, with one outlier, [16], published in 2012. An important criteria for the selection of papers was also the number of citations, as this was used as a measure of relevance in the field. Having a lot of citations does not prove that the paper is well received, as it could also be used as a "negative example" by other researchers. This possibility was not further investigated, and might be a weak point of this review. Additionally, almost all papers included an extensive set of references to existing protocols, that were not found by the methodology. This could mean, that they are either too old, did not include the relevant search terms in the title or had only few citations.

Some articles even referenced other articles that were included in the set of reviewed papers. In [19], Zhu et al. outlined a number of security flaws with the protocol proposed in [18], including a lack of protection against desynchronization attacks and improvements on power efficiency. In [28], Gope et al. critized the protocol proposed by Majzoobi et al. in 2012, mentioning its lack of protection against replay and impersonation attacks. This again suggests, that the set of selected papers is quite relevant.

The reviewed articles provided a diverse set of different interpretations on a challenge-response protocols, with varying requirements, intended applications, focuses and security strengths or weaknesses.

The articles focus two main application areas: RFID and IoT. IoT is further categorized in general IoT applications IoMT and IoV which consists of applications for traffic management, as well as V2G. The varying applications lead to a number of different underlying architectures. Protocols focused on RFID are typically based on a three layer architecture, including the RFID tags, the readers and a backend server [18] [19]. [28] provided a unique approach, where devices talk to an IoT gateway, which is connected to an external service provider over the internet for authentication. V2G also has an interesting architecture, using charging stations as a communication endpoint for the IoT devices, which then talk to a backend server. The wide variety of devices, which can be secured with PUF, like cars, medical devices and RFID tags, also highlights the versatility of PUF technology.

Looking at the different types of PUF required by the protocols, it is surprising to see, that many protocols don't have a specific requirement for a PUF implementation. This suggests, that the PUF implementation is actually not as relevant to the protocols and infrastructure as previously assumed. Some papers on the other hand include a requirement for strong PUFs [18] [19], while others require a certain set of statistical properties [16] or a specific OPUF implementation [28]. The loose requirements of many protocols do not make the choice of PUF irrelevant. There are generally strict statistical requirements for PUFs used in authentication, which have been touched on in section 2.5.2. Some protocols only provide support for ideal PUFs which are assumed to have perfect statistical properties, but do not actually exist in the real world. Near perfect PUF robustness can be achieved through extensive error correction using e.g. fuzzy extractors, but this is computationally expensive. [30]

On the verifier side, there are many approaches to managing the identity of provers. [16] provides the use of a lightweight model of the PUF used to verify responses. In [19], only one CRP is stored in the database at any time and a new CRP is agreed on between prover and verifier on each authentication run. In [24], a method is proposed, that does not require the verifier to store any CRPs.

All papers strongly focus on the security aspects of their protocols, as it is crucial in providing a secure authentication system. Table 2 shows an in-depth overview of the security related features of each protocol. For interpreting these results correctly, it is important to consider, that the security of each protocol is interpreted without extensive knowledge in the field of cyber security and cryptography. The table should therefore not be viewed as an actual representation of the security features of each protocol, but rather, as an

overview over which features were explicitly proven to be present. As most papers used some kind of security framework for proofs, there is a high probability, that some of these properties were implicitly proven in a non obvious way. These kinds of proofs could not be taken into account, which means that some protocols might actually be more secure than the table suggests.

Table 2: Results of the Literature Review from a Security Perspective

Paper	[16]	[18]	[24]	[25]	[26]	[19]	[27]	[28]	[20]
Section	4.3.1	4.3.2	4.3.3	4.3.4	4.3.5	4.3.6	4.3.7	4.3.8	4.3.9
Use Case	Any	RFID	IoT	IoT	V2G	RFID	IoV	IoMT	RFID
Required PUF	Any*	Strong	Strong	PUF-FSM	N	Any	Any	OPUF	Any
Handles Noisy PUF resp.	✓	✓	✓	✓		✓	✓	✓	
Machine Learning Attack	✓					✓		✓	
Random Guessing Attack	✓	✓	✓			✓			
Desynchronization Attack		✓				✓	?	✓	✓
Denial of Service Attack		✓		✓		✓		?	
Impersonation Attack	✓	✓	✓	✓	✓	✓	✓	?	✓
Physical Attack	✓	✓	✓	✓	✓	✓	✓	✓	✓
Cloning Attack	✓	✓	✓	✓	✓	✓	✓	✓	✓
Mutual Authentication		✓			✓	✓	✓	✓	✓
Anonymity					✓	✓	✓	✓	✓
Forward Secrecy		✓				✓		✓	✓
Scalable		✓				✓		?	✓
Security Proof	AM	AM	AE	A	AM	A	A	AS	A

* PUF must fit the strict PUF avalanche criterion
✓= Proven to be secure against attack or implement feature
? = Claimed but not proven
A = Argumentative, M = Mathematical, S = Simulation, E = Experiment, N = Not specified

All papers proved the security of their protocol in some way. Methods used for this include argumentative proofs with established frameworks like Session-Key Security or the Universal Composability Framework [24], mathematical proofs showing the negligible chances of a successful attack through probabilistic calculations [18], experimental proofs using actual hardware [24], or the use of software simulations [28]. An important distinguishing feature between protocols is mutual authentication. It means, that not only the entity proves their identity to the server, but also the other way around. This is a central requirement for privacy and anonymity, as it ensures, that entities only have to share relevant identifying information with a party they trust. This makes it useful for preventing location tracking in RFID systems. In regards to security, the table shows a clear winner in terms of being able to prove the most security related features, which is [19], published by Zhu et al. in 2019.

5 Prototype Implementation of a Protocol

5.1 Selecting a Protocol

The initial plan for this thesis was to take the knowledge gained from the literature review and use it to design a new protocol, which combines the best features of each protocol to perfectly fit the requirements of the research project. However, during the review it became clear, that the design of such a protocol is far out of scope for this thesis, as extensive knowledge of cryptography and extensive security proofs are needed to ensure the security of the protocol. Due to this fact, the methodology is changed and one of the reviewed protocols is selected, based on the requirements. The protocol is then examined closely in order to fully understand it. After that, a prototype of the infrastructure is implemented.

The goal of the research project, that this thesis is a part of, is to design a complete authentication system, that can be used in an electronic locking system for physical access protection. There are several criteria that can be derived from this goal:

- To prevent unauthorized persons from entering protected areas, security is of the highest importance. The protocol should thus not be susceptible to any known attacks that could compromise the integrity of the system.
- The system should be practical for daily use when entering buildings, floors or rooms. Therefore, authentication needs to be fast, and there needs to be an easy and secure way for the authentication device to interface with the system.
- Authentication needs to be possible at multiple physical entry points
- Privacy and anonymity are important, because users would carry these devices with them for long periods of time, potentially allowing attackers to track them.

Looking at the reviewed papers in table 2, [18], [19] and [20] immediately seem like good choices, because they are focused on the use-case of RFID. This would allow the PUF devices to be embedded in RFID tags, which could interface with RFID readers at every required physical entry point to authenticate persons against the system and grant access. In addition, [19] delivers solid proofs for all security related features, does not require any special PUF implementation, is able to handle noisy PUF responses and is proven to be scalable. Therefore, this protocol is selected.

In the next section, the protocol is thoroughly examined.

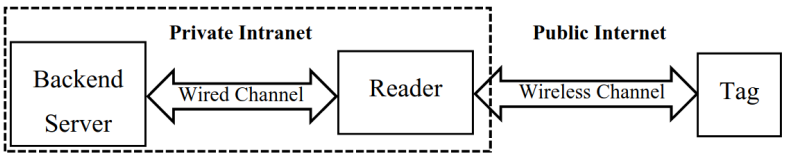
5.2 Design of the Protocol

5.2.1 Assumptions

Two versions of the protocol were proposed by Zhu et al. One version assumes an ideal PUF with intra-distance $HD = 0$, while the other uses fuzzy extractors to deal with noise in PUF responses [19, p. 6, 8]. To keep the complexity manageable for the implementation phase, the simplified version of the protocol is examined in this thesis. This means, that from now on, the PUF is considered to have ideal challenge-response characteristics. In a future work, this could easily be improved upon by implementing the enhanced version of the protocol, which is specified in great detail in the paper [19, p. 8].

Figure 12 shows the layout of the underlying RFID system, including the three parties backend server, reader and tag. The tag is a small device embedded with a PUF module, which has inherently unique characteristics. The backend server and reader are connected through a secure channel, which an attacker does not have access to. A physical attack on the tag is assumed to change the behavior of the PUF and make the tag useless. Both tag and server have access to a hash function, which generates the same output for a given input on both sides. The tag is considered to be resource limited, which means that the protocol does not specify any spatially or computationally intensive tasks on the tag [19, p. 5].

Figure 12: Layout of the RFID infrastructure



Source: [19, p. 5]

In addition to the hash function, each party requires two more operations. One is an XOR operation, denoted as $\oplus$, the other is a concatenation of two bitstrings, denoted as $\|$.

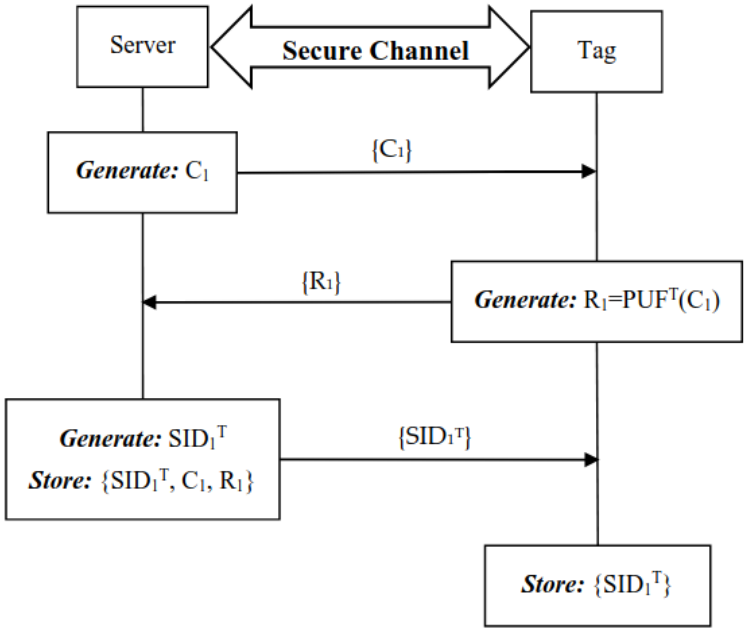
The protocol consists of two phases, the setup phase and the authentication phase. [19, p. 6-8]

5.2.2 Setup Phase

The setup phase only needs to be executed once in the beginning to initially synchronize tag and server over a secure channel. It is comparable to the enrollment phase discussed

for the basic protocol in section 2.5.3. After that, the authentication phase can happen over an insecure channel [19, p. 7]. Figure 13 shows a sequence diagram of the setup phase.

Figure 13: Setup Phase of the Proposed Protocol



Source: [19, p. 6]

In the first step, the server generates a challenge and sends it to the tag. The tag evaluates its internal PUF module using the challenge, producing a response. The response is sent back to the server. Next, the server prepares a unique session identity SID, which is used later in the first round of authentication. The server sends it to the tag, which stores it in memory. The server stores the SID, as well as the CRP for the tag. This concludes the setup phase. The tag is now enrolled in the system an can be used for authentication from now on. [19, p. 7]

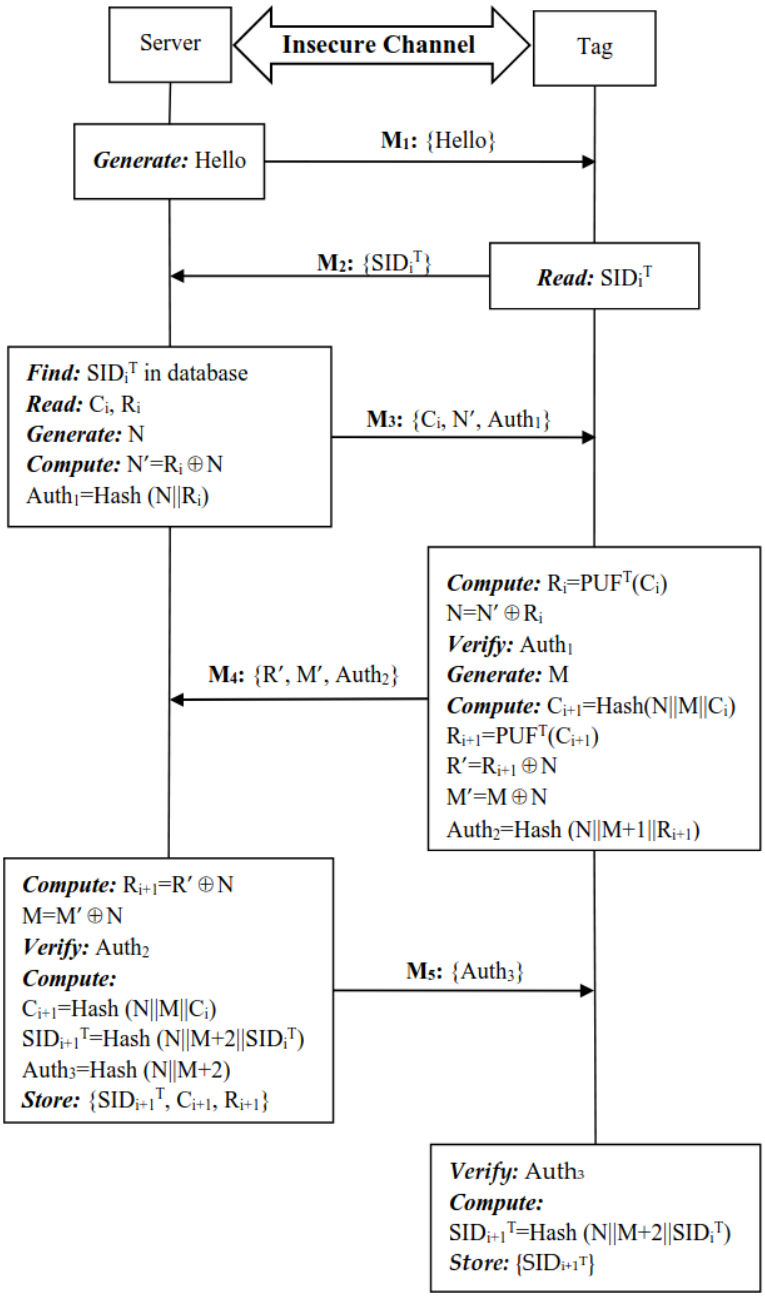
This process can be repeated for an arbitrary number of tags. The server generates a different challenge and SID for each tag and stores all of them in its database. [19, p. 7]

5.2.3 Authentication Phase

The authentication phase is described next. Note that the reader is not shown in figure 14, as it is connected with the server through a secure channel. Therefore, server and reader are seen as a single unit. In reality, the messages from the tag would be received by the reader and then forwarded to the server. Figure 14 shows the i-th round of authentication.

This means that variables like C_i and R_i are used in the current round of authentication, while variables like C_{i+1} are prepared for the next round.

Figure 14: Authentication phase of the proposed protocol



Source: [19, p. 7]

In the first step, the server sends a "Hello" message to initiate the protocol. The tag answers this with the session identifier it received during the setup phase or a previous authentication phase. [19, p. 7]

Next, the server uses the SID to find the previously stored CRP for that tag in the database. If it can't find the SID in the database, the tag is considered to be invalid and the authentication phase is terminated. Next it generates a random number N and performs an XOR operation $R_i \oplus N$ on the random number and the response to produce N'. It hashes a concatenation of N and the response to produce $Auth_1$. The challenge, N' and $Auth_1$ are sent to the tag. [19, p. 7]

The tag uses the challenge C_i sent by the server to evaluate its PUF module and produce the response R_i. Note that in this scenario, the PUF is considered ideal, so the response is equal to the one the server had stored in its database. It uses the response and N' to perform the same XOR operation and compute N. It then produces the same hash from N and the response as the server to verify $Auth_1$, as proof that the server is genuine, because knowledge of R_i is needed to calculate $Auth_1$ in the first place. After verification, it generates a second random number M and uses it to compute the next challenge C_{i+1} which is a hash of the concatenation of N, M and the challenge $C_{i+1} = Hash(N \parallel M \parallel C_i)$. The PUF is evaluated using C_{i+1} to produce the next response R_{i+1}. $R' = R_{i+1} \oplus N$ and $M' = M \oplus N$ are calculated using XOR operations on N. Lastly, the tag calculates $Auth_2 = Hash(N \parallel M + 1 \parallel R_{i+1})$. It sends R', M' and $Auth_2$ back to the server. [19, p. 7]

The server uses R' and M' received from the tag to compute R_{i+1} and M in the same way as the tag. It uses them to verify $Auth_2$, which proves the identity of the tag, as the tag would not be able to generate a correct $Auth_2$ without knowing N, which requires knowledge of the response to C_i, that it can only know using the PUF module. It computes the next challenge C_{i+1}, the next session identity SID_{i+1} and $Auth_3$. The server stores the next sid, and the next CRP in its database and sends $Auth_3$ to the tag. However, the old CRP and SID are still kept in the database, to prevent desynchronization attacks. The tag verifies $Auth_3$ in the same way, computes the next session ID and updates it internally. [19, p. 7]

If during this process, verification of any of the three $Auth$ parameters fails on either party, the authentication procedure should be terminated to prevent attacks. If all validations pass, each party has proven their identity to the other party. They are thus mutually authenticated. [19, p. 7]

5.3 Python Implementation

A basic prototype of the protocol was implemented using Python. Note that the full code is not shown in the text to improve readability and focus on the important details. For the full implementation, refer to Appendix 3.

5.3.1 Simulating the PUF Module

The PUF module was simulated using the *pypuf* Python library [21]. It provides simulations for many of the major strong PUFs, including Arbiter PUFs and Optical PUFs. For this implementation, the *ArbiterPUF* simulation was used. The following code shows how the simulation is used in code:

```
>>> from pypuf.simulation import ArbiterPUF
>>> puf = ArbiterPUF(n=64, seed=1)
>>> from pypuf.io import random_inputs
>>> puf.eval(random_inputs(n=64, N=3, seed=2))
array([ 1,  1, -1], dtype=int8)
```

The constructor takes the number of switch gates, which equals the number of challenge bits, and a seed which defines the random characteristics of the PUF module. The library additionally supports setting a level of noisyness for the PUF, however this was not used here. The PUF instance provides a $puf.eval()$ method, which takes a two dimensional array of shape (N,n), where n is the number of challenge bits and N is the number of evaluations. Each value of the array is an integer of either -1 or 1, which dictates wether that switch is turned on or off. The PUF returns an array of length N, because on each evaluation of the PUF, a single value is returned. The response consists of the results of N evaluations, making it length N.

While testing, it was noticed that for small N, it is easy to have colliding PUF responses for different challenges, which would decrease security of the implementation. In the end, $N = 8; n = 64$ was used, as it proved to have no problems with value collisions, even at a high number of evaluations.

5.3.2 Additional Dependencies

For hashing, the *sha256* hash function from the builtin Python *hashlib* was used. The library provides a number of hash functions, which can easily be imported and used to hash different types of data. There is no particular reasoning behind the choice of *sha256* for hashing. As there are no hardware constraints for this implementation, complexity was not an issue. However on a real RFID tag, one would not be able to implement *sha256*, as the number of logic gates required is far too high. Therefore, in a real-world setting, a different lightweight hash function, like *SPONGENT*, should be used [19, p. 2].

To represent the binary challenges and responses, the Python *bitstring* library was used. It provides a *BitArray* class, which allows storing binary data in a space efficient way, while

also providing methods to convert hex or integer data to binary and vice-versa, which proved useful during the implementation of the protocol steps. The *numpy* library was used to convert the PUF responses from the *ArbiterPUF* modules to and from *BitArrays*.

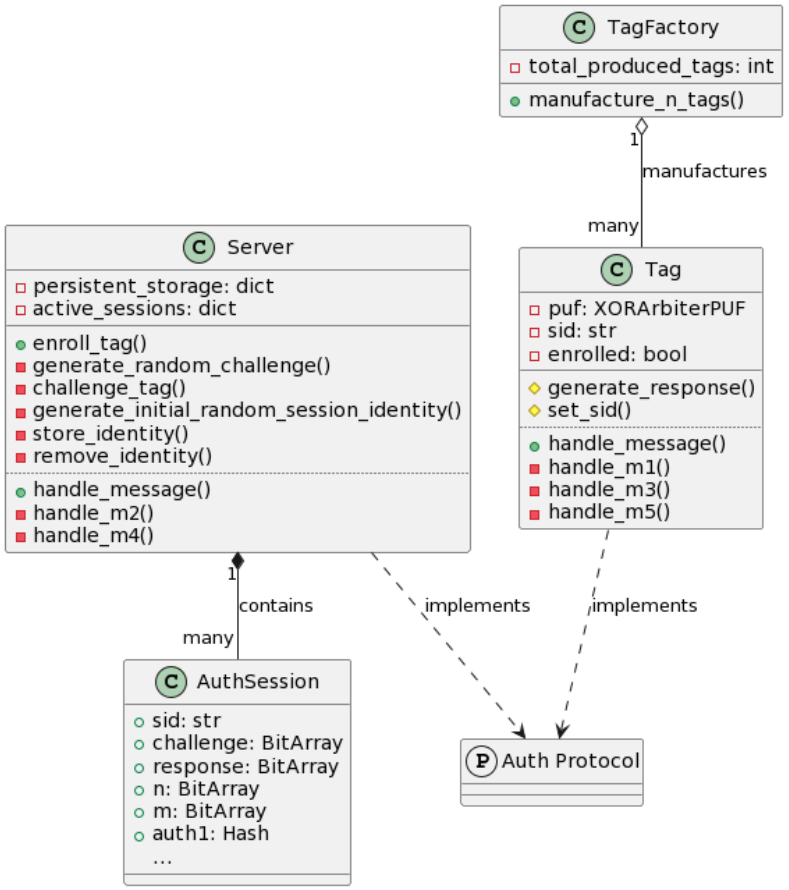
For the creation of SIDs, a source of randomness was needed. For this, the Python builtin *uuid* module, as well as the *random* module were used.

The *pytest* framework was used for testing the implementation.

5.3.3 Structure of the Program

The structure of the main program is shown in figure 15. It consists of four main classes *Server*, *Tag*, *TagFactory* and *AuthSession*.

Figure 15: UML Class Diagram showing Structure of Main Implementation



Server and *TagFactory* are implemented using a singleton pattern to ensure, that only one instance can be present at all times, which helps to keep state consistent. *TagFactory* keeps track of the number of total produced *Tags*, which is important for managing random

variations. As the *ArbiterPUF* takes a seed integer, the factory needs to ensure, that no two tags with the same seed are created, as they would have PUF modules with the same characteristics. Therefore, each produced tag is assigned a different seed based on the total number of tags in existence.

The server has two types of storage, a dictionary storing active sessions of type *AuthSession* and a persistent storage dictionary, which is used for storing SIDs and CPRs for all enrolled tags. The server also provides an *enroll_tag* method, which takes a tag and a challenge seed and executes the setup phase of the protocol with the tag, so it can be used for authentication in the future. *AuthSession* is a type responsible for storing all relevant data needed in the authentication phase. The server supports the use of multiple readers at the same time, which allows multiple *Tags* to be authenticated simultaneously. Each active authentication phase needs its own *AuthSession*. The sessions are destroyed, once the tag has been authenticated and the new SID and CRP have been stored in persistent storage.

A *Tag* consists of a PUF module, a variable for storing the SID and a boolean which keeps track of whether the tag has been enrolled by the server. It also has two methods for generating responses and setting the SID on the tag, which the server can call directly during the setup phase.

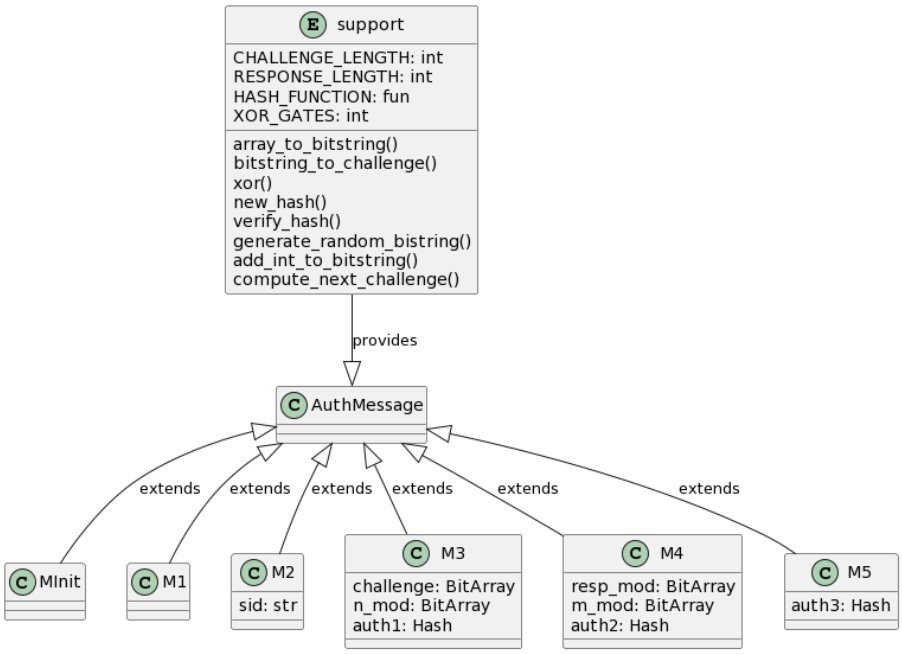
Both *Tag* and *Server* provide a *handle_message* method. It is used for communication between the two entities during the authentication phase. The server *handle_message* takes an *AuthMessage* , and a reader ID, which is needed to keep track of the different simultaneous *AuthSessions*. The tag's *handle_message* only takes messages, as it does not need to keep track of sessions.

Server and Tag both implement the Auth Protocol, but this is only included in the class diagram for the purpose of showing that both entities support the protocol. There is no specific *AuthProtocol* type implemented.

In addition to the main entities described above, there is an external *support* module which provides all necessary supporting functions and constants, that are not defined by the protocol, but both parties need to agree on. The support module structure is shown in figure 16. It includes constant values for the length of challenges and responses, as well as the hash function that should be used by each party to verify responses. It also provides functions for converting numpy arrays to *BitArrays*, as the PUF module takes and returns numpy arrays, but the rest of the program uses *BitArrays* for handling binary data, as they have better compatibility with hash functions and other calculations. The basic operations

required by the protocol are also provided, including creation of a new hash, verifying a hash, XOR and concat, among others.

Figure 16: UML Class Diagram of the Support Module



Lastly the *support* module provides types for each of the messages required for the authentication phase. These classes extend a common class *AuthMessage* and each message holds different values, like *sid*, *challenge*, or *auth* values, depending on what data is being sent in that step of the authentication phase. The *handle_message* endpoint of both tag and server takes the *AuthMessage* type and all of its children and uses the specific type internally, to ensure that communication is synchronized and that each message is handled according to protocol.

5.3.4 Enrollment of Tags

The following code sets up the infrastructure, manufactures $n = 10$ tags with unique PUF modules and enrolls them with the server:

```
factory = TagFactory()
server = Server()
n = 10
tags = factory.manufacture_n_tags(10)
for seed, tag in enumerate(tags):
    server.enroll_tag(tag, seed)
```

First, the factory and server are instantiated. Next the factory's manufacturing method is called with the number of tags that should be created.

The code for tag creation looks like this:

```
def manufacture_n_tags(self, n) -> List['Tag']:
    tags = []
    for i in range(0, n):
        seed = self.total_produced_tags
        puf = ArbiterPUF(n=CHALLENGE_LENGTH, seed=seed)
        tags.append(Tag(puf))
        print(f"Tag with seed {seed} manufactured")
        self.total_produced_tags += 1

    return tags
```

The seed for the PUF module is directly taken from the factories total production count. This makes sure that each tag has a unique challenge response behavior. It also makes the behavior of tags very predictable, but this can be ignored, because in the real world the behavior would stem from uncontrollable intrinsic variations. The PUF is created with the challenge length provided by the support module. A new tag object is instantiated with the PUF module and added to the list of tags. The list is then returned, concluding the production process.

Next, the newly created tags need to be enrolled. Therefore, the server's enroll_tag method is called directly for each tag. The following code shows that method:

```
def enroll_tag(self, tag: tag.Tag, challenge_seed: int) -> None:
    challenge = self.generate_random_challenge(challenge_seed)
    response = self.challenge_tag(tag, challenge)
    sid = self.generate_initial_random_session_identity()
    self.store_identity(sid, challenge, response)
    tag.set_sid(sid)
    print("Enrolled Tag with SID ", sid)
```

It takes the tag and a challenge seed, which it uses to generate a random challenge of the correct length. It then calls the server's challenge_tag method, which triggers the tag's PUF module to generate a response. Next, a session id is generated and SID and the CRP are stored in the server's persistent storage. Lastly, the SID is set in the tag's memory and the enrollment is finished.

5.3.5 Authentication Phase

To demonstrate the authentication phase, the mutual authentication of a single tag is shown here. We assume a fully set up infrastructure with Tag *tag* and Server *server*. The following code executes the authentication phase and returns a success message:

```
tag = tags[0]
reader = 0
m1 = server.handle_message(support.MInit(), reader)
m2 = tag.handle_message(m1)
m3 = server.handle_message(m2, reader)
m4 = tag.handle_message(m3)
m5 = server.handle_message(m4, reader)
success = tag.handle_m5(m5)
if success:
    print("Tag with SID: ", tag.sid, "successfully mutually
    authenticated.")
```

The server's and tag's handle_message methods are called in an alternating way, using the previous message returned by the other party as an argument each time. Note that the server's method takes two arguments, to account for the use of multiple RFID readers at the same time on a single server. The next piece of code shows the server's handle_message method:

```
def handle_message(self, m: AuthMessage, reader_id: int) -> None:
    if reader_id in self.active_sessions:
        session = self.active_sessions[reader_id]
    else:
        session = self.AuthSession()
        self.active_sessions[reader_id] = session
        session.expected_message = MInit
    if type(m) != session.expected_message:
        raise TypeError(
            f"Expected {session.expected_message}, got {type(m)}")
    elif (type(m)) == MInit:
        session.expected_message = M2
        return M1()
    elif (type(m)) == M2:
        session.expected_message = M4
        return self.handle_m2(m, session)
    elif (type(m)) == M4:
        m = self.handle_m4(m, session)
        del self.active_sessions[reader_id]

        print(f"Tag at reader {reader_id} successfully authenticated.")
```

```
22          print("New session identity written to persistent storage\n")
23          return m
24      else:
25          raise TypeError("Unknown Message Type")
```

It first checks if there is an active session for the given reader and creates a new one if there is none. Next it checks which type of message was sent and calls the appropriate internal method to handle that message. To ensure consistency within a session, the last received message is recorded and an expected_message variable is set, which is checked before each handling call. After handling M4, the session is deleted from the session memory, as the tag is now successfully authenticated.

The code of handle_m2 is shown next to explain handling of specific messages:

```
1  def handle_m2(self, m, session: AuthSession) -> M3:
2      session.sid = m.sid
3      try:
4          session.challenge = self.persistent_storage[session.sid]["c"]
5          session.response = self.persistent_storage[session.sid]["r"]
6      except KeyError:
7          raise ValueError(
8              "Invalid SID sent by tag.\nTag might be invalid.
       Authentication terminated.")
9
10     session.n = generate_random_bitstring(RESPONSE_LENGTH)
11     session.n_mod = xor(session.response, session.n)
12     session.auth1 = new_hash(
13         bytes(concat(session.n, session.response).hex, 'utf-8'))
14     return M3(session.challenge, session.n_mod, session.auth1)
```

M2 is sent by the tag after the initial hello message and contains the SID stored in the tag's memory. It is stored in the session, then the persistent_storage of the server is queried, to find the CRP stored during enrollment or a previous authentication phase. If it is not found, the tag is assumed to be invalid and needs to be re-enrolled. The session generates N not as an integer, but in the form of a random BitArray of length *RESPONSE_LENGTH*. This is done so it is possible to easily perform XOR operations on the random number with the challenge. *n_mod* represents N' from the protocol definition. *auth1* is created by hashing the hexadecimal representation of the concatenation of n and the $response$. n, n_mod and *auth1* are all stored in the session and returned in *M3*.

Only a small part of this implementation has been shown in this section. To look at the full code, please reference Appendix 3.

5.3.6 Testing the Implementation

To test the implementation, some rudimentary test scenarios were developed using the *pytest* framework. It provides methods for asserting, wether a variable has taken on the correct value, or if the correct errors were raised during execution. The following tests were implemented:

- Enrolling 100 tags and authenticating them in sequence.
- Enrolling 100 tags and authenticating them in parallel, using 100 different readers.
- Enrolling a single tag and authenticating it 100 times in sequence, to test if generation of new CRPs and SIDs works correctly.
- Enrolling two tags and switching them in the middle of the authentication process on the same reader. This should raise a ValueError, as the hashes should not be able to be verified.
- Enrolling a tag and changing its SID before the authentication phase. This should again lead to a ValueError, as the SID should not be found in the server's database.
- Enrolling a tag and modifying its internal PUF module before authentication. This would be the equivalent of an attacker tampering with the tag and changing the PUFs characteristics. This should raise a ValueError, as the challenge response behavior has changed.

As an example the last test is shown here:

```
1  def test_tag_with_invalid_puf():
2      """An enrolled tag with a PUF module, that was modified by an
       attacker,
3      should not be able to authenticate."""
4      factory = TagFactory()
5      server = Server()
6      tag = factory.manufacture_n_tags(1)[0]
7      server.enroll_tag(tag, 0)
8
9      attacker_puf = ArbiterPUF(n=CHALLENGE_LENGTH, seed=1)
10     tag.puf = attacker_puf
11
12     m1 = server.handle_message(MInit(), 0)
13     m2 = tag.handle_message(m1)
14     m3 = server.handle_message(m2, 0)
15     with pytest.raises(ValueError):
16         tag.handle_message(m3)
```

After enrollment, the puf variable of the tag is reassigned to a newly created ArbiterPUF instance with a different seed. Following this, a ValueError is raised, because the authentication hash can not be verified on the tag's side. This could still be circumvented by an attacker, by disabling the tag-side hash verification, but the tag still would not be able to generate $Auth_2$ correctly, due to the different PUF response.

Note, that these tests do not represent a comprehensive security analysis of the protocol, but give some sense of its capabilities. For the complete set of tests, refer to Appendix 3.

6 Conclusion

The thesis was able to lay a broad theoretical foundation about concepts surrounding PUFs, their types, their implementations and PUF-based authentication. The systematic review of the existing literature on PUF-based authentication protocols provided an overview of the wide variety of applications, focuses, security properties and performance factors. Through the literature review, it became clear, that the central part of any infrastructure is the protocol, as it defines and limits the technical requirements of the underlying hardware, backend systems and databases. Initially, it was planned to develop a new protocol fitting the requirements of the research project. However, this goal was not realized, because the review showed the complexity and knowledge required for designing a secure protocol goes far beyond the scope of this thesis. A protocol fitting the requirements of the project was selected, analyzed and a prototype was implemented.

When analyzing the methodology of the thesis critically, a couple of possible weak points can be identified. Small parts of the fundamentals section were only based on few sources, which leaves a possibility of incorrect and unverified information. However, the scientific quality and relevance of sources was always ensured and no contradicting information was encountered, making this risk small. When looking at the final selection of the protocol, it is possible, that the limited knowledge about cryptography and security has lead to a false evaluation of certain protocols. Additionally, the defined search string could have excluded relevant sources, opening the possibility, that the best protocol was not found. Looking at the implementation, the security aspects of the prototype were not closely examined.

To conclude the thesis, another look at the research questions defined in section 1.2 is necessary to evaluate the success of the thesis. *RQ1* is only partially answered by the thesis. Results almost exclusively focused on the protocol. Other requirements for a full infrastructure, like the database implementation, the hardware required for supporting RFID, or the requirements of access control systems, were only partially considered.

RQ2 required the implementation of a prototype and a demonstration of its functionality. This goal was achieved as the selected protocol was implemented using python and evaluated using a set of tests. However, the prototype has its limitations. It has not been extensively tested to ensure resilience against all possible types of attacks. Additionally, hardware constraints, like a limited number of logic gates on RFID tags and the imperfect PUF responses in the real world, have not been considered.

Appendix

Appendix 1: Guidelines for Literature Reviews by Brocke et al.

This is a summary of the guidelines for literature reviews in the information systems field, proposed by Brocke et al. in [3]. The methodology for the literature review in this thesis is loosely based on these five-steps.

1. Analyze the scope and purpose of the review by using the taxonomy on literature reviews proposed in [31]. This acts as a basis for further steps and helps to keep the review focused on a specific goal

2. Establish concepts about the topic, that are already widely known and recognized in the literature. This is best done using publications like existing review articles, textbooks or encyclopedias.

3. After having gained a good understanding of the concepts and fundamentals of the topic, search terms should be devised, that can be used for the literature searching process. These search terms should then be applied in the following way:

    a) Find all journals, which publish articles related to the topic.

    b) Find all databases, which include articles from those journals.

    c) Query these databases using keywords and search terms devised from the previously acquired knowledge. Estimate the relevance of articles by analyzing title, abstract, number of citations and recency. Include all relevant articles in the review.

    d) Use backward and forward search to find additional relevant literature.

4. Analyze all collected literature and synthesize new knowledge from it. This process is enabled by using a concept matrix (proposed in [32]). Make sure to focus on the focus and goal specified in *step one*.

5. Create a research agenda that outlines, which questions have been answered by the existing research and which topics future research should focus on.

Appendix 2: Exact Search Strings used in Literature Review

IEEE Xplore

("Document Title":physical unclonable function OR "Document Title":puf OR "Document Title":physically unclonable function) AND ("Document Title":authentication OR "Document Title":challenge response OR "Document Title":challenge-response) AND ("Document Title":protocol)

ACM Digital Library

[[Title: physical unclonable function] OR [Title: puf] OR [Title: physically unclonable function]] AND [[Title: authentication] OR [Title: challenge response] OR [Title: challenge-response]] AND [Title: protocol]

Google Scholar

intitle:protocol AND (intitle:authentication OR intitle:challenge-response OR intitle:challenge OR intitle:response) AND (intitle:puf OR intitle:physically unclonable function OR intitle:physical unclonable function)

Appendix 3: Source Code for Implementation

Appendix 3.1: Pipfile

This pipfile can be used to easily set up the appropriate environment using pipenv.

```
[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[packages]
pypuf = "*"
numpy = "*"
bitstring = "*"

[dev-packages]
ipykernel = "*"
autopep8 = "*"
pytest = "*"

[requires]
python_version = "3.10"
```

Appendix 3.2: support.py

```
import hashlib
import random
import numpy as np
from bitstring import BitArray

CHALLENGE_LENGTH = 64
RESPONSE_LENGTH = 16  # number of evaluations of arbiter puf per response
HASH_FUNCTION = hashlib.sha256


def array_to_bitstring(np_array: np.ndarray) -> BitArray:
    """
    Flattens a nested array consisting of a challenge or response for an Arbiter PUF
    Arbiter PUF output -1 is interpreted as binary 0, output 1 is interpreted as binary 1)
```

```
    """

    np_array = np_array.flatten()
    np_array[np_array == -1] = 0
    return BitArray(np_array)


def bitstring_to_challenge(b: BitArray, n: int = CHALLENGE_LENGTH, N: int = RESPONSE_LENGTH) -> np.ndarray:
    """Converts a BitArray to a numpy array of shape (N, n)"""
    flat = []
    shaped = [None]*N
    for c in b.bin:
        flat.append(c)
    for i in range(0, N):
        shaped[i] = flat[n*(i):n*(i+1)]
    shaped = np.array(shaped, dtype='int8')
    shaped[shaped == 0] = -1
    return shaped


def xor(n: BitArray, m: BitArray) -> BitArray:
    """Takes two BitArrays of length n and applies XOR logic to each bit"""
    if len(n) != len(m):
        raise ValueError("XOR inputs of different lengths are not supported!")
    return BitArray(bin=bin(int(n.bin, 2) ^ int(m.bin, 2))[2:].zfill(len(n)))


def concat(n: BitArray, m: BitArray) -> BitArray:
    """Concatinates two BitArrays"""
    return n + m


def new_hash(v) -> HASH_FUNCTION:
    """Returns a new hash of value v"""
    h = HASH_FUNCTION()
    h.update(v)
    return h


def verify_hash(h1, h2):
    """Checks if hashes h1 and h2 are matching"""
```

```
    try:
        assert h1.hexdigest() == h2.hexdigest()
    except:
        raise ValueError(
            f"Hash could not be verified!\nHash 1: {h1.hexdigest()}\
nHash 2: {h2.hexdigest()}")


def generate_random_bitstring(l: int) -> BitArray:
    """Generates a random BitArray of length l"""
    n = [0]  # first bit is always zero to prevent overflows on
addition
    for i in range(0, l-1):
        n.append(random.randint(0, 1))
    n = BitArray(n)
    return n


def add_int_to_bitstring(b: BitArray, i: int):
    """
    Increases binary value of BitArray by bin(i)
    If resulting BitArray is too large to fit in len(b), an error is
thrown
    """
    return BitArray(int=b.int+i, length=len(b))


def compute_next_challenge(n, m, c):
    """
    Computes the next challenge based on a concatination of two random
numbers and the previous challenge.
    This is done by hashing the resulting bitstring and multiplying it
until the required challenge length is reached.
    """
    h = new_hash(bytes(concat(concat(n, m), c).hex, 'utf-8'))
    h = BitArray(h.digest())
    s = CHALLENGE_LENGTH * RESPONSE_LENGTH
    while len(h) < s:
        h = h + h
    return h[0:s]


class AuthMessage:
    """Base class for authentication messages"""
    pass
```

```
96
97
98  class MInit(AuthMessage):
99      pass
100
101
102 class M1(AuthMessage):
103     pass
104
105
106 class M2(AuthMessage):
107     def __init__(self, sid) -> None:
108         super().__init__()
109         self.sid = sid
110
111
112 class M3(AuthMessage):
113     def __init__(self, challenge, n_mod, auth1) -> None:
114         super().__init__()
115         self.challenge = challenge
116         self.n_mod = n_mod
117         self.auth1 = auth1
118
119
120 class M4(AuthMessage):
121     def __init__(self, resp_mod, m_mod, auth2) -> None:
122         super().__init__()
123         self.resp_mod = resp_mod
124         self.m_mod = m_mod
125         self.auth2 = auth2
126
127
128 class M5(AuthMessage):
129     def __init__(self, auth3) -> None:
130         super().__init__()
131         self.auth3 = auth3
```

Appendix 3.3: server.py

```
1 import tag
2 from pypuf.io import random_inputs
3 from uuid import uuid4
4 from bitstring import BitArray
5 from support import *
6
```

```
7
8  class Server:
9      # Singleton pattern allows only one server to exist
10     def __new__(cls):
11         if not hasattr(cls, 'instance'):
12             cls.instance = super(Server, cls).__new__(cls)
13         return cls.instance
14
15     def __init__(self) -> None:
16         self.persistent_storage = {}
17         self.active_sessions = {}
18         pass
19
20     class AuthSession():
21         def __init__(self) -> None:
22             self.sid: str = None
23             self.next_sid: str = None
24             self.challenge: BitArray = None
25             self.next_challenge: BitArray = None
26             self.response: BitArray = None
27             self.next_resp: BitArray = None
28             self.n: BitArray = None
29             self.n_mod: BitArray = None
30             self.resp_mod: BitArray = None
31             self.m: BitArray = None
32             self.m_mod: BitArray = None
33             self.m_plus_1: BitArray = None
34             self.m_plus_2: BitArray = None
35             self.auth1: HASH_FUNCTION = None
36             self.auth2_tag: HASH_FUNCTION = None
37             self.auth2: HASH_FUNCTION = None
38             self.auth3: HASH_FUNCTION = None
39             pass
40
41     def enroll_tag(self, tag: tag.Tag, challenge_seed: int) -> None:
42         challenge = self.generate_random_challenge(challenge_seed)
43         response = self.challenge_tag(tag, challenge)
44         sid = self.generate_initial_random_session_identity()
45         self.store_identity(sid, challenge, response)
46         tag.set_sid(sid)
47         print("Enrolled Tag with SID ", sid)
48
49     def generate_random_challenge(self, seed: int) -> BitArray:
50         return array_to_bitstring(random_inputs(
51             n=CHALLENGE_LENGTH,
```

```
52              N=RESPONSE_LENGTH,
53              seed=seed))
54
55      def challenge_tag(self, tag: tag.Tag, challenge: BitArray) ->
    BitArray:
56          return tag.generate_response(challenge)
57
58      def generate_initial_random_session_identity(self) -> str:
59          h = new_hash(bytes(str(uuid4()), 'utf-8'))
60          return h.hexdigest()
61
62      def store_identity(self, sid: str, challenge: BitArray, resp:
    BitArray) -> None:
63          self.persistent_storage[sid] = {
64              "c": challenge,
65              "r": resp
66          }
67
68      def remove_identity(self, sid: str) -> None:
69          del self.persistent_storage[sid]
70
71      def handle_m2(self, m, session: AuthSession) -> M3:
72          session.sid = m.sid
73          try:
74              session.challenge = self.persistent_storage[session.sid]["c
    "]
75              session.response = self.persistent_storage[session.sid]["r"
    ]
76          except KeyError:
77              raise ValueError(
78                  "Invalid SID sent by tag.\nTag might be invalid.
    Authentication terminated.")
79
80          session.n = generate_random_bitstring(RESPONSE_LENGTH)
81          session.n_mod = xor(session.response, session.n)
82          session.auth1 = new_hash(
83              bytes(concat(session.n, session.response).hex, 'utf-8'))
84          return M3(session.challenge, session.n_mod, session.auth1)
85
86      def handle_m4(self, message: M4, session: AuthSession) -> M5:
87          session.resp_mod = message.resp_mod
88          session.m_mod = message.m_mod
89          auth2_tag = message.auth2
90
91          session.next_resp = xor(session.resp_mod, session.n)
```

```
92          session.m = xor(session.m_mod, session.n)
93          session.m_plus_1 = add_int_to_bitstring(session.m, 1)
94
95          session.auth2 = new_hash(bytes(concat(
96              concat(session.n, session.m_plus_1), session.next_resp).hex
    , 'utf-8'))
97          verify_hash(auth2_tag, session.auth2)
98
99          session.next_challenge = compute_next_challenge(
100             session.n, session.m, session.challenge)
101         session.m_plus_2 = add_int_to_bitstring(session.m, 2)
102         session.next_sid = new_hash(bytes(concat(concat(
103             session.n, session.m_plus_2), BitArray(hex=session.sid)).
    hex, 'utf-8')).hexdigest()
104         session.auth3 = new_hash(
105             bytes(concat(session.n, session.m_plus_2).hex, 'utf-8'))
106
107         self.store_identity(
108             session.next_sid, session.next_challenge, session.next_resp
    )
109
110         print("Authentication of tag successful on server side!")
111         print("Identity for next auth session: ", session.next_sid)
112
113         return M5(session.auth3)
114
115     def handle_message(self, m: AuthMessage, reader_id: int) -> None:
116         if reader_id in self.active_sessions:
117             session = self.active_sessions[reader_id]
118         else:
119             session = self.AuthSession()
120             self.active_sessions[reader_id] = session
121             session.expected_message = MInit
122         if type(m) != session.expected_message:
123             raise TypeError(
124                 f"Expected {session.expected_message}, got {type(m)}")
125         elif (type(m)) == MInit:
126             session.expected_message = M2
127             return M1()
128         elif (type(m)) == M2:
129             session.expected_message = M4
130             return self.handle_m2(m, session)
131         elif (type(m)) == M4:
132             m = self.handle_m4(m, session)
133             del self.active_sessions[reader_id]
```

```
134
135             print(f"Tag at reader {reader_id} successfully
    authenticated.")
136             print("New session identity written to persistent storage\n
    ")
137             return m
138         else:
139             raise TypeError("Unknown Message Type")
```

Appendix 3.4: tag.py

```
1  from bitstring import BitArray
2  from pypuf.simulation import ArbiterPUF
3  from support import *
4  from typing import List
5
6
7  class TagFactory:
8      def __new__(cls):
9          if not hasattr(cls, 'instance'):
10             cls.instance = super(TagFactory, cls).__new__(cls)
11         return cls.instance
12
13     def __init__(self) -> None:
14         pass
15
16     # ensure uniqueness of PUFs
17     total_produced_tags = 0
18
19     def manufacture_n_tags(self, n) -> List['Tag']:
20         tags = []
21         for i in range(0, n):
22             seed = self.total_produced_tags
23             puf = ArbiterPUF(n=CHALLENGE_LENGTH, seed=seed)
24             tags.append(Tag(puf))
25             print(f"Tag with seed {seed} manufactured")
26             self.total_produced_tags += 1
27
28         return tags
29
30
31 class Tag:
32     def __init__(self, puf_module: ArbiterPUF) -> None:
33         self.puf = puf_module
34         self.sid = None
```

```
35          self.enrolled = False
36
37      def generate_response(self, challenge: BitArray) -> BitArray:
38          challenge = bitstring_to_challenge(challenge)
39          return array_to_bitstring(self.puf.eval(challenge))
40
41      def set_sid(self, sid: str) -> None:
42          self.sid = sid
43
44      def handle_message(self, m: AuthMessage) -> None:
45          if (type(m)) == M1:
46              return self.handle_m1()
47          elif (type(m)) == M3:
48              return self.handle_m3(m)
49          elif (type(m)) == M5:
50              return self.handle_m3(m)
51          else:
52              raise TypeError
53
54      def handle_m1(self) -> M2:
55          return M2(self.sid)
56
57      def handle_m3(self, message: M3) -> M4:
58          challenge = message.challenge
59          n_mod = message.n_mod
60          auth1 = message.auth1
61          resp = self.generate_response(challenge)
62          n = xor(n_mod, resp)
63          self.n = n
64          verify_hash(auth1, new_hash(
65              bytes(concat(n, resp).hex, 'utf-8')))
66          m = generate_random_bitstring(RESPONSE_LENGTH)
67          next_challenge = compute_next_challenge(n, m, challenge)
68          next_response = self.generate_response(next_challenge)
69          resp_mod = xor(next_response, n)
70          m_mod = xor(m, n)
71          m_plus_1 = add_int_to_bitstring(m, 1)
72          self.m_plus_1 = m_plus_1
73          auth2 = new_hash(bytes(concat(
74              concat(n, m_plus_1), next_response).hex, 'utf-8'))
75          return M4(resp_mod, m_mod, auth2)
76
77      def handle_m5(self, m: M5) -> bool:
78          m_plus_2 = add_int_to_bitstring(self.m_plus_1, 1)
79          auth3_server = m.auth3
```

```
        auth3_comp = new_hash(
            bytes(concat(self.n, m_plus_2).hex, 'utf-8'))
        verify_hash(auth3_server, auth3_comp)
        next_sid = new_hash(bytes(concat(concat(
            self.n, m_plus_2), BitArray(hex=self.sid)).hex, 'utf-8')).
    hexdigest()
        self.sid = next_sid

        print("Authentication on tag side successful:")
        print("New session identity provisioned: ", self.sid)

        return True
```

Appendix 3.5: test_auth.py

```
from server import Server
from tag import TagFactory
import pytest
from support import *
from pypuf.simulation import ArbiterPUF


def test_auth_100_tags():
    """Different tags should be able to authenticate in sequence."""
    factory = TagFactory()
    server = Server()
    tags = factory.manufacture_n_tags(100)

    for seed, tag in enumerate(tags):
        server.enroll_tag(tag, seed)

    for tag in tags:
        terminal = 0
        m1 = server.handle_message(MInit(), terminal)
        m2 = tag.handle_message(m1)
        m3 = server.handle_message(m2, terminal)
        m4 = tag.handle_message(m3)
        m5 = server.handle_message(m4, terminal)
        success = tag.handle_m5(m5)
        assert success


def test_auth_100_tags_parallel():
    """Many tags should be able to authenticate at the same time on
    different readers.
```

```
    In reality this would be limited by the number of deployed readers.
    """
    factory = TagFactory()
    server = Server()
    tags = factory.manufacture_n_tags(100)

    for seed, tag in enumerate(tags):
        server.enroll_tag(tag, seed)

    messages = {}
    for reader, tag in enumerate(tags):
        messages[reader] = {}
        m1 = server.handle_message(MInit(), reader)
        m2 = tag.handle_message(m1)
        messages[reader]['m2'] = m2

    for reader, tag in enumerate(tags):
        m2 = messages[reader]['m2']
        m3 = server.handle_message(m2, reader)
        m4 = tag.handle_message(m3)
        messages[reader]['m4'] = m4

    for reader, tag in enumerate(tags):
        m4 = messages[reader]['m4']
        m5 = server.handle_message(m4, reader)
        success = tag.handle_m5(m5)
        assert success


def test_auth_one_tag_100_times():
    """One tag should be able to authenticate an unlimited number of
    times in sequence."""
    factory = TagFactory()
    server = Server()
    tag = factory.manufacture_n_tags(1)[0]
    server.enroll_tag(tag, 0)

    for i in range(0, 100):
        m1 = server.handle_message(MInit(), 0)
        m2 = tag.handle_message(m1)
        m3 = server.handle_message(m2, 0)
        m4 = tag.handle_message(m3)
        m5 = server.handle_message(m4, 0)
        success = tag.handle_m5(m5)
        assert success
```

```
73
74
75  def test_switch_tag_during_authentication():
76      """If the tag is switched during the authentication phase,
77      authentication should be terminated"""
78      factory = TagFactory()
79      server = Server()
80      tags = factory.manufacture_n_tags(2)
81
82      for seed, tag in enumerate(tags):
83          server.enroll_tag(tag, seed)
84
85      tag1 = tags[0]
86      tag2 = tags[1]
87
88      m1 = server.handle_message(MInit(), 0)
89      m2 = tag1.handle_message(m1)
90      with pytest.raises(ValueError):
91          m3 = server.handle_message(m2, 0)
92          m4 = tag2.handle_message(m3)
93          m5 = server.handle_message(m4, 0)
94          success = tag.handle_m5(m5)
95
96
97  def test_tag_with_invalid_sid():
98      """If the sid of the tag is changed after enrolling it,
99      a ValueError should be thrown on the server side when handling M2.
        """
100     factory = TagFactory()
101     server = Server()
102     tag = factory.manufacture_n_tags(1)[0]
103     server.enroll_tag(tag, 0)
104
105     tag.sid = "changed_sid"
106
107     m1 = server.handle_message(MInit(), 0)
108     m2 = tag.handle_message(m1)
109     with pytest.raises(ValueError):
110         server.handle_message(m2, 0)
111
112
113 def test_tag_with_invalid_puf():
114     """An enrolled tag with a PUF module, that was modified by an
        attacker,
115     should not be able to authenticate."""
```

```
116     factory = TagFactory()
117     server = Server()
118     tag = factory.manufacture_n_tags(1)[0]
119     server.enroll_tag(tag, 0)
120 
121     attacker_puf = ArbiterPUF(n=CHALLENGE_LENGTH, seed=1)
122     tag.puf = attacker_puf
123 
124     m1 = server.handle_message(MInit(), 0)
125     m2 = tag.handle_message(m1)
126     m3 = server.handle_message(m2, 0)
127     with pytest.raises(ValueError):
128         tag.handle_message(m3)
```

Bibliography

[1] S. R. Basavala, N. Kumar, and A. Agarrwal, 'Authentication: An overview, its types and integration with web and mobile applications', in *2012 2nd IEEE International Conference on Parallel, Distributed and Grid Computing*, IEEE, 2012-12. DOI: 10.1109/pdgc.2012.6449853.

[2] Y. Gao, S. F. Al-Sarawi, and D. Abbott, 'Physical unclonable functions', *Nature Electronics*, vol. 3, no. 2, pp. 81–91, 2020-02. DOI: 10.1038/s41928-020-0372-5.

[3] J. v. Brocke, A. Simons, B. Niehaves, K. Riemer, R. Plattfaut, and A. Cleven, 'Reconstructing the giant: On the importance of rigour in documenting the literature search process', 2009-06.

[4] R. Maes, *Physically Unclonable Functions*. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-41395-7.

[5] U. Rührmair, H. Busch, and S. Katzenbeisser, 'Strong PUFs: Models, constructions, and security proofs', in *Information Security and Cryptography*, Springer Berlin Heidelberg, 2010, pp. 79–96. DOI: 10.1007/978-3-642-14452-3_4.

[6] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, 'FPGA intrinsic PUFs and their use for IP protection', in *Cryptographic Hardware and Embedded Systems - CHES 2007*, Springer Berlin Heidelberg, pp. 63–80. DOI: 10.1007/978-3-540-74735-2_5.

[7] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld, 'Physical one-way functions', *Science*, vol. 297, no. 5589, pp. 2026–2030, 2002-09. DOI: 10.1126/science.1074376.

[8] T. McGrath, I. E. Bagci, Z. M. Wang, U. Roedig, and R. J. Young, 'A PUF taxonomy', *Applied Physics Reviews*, vol. 6, no. 1, p. 011 303, 2019-03. DOI: 10.1063/1.5079407.

[9] A. Aysu, N. F. Ghalaty, Z. Franklin, M. P. Yali, and P. Schaumont, 'Digital fingerprints for low-cost platforms using MEMS sensors', in *Proceedings of the Workshop on Embedded Systems Security - WESS '13*, ACM Press, 2013. DOI: 10.1145/2527317.2527319.

[10] J. Lee, D. Lim, B. Gassend, G. Suh, M. van Dijk, and S. Devadas, 'A technique to build a secret key in integrated circuits for identification and authentication applications', in *2004 Symposium on VLSI Circuits. Digest of Technical Papers (IEEE Cat. No.04CH37525)*, Widerkehr and Associates. DOI: 10.1109/vlsic.2004.1346548.

[11] H. Shahoei and J. Yao, *Delay lines*, 2014-12. DOI: 10.1002/047134608x.w8234.

[12] C. Herder, M.-D. Yu, F. Koushanfar, and S. Devadas, 'Physical unclonable functions and applications: A tutorial', *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1126–1141, 2014-08. DOI: 10.1109/jproc.2014.2320516.

[13] J. Singh, S. P. Mohanty, and D. K. Pradhan, *Robust SRAM Designs and Analysis*. Springer New York, 2013. DOI: 10.1007/978-1-4614-0818-5.

[14] A. Bhavnagarwala, X. Tang, and J. Meindl, 'The impact of intrinsic device fluctuations on CMOS SRAM cell stability', *IEEE Journal of Solid-State Circuits*, vol. 36, no. 4, pp. 658–665, 2001-04. DOI: 10.1109/4.913744.

[15] S. Devadas, E. Suh, S. Paral, R. Sowell, T. Ziola, and V. Khandelwal, 'Design and implementation of PUF-based "unclonable" RFID ICs for anti-counterfeiting and security applications', in *2008 IEEE International Conference on RFID*, IEEE, 2008-04. DOI: 10.1109/rfid.2008.4519377.

[16] M. Majzoobi, M. Rostami, F. Koushanfar, D. S. Wallach, and S. Devadas, 'Slender PUF protocol: A lightweight, robust, and secure authentication by substring matching', in *2012 IEEE Symposium on Security and Privacy Workshops*, IEEE, 2012-05. DOI: 10.1109/spw.2012.30.

[17] S. Katzenbeisser and A. Schaller, 'Physical unclonable functions', *Datenschutz und Datensicherheit - DuD*, vol. 36, no. 12, pp. 881–885, 2012-11. DOI: 10.1007/s11623-012-0295-z.

[18] P. Gope, J. Lee, and T. Q. S. Quek, 'Lightweight and practical anonymous authentication protocol for RFID systems using physically unclonable functions', *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 11, pp. 2831–2843, 2018-11. DOI: 10.1109/tifs.2018.2832849.

[19] F. Zhu, P. Li, H. Xu, and R. Wang, 'A lightweight RFID mutual authentication protocol with PUF', *Sensors*, vol. 19, no. 13, p. 2957, 2019-07. DOI: 10.3390/s19132957.

[20] C. Hristea and F. L. Ţiplea, 'A PUF-based destructive private mutual authentication RFID protocol', in *Innovative Security Solutions for Information Technology and Communications*, Springer International Publishing, 2019, pp. 331–343. DOI: 10.1007/978-3-030-12942-2_25.

[21] N. Wisiol, C. Gräbnitz, C. Mühl, B. Zengin, T. Soroceanu, N. Pirnay, K. T. Mursi, and A. Baliuka, *pypuf: Cryptanalysis of Physically Unclonable Functions*, version v2, 2021. DOI: 10.5281/zenodo.3901410. [Online]. Available: https://doi.org/10.5281/zenodo.3901410.

[22] Y. Zhang, B. Li, B. Liu, Y. Hu, and H. Zheng, 'A privacy-aware PUFs-based multi-server authentication protocol in cloud-edge IoT systems using blockchain', *IEEE Internet of Things Journal*, vol. 8, no. 18, pp. 13 958–13 974, 2021-09. DOI: 10.1109/jiot.2021.3068410.

[23] S. Yu and Y. Park, 'A robust authentication protocol for wireless medical sensor networks using blockchain and physically unclonable functions', *IEEE Internet of Things Journal*, vol. 9, no. 20, pp. 20 214–20 228, 2022-10. DOI: 10.1109/jiot.2022.3171791.

[24] U. Chatterjee, V. Govindan, R. Sadhukhan, D. Mukhopadhyay, R. S. Chakraborty, D. Mahata, and M. M. Prabhu, 'Building PUF based authentication and key exchange protocol for IoT without explicit CRPs in verifier database', *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 3, pp. 424–437, 2019-05. DOI: 10.1109/tdsc.2018.2832201.

[25] A. Braeken, 'PUF based authentication protocol for IoT', *Symmetry*, vol. 10, no. 8, p. 352, 2018-08. DOI: 10.3390/sym10080352.

[26] G. Bansal, N. Naren, V. Chamola, B. Sikdar, N. Kumar, and M. Guizani, 'Lightweight mutual authentication protocol for v2g using physical unclonable function', *IEEE Transactions on Vehicular Technology*, vol. 69, no. 7, pp. 7234–7246, 2020-07. DOI: 10.1109/tvt.2020.2976960.

[27] Q. Jiang, X. Zhang, N. Zhang, Y. Tian, X. Ma, and J. Ma, 'Two-factor authentication protocol using physical unclonable function for IoV', in *2019 IEEE/CIC International Conference on Communications in China (ICCC)*, IEEE, 2019-08. DOI: 10.1109/iccchina.2019.8855828.

[28] P. Gope, O. Millwood, and B. Sikdar, 'A scalable protocol level approach to prevent machine learning attacks on physically unclonable function based authentication mechanisms for internet of medical things', *IEEE Transactions on Industrial Informatics*, vol. 18, no. 3, pp. 1971–1980, 2022-03. DOI: 10.1109/tii.2021.3096048.

[29] S. Chatterjee and P. Sarkar, *Identity-Based Encryption*. Springer US, 2011. DOI: 10.1007/978-1-4419-9383-0.

[30] Y. Dodis, L. Reyzin, and A. Smith, 'Fuzzy extractors: How to generate strong keys from biometrics and other noisy data', in *Advances in Cryptology - EUROCRYPT 2004*, Springer Berlin Heidelberg, 2004, pp. 523–540. DOI: 10.1007/978-3-540-24676-3_31.

[31] H. M. Cooper, 'Organizing knowledge syntheses: A taxonomy of literature reviews', *Knowledge in Society*, vol. 1, no. 1, pp. 104–126, 1988-03. DOI: 10.1007/bf03177550.

[32] J. Webster and R. T. Watson, 'Analyzing the past to prepare for the future: Writing a literature review', *MIS Quarterly*, vol. 26, no. 2, pp. xiii–xxiii, 2002, ISSN: 02767783. [Online]. Available: http://www.jstor.org/stable/4132319 (visited on 2022-11-04).

Declaration in lieu of oath

I hereby declare that I produced the submitted paper with no assistance from any other party and without the use of any unauthorized aids and, in particular, that I have marked as quotations all passages which are reproduced verbatim or near-verbatim from publications. Also, I declare that this paper has never been submitted before to any examination board in either its present form or in any other similar version. I herewith agree that this paper may be published. I herewith consent that this paper may be uploaded to the server of external contractors for the purpose of submitting it to the contractors' plagiarism detection systems. Uploading this paper for the purpose of submitting it to plagiarism detection systems is not a form of publication.

Düsseldorf, 11.11.2022

(Location, Date)

(handwritten signature)