FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Use of physically unclonable function to secure wireless communication
Student:	František Kovář
Supervisor:	Ing. Jiří Buček, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2019/20

Instructions

1. Study the topic of physically unclonable functions (PUFs).
2. Design and implement a security device consisting of a WiFi-enabled microcontroller module [1] and an FPGA board containing a PUF [2]. The microcontroller will be connected to the FPGA using an SPI bus and will communicate with other devices over WiFi using TCP/IP.
3. Design and implement a system consisting of an authentication authority (on a PC) and two or more security devices described above. Implement an authentication protocol under the guidance of the supervisor.
4. The system will support the following functions:
* Enrollment of the security devices to the authentication authority.
* Authentication of the security devices by the authentication authority using PUF.
5. Perform automated testing of the system and its functionality in order to measure reliability parameters, such as false acceptance ratio and false rejection ratio.

References

[1] Wemos D1 mini or similar, https://wiki.wemos.cc/products:d1:d1_mini
[2] Kodýtek, F. and Lórencz, R.: A design of ring oscillator based PUF on FPGA. In *Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2015 IEEE 18th International Symposium on* (pp. 37-42). IEEE.

prof. Ing. Pavel Tvrdík, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 14, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Use of physically unclonable function to secure wireless communication

František Kovář

Department of Information Security
Supervisor: Ing. Jiří Buček, Ph.D.

June 4, 2020

Acknowledgements

I would like to thank everyone who supported me during my work on this thesis. First of all, I would like to thank my supervisor Ing. Jiří Buček, PhD. for his incredible patience and every valuable advice, that helped me to finish my thesis. An exclusive thanks belong to my sister, Bc. Michaela Kovářová, for her endless support and unbelievable motivation during all my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on June 4, 2020 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Kovář, František. *Use of physically unclonable function to secure wireless communication.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

V této bakalářské práci jsme zdokumentovali návrh a implementaci bezpečnostních zařízení, které využívají fyzicky neklonovatlnou funkci k zabezpečení bezdrátové komunikace. Implementace proběhla v jazyce C++ a v jazyce VHDL na zařízeních Wemos D1 mini s ESP8266 wifi čipem a na FPGA Digilent Basys 2. Wemos D1 mini komunikuje přes s FPGA pomocí sériové linky a tím získáva data k prokázání své identity. Zařízení bylo otestováno na lokální wifi síti a úspěšnost takovéto zabezpečené komunikace dosahovala 96,7% kvůli nestabilitě fyzicky neklonovatelné funkce.

Klíčová slova PUF, fyzicky neklonovatelná funkce, FPGA, IoT, Wemos D1 mini, zabezpečení, WiFi komunikace, TCP/IP protokol, samoopravný kód

Abstract

In this bachelor's thesis, we documented the design and the implementation of security devices that use a physically unclonable function to secure wireless communication. The implementation is in C ++ language and VHDL language on Wemos D1 mini device with ESP8266 WiFi chip and on FPGA Digilent Basys 2. Wemos D1 mini communicates with FPGA via serial line and thus obtains data to prove its identity. The device was tested on a local

WiFi network, and the success of such secure communication reached 96.7% due to the instability of a physically unclonable function.

Keywords PUF, physically unclonable function, FPGA, IoT, Wemos D1 mini, security, WiFi communication, TCP/IP protocol, error correction code

Contents

List of Figures

List of Tables

Introduction

The topic of this bachelor thesis is to secure wireless communication with the use of a physically unclonable function. The goal of the theoretical part of this thesis is to study the physically unclonable functions and other tools that can be used to secure the communication.

The goals of the practical part are to design and create a security device, which will be able to communicate over the TCP/IP protocol with other devices and will use its physical properties to secure the connection. We furthermore need an authentication protocol and authentication authority, so the devices can use it as a trusted third party to authorize themselves and at the end share a secret message.

This thesis is divided into three chapters. In the first chapter, *Analysis*, we focus on the theoretical background that we need to design and implement all our goals. In the second chapter, *Design and implementation*, we look at the design and subsequent implementation individual building blocks of our security device, protocol and authentication authority. In this part, we entirely use the knowledge gained from the first chapter. In the last chapter, *Measurements*, we have some measurements that were needed to do during the thesis, to find out some implementation details and the reliability of the protocol used.

Chapter 1

Analysis

1.1 Field programmable gate array

An FPGA is an integrated circuit, which is designed to be reconfigured by users after manufacturing. People typically use HDL, such as Verilog [6] or VHDL [7], to describe hardware behaviour. An FPGA contains a set of logic blocks and a hierarchy of interconnects which are configurable. Modern FPGAs also consist of memory elements, such as simple flip-flops or complete memory blocks [8]. A considerable advantage of FPGAs is their flexibility, reprogramming, and cost-effectiveness. The FPGA is often used to create prototypes so that the designs can be fully debugged, tested, and updated before manufacturing.

In this thesis, we only consider the device Digilent Basys 2 with Xilinx Spartan-3E [9] and the software needed for development ISE Design Suite: WebPACK Edition 14.7 [10] of company Xilinx.

1.2 Physically unclonable function

Physical Unclonable Function (PUF) generates a digital fingerprint using a device's characteristics which were created during the manufacturing process. Therefore, identical integrated circuits from the same fabrication facility, using the same manufacturing process can generate different challenge-response pair, as there will always be a slight difference in the manufacturing process. As stated in [8], secret keys are usually stored in a nonvolatile memory (NVM), which can be hard to secure. Since a PUF can produce a cryptographic key or digital fingerprint, which should be unique from device to device, it can eliminate the need of storing such a key in an NVM as it is always generated on the fly. A PUF, unlike an NVM, is resistant to a physical attack, as the attack, even slightly, changes the small differences that made the PUF and could drastically change the PUF's response.

We divide PUFs into two groups. A *weak* PUF and a *strong* PUF. This is determined on the challenge-response pair (CRP)-space. A weak PUF has its CRP-space is extremely small, an example of a weak PUF is an SRAM PUF. In contrast, a strong PUF should handle a huge number of challenges. At best the CRP-space should grow exponentially with the length of the challenge itself [8]. An example of a strong PUF is the Arbiter PUF and Ring oscillator PUF.

The PUFs can be used for secret key generation. Still, this process should be provided with an ECC since even with a slight change of the output, the resulting cryptographic application and its messages would be corrupted. Strong PUFs are excellent candidates to provide device authentication based on the hardware, and the response can be considered to be its signature or fingerprint.

1.2.1 Arbiter PUF

As stated in [11], this type of PUF is composed of delay paths, multiple switches that can change the path, and an arbiter located at the end of the delay paths. The PUF takes as an input 64bit long challenge and produces a 1bit response as an output. The resulting bit is decided as from what path the signal came first. To achieve the maximum variation of PUF responses the delay paths must be placed as symmetrically as possible, to minimize the delay differences.

1.2.2 Ring Oscillator PUF

The basic idea behind Ring Oscillator PUF (RO PUF) [12] is that given two same symmetrical oscillators, the frequency of the two oscillators will differ due to the manufacturing process, and two counters. Comparison of the counter, which oscillator gave a bigger number of cycles, determines a state of the response as a single bit.

The RO PUF [5] uses two oscillators, which are selected by the challenge, composed of 1 NAND gate and four inverters. These two oscillators are connected to their respective counters, and the response is the value of the counter that did not overflow a specified threshold. Using this approach, we not only get one bit of response but multiple bit response. The response must be processed appropriately and is documented in [5].

1.2.3 SRAM PUF

The principle for SRAM PUF is based on the content of the uninitialized SRAM memory after startup as its behaviour is unpredictable. The SRAM snapshot is taken to authenticate the device. An SRAM cell consists of six transistors as where four internal behave like a small oscillator [13]. As some of

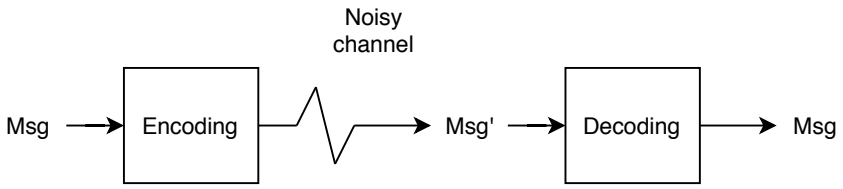

Figure 1.1: Example usage of encoding and decoding of sent message over a noisy channel.[1]

the SRAM cells after startup are stable, since the strength of a cross-coupled inverter is different for each SRAM bitcell[14], they will have a preferred state. Due to noise, temperature and voltage fluctuations, the value can be different. Thanks to stable and even the unstable SRAM cells we can identify the device. "*The chip is identified if: a) the Hamming distance between all values in the database (except one) and the request value is bigger than 39%, and b) there is one entry in the database where the Hamming distance between the entry and the request ID is below 25%.*"[13].

1.2.4 Butterfly PUF

According to [5], it is not always possible to implement an SRAM PUF on the FPGA since they usually initialize memory to some default values. The concept of the Butterfly PUF (BPUF) [15] is based on the idea of creating structures with the FPGA matrix, which behave similarly to an SRAM cell during the startup phase. Its benefits versus an SRAM PUF is that we do not need to power on and off the device to generate the output. A BPUF cell is a cross-coupled circuit which can be brought to an unstable state before allowing it to settle to one of the two stable states that are possible. It is necessary to have as symmetrical circuit as possible, to ensure the best results of the BPUF.

1.3 Error correction codes

"*Communication of a piece of information begins with a sender writing it, goes on with its transmission through a channel and ends with the reconstruction of the message by the recipient.*"[1].

When we transfer data from one device to another, Error Correction Codes (ECCs) are important for detection and if possible correction of occurred errors on the data. An error can emerge in different situations. The message is modified by a third party or some bits are modified over a noisy channel. We will discuss only the second option, that some bits are distorted by an accident while transferring over a noisy channel. ECCs aim to fix or at least detect errors in a message using redundancy data and simultaneously

ensure that the initial size of the data doesn't change much. According to used ECC it can fix and detect limited amount of errors depending on the size of redundancy data it uses.

1.4 Random number generator

In this section we will discuss one of the main tools in cryptography. Randomness, random numbers and their generation using true random number generator (TRNG) or pseudo random number generator (PRNG). The next figure fig. 1.2 shows an example of such generation.

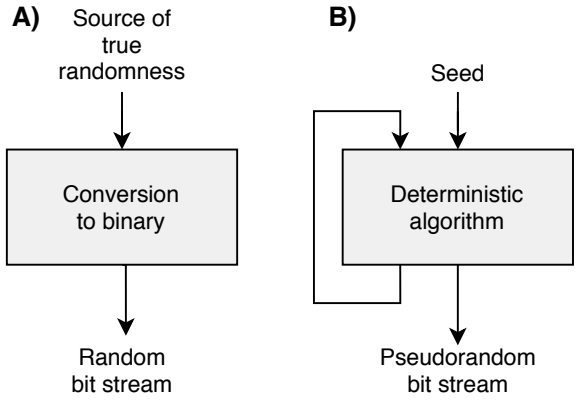

Figure 1.2: Basic model of a true random number generator(A) and a pseudo random number generator(B)[2].

1.4.1 True random number generator

First, we look at true random number generators, and these generators rely on unpredictable sources in the physical world. "*A TRNG is based on the randomness produced by physical phenomena and therefore provides a source of randomness*"[3]. "*A TRNG is a hardware primitive widely used in security and cryptographic application to generate session keys, onetime pads, random seeds, nonces, challenges to PUFs, and so on, and these applications are growing in number with time*"[8]. To produce a truly random output, the TRNG must rely on a non-deterministic source of randomness, such as:

- time intervals in the radioactive decay of a nuclear atom,
- semiconductor thermal noise,
- measurements of running oscillators.

That were so far sources of randomness for the hardware implementation of a TRNG. According to [3], software implementation of a TRNG relies on physical phenomena of connected hardware to a computing device. These sources could be, for example:

- times between keystrokes,
- times between interrupts,
- mouse movements.

1.4.2 Pseudo random number generator

As the price of a TRNG can be expensive, we can use a pseudo-random number generator (PRNG) instead. A PRNG is an algorithm which outputs a pseudo-random bit string. The bit string is required to have no apparent structure; however, the output is certainly not random. If the PRNG is run twice with the same input data, the corresponding output data will follow as before. Anyone who knows the initial seed can completely predict the output; thus, the initial seed must remain secret, so its output appears to have been randomly generated. Usually, the PRNG seed is first generated using a TRNG output. There are many approaches to construct a PRNG, such as linear congruential generators, block ciphers or stream ciphers.

1.5 Data Encryption

Encryption is applied in two forms of using a symmetric key or an asymmetric key. Symmetric cipher uses the same key for encryption and decryption, whereas asymmetric cipher uses two keys, one key for encryption, which is publicly known, and one key for decryption, which the receiver only posses. The symmetric encryption is usually cheaper to implement than asymmetric. In this thesis, we focus on using only symmetric encryption.

Symmetric encryption algorithms, according to [16], we have two different symmetric encryption approaches: stream cipher or block cipher, an example can be seen in fig. 1.3. Stream ciphers perform a series of operations on one bit of plaintext using a pseudorandom key. Block ciphers, data is divided into fixed length of bits, also called blocks, the block is then transformed by the cipher to produce an output. The output size of the cipher is usually the same as the input size, for the block ciphers we typically need to add extra padding to the end of the message.

The strengths, of the symmetric encryption, are: they are fast, hard to crack and cheap to implement. On the other hand, since the symmetric encryption uses the same secret key for both encryption and decryption, the key must be therefore distributed among all parties who want to exchange encrypted messages. That is a potential risk as if we distribute the secret key

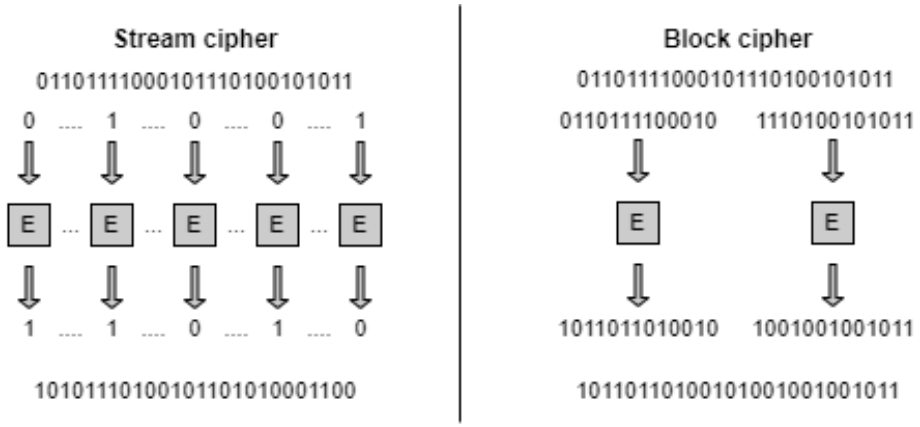

Figure 1.3: Stream cipher versus block cipher.[3]

among multiple parties, which is not secure, and we should use a new key for everyone. In that case, if only a single device were compromised, the secret key and all other devices could be fooled by the impersonated device. The strength of any symmetric cipher relies on the secrecy and strength of the key.

In these days we have multiple block cipher algorithms, but we only present current Advanced Encryption Standard (AES). Rijndael algorithm, also known and interchangeable with AES, is a symmetric block cipher that processes 128 bits size data blocks. It can use keys of different lengths of 128, 192, and 256 bits. The complete design and implementation can be found here [17].

1.5.1 Block cipher modes

Electronic Code Book (ECB) is the easiest and fastest mode to use since it can run in parallel. Although it is fast, it has a fatal security issue; every block is encrypted with the same key. For long messages, multiple patterns could emerge, thus increasing the risk of the attacker discovering the plain text and then deduce the secret key.

Cipher Block Chaining (CBC) is, on the other hand, slower, but it does not have these issues. This mode uses an initialization vector(IV), and every block of the ciphertext is chained with the previous block, except for the first, which is chained with the IV. In this mode, there is no pattern, and with the use of a new IV, even the same message encrypted with the same secret key will always be different.

Cipher Feedback (CFB), according to [3], has similar properties to the CBC mode, but has a little different way of operation. The basic version of CFB mode is, as in CBC mode using an IV. In this mode XORs plain text

with the encryption of the previous block, in the case of the first block, on the IV.

Counter (CTR), can be thought of as a counter-based version of CFB mode without feedback. The feedback is interchanged for a counter, as we assume both sender and receiver have the same counter with the same state, and this counter does not need to be secret. Still, both sides must keep the counter synchronized as it needs to compute a new value each time a cipher-block is exchanged.

1.6 Data integrity

According to [18], integrity ensures that data is protected from unauthorized modification or data corruption. The goal of integrity is to preserve the consistency of data, including data stored in files, databases, systems, and networks.

1.6.1 Hash functions

A hash function is a cryptographic algorithm that processes data of any arbitrary length and outputs a fixed-length output. "*A one-way hash function reduces a message to a hash value*"[18]. Since they reduce the original message to a fixed hash, there is a possibility of a collision happening. A collision is defined as when two different messages have the same hash value. An example of hash functions:

- MD5 – produces a 128-bit hash value, performs four rounds of computations, is not collision-free;
- SHA-1 – produces a 160-bit hash value, performs 80 rounds of computation;
- SHA-2 – this is a family of hash functions, each provides different functional limits.

Most of the hash functions are easy to compute, but they might be a bit weaker in the cryptographic context. Since all hash functions are publicly computable, they provide only a weak notion of data integrity. Results of hash functions must then be protected with the use of another security mechanism, to prevent the hash value from being manipulated by a malicious party.

1.6.2 Message authentication code

A message authentication code (MAC) is a little different from a hash function. It is a cryptographic checksum where a secret key is now involved in the *hash* generation. The usage can be seen in fig. 1.4. "*These are symmetric cryptographic primitives designed to provide data origin authentication, which*

is a stronger notion than data integrity"[3]. Usually, to implement a MAC algorithm is to use a block cipher. The MAC does not provide identification of a problem occurred so the receiver can not determine if the message was altered on purpose or if it was an accident, but they can for sure tell the message is somehow modified.

So far, we talked about MAC based on block ciphers. The second version of MACs are based on hash functions and are called HMAC. The strength of an HMAC is bound to the hash function used. "*This type of MAC can, at least in theory, be constructed from any cryptographic hash function*"[3].

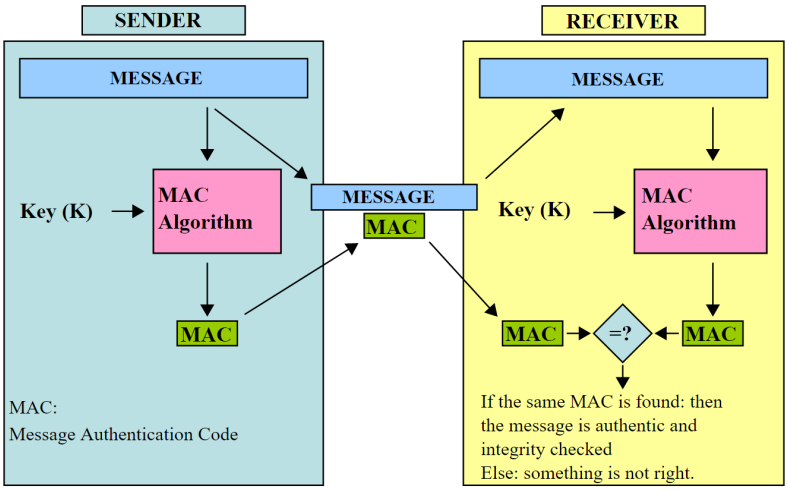

Figure 1.4: An example of MAC use[4].

1.7 ESP-8266

According to [19], the Wemos D1 mini is an inexpensive ESP8266-based WiFi board that is powerful as any NodeMCU or ESP8266-based microcontroller. The D1 mini is very cheap, supports wifi and is fully compatible with the Arduino platform.

According to [20], there are multiple versions of WeMos D1 Mini, which differs by ESP chip, size of flash memory, if they contain LED pin, and what antenna they use.

To use the Wemos D1 mini, one must install the driver for serial port, Arduino IDE and python. The Arduino IDE offers many sketch files to try out the microcontroller. All requirements and documentation can be found online [21, 22].

1.8 Authentication

"*Authentication is the binding of an identity to a subject*"[23]. The authentication process determines if someone is who they say they are. "*Limiting access to unauthorized devices to important assets is the most fundamental security step that you can take*" [24]. To be able to confirm someone identity, they usually should provide some authority one or more of the following:

- something they know,
- something they have,
- something they are.

The process to authenticate someone is first obtaining some of the above data and then determine if it is associated with that entity. For example, we receive username and password, and it matches the entry in the database, or a bank card accepts the given PIN. Not only we can authenticate people, but we also can, and mostly should authenticate software, devices, drivers. These are crucial since if we would not verify them, we could run unauthorized software on our devices which was substituted for the original.

"*One valuable use of encryption is in the authentication process*"[16]. The authentication information, such as passwords or PINs, must always be kept secret from everyone. We have to prevent exposure of sensitive information; otherwise, the authentication would be useless. Such as not keeping passwords in raw format, but for example properly hashed or encrypted.

We divide authentication into two categories. We have single-factor authentication, well known is providing a password to a system. This could be seen as a user is trying to connect to a wireless access point. On the other hand, we also have multifactor authentication. This authentication process requires two or more different identity authenticators. This could be logging into a personal computer using a smart card and the assigned PIN code. We now provided *something we have* and *something we know*. It does not only have to be a smart card, but it could be, for example:

- a USB key dongle,
- an SMS code,
- a fingerprint or iris scanned,
- accepting login attempt in an application on the phone.

1.9 Communication over internet

According to [24], many people think of the Internet as it is only their email and the Web, but on the contrary, there have always been many protocols,

and more are created every day. It should not matter what type of device or software is used; information must move through the Internet in the form of TCP/IP packets.

According to [25], one of the core protocols is the Internet Protocol, generally used the term as TCP/IP. As the title of the protocol indicates, TCP/IP stands for two protocols, Transmission Control Protocol and the Internet Protocol. This term can also include other protocols, applications, and even the network medium. TCP protocol provides reliable delivery and delivery in the correct order, and the IP protocol provides proper routing using its routing table.

A collection of devices that can exchange data freely is called a network, and the Internet is nothing more than a group of networks. Since all data can be freely transferred, it makes the Internet a dangerous place.

Internet of Things (IoT) is a network of connected devices. In most cases, this network is also connected to the Internet. As a result, IoT devices are accessible from the Internet. For that one can, for example, turn on lights, start making their coffee or cook dinner before they even get home. "*As IoT continues to grow in popularity, we will see more and more attacks on the device class*"[16]. These devices could cause potentially DDoS if a vulnerability is found. An example is the Mirai malware discovering and exploiting IoT devices using [26, 27].

Chapter 2

Design and implementation

In this chapter, we design and implement a security device, a communication protocol over which will the devices communicate, and an authentication authority which provides authentication to all enrolled devices. The design part begins with defining our authentication protocol, where we need to identify all parts of the protocol and continues with the design and then implementation of all required components.

2.1 Basic structure of the system

In this section, we show the basic structure of our system and devices. The fig. 2.1 shows the connection of all the devices in the domain to the cloud and are able to communicate wirelessly.

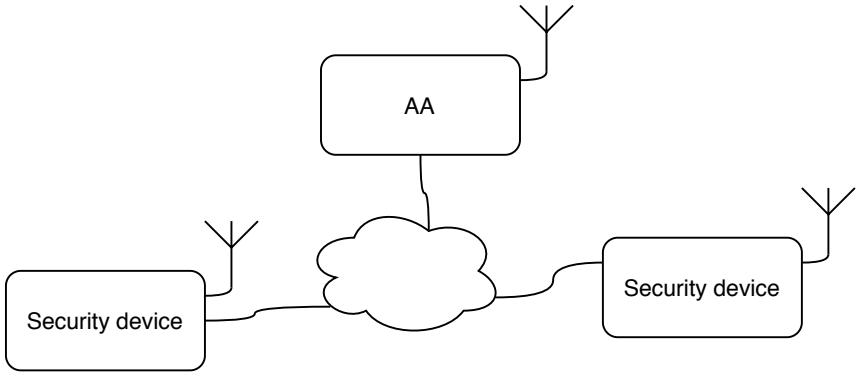

Figure 2.1: Design of our system consisting of at least two security devices and an authentication authority.

In the fig. 2.2 we can see the design of our security device. It is able to communicate with its FPGA and is able to connect to the cloud using its wireless connection.

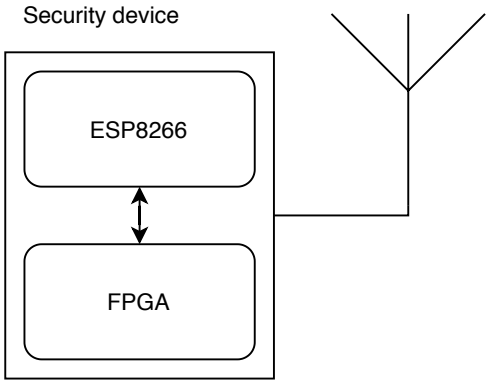

Figure 2.2: Design of the security device consisting of FPGA and ESP8266.

The fig. 2.3 shows our expectation of the enrollment process. The AA sends individual challenges and receives their corresponding responses. This process must take place in a secure environment as these responses are crucial in the security of our usage.

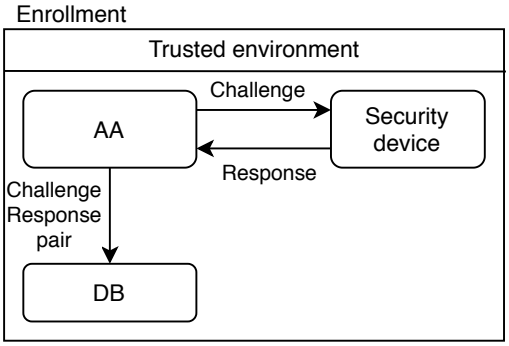

Figure 2.3: Design of the enrollment process between the AA and security device.

2.2 Authentication protocol

Let's summarize what our expectations of the proposed protocol are and how it works. We expect the protocol to provide at least confidentiality and integrity and to force the devices to use PUF as it will be the main source of identification.

We expect that both parties **A** and **B** use their PUF for key extraction from the *helper data* provided by **AA**. That means except for the security devices itself, the only external entity is the AA that has access to all the challenges and responses of these devices across a given domain.

We expect this protocol to be able to resist replay attack since every session starts with a random nonce, the nonce is part of the encrypted message from the **AA** and no one except for the **AA** could potentially encrypt it this way.

We have to make sure both **A** and **B** have the same shared key; there is a simple exchange of an encrypted random number, where the issuer has to increase the numbers by one and encrypt them and send them back, so the receiver knows the other side has the same key.

Despite the beginning, where we had nothing, we have correctly designed a protocol, which provides integrity and confidentiality and basic protection against simple attacks. As from the discussed protocol, we have now defined what we want to achieve and need to design and to implement it successfully.

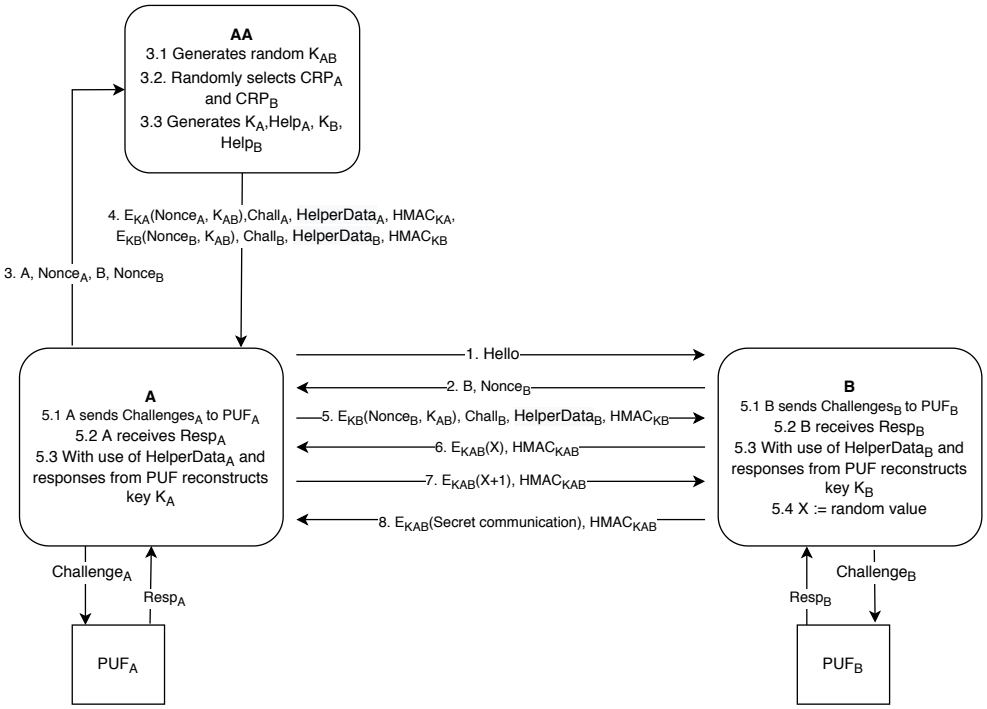

Figure 2.4: Initialization of the communication from device A to the device B using AA

2.3 Synthesized design on the FPGA

The first of the two components of our security device is the FPGA, in the fig. 2.4 shown as PUF only. This device contains the implementation of the required hardware, such as a control block, a PUF and more. In this section, we will discuss the HW design and its implementation. We expect our design and implementation to work only with an external clock source as we want

every operation to be synchronized to this source. In the following subsections, we discuss only the essential parts of the proposed solution. In the fig. 2.5 we can see our design of FPGA.

Even though we want everything synchronized to the external clock, we have an asynchronous reset. This reset clears all flags, counters and memory. The need for this is more elaborated in the implementation of the security device in section 2.8.

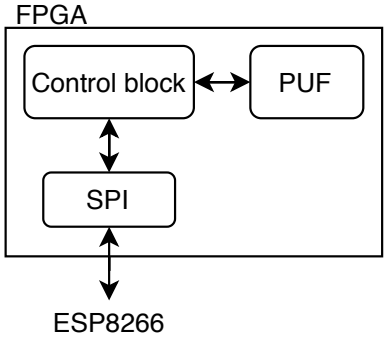

Figure 2.5: FPGA design containing SPI interface and PUF

2.3.1 Data transmission

All of the signals, MOSI, CLOCK, MISO, used in this section, are part of the SPI interface. All received data are stored into an 8bit shift register. Every received bit from the MOSI wire is saved upon the rising edge of the CLOCK. The data are then propagated to the other components, which need to work with incoming data. During every rising edge, already received data are then shifted left, and the new bit is concatenated from the right. As a result of this, the first bit we receive is treated as the most significant bit in the data as can be seen in fig. 2.6.

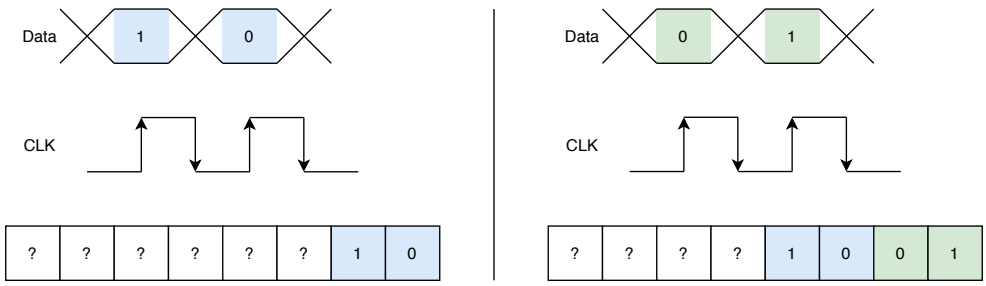

Figure 2.6: An example of receiving 2bits of data followed by another 2bits.

Sending data, on the other hand, is different than storing. We send only the PUF responses and PUF's state. As the PUF responses have two bytes, we send it in two parts. For this, we have a send selector, which is controlled from the control block as the security device asks for it. Every bit of the sending data is ready to be sent on the rising edge of the clock, and as it gets to the falling edge, it puts the n-th bit of a byte on the MISO wire.

2.3.2 Control block

We have decided to implement the control block as a finite state machine(FSM). In the project implementation, this control block is called *Automat*. To communicate with the control block, we use a set of commands that then decide where does the FSM go and what happens with the received data. In the fig. 2.7 we can see the design of the FSM.

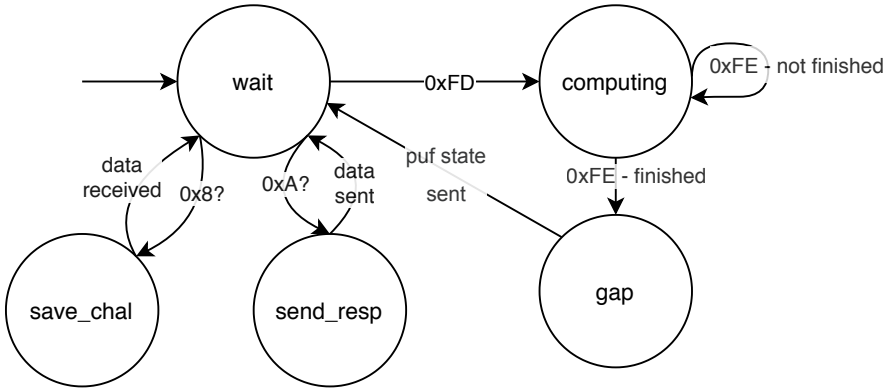

Figure 2.7: The final design of the finite state machine of the control block.

Let's first take a look at the commands, that define the behaviour of the FSM and what we want to achieve with them.

- 0x80 – changes the FSM state into *save_chal*, selects the most significant byte of the challenge and waits for the data. After the byte is received, the FSM state changes to *wait*.
- 0x81 – changes the FSM state into *save_chal*, selects the least significant byte of the challenge and waits for the data. After the byte is received, the FSM state changes to *wait*.
- 0xA0 – changes the FSM state into *send_resp*, selects the most significant byte of the PUF response and on next eight ticks of the clock sends the data. After the byte is received, the FSM state changes to *wait*.
- 0xA1 – changes the FSM state into *send_resp*, selects the least significant byte of the PUF response and on next eight ticks of the clock sends the data. After the byte is received, the FSM state changes to *wait*.

- 0xFD – changes the FSM state into *computing*. When this is enabled, all previous commands are ignored.

- 0xFE – by default does not change the FSM state, as long as the PUF process did not finish, then it changes its state into *wait*. This command is available only during PUF computing and sends PUF state on next byte.

And now arises a little problem, we want to process everything during the rising edge, but during the last rising edge, we just have the data prepared and not yet processed. That causes us to use a filling block of eight bits for only one tick of the clock. So from now on, all commands should have an additional filling byte of data, and their content is not determined. The additional byte will is omitted in the commands, that send data from the FPGA, where we can utilize the sending immediately on the first falling edge.

2.3.3 Physically unclonable function

For our physically unclonable function (PUF), since we are using FPGA, we have chosen the RO PUF. We chose this type of PUF because it can be implemented on the FPGA and already is implemented in *Physically unclonable functions on an FPGA*[5]. This implementation is then connected using its modified interface to our control block, from where we furthermore control the action flow. We have made a series of our own measurements with this PUF that can be seen in the chapter 3 and we used these results as some building blocks for our next work.

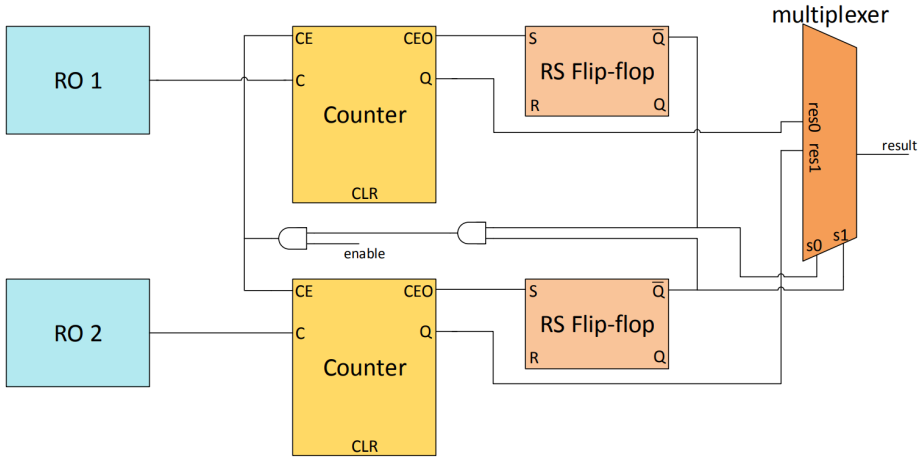

Figure 2.8: Implementation of RO PUF using [5] realization.

2.4 Encryption, decryption, integrity

For the use of encryption and decryption, we decided to use the advanced encryption standard (AES). For integrity, we use HMAC with the same secret key as is used in encryption, respectively decryption.

For this purpose, we chose a lightweight crypto library [28] that provides this functionality. This library offers many utilities such as AES with key length 256 bits, SHA256 hash and more. This library is made especially for ESP8266 which we use.

Since AES requires not only the secret key but as well the initialization vector, we decided to leave the initialization vector as simple as possible, thus leaving it as a vector of zeroes, so devices do not need to send additional data. We use this cipher with 128bit long key, alongside as stated before with initialization vector of binary zeroes. The cipher uses CBC mode. Although AES encrypts in blocks of a fixed length, so it might sometimes need to use extra padding, we chose to use no extra padding after the cipher text so we can store everything in prepared structures, see listing 2.1. There is the only requirement that all sizes are divisible by sixteen, as the library requires that. After every encryption, we create a control HMAC block, with use of the secret key, which we send together with the data, as we do the integrity check of the message upon decryption. That ensures if any malicious third party were to change transmitted data, we would know about it and act accordingly; for example, we always close the connection.

```
struct INIT_DATA_RECV;
struct SESSION_DATA;
struct KEY_AGREED;
struct MESSAGE;
```

Listing 2.1: Data structures which work with encrypted data.

2.5 Replay

According to [29], secret keys, in our purpose, could be called session keys, are possible to be vulnerable to attack. We have to ensure that even if some adversary does cryptanalysis and could potentially break the secret key, has only access to as fewer sessions as possible. Thus every run of the protocol should generate a new secret key between all parties. A new secret key is a must since we do not want any potential adversary to use old setup messages to change the secret key to one of their choice and if they tried to do that, we would detect that.

For this, we use a random nonce of length 128bits, that gives us a lot of usages, which we randomly generate using fizziness of PUF response. The random nonce is sent to the initiator of the session as well as later sent to

the AA. We expect this nonce to be part of the encrypted message alongside the secret key. We trust, only the AA has access to the first secret key, which is generated by her, and only the AA has access to the PUF responses from the SDs. So the secret key we get from the encrypted message from the AA must be valid new key if and only if we decrypted our unique random nonce from the message. If this message contains an incorrect nonce, we discard the session.

As from this part, we can assume, we always have a fresh new secret key for our sessions.

2.6 Error correction code

Since the responses from the PUF can vary and we want to use the PUF responses to establish a shared key between two entities, we have to ensure we get the same response every time or at least a close response as possible to the original one. Instability of the PUF leads us to the need to implement an error correction code (ECC). One of the simplest ECC is a repetition code as written in section 1.3. The repetition code simply repeats a bit, a block of data, respectively, n-times.

We do not want to correct the PUF response. We instead want to transfer somehow securely secret key from the AA to the security device. We use the PUF response to enrich the shared secret key as *helper data*, which nobody else, except for the legitimate receiver, can reproduce. So we form our $\mathbf{h} \coloneqq$ *helper data* in the form of XORed $\mathbf{r} \coloneqq$ *response* with $\mathbf{k} \coloneqq$ *a secret key* as shown bellow.

$$\begin{array}{rl} & r_0 \ldots r_{n-1} \\ \oplus & k_0 \ldots k_{n-1} \\ \hline & h_0 \ldots h_{n-1} \end{array}$$

2.6.1 Encoding

As written in section 2.3.3, we have three bits from each response of the PUF. That goes in hand with our chosen repetition code with a majority of nine. As discussed in section 2.4, we use 128bit long key for encryption and decryption. This combination leads us to the total length of 1152 bits long *helper data*.

Since manipulation of individual bits is expensive and could slow down this process, we have decided to work with them as words, rather than bits. Suppose we have 32bit size words, next fig. 2.9 explains in detail how it works. The first step is to repeat the secret key n-times, then we use $\oplus$ to enrich this key with PUF response, so the secret key is safe from all malicious parties.

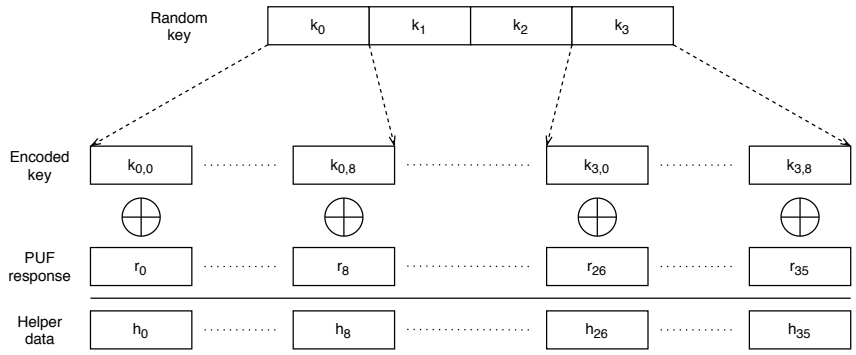

Figure 2.9: Key encoding using a random key k, divided into four groups, and a PUF response r to form helper data h.

2.6.2 Decoding

For the decoding part, we now have the unmodified *helper data* and we have a freshly generated PUF response, which is a possibly little bit different than the response, we used in the encoding part. This new response furthermore labelled as **r'** is going to be used to recover the secret key to its original form partly. We have $k' \coloneqq h \oplus r'$. After this part, we need to create a mechanism, that will allow us to choose if the bit is *one* or *zero*, applying this on the whole **k'** reveals original key **k**. For illustration purposes, we use only one bit as it can be simply replaced with a word of a different size than one to work the same way.

We now face the problem where we have nine bits, and we need to determine if there are over five ones or not. A naive approach is to construct a function, where we use all nine bits at the same time and use a logical expression to determine whether there is the result one or zero. For the majority of three, the formula looks like this:

$$f(a,b,c) = (a\&b)|(a\&c)|(b\&c)$$

For more complex, as we use the majority of nine, we would need the formula to check if any five of the nine inputs are ones and set the result appropriately. The formula for this majority looks like this:

$$\begin{aligned} f(a,b,c,d,e,f,g,h,i) = (a\&b\&c\&d\&e)|(a\&b\&c\&d\&f)|(a\&b\&c\&d\&g) \\ |(a\&b\&c\&d\&h)|(a\&b\&c\&d\&i)|(a\&b\&c\&e\&f) \\ |(a\&b\&c\&e\&g)|\ldots|(d\&e\&f\&g\&i)|(e\&f\&g\&h\&i) \end{aligned}$$

This formula is constructed using $4 \times \binom{9}{5} = 504$ *and* operators and 125 *or* operators. As this formula would be very ineffective, we have used another approach.

A better approach comes with an easy solution as we can use binary adder, exactly for three input bits. The fig. 2.10 shows precisely the implementation

of the fully binary adder and the complete connection of the individual adders to construct a 9-4 parallel counter which input is nine single bits and the output is a binary number as a count of ones. The fig. 2.11 shows how to handle the result of the 9-4 parallel counter to decide if the threshold was crossed as the bit is then *one* or *zero*, respectively, what every bit of a given word is. Everything is implemented in the context of a logical circuit but on software. The resulting formula is then for the fully binary adder as follows.

$$Carry = (a\&b)|(a\&c)|(b\&c)$$

$$Sum = a \oplus b \oplus c$$

For the result of the majority threshold, as shown in fig. 2.11, we use the following formula:

$$f(s_3, s_2, s_1, s_0) = s_3|(s_2\&s_0)|(s_1\&s_0)$$

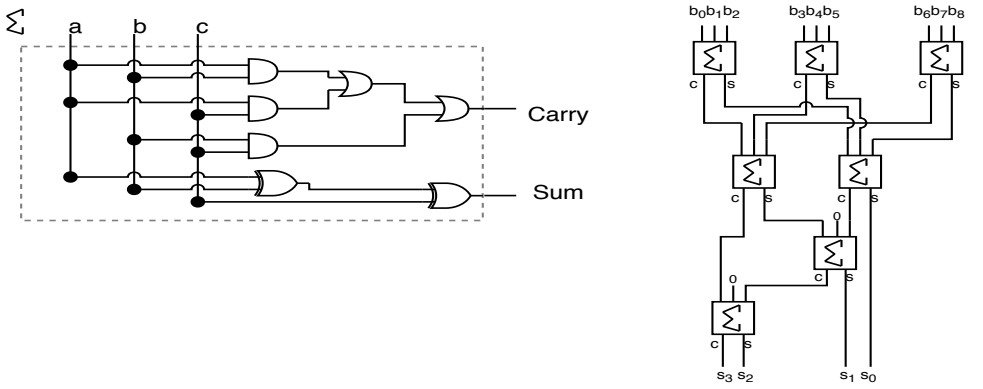

Figure 2.10: Implementation of the full binary adder and 9-4 parallel counter.

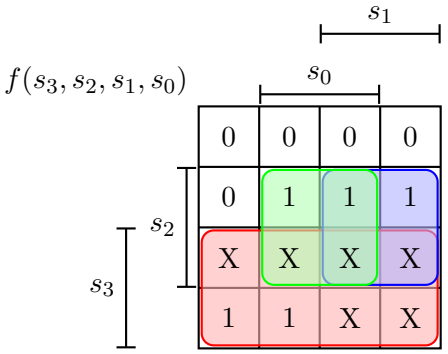

Figure 2.11: Karnaugh map for treshold of the 9-4 parallel counter.

As we now established all the tools for decoding, we show an example of decoding using a 2bit word as longer could be a little bit confusing. We have a word {10}, and this word is first encoded into a new word {10 10 10 10 10 10 10 10}. We add some fuzziness, so the numbers are not always the same, and we end up with {10 11 10 01 11 10 10 10 01}. Usually, we would have to work with single bits, but bit operations are able to work with whole vectors of bits as if we do the operation & on a vector of bits {10} and {11}, the result is also a vector of bits {10}.

We divide the word into three groups $\{\{10\ 11\ 10\}_0\{01\ 11\ 10\}_1\{10\ 10\ 01\}_2\}$. We use the full binary adder (FBA) to form a result vector of carries and sums. We apply it to all groups.

$$FBA(\{10\ 11\ 10\}_0) = \left\{ \begin{array}{ll} carry & (10\&11)|(10\&10)|(11\&10) \\ sum & 10 \oplus 11 \oplus 10 \end{array} \right\} = \{\{10\}_{c0}, \{11\}_{s0}\}$$

$$FBA(\{01\ 11\ 10\}_1) = \{\{10\}_{c1}, \{10\}_{s1}\}$$

$$FBA(\{10\ 10\ 01\}_2) = \{\{10\}_{c2}, \{10\}_{s2}\}$$

As we look at the carries part, this part now shows not that there is a one, but there are two ones, as it represents higher order. If we continued more times, we would construct a binary number telling us how many ones there are in every position. To continue with the process, we present the solution $\{00_{s3}11_{s2}10_{s1}10_{s0}\}$ to be a vector of four two-bit words, representing the number of ones in the artificially noisy sequence. If we take a closer look at the result, we can take the first bit from all words, then the second bit from all words and reconstruct it as a binary number. The first is $0111_2 = 7_{10}$ and the second is $0100_2 = 4_{10}$. From that, we now decide, if the majority of nine threshold was crossed. Now we can use the 9-4 parallel counter and its formula.

$$\begin{aligned} f(00_{s3}, 11_{s2}, 10_{s1}, 10_{s0}) &= 00_{s3}|(11_{s2}\&10_{s0})|(10_{s1}\&10_{s0}) \\ &= 00_{s3}|10_{s2,s0}|10_{s1,s0} = 10 \end{aligned}$$

We have now finished the decoding process and we end up with {10} which was our original sequence.

2.7 Authentication authority

For our purposes, we are using a PC as a server. The server and the SD have some parts of the code in common. The shared functionalities are in a separate header file “common.h” and implementation file “common.cpp”. As for part being, there are some Arduino based functionalities which had to defined or redefined to work in the AA environment.

The primary purpose of the AA is to generate random numbers and take care of the PUF challenges and responses. We now know, to encode one secret key, we need many PUF challenges and responses. Since it would not be space-efficient, to save all challenges for one enormous challenge, we have decided to save some space, to keep only the first challenge, since we can derive all other challenges, of the enormous challenge and the whole PUF response. The fact to save challenge space also benefits the ESP8266, since we are constrained.

The AA is a socket server, which can accept multiple connections at once. For every connection, it will fork itself to a new process and handle that connection. After the connection finishes in any way, either prosperous or not, the process ends as well.

Since we are using a strong PUF, but with very limited amount of challenges and response length, we should treat it as a weak PUF and use every PUF challenge-response pair only once. But on the other hand, we do not present raw responses from the AA to any other party. The responses are always XORed with the random key we just generate for the SDs, and this means we can repeatedly use the challenge-response pairs.

The AA follows the protocol. It generates two required random keys. The first is encrypted with attached random nonce from SD using the second key. Then the second key encodes using encoding from section 2.6 and using $\oplus$ with random challenge-response pair to assemble the *helper data*, of course attaching the challenge to the message. And after it finishes the whole message for the first device, using the second key creates the HMAC for integrity control. In the end, when the AA completes assembling data for one SD and then the same process for the second SD, the AA sends back the required data.

If the AA does not receive correct data, for example, it gets a request for a non-existing device, does not receive some data, it will immediately drop the connection, as it can no longer serve the request.

2.7.1 Enrollment

The first meeting point between the AA and SD is when we want to add new SD into our database. This is happening in a secure and trusted environment, where no secret data can leak. Although we expected in fig. 2.3 the AA will send all challenges to the security device, we left the generation of the challenges on the security device.

The first step is to upload software for the security device in an enrollment mode. Before uploading the software, an administrator has to assign a number, so-called ID, to the device, which is going to be used in the future for its identification and during the enrollment process for saving the generated data. Turning-on the SD while in enrollment mode will print on the serial output challenge-response pair in hexadecimal numbers. Then the authority has a folder structure for its devices:

- dataFiles,
- deviceFiles.

The folder *dataFiles* contains data in plain text hexadecimal numbers, most likely UTF-8 or ASCII. The names of devices do not matter in this case, but the administrator should be able to assign the files to their correct devices. The folder *deviceFiles* contains binary data, the same as data saved in *dataFiles*, the data must not be changed. If associated files from *deviceFiles* and *dataFiles* should be compared, there is no difference.

The files in the folder *deviceFiles* must follow a strict naming convention. It is compulsory to use this format "device-ID[.bin]" or change configuration in the authority source code. An example of correct names "device-00.bin", "device-24.bin", "device-1". An example of a wrong name, "device24.bin".

2.7.2 Start

The server in its initialization expects to have all enrolled devices passed in an argument list upon its start. As it gets harder, with a growing number of enrolled devices, to manage and always run the AA with all its SDs and we need to ensure we still use the correct version of the software. For this, we created a makefile that contains sufficient functions for this use. The functions calls are as follows:

- make compile – only compiles the project if any of the required files were changed,
- make clean – remove all compiled files,
- make run – starts the server and passes to the argument list all devices from the *deviceFiles* folder.

2.8 Security device

For our security device, we use, as discussed in section 1.7, a Wemos D1 mini v3 with ESP8266 enabled WiFi chip. In the fig. 2.2 we can see our design of the security device. As for the second part of the security device, we need to maintain a few things. The first is to connect the device to the FPGA, so we are able to manage and use the PUF. The SD will communicate over SPI with the FPGA. Our security device has multiple modes. For our use, we need the following modes:

- enrollment,
- server,
- client.

Afterwards, our security device needs to communicate over TCP/IP. For that, we connect the SD to any WiFi network. In this network, the SD modes will start their respective functions, such as a web server as an interface for *client* mode or a WiFi client for *server*.

2.8.1 Serial peripheral interface

The SPI is before its first use adequately initialized. That means, as our FPGA is configured see section 2.3.1, we have to configure the SPI accordingly. We use the library provided by Arduino, and its documentation is available online [30]. The SPI is available in all SD modes. The configuration is set to send the most significant byte first and is in *SPI_MODE0* mode, that means all output is set on falling edges, and all data captures are received on rising edge. Unfortunately, after every reset of the device, there is a short CLK signal sent, so the FPGA receives a CLK on its clock, this must be handled appropriately; thus we implemented a reset function which after all initialization of the SPI we use. Its job is to reset the FPGA to initial state.

As we can see the pinout in [31], we have to connect necessary pins from the SD to the FPGA. As using wires for each individual pin would be time-consuming and could lead to potential hazards, we have a hardware connector, where we can place the SD and which can be inserted into the FPGA. All three devices can be seen in fig. 2.12. The connection between every pin of the SD and the FPGA is specified in the FPGA project directory, in the *top_view.ucf* file of the FPGA implementation.

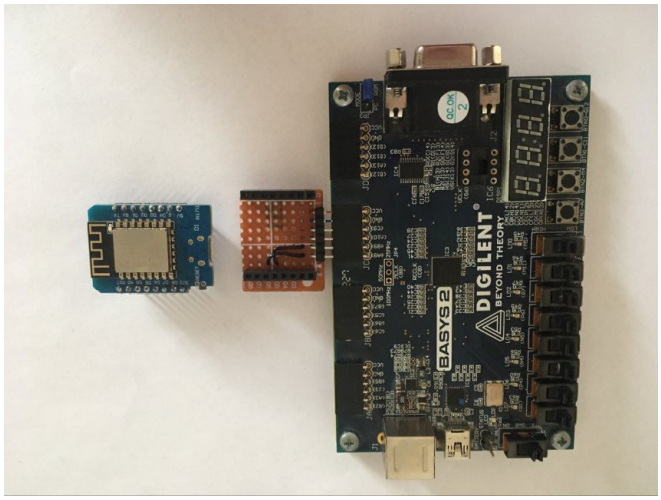

Figure 2.12: Digilent Basys 2 with FPGA, hardware connector, Wemos D1 mini with ESP8266.

2.8.2 Enrollment

This device mode is designed for every new device we want to add to our network of security devices. This mode is used in a trusted environment as there is a potential risk of losing secret responses of the new device. In this device mode, the device has access to only the SPI connection to the FPGA and the serial output. As this mode is for initial purposes only, it does not need any other tools to work appropriately. And the generation function is called only once.

For one enormous challenge for encoding and decoding, all the challenges follow incrementally by one. For our generation of the enormous challenge, we need to use 384 (0x180) challenges as we get only three bits from each response. So for the first enormous challenge-response pair, we use challenges 0x0000, 0x0001, 0x0002, ..., 0x0180. For the next enormous challenge-response pair, we start with the first challenge 0x0181. During this process, the first 16bits of the output is the first challenge, and then it is followed with 288 hexadecimal numbers, this is repeated fifty times, so we have a total of 50 challenge-response pairs for the AA.

2.8.3 Server

This device mode is for servers which can contain sensitive information, such as data from sensors, secret information, etc... These devices are using multiple tools. They are using the serial port for debugging and logging purposes, SPI for communication with the FPGA, WiFi for connection to the WiFi network and at last it uses WiFi server so that other devices in client mode can connect to this server. Documentation to these tools is available online at [32, 30].

The server provides a connection to all devices but serves only one device at any given moment. When the communication is over, the server can then handle another client. Because of this, the server can be a target of an attack by a DOS-type attack. When a connection is received, it immediately starts following the designed protocol and starts exchanging data with the other device. If any part of the communication is unsuccessful, such as decrypting, or not receiving all required data, the connection is closed. That begins with an unsafe communication, but with successive stages, it becomes a safe communication with a shared secret key between both security devices. Since this work is to use a PUF for securing the communication, not designing all possible communication protocol, after the initiation of secure communication, the server sends secret data to the client and ends the connection.

When the server starts up, it first connects to a defined WiFi access point and prints its IP address on the serial output. Then it waits for any incoming connection.

2.8.4 Client

The client mode is very similar to the setup of the server mode, except now we do not need a WiFi server to where we would be connecting. The IP address of the client is first shown on the serial output. For more comfortable use, we could use DNS records and reserve IP addresses for the SDs and name them appropriately, but we do not use it. The client is able to connect to the server, and all stages of the communication are logged, including the secret message from the server, which is our goal.

To demonstrate the usage of the client, as we need a direct connection to the client, we have created a web interface, which all users can use to try to connect from the client to the server. For this web interface, there is a running web server on the SD, and this server is available only in *client* mode. The web server contains an interface containing a simple form where we need to insert an IP address or an URL of the server we want to connect to. In any possible scenario, the interface shows all important steps during the communication, including the secret message, if received correctly.

2.9 Authentication

There is no direct authentication between the SDs and the AA. The AA is only a trusted third party, from where we receive a secret key for communication between the SDs. Indirect authentication to the AA comes in play in two steps. The first when the SDs have successfully checked that their messages were not modified using HMAC. The second decrypted their messages and got their random nonce, which is the same as they generated, and the shared secret key for communication between SDs **A** and **B**. From this point, the SDs believe they received the message and thus the random shared secret key from the AA.

Since the SDs are now indirectly authenticated to the AA, they also have to authenticate themselves. The server **B**, who is receiving communication, generates random numbers of length 128bits. These numbers then encrypt with the shared secret key and send them to the client **A**. The client **A** proves herself with decrypting the received numbers and incrementing them by one, as well as encrypting them and sending them back to the server. When the server gets correctly incremented numbers, the client **A** has just authenticated to the server **A**. The server now knows they have acquired the same secret key and communicate safely.

Chapter 3

Measurements

In this chapter, we will measure the usability of our PUF and its usage in the implementation. Also, we will test the complete functionality of our protocol. For PUF measurements, we used six FPGA boards to determine the difference between PUF stable bits and stable bits for all devices. For some measures, we used only the SD with one FPGA since we needed to observe its capabilities. In the end, we tested two SDs alongside with AA, to measure the acceptance ratio of the proposed protocol.

3.1 Physically unclonable function

In this section, we will mainly focus on the PUF and its response generation. We test the stability bits, difference across the responses from the stable bits and time spent generating responses.

3.1.1 Stable bits

In the first table 3.1 we can see, that the bits labelled 0 – 8 are for many PUFs stable almost over 93%. Bits 9 – 10, as shown in the table, are getting less and less stable, the rest of the bits are even less stable. Next step is to determine, what bits are the same for all challenges and from where they start to change. We expect to have about 50% difference of the bits among the responses, to consider them stable for the challenge and not the PUF. For that, we have created table 3.2. From here we can see, that the bits on positions 0 – 4 do not change most of the time. The first major changes can be seen from the bit 5 and 6, which have the potential to change to one in 47% times, followed by 52.9% change to change to one on bits 7 and 8. These four bits are good enough, and we will use them in the next measurement.

Device	Bit stability in % From MSB to LSB										
	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.
B-00	100	100	100	100	100	100	100	99	96.4	92.1	78.6
B-21	100	100	100	100	100	100	100	99.2	97.6	91.3	82.6
B-24	100	100	100	100	100	98.2	98.2	98.5	89.2	84.5	71.1
B-26	100	100	100	100	100	98.6	96.1	96.1	93.4	87.8	78.7
B-35	100	100	100	100	100	100	100	98.6	97.7	84.4	80.7
B-37	100	100	100	100	100	100	100	100	93.7	85.9	74

Table 3.1: PUF response bit stability.

Challenge	Binary response				Challenge	Binary response			
0xFCD9	110	000	011	00x	0x58C8	101	101	101	001
0xEF6B	101	111	101	11x	0x567A	110	011	011	xxx
0xC80F	110	101	100	111	0x4BA6	110	110	100	01x
0xB513	110	100	110	1xx	0x4A4F	111	111	100	10x
0xABCD	110	000	010	1xx	0x49AF	110	000	000	0xx
0xAB47	100	110	001	10x	0x3A09	110	011	011	0xx
0xAAA1	110	000	011	xxx	0x36E8	101	111	010	1xx
0x7646	101	111	011	110	0x3220	110	110	110	0xx
0x5A99	110	110	101	01x					

Table 3.2: PUF responses from the same PUF.

3.1.2 Response differences

Since we know, we need to uncover the secret key with the PUF response before decoding; we also need to know what is the maximum of wrong bits in the generated response. We are using a majority of nine. That means, we can always have at most four bits out of nine wrong. In that case, we have an upper bound of wrong bits at $\frac{4}{9} \approx 44,4\%$. But this is an optimistic view, and we would have to be lucky.

On the other hand, pessimistic view on this is we can have only four wrong bits at most to be able to decode the secret key correctly, that is maximum error rate at $\frac{4}{1152} \approx 0,35\%$. Also, for this view, we would have to be extremely lucky.

From section 3.1.1 we know, what stable bits we can use. As we want to use at least three bits of the response, we first measure the percentage of wrong bits using bits 5 — 7, then bits 5 – 8 and at the end bits 6 – 8. From the results in the table 3.3 we can see, the best option is using bits 5 – 7.

First bit	Last bit	Wrong bits
5	7	1,64%
5	8	5,25%
6	8	5,92%

Table 3.3: PUF response difference using different bits.

3.2 Response generation times

One of the most time-consuming activity, excluding the time spent waiting for other devices, is the PUF response generation time. We want the time to be as low as possible, and we could achieve that removing a few invertors in the RO PUF of the cost of stable bits. We have made a series of tests on this topic, that revealed the actual time of the generation.

For one 16bit challenge, we ran 1000 runs to get a more precise result. For the enormous challenge consisting of 384 small challenges, we ran 100 runs, but it also includes the time spent on parsing bits. The table 3.4 shows the results. The measurements were measured separately.

challenges\time[μs]	$\mathbb{E}$X	σ
1	791,5	0,31
384	295038,95	192,09

Table 3.4: Time measurements of a response computing in μs.

3.3 Authentication of the security device

In this section, we measured how many times it happens, that we successfully establish a connection between two security devices. A successfully established connection is considered, when the client receives a secret message, everything else is considered to be a failed connection. In this test, we ran 1000 attempts from the test client to the server. Meanwhile, we tested using different bits of the response, to determine, which bits are most stable and guarantee the connection.

First bit	Last bit	Successful authentication
6	8	28.4%
5	8	40.3%
5	7	96.7%

Table 3.5: Successful authentication using different bits.

In the table 3.3 we can see, that even with nearly 4% difference of responses, makes a massive difference in the successful connection attempts as we can see in table 3.5. We do not want to use more stable bits, as we would be at risk of having false acceptance, which we do not wish to.

Conclusion

The aim of the thesis was to study the usability of physically unclonable functions to secure wireless communication. Implement a system consisting of at least two security devices and authentication authority, and together they are communicating over the TCP/IP protocol using a designed authentication protocol.

The goal of this thesis was fulfilled; we designed an authentication protocol, which we then used to describe the communication. The security devices were properly programmed in multiple modes and can be easily switched between modes from the source code using constants in the header file of the security device. Authentication authority is able to enrol new devices into its database and provides authentications means to all enrolled devices when they ask for it.

In the beginning, the reader was introduced to the theoretical background. In this chapter, we discussed physically unclonable functions and their use in a real environment, and then we discussed all the necessary tools for the authentication protocol to work safely. After this part, we moved on the second part, and we discussed all the design and implementation of all the blocks before we assembled them into a security device. In the end, we could see measurements, such as the success rate of the authorization protocol.

The designed protocol is able to protect itself from attacks like replay attack and man in the middle attack for devices that are not part of the domain. The security devices can detect any tampering with the data from unauthorized entities because of the use of HMAC. Unfortunately, the server is vulnerable to DOS type attacks as it is waiting for the communication and is able to handle only one connection at a time.

Bibliography

[1] Dumas, J.-G.; Roch, J.-L.; et al. *Foundations of coding: compression, encryption, error correction.* Book, Whole, Hoboken: Wiley, 2015.

[2] Stallings, W. *Cryptography and network security.* Boston: Prentice Hall, fifth edition, c2011, ISBN 978-0-13-705632-3.

[3] Martin, K. M. *Everyday cryptography.* Oxford, United Kingdom: Oxford University Press, second edition edition, 2017, ISBN 978-0-19-878800-3.

[4] Wikipedia. Message authentication code — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/wiki/Message_authentication_code`, [Online; accessed 03-June-2020].

[5] Kodýtek, F. *Fyzicky neklonovatelné funkce na FPGA*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2014.

[6] Wikipedia. Verilog — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Verilog`, [Online; accessed 03-June-2020].

[7] Wikipedia. VHDL — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=VHDL`, [Online; accessed 03-June-2020].

[8] Bhunia, S.; Tehranipoor, M. H. *Hardware security.* Cambridge, MA, United States: Morgan Kaufmann/Elsevier, [2019], ISBN 978-0-12-812477-2.

[9] Xilinx, Inc. Basys 2^{TM} FPGA Board Reference Manual. [Online; accessed 03-June-2020]. Available from: `https://reference.digilentinc.com/_media/reference/programmable-logic/basys-2/basys2_rm.pdf`

[10] Xilinx, Inc. ISE Design Suite. [Online; accessed 03-June-2020]. Available from: https://www.xilinx.com/products/design-tools/ise-design-suite.html

[11] Lim, D.; Lee, J.; et al. Extracting secret keys from integrated circuits. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, volume 13, 11 2005: pp. 1200 – 1205, doi:10.1109/TVLSI.2005.859470. Available from: https://www.csl.cornell.edu/~suh/papers/tvlsi05.pdf

[12] Eiroa, S.; Baturone, I. An analysis of ring oscillator PUF behavior on FPGAs. In *2011 International Conference on Field-Programmable Technology*, 2011, pp. 1–4. Available from: https://ieeexplore.ieee.org/document/6132673

[13] Platonov, M.; Hlaváč, J.; et al. Using Power-Up SRAM State of Atmel ATmega1284P Microcontrollers as Physical Unclonable Function for Key Generation and Chip Identification. *Information Security Journal: A Global Perspective*, volume 22, no. 5-6, 2013: pp. 244–250, doi:10.1080/19393555.2014.891279, https://doi.org/10.1080/19393555.2014.891279. Available from: https://doi.org/10.1080/19393555.2014.891279

[14] Jang, J.; Ghosh, S. Design and analysis of novel SRAM PUFs with embedded latch for robustness. In *Sixteenth International Symposium on Quality Electronic Design*, 2015, pp. 298–302. Available from: https://ieeexplore.ieee.org/document/7085443

[15] Kumar, S. S.; Guajardo, J.; et al. Extended abstract: The butterfly PUF protecting IP on every FPGA. In *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, 2008, pp. 67–70.

[16] Oriyano, S.-P.; Solomon, M. *Hacker techniques, tools, and incident handling*. Burlington, MA: Jones & Bartlett Learning, third edition edition, [2020], ISBN 978-1-284-14780-3.

[17] PUB, NIST FIPS. 197: Advanced encryption standard (AES). volume 197, no. 441, 2001. Available from: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[18] McMillan, T. *CCNA security study guide*. Indianapolis, Indiana: Sybex, [2018], ISBN 978-1-119-40993-9.

[19] WeMos D1 Mini ESP8266 Arduino WiFi Board. Available from: https://makersportal.com/blog/2019/6/12/wemos-d1-mini-esp8266-arduino-wifi-board

[20] ESP8266 Pinout Overview. Available from: https://diyi0t.com/what-is-the-esp8266-pinout-for-different-boards/

[21] Get started with Arduino [D1/D1 mini series]. Available from: https://www.wemos.cc/en/latest/tutorials/d1/get_started_with_arduino_d1.html

[22] Wemos D1 Mini. 2019. Available from: https://wiki.lvl1.org/Wemos_D1_Mini

[23] Bishop, M. *Computer security*. Boston: Addison-Wesley, second edition edition, [2019], ISBN 978-0-321-71233-2.

[24] Brooks, C. J.; Grow, C.; et al. *Cybersecurity essentials*. Indianapolis, Indiana: Sybex, John Wiley & Sons, [2018], ISBN 978-1-119-36239-5.

[25] Socolofsky, T. J.; Kale, C. J. TCP/IP tutorial. RFC 1180, RFC Editor, January 1991, http://www.rfc-editor.org/rfc/rfc1180.txt. Available from: http://www.rfc-editor.org/rfc/rfc1180.txt

[26] CVE-2014-8361. Available from MITRE, CVE-ID CVE-2014-8361. Available from: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-8361

[27] CVE-2017-17215. Available from MITRE, CVE-ID CVE-2017-17215. Available from: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17215

[28] Ellis, C. ESP8266 Crypto. [cit. 2020-05-27]. Available from: https://github.com/intrbiz/arduino-crypto

[29] Boyd, C.; Mathuria, A.; et al. *Protocols for Authentication and Key Establishment*. Berlin, Heidelberg: Springer Berlin / Heidelberg, second edition, 2019;2020;, ISBN 3662581450;9783662581452;.

[30] Arduino. SPI library. Available from: https://www.arduino.cc/en/reference/SPI

[31] Arduino. Wemos D1 mini pinout. Available from: https://escapequotes.net/esp8266-wemos-d1-mini-pins-and-diagram/

[32] Arduino. WiFi library. Available from: https://www.arduino.cc/en/Reference/WiFi

Appendix A

Acronyms

PUF Physically unclonable function

AES Advanced encryption standard

ECC Error correction code

HDL Hardware Description Language

VHSIC Very High Speed Integrated Circuit

VHDL VHSIC HDL

PUF Physical Unclonable Function

NVM Nonvolatile memory

CRP Challenge Response Pair

SRAM Static Random Access Memory

BPUF Butterfly PUF

ECB Electronic Code Book

CBC Cipher Block Chaining

CFB Cipher Feedback

CTR Counter

IDE Integrated development environment

USB Universal Serial Bus

SMS Short Message Service

PIN Personal Identification Number

IoT Internet of Things

DoS Denial of Service

DDoS Distributed DoS

MAC Message authentication code

HMAC Hash-based message authentication code

MD Message digest

SHA Secure hash algorithm

TCP/IP Transmission control protocol/Internet protocol

AA Authentication authority

SD Security device

FPGA Field-programmable gate array

PC Personal computer

RO Ring oscillator

FSM Finite state machine

MOSI Master out, slave in

MISO Master in, slave out

SPI Serial peripheral interface

XOR Exclusive OR

FBA Full Binary Adder

Appendix B

Contents of enclosed CD

- `data` ..... the directory with measured data
- `scripts` ..... scripts for statistic calculation
- `src` ..... the directory of source codes
  - `authority` ..... the source code of the authentication authority
  - `FPGA` ..... the project for Xilinx ISE
  - `secureDevice` ..... the source code of the security device
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `Frantisek_Kovar_BT_2020.pdf` ..... the thesis text in PDF format