FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Using physical unclonable functions in TLS on ESP32
Student:	Matěj Týfa
Supervisor:	Ing. Jiří Buček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2022/2023

Instructions

Physical unclonable functions are an emerging cryptographic primitive that can be used for device identification, authentication, and key generation in digital devices.

* Study the topic of physical unclonable functions (PUFs). Focus primarily on key generation.
* Select a suitable TLS library for use with ESP32.
* Implement generation of asymmetric keys from PUF responses.
* Design and implement an enrollment procedure for key certification.
* Test authentication and connection establishment using the TLS library with a private key generated from PUF.
* Use a mock-up PUF model in your development, final PUF implementation will be adapted from a parallel thesis [1] by Ondřej Staníček.

[1] Ondřej Staníček: Physical unclonable functions on ESP32, bachelor thesis, CVUT FIT 2022.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 3 February 2022 in Prague.

Bachelor's thesis

USING PHYSICAL UNCLONABLE FUNCTIONS IN TLS ON ESP32

Matěj Týfa

Faculty of Information Technology
Department of Computer Systems
Supervisor: Ing. Jiří Buček, Ph.D.
May 12, 2022

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Týfa Matěj. *Using physical unclonable functions in TLS on ESP32.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Contents

List of Figures

List of Tables

List of code listings

I would like to thank my supervisor, Ing. Jiří Buček, Ph.D, without his help this work would not be possible. I also need to thank my family and friends, who provided all the moral support in the world. Thank you! For all the times you helped me. No matter if it was a technical question, discussion or simple reassurance.

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 particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 12, 2022 .....

Abstract

This thesis focuses on key generation using a hardware security primitive known as physical unclonable function. Proof of concept application is created to test the functionality of physical unclonable function – inside cryptographic protocol known as Transport Layer Security – on ESP32 platform.

The theory behind physical unclonable functions in the context of key generation is examined to define essential properties. An enrollment protocol for the keys generated on ESP32 is proposed to enable the use of generated keys in Transport Layer Security. Available Transport Layer Security libraries are investigated to select the most suitable choice for the proof of concept application.

Both a mock-up (simulating ideal properties) and a real physical unclonable function are used inside the proof of concept application to test the functionality on the ESP32 platform.

Our application proves that physical unclonable functions can be successfully used to generate keys for Transport Layer Security on ESP32. Mock implementation is working almost perfectly. However, the real physical unclonable function poses some significant implementation challenges that decrease the usability of these solutions.

Keywords cryptography, key generation, physical unclonable function, Transport Layer Security, ESP32

Abstrakt

Tato práce se soustředí na generaci klíčů za pomoci hardwarového bezpečnostního prvku známého jako fyzická neklonovatelná funkce. Testovací aplikace je vytvořena s cílem otestování funkčnosti fyzické neklonovatelné funkce – uvnitř kryptografického protokolu známého jako Transport Layer Security – na platformě ESP32.

Teorie o fyzických neklonovatelných funkcích v kontextu generování klíčů je zkoumána za účelem definice důležitých parametrů. Je navržen protokol, který umožňuje použití klíčů generovaných na ESP32 v Transport Layer Security. Dostupné knihovny implementující Transport Layer Security jsou prozkoumány s cílem volby nejvhodnější knihovny pro naší aplikaci.

Uvnitř testovací aplikace jsou použity jak imitace (simulující ideální parametry), tak opravdová fyzická neklonovatelná funkce, s cílem otestovat jejich funkčnost na platformě ESP32.

Naše aplikace dokázala, že je možné úspěšně použít fyzickou neklonovatelnou funkci ke generování klíčů pro Transport Layer Security na platformě ESP32. Použití imitace funkce je funkční téměř dokonale. Ovšem použití opravdové fyzické neklonovatelné funkce přináší zásadní implementační výzvy, které snižují použitelnost takovýchto řešení.

Klíčová slova kryptografie, generace klíčů, fyzická neklonovatelná funkce, Transport Layer Security, ESP32

Abbreviations

PUF	Physical Unclonable Function
SSL	Secure Sockets Layer
TLS	Transport Layer Security
HD	Hamming Distance
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
SRAM	Static Random-Access Memory
AES	Advanced Encryption Standard
RSA	Rivest–Shamir–Adleman
ECC	Elliptical Curve Cryptography
XOR	Exclusive Or
KDF	Key Derivation Function
HKDF	HMAC-based Key Derivation Function
HMAC	Hash-based Message Authentication Code
CA	Certification Authority
CSR	Certificate Signing Request
UART	Universal Asynchronous Receiver-Transmitter
PEM	Privacy-Enhanced Mail
ESP-IDF	Espressif IoT Development Framework
IDE	Integrated Development Environment
SSID	Service Set Identifier
IP	Internet Protocol
DNS	Domain Name System
mDNS	Multicast DNS
DER	Distinguished Encoding Rules
TCP	Transmission Control Protocol

Chapter 1
Introduction

The rise of Internet connected devices and their presence in our day-to-day life highlighted the need for privacy and security. Especially on low-power and low-cost devices such as an ESP32. Historically, the security aspect of these devices was overlooked, usually due to the significant costs associated with the development and distribution of security elements. Nevertheless, their popularity is rising each year.

This work could be used in network connected devices by developers and manufacturers worldwide. It can provide a relatively straightforward (and at no additional cost) way to use industry standard cryptographic protocols for secure network communication on their device. It also has the potential to improve privacy (against an unknown adversary) for anyone using such devices.

The topic of this thesis was chosen because use of of physical unclonable function (PUF) on an ESP32 platform was not sufficiently explored. Furthermore, such devices could greatly benefit from the advantages provided by physical unclonable functions.

In this thesis, we describe what a physical unclonable function is. We study properties required to securely generate something known as a private key[1]. A private key is something only the device knows, and it is also used to prove that the device is the one it says it is.

We design an enrollment procedure that establishes a way for the outside world to verify the device's identity. If you are deploying the device inside your business infrastructure, you can verify the identity of the device when connecting to it from anywhere inside the infrastructure.

Then we investigate available options for implementing Transport Layer Security[2] (TLS) on ESP32. TLS is a cryptographic protocol that provides a secure connection over computer networks, and it can be found implemented on most websites accessible over the Internet. We weigh the positive and negative factors of each available option to choose the most suitable one.

And finally, we create a proof of concept application that uses a private key – that was generated using the physical unclonable function – in TLS. The application also implements our enrollment procedure, after which the device is prepared to establish a secure connection using Transport Layer Security protocol.

We also perform tests on our implementation to establish whether or not is the use of physical unclonable function inside Transport Layer Security on ESP32 platform is even possible and practical.

This work makes use of a parallel Bachelor's thesis [1] by Ondřej Staníček. His thesis provided a usable implementation of a physical unclonable function that will be used as part of our proof of concept application.

[1] sometimes referred to as a secret key, or simply a secret, depending on context and application
[2] successor of Secure Sockets Layer (SSL)

Chapter 2
Aims of This Thesis

The main aim of this thesis is to describe physical unclonable function and to analyze properties and procedures that are required to use PUF as a secret key generator. It also aims to create a proof of concept application to demonstrate the usability of PUF inside Transport Layer Security protocol on ESP32 platform.

In the theoretical section, we attempt to define a physical unclonable function and its properties in the context of key generation. We analyze different approaches to repeatable secret key generation from responses of physical unclonable function. We also propose an enrollment procedure for devices to integrate them into already existing infrastructure. Finally we compare available options for implementing Transport Layer Security on the ESP32 platform.

The practical section aims to develop a proof of concept application to demonstrate the device's life cycle in a typical deployment. We aim first to implement the proposed enrollment procedure using a mock[1] implementation of physical unclonable function. The application should demonstrate the device's ability to establish secure communication using Transport Layer Security.

Finally, we aim to test and evaluate the usability of our application with a real physical unclonable function created by Ondřej Staníček as part of his Bachelor's thesis [1].

[1] an imitation with ideal characteristics

Chapter 3

Physical Unclonable Function

In this chapter, we attempt to describe an emerging security primitive known as physical unclonable function (PUF). We will study properties that should be present in the PUF when it is being used as a key generator. We also describe different types of constructions and consider their suitability for this application.

When reading previously published works, it is possible to come across multiple different names for the same (or at least very similar) concept. The terminology is evolving with different properties that are required from what we call *physical unclonable function* (PUF). Some of previously used names include, but are not limited to:

- physical one-way functions
- physical random function
- physically obfuscated key
- physically unclonable functions

The label of physical unclonable functions can be used to describe a wide range of constructs. While the term is mainly limited to the field of hardware/computer security, it might be applicable to other fields. [2]

3.1 What is a Physical Unclonable Function?

Maes [2] devises an interesting comparison: *"a PUF is an object's fingerprint"*. The name – physical unclonable function – already contains three points of reference:

1. *physical*: A fingerprint is a physical property of an individual, and it is a "component" that exists in real world. [2]
2. *unclonable*: We are unable to create two humans with identical fingerprints, and the fingerprint is "selected" randomly at birth. [2]
3. *function*: Fingerprint functions as an identifier of a specific individual. Each individual fingerprint contains markers that we can quantify and compare to ascertain if two samples match. [2]

While this comparison is quite useful to illustrate the essence of physical unclonable functions, it can not be used as a proper definition.

Over the years, there have been numerous attempts[1] at defining PUF at varying levels of complexity. Neither one of them seems to be successful enough to see widespread adoption. We will to follow the unwritten tradition and combine multiple definitions to create our own.

▶ **Definition 3.1** (Physical unclonable function)**.** *Physical unclonable function (PUF) is a hardware component that maps a challenge to a response using device-intrinsic process variations*[2] *as an entropy source. [3, 4]*

Let us revisit the three defining words now that we have a definition of physical unclonable functions:

1. *physical*: PUF is a physical component with some physical properties. [2]

2. *unclonable*: While we are still unsure what "clone" precisely means. However, the fact that PUF should map answers using device specific characteristics suggests that it should be impossible to copy those characteristics to another device. [2]

3. *function*: A PUF is not a function in a mathematical sense. Since all PUF constructions rely on physical variance to generate responses. Moreover, due to fluctuations, the response might slightly change from time to time. However, in simplified terms, we can still think about it as such. [2]

3.2 Challenge-Response Pair

Physical unclonable function has two inputs. The first one is a challenge selected by the user (or a program, protocol, etc.). The second input is its *device-intrinsic variation*.

The variation is inherited from the manufacturing process and is consistent across the devices life. Small inaccuracies that originated in production, outside the manufacturer's control, lead to different characteristics across units. A typical example of such variation is resistance, leakage current or switching delay. [3]

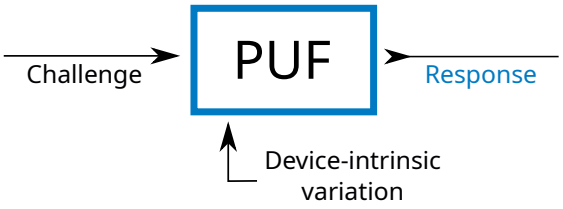

Figure 3.1 Challenge-Response pair in PUF, adapted from [5, p. 5]

As illustrated in figure 3.1, a physical unclonable function creates a response based on a challenge and the device's manufacturing variations. Since these variations are device specific and static they are usually omitted as an input. The physical unclonable function, therefore, create *Challenge-Response pairs*.

Since the device-intrinsic variation is device specific, no two devices should create identical challenge-response pairs with a bigger probability than random chance.

[1] almost every publication we encountered attempts to define PUF itself
[2] manufacturing variations that are integral to a device

3.3 Strong and Weak PUF

Based on the amount of possible challenge-response pairs, it is possible to differentiate physical unclonable functions into two types.

▶ **Definition 3.2** (Strong PUF)**.** *A physical unclonable function is classified as strong if the size of challenge-response pairs is large. It must be impossible to evaluate all possible challenges in a reasonable time. [6]*

▶ **Definition 3.3** (Weak PUF)**.** *A physical unclonable function is classified as weak if the size of challenge-response pairs is small (or in some cases even 1). [6]*

3.4 Properties

While the general behavior of physical unclonable functions is simple, we need to specify some more specific properties that "good" PUF architecture should achieve.

As noted previously, the term physical unclonable function is rather broad, and because of that, it is extremely difficult to define properties that would fit all applications. On one end, we might make the properties too strict to the point of excluding some constructions that still deserve to be considered physical unclonable functions. And on the other end, if we are not specific enough, we end up permitting constructions that might hinder security. [2]

We, therefore, included properties that are interesting (or beneficial) in the context of key generation. A mixture of formal and more relaxed definitions will be used. The aim is to convey why these properties are required, rather than creating a set of strict definitions that all physical unclonable functions must adhere to.

When possible, we tried to include metrics to quantify the PUF compliance with a given property. But the requirements, whether or not does PUF satisfy the property, were kept intentionally ambiguous to allow for different security or implementation requirements.

Multiple metrics might be used when comparing two responses and attempting to quantify their difference. Probably the best known metric might be Hamming distance proposed by Hamming in [7]. Let us modify the definition slightly to account for the different contexts.

▶ **Definition 3.4** (Hamming distance)**.** *Suppose that a and b are two responses consisting of a sequence of bits (of the same length). Then the Hamming distance between a and b is:*

$$HD(a, b) = \textit{number of bits where } a \textit{ and } b \textit{ differ}$$

3.4.1 Constructibility

While constructibility might be a natural expectation, some aspects are not only interesting but also important.

▶ **Definition 3.5** (Constructibility)**.** *Physical unclonable function is constructible if, for a given PUF architecture, it is easy to construct a random unit. [2]*

The adjective *random* is important. We want it to be easy to create multiple units for manufacturing needs. However, at this time, we pose no requirements on the creation of *specific* unit – a unit where some challenge-response behavior is required. [2]

3.4.2 Evaluability

Also evaluability is quite simple yet important. If it would not be possible to retrieve a response corresponding to a valid challenge, the whole physical unclonable function would be pointless.

▶ **Definition 3.6** (Evaluability). *Physical unclonable function is evaluable if it is easy to retrieve a response for any valid challenge. [2]*

The *easy* requirement is highly use case dependent.

3.4.3 Uniqueness

"Uniqueness represents the ability of a PUF to uniquely distinguish a particular chip among a set of chips of the same type." [8]

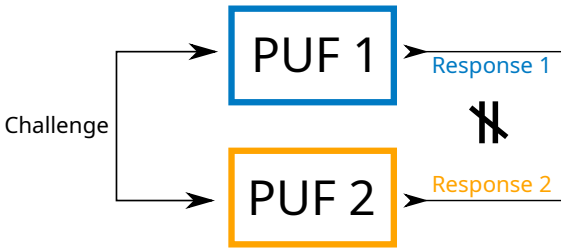

Figure 3.2 Uniqueness in PUF, adapted from [5, p. 8]

As you can see illustrated in figure 3.2, two different devices that are given an identical challenge should not prove identical responses. Further down we actually make the definition even stricter. The responses should differ as much as possible.

When using a physical unclonable function to generate a private key on multiple identical devices it is critical that the generated key is unique for each device. If this property is not satisfied, it could mean that multiple devices share the same private key. And we would no longer be able to guarantee the identity of the device.

To measure the variance between multiple responses (to the same challenge) from different PUFs, we can use Inter Hamming distance. [8]

▶ **Definition 3.7** (Inter Hamming distance). *Suppose that i and j (where $i \neq j$) are physical unclonable functions. And R_i and R_j, with bit length of n, are their respective responses to an identical challenge. With $HD(R_i, R_j)$ representing Hamming distance, from definition 3.4, between the two responses. Then the average Inter Hamming distance for k PUFs is [8]:*

$$HD_{inter} = \frac{2}{k(k-1)} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{HD(R_i, R_j)}{n} \times 100\%$$

The Inter Hamming distance of 0% would mean that all responses are identical ($R_i == R_j$), while distance of 100% implies that the first response is the exact opposite the of second response (i.e. bit-wise negation).

▶ **Definition 3.8** (Uniqueness). *Physical unclonable function is unique if Inter Hamming distance is close to 50%. [3]*

3.4.4 Unclonability

Physical *unclonable* function already contains the required property in the name. But what does it mean?

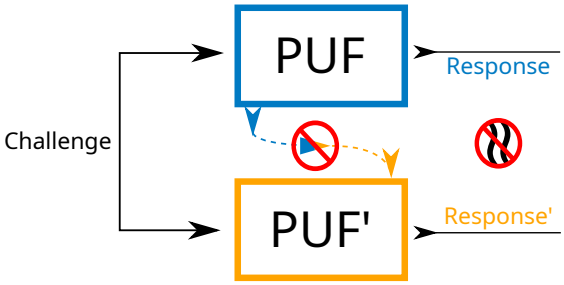

Figure 3.3 Unclonablility of PUF, style inspired by [5]

As you can see illustrated in figure 3.3, it should be ideally impossible to create a copy of physical unclonable function that creates the same challenge-response pairs.

Maes et al. [2, 9] suggest that two distinct types of unclonablility can be formulated.

▶ **Definition 3.9** (Physical unclonablility)**.** *PUF architecture is physically unclonable if it is extremely difficult to create two units that – for every challenge – have minimal* HD_{inter}*. [9]*

This property is sometimes called *manufacturer resistance* because it protects against malicious manufacturers. The manufacturer would be unable to create two identical PUFs, and therefore the manufacturer does not need to guarantee the uniqueness of our unit. [2]

As we previously mentioned, when discussing constructibility in subsection 3.4.1, it should be *easy* to construct a random unit; however *extremely difficult* to construct a unit with specific characteristics. [2]

▶ **Definition 3.10** (Mathematical unclonablility)**.** *PUF architecture is mathematically unclonable if it is extremely difficult to create a mathematical model of another PUF (with unlimited access) that – for every challenge – has minimal* HD_{inter}*. [2]*

This is especially problematic with *weak* physical unclonable functions from definition 3.3. If we have unlimited physical access, it is trivial to save every challenge-response pair to memory. This would enable us to have a perfect mathematical model. [2]

3.4.5 Reproducibility

If we wish to use the physical unclonable function more than once, for example, when we use it to generate a private key, it would be beneficial to use PUF every time we need the private key. This way, there is no need to store the sensitive key in memory.

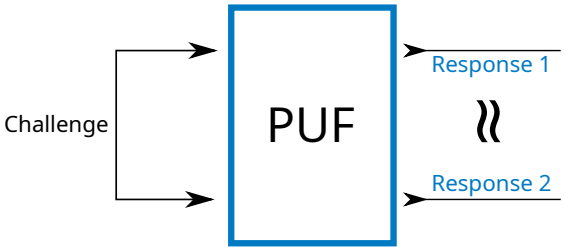

Figure 3.4 Reproducibility in PUF, adapted from [5, p. 7]

As you can see illustrated in figure 3.4, a device should always generate identical responses for the same challenge always. Because the response should be dependent on a challenge and device-intrinsic variations that should be constant across the device's life.

As a metric to asses the reproducibility (or reliability) of a physical unclonable function, we can use Intra Hamming Distance. It uses Hamming distance, defined in definition 3.4, between multiple responses to the same challenge of a single PUF, to quantify their similarity. [8]

Due to the physical properties of PUF, the response is likely at least partially dependent on environmental factors – temperature, supply power fluctuations, etc. We therefore establish a reference response R_{ref} at a normal operating conditions. Which we can later compare to samples taken across operating conditions. [8]

▶ **Definition 3.11** (Intra Hamming distance)**.** *Suppose that we have a physical unclonable function with a response bit length of n. Let us have a single challenge with reference response R_{ref} taken at normal operating conditions, and responses $R_1 \dots R_m$ taken at different operating conditions. Then the average Intra Hamming distance is [8]:*

$$HD_{intra} = \frac{1}{m} \sum_{i=1}^{m} \frac{HD(R_{ref}, R_i)}{n} \times 100\%$$

Intra Hamming distance represents how big portion of a PUF response is unreliable; this portion does not provide stable bits across the measured range [8]. We can express the reproducibility of physical unclonable function as:

▶ **Definition 3.12** (Reproducibility)**.** *Physical unclonable function should be 100% reproducible. Using the Intra Hamming distance metric, we can represent reproducibility of the PUF as [8]:*

$$Reproducibility = 100\% - HD_{intra}$$

The Intra Hamming Distance of 0% would mean that all responses are identical (the PUF is 100% reproducible), with rising Intra HD the collected responses are drifting away from reference response (the PUF is $100\% - HD_{\text{intra}}$ reliable)

To achieve ideal reproducibility, the average Intra Hamming distance should be at 0%. This value signals the responses are stable across tested environments. [3]

3.4.6 Unpredictability

We have chosen to name the property, illustrated in figure 3.5, *unpredictability*, because we wish to convey that it should be impossible to predict the response (or even a part of the response) regardless of how much information is known about the physical unclonable function (short of knowing the challenge-response pair itself).

Figure 3.5 Unpredictability in PUF, adapted from [5, p. 9]

Different publications refer to this property differently. It is often split into multiple simpler properties. For example:

- Hori et al. [10] used two properties *randomness* and *diffuseness*;
- Maiti et al. [11] already included uniform distribution and bit-aliasing requirements as part of *uniqueness*;
- Maes [2] made use of *unpredictability* of next response with *uniqueness* of individual PUFs.

The following definition is constructed by combining multiple separate definitions [10, 11, 2, 5] into a single simpler definition.

▶ **Definition 3.13** (Unpredictability)**.** *Suppose that some number of previously seen challenge-response pairs is known. Physical unclonable function is unpredictable if the probability of correctly predicting a response to a new challenge is close to a random choice* ($\frac{1}{|all\ responses|}$)*.*

It is possible to separate this property into two parts. Firstly it should not be possible to predict (or partially predict) the next response after studying previously used challenge-response pairs. This is especially important if a large number of challenge-response pairs is used (e.g. repeated authorization with pre-shared pairs). And secondly, it should be impossible to predict a response based on a challenge.

3.4.7 Other Properties

There is no shortage of possible properties. We decided to include a few that might be interesting for some applications.

One-Wayness by Maes [2] requires that it should be *very hard* (ideally impossible) to extract the challenge from a response;

Tamper evidence by Maes [2] requires that any attempted modification or security circumvention of the PUF is either blocked, confidential data is cleared, or device-intrinsic characteristics are irreparably changed;

Bit-Aliasing by Maiti et al. [8] states that every bit of response should be a random choice between 1 and 0.

3.5 Constructions

Now that we know *what* is a physical unclonable function we need to discuss how can PUF be constructed.

Since 1993 there have been at least 40 suggested concepts for physical unclonable function construction. While not all of them used the term *physical unclonable function*, it is possible to find similarities in their functional principles and/or intended usage to label them as such. [12]

Due to the availability of a wide variety of PUF construction, it is highly probable that a suitable construction for your specific need already exists. It would be impossible for us to include every single one in this work.

We, therefore, selected just a few interesting proposals. If you wish to explore more construction "A PUF taxonomy" by McGrath et al. [12] is an excellent starting point.

3.5.1 Optical PUF

Pappu et al. [13] proposed in 2002 a so called *Physical One-Way Function*, this led to what we call today a *physical unclonable function*.

The proposed constructions use a piece of optical epoxy containing randomly dispersed glass spheres of varying size (500–800 µm). This *token* is the source of device-intrinsic variations.

As you can see illustrated in figure 3.6, a laser beam (HeNe, 632.8 nm) is emitted through the token at an angle. This angle is the *challenge* of this construction. Due to a difference between the refractive indexes of the glass and the used optical epoxy (and other imperfections), the laser disperses and creates a so called *speckle*. This speckle is then recorded using a camera. A final 2400-bit *response* is created from the picture using a Gabor transform. [13]

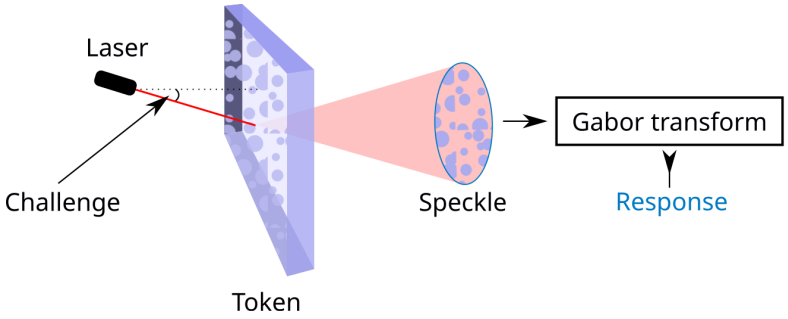

Figure 3.6 Functional principle of optical physical unclonable function, adapted from [9, p. 8]

Challenge-response space relies on the accuracy that can be achieved when positioning the laser. The mount allows movement in six degrees of freedom. [13]

The properties of optical PUFs are exceptional. This construction meets all properties we previously outlined. [2]

Unfortunately, it is rather impractical for the decoder to be used in small and/or low cost devices [6]. It would be possible to use this PUF as an access control system, where cheap tokens could be issued to individuals. The downsides in the form of required scratch protection, periodical cleaning of optical systems, etc., make the solution less appealing.

3.5.2 SRAM PUF

Static random-access memory (SRAM) is a *"digital memory technology based on bistable circuits."* [2]. This storage technology is widely used in a wide range of products, including micro controllers and field-programmable gate arrays [3].

A single cell – a component that can be used to save 1 bit of information – is usually constructed using four transistors that create a flip-flop circuit, and two additional transistors are used to read and set the information. The flip-flop circuit consists of two parts, where one is always the opposite of the second part. Based on which side is high[3] we can decode the stored information. [3]

When power is applied to a cell, both parts of the cell attempt to enter their high state. But due to the flip-flop circuitry, only half of the circuit can be high. Based on manufacturing variations of the transistors, the cell will assume a "random" state. [3]

The strength of transistors that decides the state of a cell is susceptible to voltage noise. When the variance between transistors is small, the outcome of cell power up might become random. We can repeatably measure the outcome and calculate the stability of a cell as illustrated in figure 3.7. [2]

[3] a powered on state

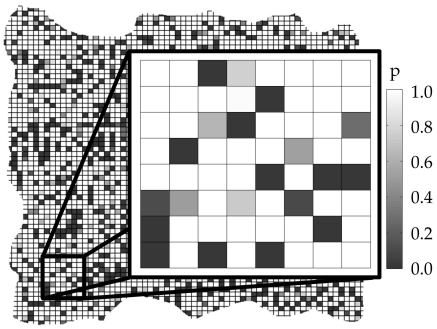

Figure 3.7 Stability of SRAM cells after power up, reproduced from [14, Figure 3]

Since the SRAM PUF is using a memory component that is already available on low-power and low-cost devices such as ESP32, it is a suitable candidate. But does it comply with the properties that we previously described?

physical unclonable function Yes, a challenge is a selected part of memory; response is the state after power-up. [2]

constructibility Yes, we are able to construct a SRAM. [2]

evaluability Yes, we need to temporarily remove power from the cells, and after powering them up we only need to read the memory. [2]

uniqueness Yes, based on measurements by Maes [2], the mean Inter Hamming distance was in the range of 49.59% to 49.72%. Measurements by Holcomb [14] also confirm that it is possible to use responses to identify a unit.

unclonablility No, since the challenge-response space is fairly small (limited by the size of memory), it is trivial to create a lookup table by iterating over every possible challenge. This is in conflict with the definition of *mathematical unclonablility*. [2]

Unfortunately, also the aspect of *physical unclonablility* is in danger. It would be possible for a manufactures to include circuitry to manually initialize cells at power up.

Helfmeier et al. [15] demonstrated the creation of physical copy using Focused Ion Beam circuit edit. This process modifies the transistors of a new SRAM to match extracted characteristics.

reproducibility Yes, with the mean Intra Hamming distance of 5.46% as measured by Maes [2], it is possible to reproduce with a relatively high success rate. With the use of error correction, it should be possible to achieve almost flawless reproducibility.

unpredictability Yes, measurements by Holcomb [14] indicate uniform distribution of 1 and 0. Furthermore, there is no corelation between multiple challenge-response pairs.

A SRAM PUF is an example of weak physical unclonable functions defined in definition 3.3. As such, it is useful as a device identifier or a physically obfuscated key. The advantage of already being present (as a form of memory) on many devices might outweigh the possibility of being cloned.

3.6 Usage

There are many potential use cases for a physical unclonable function. While this work focuses on key generation (on which we focus in chapter 4), we have also prepared a selection of other possible use cases.

3.6.1 Device Identification

Mainly on the grounds of *uniqueness* from definition 3.8 and *reproducibility* from definition 3.12, physical unclonable functions can act as an identifier. Suppose such PUF is included in a device. In that case, the simple fact that response to a predefined challenge is unique among multiple devices means that the response can act as a devices identifier. [2]

The usage of physical unclonable function, to identify a device, have a few significant advantages. The need for an algorithm that generates unique identifiers is eliminated. When the device is deployed, it is also no longer required to make physical (e.g. a write to a permanent storage) changes to the device itself, simplifying and speeding up the process. [2]

3.6.2 Device Authorization

When we add stricter requirements to identification introduced in subsection 3.6.1 we can, in addition to identification, also authorize[4] the device. The added properties usually require the use of strong PUF definition 3.2 and unclonablility definition 3.9 to guarantee that it is impossible to impersonate another device. [2]

In an authorization protocol proposed by Pappu et al. [13], the enrollment[5] process consists of generating multiple challenge-response pairs that are send over a secure channel and stored inside the system.

When an attempt to authorize the device is made, the system selects one of the previously stored challenges and requests a response from the device. The received response is then compared to the one previously generated. The device is authorized only if the responses match. [13]

After a challenge is used it is deleted from the system. This allows us to authorize the devices over untrusted connection since the challenge-response pairs are not reused. [13]

3.6.3 Counterfeit Protection

A physical unclonable function can be used to combat counterfeiting. When a manufacturer manufactures a new unit, its challenge-response pairs can be stored inside a database. When devices authenticity needs to be tested, response to one of the previously stored challenges is evaluated by the user and sent to validation to the manufacturer. [3]

This is principally similar to identification introduced in subsection 3.6.1 and authorization from subsection 3.6.2. The difference is that the manufacturer can validate a devices authenticity based on information from customers.

[4] validate that it is whom it claims to be
[5] the initialization of a device in a system

Chapter 4
Key Generation from PUF

In this chapter, we simply describe what a cryptographic key is. Then we take a look at what we need to consider to generate a cryptographic key from physical unclonable function.

Cryptography creates constructs (protocols and algorithms) to protect information [2]. Information that we wish to protect is called *plain text*. A construct takes this information and encrypts it using a *key* creating a *cipher text*. [16]

Cryptographic constructs aim to provide the ability to keep the plain text secret, even if the inner workings of the construct and cipher text is disclosed to the public (i.e. attacker). The problem is reduced from trying to protect every information to protecting the key. [2]

4.1 Cryptographic Keys

A cryptographic key is a parameter to a cryptographic construct that is used to transform information to a cipher text that can not be transformed back without knowledge about the used key [2]. Based on the procedure that is used to decrypt – transform cipher text back to plain text – we can distinguish two different types of keys [16].

4.1.1 Symmetric Key

A symmetric key (sometimes called a shared key) is a key used in symmetric cryptography. As you can see illustrated in figure 4.1, in this type of cryptography, the key that is used to encrypt the plain text and decrypt the cipher text is identical. [16]

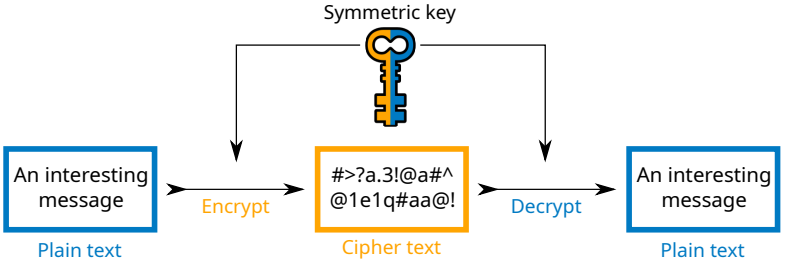

Figure 4.1 Encryption using an symmetric key cryptography, adapted from [16, p. 84]

Symmetric key cryptography allows secure information transfer between users/devices that

know the used key. The transformation between plain and cipher text can be done only by someone who knows the symmetric key. [2]

Symmetric cryptography is usually faster and less resource intensive. Multiple constructions using a symmetric key were proposed and used over the years. Probably the best known example is Advanced Encryption Standard (AES). [16]

4.1.2 Asymmetric Key

In asymmetric cryptography, we use two different keys. The first key is called a *private key* and is meant to be kept private by a single user/device. The second key is called a *public key* and is meant to be shared to other users/devices. It is often called a public-private key pair. [16]

There are two modes of operation that can be used with asymmetric cryptography:

encryption mode In the mode illustrated in figure 4.2 anyone can encrypt plain text message using a public key. This cipher text can be decrypted only with the private key. [2]

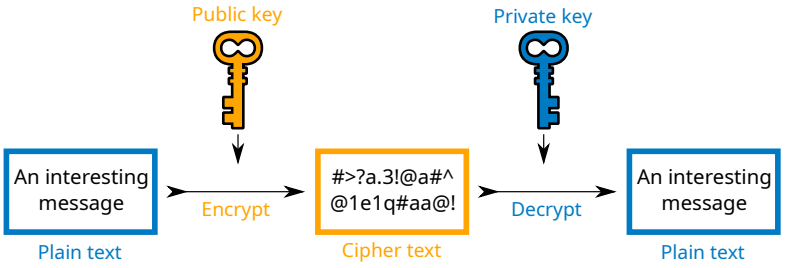

Figure 4.2 Encryption using an asymmetric key cryptography, adapted from [16, p. 84]

signature mode In the mode illustrated in figure 4.3 a user/device signs a message using a private key. And anyone, who knows the public key, can verify that the message was signed with the key that belongs to the key pair. [2]

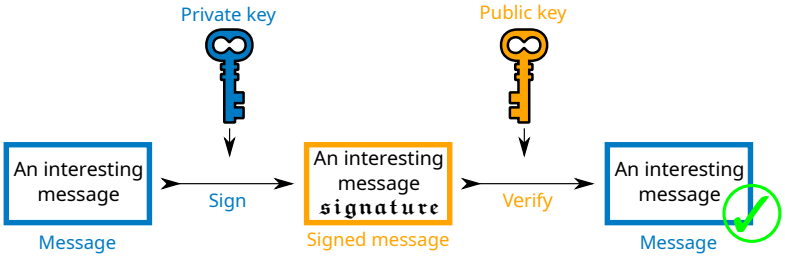

Figure 4.3 Signature using an asymmetric key cryptography, inspired by [16, p. 84]

The use of asymmetric cryptography is more suited for instances where we want to be able to identify an individual user/device.

Notable examples of asymmetric cryptography include Rivest–Shamir–Adleman (RSA) and Elliptical Curve Cryptography (ECC). [16]

4.2 Error Checking and Correction

When we defined reproducibility of physical unclonable functions in definition 3.12 we recommended that the reproducibility should be at 100%[1]. Unfortunately, measurements by Maes [2]

[1] 100% reproducibility means that every generated response for a repeated challenge is identical

demonstrated that even the best performer in reproducibility, SRAM physical unclonable functions only reached 90% reproducibility. This means that if we need an identical response every time, an additional error checking and correction procedure must be implemented.

The recommended reproducibility value of 100% was chosen because cryptographic protocols/algorithms require the key to be identical every time. Even if a single bit of key was changed between encryption and decryption, the decrypted text should have no resemblance to the original plain text. [3]

As a side note, if we wish to use the key only once, this requirement would not be important. If the intended use is an authorization using pre-shared challenge-response pairs an error correction may not be even necessary. Similarity between expected and actual response could be calculated, and the device could be authorized based on a threshold. [17]

It would also be possible to generate a key using physical unclonable functions once and then store it inside non-volatile[2] memory. However, this approach enlarges the attack surface on the device. If the key is not in memory (e.g. when the device is off or not connected) any memory disclosure attack would be unable to reveal it. [3]

The aim of error checking and correction is to stabilize and repair the subsequent response that match the original. It allows us to detect and repair erroneous data. A helper data[3] is usually created to aid in response restoration. This data must be considered public and therefore contain no information that can be used to predict the response. [17]

The following overview of approaches to error checking and correction is not exhaustive.

4.2.1 Secure Sketch

Information in this subsection is based on a publication by Dodis et al. [18], in which they propose a *secure sketch* which converts noisy inputs to reproducible outputs. This transformation is additionally done in a way that does not significantly decrease the entropy of the input.

Secure sketch consists of two randomized procedures:

sketch This procedure generates a helper data h for input $w \in W$. The helper data is a sequence of bits that can be potentially publicly disclosed.

recover This procedure generates an output for input w' using the previously generated helper data. If w and w' are similar enough, the output of the recover procedure is identical to the original w.

Disclosure of the helper data h, for randomly selected w with min-entropy[4] of m, should not increase the prediction probability of w above $2^{-\tilde{m}}$. Where $\tilde{m}$ is a min-entropy of response when helped data is known.

There are multiple possible secure sketch implementations. Two main implementations are using error correcting block code C with 2^k distinct n-bit codewords. Each codeword is at least $2t+1$ bits apart from every other codeword. This distance can be measured by Hamming distance defined in definition 3.4. More specific information about the used error correcting codes can be found in publications by Dodis et al. [18] and Hamming [7].

code-offset construction This construction calculates offsets from code words.

1. *Sketching*: A random codeword c is selected. The helper data (h) is constructed as the change that is required to reach the input (w) from the codeword (c). $h = w - c$
2. *Recovering*: When attempting to recover word w', the change stored in the helped data (s) is applied in reverse to the word: $c' = w' - h$. This new potential codeword c' can

[2] memory that does not need power to retain data
[3] sometimes called parity and/or redundancy
[4] later defined in definition 4.1

then be decoded using error correcting code C. If $\mathrm{HD}(w, w') \leq t$, the original codeword is restored ($c' = c$). At this point if the change is applied to the codeword (c) the original input is recovered. $w = c + h$

syndrome construction This construction is only possible if the used code C is linear.

1. *Sketching*: The helper data (h) is constructed from input (w), using parity check matrix H from the selected code C, as: $h = H \cdot w$
2. *Recovering*: When attempting to recover input (w'), a n-bit error vector e is defined using: $H \cdot e = H \cdot w' - s$. The corrected word ($w$) then can be calculated as: $w = w' - e$.

In the case of binary strings additions and subtractions can be achieved using bit-wise exclusive or (XOR) operations. Both constructions are equivalent and provide error correction up to t bits. A w' input can be corrected back to the original w if: $\mathrm{HD}(w, w') \leq t$.

4.2.2 Repeated Sampling

Information in this subsection is based on a publication by Price et al. [19], in which they propose a new approach to error correction for physical unclonable functions. The proposed algorithm allows the use of all response bits regardless of their noisiness.

The process requires a test that can verify the accuracy of the generated key. This means that in the case of asymmetric cryptography, only the public key is required. If the generated key is used inside symmetric cryptographic protocol, additional helper data, such as plain and cipher text pair encrypted using the key, is required. This approach can lead to reduced storage requirements on most applications since a public key is already stored for other reasons. The maximum number of restored bits is not defined at enrollment but rather at reconstruction.

enrollment Multiple samplings of PUF response are performed to establish the most probable response. Since this procedure is not done often, it is possible to perform a large number of samplings to increase the accuracy of this correct response.

This generated response is treated as correct. A private key (or in the case of symmetric cryptography plain and cipher text) is generated and stored for future correctness tests.

reproduction During the reproduction stage, multiple samplings of the physical unclonable function are performed to establish the stability of each response bit. The number of samples can usually be lower to allow faster reproduction, and this repeated sampling is the main error correction mechanism employed.

When the most probable response is calculated, the generated key is tested. If the public key (or cipher text) match the stored data, it is correct.

Two additional error correction mechanisms can be performed to repair a key that was unable to be corrected by repeated sampling:

1. broadening search for $1 \ldots n$ bit errors
2. exhaustive search ordered by bit stability

The number of required repetitions in the enrollment stage should be greater than the number of repetitions in the reproduction stage. The measurements conducted by Price et al. [19] on a latch based physical unclonable function suggests that 353 samplings at reproduction make any bit error improbable.

The required sample count is however dependent on PUF construction as well as external noise, temperature and age. Since the amount of samplings can be adjusted without re-enrollment – provided that the device was originally enrollment with a sufficiently large number of samples

– this value can be modified at runtime in an attempt to find the balance between sampling and error correction.

The broadening search is designed to correct only a small number of bits. The probability of a change of a small subset of response bits is much higher than the probability of large instability. If this process is unable to recover the key, an exhaustive search is performed based on bit stability. This, in theory, guarantees the recovery of the key. In practice, this is only effective if erroneous bits indicated higher instability than the rest of the response.

At some point, this search becomes more of a brute force attack. And since most used keys are long enough to resist these attacks, in a reasonable time frame, a time limit should be implemented.

4.3 Generating Key from Response

We can construct physical unclonable functions based on information in chapter 3. Small errors that are common to every PUF construction can also be repaired using error checking and correction introduced in section 4.2. At this state, we are able to create unique responses repeatably. In this section, we take a look at different approaches to using the generating response as a key in a cryptographic protocol.

Most physical unclonable functions, unfortunately, exhibit some amount of predictability. Sometimes it might be in form of bit preference (e.g. bits of value 1 are more likely to appear), other times, it can manifest itself in the form of increased probability for groups of identical bits. [8]

Different cryptographic algorithms/protocols impose different requirements and restrictions on the keys. However one common requirement is usually high entropy. [2]

A *min-entropy* can be used to quantify this requirement. Entropy is a measurement of "uncertainly" of information. [18]

▶ **Definition 4.1** (Min-entropy). *The min-entropy is based on the entropy of the most probable response [18]:*

$$H_\infty(A) = -log_2(MAX_a(P(A = a)))$$

Let us suppose that PUF generates 4-bit responses and that the responses are uniformly distributed (all responses have the same probability of generation). In that case the min-entropy is $H_\infty(A) = -log_2(0.0625) = 4$. This means that the response bit sequence contains as much information as possible.

If the generated responses are not uniformly distributed, the min-entropy metric changes. If, for example, the response 0000 is generated with probability 25%, and the rest is then uniformly distributed (each with a probability of 5%). The min-entropy for this physical unclonable function would be $H_\infty(A) = -log_2(0.25) = 2$.

Depending on the algorithm/protocol, and characteristics of the physical unclonable function, multiple approaches are possible when transforming PUF response to a cryptographic key. Some of which are discussed below.

4.3.1 Response as a Key

The simplest way to utilize a response is to simply use it as a key. There are some challenges that should be considered.

length Unfortunately, algorithms usually require a predefined key length. For example, Advanced Encryption Standard offers 3 different key lengths: 128, 192 and 256 bits [16].

If the selected response is shorter than the required key length, we must extend it to match. This can be done in multiple ways. If we have the ability to use multiple challenges, another

response could be chained to double the length. In the case of weak PUF, with a single response, this would not be possible.

Another solution to a short response is to simply repeat the response or create a simple transformation (e.g. switch every other bit, use response in reverse). However, both of these options are strongly not recommended. Part of the key is now more predictable because it depends on another part of the key.

Let presume that the attacker has access to the implementation (either by it being open source, being reverse engineered or by simple information leak). He can take a look at the key usage passage of the application and see that the 128-bit key is comprised of two identical sequences of 64 bits. His brute force attack is now reduced from $2^{128} \approx 3.40 * 10^{38} \approx$ 340 undecillion[5] to mere $2^{64} \approx 1.84 * 10^{19} \approx 18$ quintillion[6]. While those numbers might seem relatively large, the decrease in required time complexity is by 2^{64}.

In a report [20] published by the European Union Agency for Network and Information Security published in 2014[7] a minimal 80-bit key length for *legacy* symmetric ciphers (such as AES) was recommended. With 128-bit keys being recommended for near term future use. Our, at most, 64-bit key does not meet those safety criteria.

The situation is slightly better if the response is longer than the required key length. The unused part could simply be left unused. This would be slightly ineffective as part of the response that had to be generated and corrected is now not being used.

properties Another factor to consider is that not all technically possible key values (e.g. all values in a range $0 \dots 2^{128}$ for 128-bit key) are possible or safe to use. Some cryptographic constructions have a small subset of keys that lower the provided security. Those keys should therefore not be used. Some constructions also require the keys to fulfil some additional requirements. For example, the Rivest–Shamir–Adleman (RSA) uses two prime numbers (with some additional properties for added security) to generate the private-public key pair. [20]

entropy Key entropy must be taken into consideration. Even if the response has the same (or larger) length as the required key, it does not mean that the security provided is identical to a randomly generated key. An ideal min-entropy would be identical to the key length. [18]

4.3.2 Strong Extractor

A strong extractor (sometimes called randomness extractor) can be combined with a secure sketch introduced in subsection 4.2.1 to form a single construct called *fuzzy extractor*. [18]

A strong extractor uses a seed sequence, of r random bits, to transform nonuniform n-bit sequence into l-bit sequence of mostly (with ε error) uniform distribution. The output length l is limited by [18]:

$$l \leq m - 2\log\left(\frac{1}{\varepsilon}\right) + O(1)$$

If a high quality (i.e. uniform distributed) seed of sufficient length is available, a universal hash functions creates a strong extractor with maximal output length $l = m - 2\log(\frac{1}{\varepsilon}) + 2$. The hash function is universal if the probability of two different inputs generating the same hash is equivalent to a random choice. [18]

Other constructions of strong extractors are possible. However, only possible improvement is to the required length of the seed sequence.

[5] $2^{128} = 340\,282\,366\,920\,938\,463\,463\,374\,607\,431\,768\,211\,456$
[6] $2^{64} = 18\,446\,744\,073\,709\,551\,616$
[7] 7.5 years ago at the time of writing

4.3.3 Key Derivation Function

A key derivation function (KDF) uses a *source keying material* with less than ideal properties (e.g. attacker has some knowledge about it, it is not distributed uniformly) and creates keys that are suited for cryptographic purposes. The process can be split into two phases. The first phase extracts uniformly distributed keys that can be used in the second phase to expand into multiple independent cryptographic keys. [21]

The extraction phase is based on strong extractors introduced in subsection 4.3.2. And the second phase uses a pseudo random function to expand the input into multiple usable keys. In practice, many KDF do not follow the two phase design, and combine the process into a single step. [21]

An HMAC-based key derivation function (HKDF) was introduced in a work by Heidelberg [21] and later published as RFC 5869. As the name suggests, the HKDF is a key derivation function that uses hash-based message authentication code (HMAC) in the second phase. [21]

As it might be already clear, every proposed solution improves the previous one. A raw key suffers from uniformity and length problems. A strong extractor can combat poor uniformity. And key derivation function adds additional key derivation functionality.

This key derivation functionality means that multiple pseudo random keys can be generated using a single response from physical unclonable function. This allows us to select another derived key in the case of secret key disclosure or to use multiple different keys for different purposes.

4.4 Key Generation Recommendations

In this section we argue our approach to key generation we use in our proof of concept application. We explain our reasoning behind our selection.

This should not be taken as a definitive solution as the field of physical unclonable function is still evolving. New PUF constructions can be faster, cheaper, simpler and/or more secure. New approaches to error checking and correction can increase the stability of responses over time. And with new cryptographic algorithms/protocols come new requirements on keys.

Another factor to consider is security vulnerabilities that were discovered in either component, from the PUF construction to the selected cryptographic protocol.

4.4.1 Best Construction

This is a complex problem that does not have a definitive solution. Measurements by Maes [2] provide a reference to different constructions. And indicate that most of them are a suitable choice with varying level of error checking and entropy extracting required.

It usually comes down to what is available for your application. We especially like the SRAM construction, for its mostly ideal properties and high availability.

4.4.2 Best Error Checking and Correction

We would recommend to use the combination of repeated sampling and a secure sketch.

Since the repeated sampling is usually fast, it can provide a nice starting point (especially if used in the enrollment phase). This helps to filter out a random errors.

However, it can not significantly combat a larger change in environmental factors. For those, we would use a secure sketch. This not only helps with the environment but it can also combat permanent changes to the characteristics caused by age. If a bit, in the devices life, switches from (mostly) generating 1 to generating 0, repeated sampling is useless, but a secure sketch can correct this error.

4.4.3 Best Process to Generate a Key from Response

We recommend that other cryptographic systems than RSA are used. The key generation for RSA is rather complex. While it is possible, the added complexity of generating two prime numbers from a response enlarges the surface where any mistake can completely neutralize any cryptographic properties. Over the years, many different attacks were developed against weak RSA keys. [16, 22]

On the other hand, a private key in elliptic curve cryptography (ECC) is a simple integer. We only need to select a number that is inside a certain range. This provides an undoubtable advantage. ECC also provides other advantages. In contrast with RSA, the encryption, decryption and sign operations are easier to calculate, and the keys – and subsequently certificates – are smaller. [23]

If the used physical unclonable function demonstrates a sufficient unclonablility and the response is at least the same length as is the key length, we can recommend to use the response – without any additional processing – as a key inside an ECC cryptographic system.

Chapter 5

Transport Layer Security on ESP32

In this chapter, we explore different options to support Transport Layer Security (TLS) on ESP32. We weigh the positives and negatives of all options and then choose one that is the best match for our use case.

5.1 Transport Layer Security

Transport Layer Security, commonly referred to as TLS, is a security protocol used to provide security in network transmissions. It creates a secure connection between two parties across a computer network. It can be used to encrypt any number of protocols (e.g. email, Voice over Internet Protocol) to securely transport information across computer networks. TLS is best known for its use with Hypertext Transport Protocol (HTTP). Together they form Hypertext Transport Protocol Secure (HTTPS) which is used all over Internet. The HTTPS protocol is used to encrypt communication between web browser and web server, for example between you and your bank. [24]

Transport Layer Security is a evolution of a protocol known as Secure Sockets Layer (SSL). While the SSL protocols are almost never used nowadays, the term SSL is still commonly used to refer to TLS. [25]

5.1.1 Functions of TLS

A Transport Layer Security is utilized for 3 main reasons [24]:

1. *encryption*: All communication inside TLS is encrypted to hide it from adversaries.
2. *authentication*: A mechanism exists to authenticate – confirm that they are who they say they are – both sides of connection.
3. *integrity*: Any modifications and/or damage to the transmitted data can be detected.

The encryption and integrity functions are simple to imagine, at least if you know how common encryption algorithms work. However how do we authenticate someone across the network?

If we wish to authenticate using TLS we can use something called a *certificate* [24]. A certificate contains information (e.g. name, address, domain) and a public key of the subject.

Certificates are generally issued by certificate authority (CA). The CA signs the certificate using asymmetric cryptography introduced in subsection 4.1.2. [24, 26]

The CA guarantees that the information stored in the certificate is valid for the subject that has the private-public key pair, from which the public part is stored inside the certificate. If we decide to trust the CA, we can authorize the subject in the certificate. [24, 26]

5.1.2 TLS Version 1.3

Multiple versions of both SSL and TLS exist. A newest version[1] is TLS 1.3 from August of 2018, standardized as RFC 8446. TLS 1.3 is a large step forward from the previous versions. Features that are not commonly used or contain known vulnerabilities are removed to improve the security. Faster and more effective mechanism to establish a secure connection is used. All with the aim to improve performance, simplify the protocol and improve security. [25]

In a recommendations published by Mozilla Organization [27] no versions before TLS 1.2 is recommended to be used. If the service is expected to communicate with modern devices only and/or an high level of security is desired, TLS version 1.3 should be used exclusively.

It must also be noted that another configuration, apart from version selection, should be applied. Namely the used ciphers and authentication mechanisms should be limited to those with good support, security and no known vulnerabilities. [27]

Significant improvement in terms of security in TLS 1.3 is the mandatory use of forward secrecy (apart from few specific instances). This change means that, even in the case of server private key disclosure to the adversary, no previous transmissions can actually be decrypted by the adversary. To decrypt a transmissions a specific one time key needs to be discloses. Those keys are however discarded after the communication ends. [28]

5.2 Self Implementation of TLS

First idea might to simply implement Transport Layer Security on our own. Let us suppose that we would like to implement TLS 1.3. The rationale behind this choice would be quite simple. It does not make much sense to develop complicated library for a protocol that is already being replaced.

The starting point would be RFC 8446 [28], this document outlines all the required components that together create Transport Layer Protocol version 1.3. It defines the handshake protocol, possible extensions, what cryptographic components are used for authentication and encryption. It also references many different standards that we would need to implement also.

Any mistake, in any component, could lead to a security incident. In which private data might be disclosed, or a adversary might take over the control of any device that is using our implementation. Even commonly used TLS libraries as OpenSSL commonly suffer from vulnerabilities. Over the last 20 years that OpenSSL existed, over 200 vulnerabilities were found [29].

Developing your own TLS library might certainly be possible, however the dangers and complexity it brings does not make it practical. A possible compromise might be to create your own version of open source library (given that proper license was followed), to customize the library for your purposes.

5.3 Available Libraries

If we wish to use Transport Layer Security on ESP32, but we do not want to go to all the trouble of implementing our own TLS library, we can turn to one of the already existing solutions.

[1] at the time of writing

Simple comparison between libraries can be found in Table 5.1.

5.3.1 ESP-TLS

ESP-TLS is a component provided as part of ESP-IDF development framework by Espressif Systems (Shanghai) Co., Ltd [30].

It provides a very limited interface for Transport Layer Security. It uses Mbed SSL (default) or wolfSSL as underlying layer. [30] This abstraction layer is available under Apache 2.0 license [31].

While this might be useful for simple applications that require TLS, and therefore do not use any special features, for any more in-depth applications it does not provide much benefits.

5.3.2 WolfSSL

WolfSSL [32] developed by wolfSSL Inc. is an open source TLS library targeted at low power and embedded devices.

The advantages highlighted by the developers include, but are not limited to: high portability, small size, support for modern protocol and ciphers and Federal Information Processing Standards certifications. [32]

WolfSSL provides a comprehensive documentation both as learning materials and function documentation. Both are accessible online, with support for generating your own via Doxygen. Both forum and direct support contact are also provided. [33]

5.3.3 Mbed TLS

Mbed TLS developed by TrustedFirmware is a small TLS library. Library was previously known as PolarSSL and was maintained by Arm. [34]

Since the inclusion of Mbed TLS in the Trusted Firmware project in 2020[2] [35], a new major (and multiple minor) version were published. [36]

"Mbed TLS provides a minimum viable implementation of the TLS 1.3 protocol" [37]. The TLS 1.3 support is not complete, however the set of implemented features is growing fast. [37]

The state of documentation for this library is less than ideal. Documentation hosted by Arm [38] is still accessible, however the documented version (2.16.1) is quite behind the newest 3.1.0. For example the documented version does not provide any TLS 1.3 support. On the other side, documentation hosted by Trusted Firmware is *still* not available [34].

All the documentation is provided in the form of Doxygen, as part of the source code, it means that if we wish to have up to date documentation available, we need to download the source code and generate it ourself. [39]

5.3.4 BearSSL

BearSSL developed by Thomas Pornin [40] is a small TLS library aimed solely on small and embedded systems.

The library is published under MIT license and is still considered in *beta*. It does however support client and server side TLS 1.0, TLS 1.1 and TLS 1.2. Server and client side certificates are implemented including simple validation. This library operates without the need for dynamic memory allocation. [40]

A TLS 1.3 support is underway. However a estimated time of implementation is not provided. [41]

[2] almost 2 years at the time of writing

5.3.5 CycloneSSL

CycloneSSL developed by Oryx Embedded [42] is a ANSI C compliant TLS library.

The library is published under GPLv2 license. It supports TLS versions 1.0 to 1.3 on both server and client with wide range of ciphers to choose from (including Suite B). The library does not use any processor depended code. [42]

Table 5.1 Comparison between TLS libraries on ESP32

Library	License[a]	TLS support	Certifications	Days since last release[b]
wolfSSL	GPLv2	1.3	DO-178[c], FIPS 140-2[c]	2
Mbed TLS	Apache 2.0	experimental 1.3	No[d]	139
BearSSL	MIT	1.2	No	1360
CycloneSSL	GPLv2	1.3	No	92

[a] Commercial licenses are also available.
[b] As of 2022-05-05. [43, 36, 40, 44]
[c] Only for wolfCrypt component. [45]
[d] The official FIPS 140-2 tests are included in automatic testing. [46]

5.4 The Best TLS Library?

Based on the information we gathered in section 5.3 we carefully considered our choice for TLS library.

While all options would be potentially suitable, we want to target TLS version 1.3 to demonstrate the usability in modern protocols. This allows this work to remain relevant for years to come. Another point to consider it a license. We wanted to use an open source library, however the license does not make much of a difference for us.

wolfSSL is quite popular on the ESP32 platform. This is important in the development phase, since many resources are available. It also supports a TLS 1.3.

Mbed TLS is probably the most popular on ESP32 (maybe due to its default inclusion in ESP-TLS). However its support of TLS 1.3 is somewhat lacking. It is also released under the Apache 2.0 license.

BearSSL does not unfortunately support TLS 1.3. The development is also rather slow. However it is released under MIT license if that is something that your project requires.

CycloneSLL does support TLS 1.3 and is released under GPLv2 license.

Mbed TLS might be a suitable open source option, if a GPLv2 license, or other commercial license options, do not suit your needs. However we decided to use *wolfSSL* for the TLS 1.3 support, active development and support it offers.

Chapter 6

Design of a Key Enrollment Procedure

In this chapter, we design a key enrollment procedure. This allows us to use private keys, that were generated on the device, within TLS to facilitate authentication.

Transport Layer Security encrypts traffic between two parties with the aim of creating a secure communication channel. It also support an authorization using TLS certificates.

We designed a key enrollment procedure to allow us to authorize the device, within already established system, using industry standard certificate mechanisms. The device creates a certificate signing request (CSR) that can be used to receive a certificate from a certificate authority.

The EPS32 is generally not used as end user device and thus do not posses any human usable interface (e.g. keyboard, screen). For this reason, another device is needed to "talk" with the device. While we intended for the device to be a computer, the protocol is device independent.

6.1 Aims of the Procedure

When designing the procedure we wanted to keep it simple.

Simple means easier implementation, which in turn leads to safer and less error prone code. While more sophisticated procedure could implement a error recovery or modification subroutines we did not think that such features would provide significant benefit.

The enrollment procedure is most likely done only once (over the life of the device). And since it is not a complicated process, it does not, in our opinion, warrant the increased complexity that would additional features bring. If at any phase a error occurs, it is simply a matter of resetting the device and repeating the procedure.

6.2 Communication Medium

The ESP32 platform itself offers multiple possible communication mediums.

There are possibly 3 main options: Bluetooth, Wi-Fi, and Universal asynchronous receiver-transmitter (UART) [47]. Other options exist, but most of them would require some development work and/or additional hardware.

A care must be taken to secure this medium against a man in the middle attack. If an adversary takes over the communication, he can create his own certificate signing request. If this CSR then gets signed it allows the adversary to impersonate the device. While this device

would be then unusable (the devices public keys would not match the public key stored inside the certificate), it still allows the adversary to have a valid certificate.

We therefore recommend the use of a UART interface. The interface is accessible only locally, so the adversary would require a physical access (or a compromised computer on the other side) to the device at the time of the enrollment.

A significant advantage in favor of UART is the presence of a USB to serial converter on most ESP32 development boards. This is mainly useful in the development phase. However a single affordable external USB to serial converter can be used for numerous deployment, leading to no significant additional cost for a manufactured unit. However, as noted previously, this enrollment procedure can be – with sufficient precautions – implemented over any medium of choice.

6.3 Procedure

You can find a diagram of the procedure in figure 6.1. In the following subsection we describe each phase in depth.

Since the aim of the procedure was to be simple, we did not want to require and significant software on the opposite computer side. We therefore decided to transfer all information as in text form. This way any serial monitor application could be used to enroll a device.

The format of the messages send from a device to a computer is inspired by the Privacy-Enhanced Mail (PEM) encoding format [48], that is commonly used with certificates and private keys. Every message is ended by a new line character ('\n'). The messages from the computer to the device are of two types. In the first section of the procedure a simple textual strings are transmitted. In the last section, an actual PEM encoded certificates are transmitted.

We call the single line messages from the device to the computer a header. All the headers look like this: "----- {PLACEHOLDER} -----". Where "{PLACEHOLDER}" is a variable length string of upper and lower case letters, numbers, spaces, and underscores.

All messages transmitted by the device are using an 8-bit ASCII encoding.

If at any stage in the enrollment procedure the devices encounters an error. It signals it via a "-----FAIL ENROLLMENT-----" header and terminates the procedure.

An example of full enrollment procedure between the device and the computer can be found in Appendix A.

6.3.1 Triggering Key Enrollment

For a security reasons the enrollment procedure must be triggered manually on the device. It is intended as a countermeasure to a remote denial of service. We suggest that a push button is installed, which is required to be pressed at the device startup, to enter the enroll procedure.

The device establishes connection across the selected medium (e.g. UART serial communication) and sends "-----BEGIN ENROLLMENT-----" header to signalize the start of the enrollment procedure.

6.3.2 Creating Certificate Signing Request

Now that the enrollment procedure is started, we need to get information from the computer to create the certificate signing request. Each CSR consists of multiple fields that need to be filled with information. The fields include identification information (e.g. country, organization, common name).

The device will send headers in the form of "-----INPUT {PLACEHOLDER}-----" where "{PLACEHOLDER}" is replaced with the requested field name. The field names adhere to "LDAP-NAME" of attributes defined in Section 6 of X.520 [49] recommendation. If multiple

versions are defined a longer version is used. All names are converted to upper case letters before usage. The order of fields is consistent with the ordering of X.520 [49].

If any additional fields are supported by the device, they are ordered lexicographically, using the most common field names, and filled last.

For example if fields country name, common name, and custom are supported, the headers are ordered as follows:

1. "-----INPUT COMMONNAME-----" as common name is defined first in Section 6.2.2 with two possible names (longer is used and capitalized).
2. "-----INPUT C-----" as country name is defined in Section 6.3.1.
3. "-----INPUT CUSTOM-----" as custom is not part of the recommendation.

After the header is sent, the device awaits a response. If the response is only an empty line that means that the field should remain unused.

After the response have been received it is validated against requirements defined in X.520 [49] (mainly maximal length and encoding). The process repeats until all supported fields are filled.

6.3.3 Exporting Certificate Signing Request

After all fields of the certificate signing request are filled the device assembles and completes the CSR. The fully prepared CSR is then transmitted as a PEM encoded multiline message. Afterwards all currently stored certificates are deleted from the system.

The message is, in accordance with RFC 7468 [48], started with header "-----BEGIN CERTIFICATE REQUEST-----" and ended with header "-----END CERTIFICATE REQUEST-----".

This certificate signing request can then be transmitted to certificate authority of choice. The CA then should perform signing a certificate with the rest of certificate chain of trust.

6.3.4 Receiving Certificate Chain

The device prompts for a count of certificates in the chain with header "-----INPUT CERTIFICATE COUNT-----". An positive whole number, larger than 0, is expected as a response.

A device now sends a header "-----INPUT CERTIFICATE CHAIN-----" prompting the computer to send the certificate chain. The certificate chain should consist of PEM encoded certificates ordered from the devices certificate to the root certificate.

Device reads all the certificates, the count was specified previously with the response to the INPUT CERTIFICATE COUNT message, and saves them for future use within TLS.

6.3.5 Validating Certificate Chain

Last part of the enrollment procedure is for the device validate that the certificate was intended for this device, a devices public key stored in the certificate should therefore be validated against devices private key.

If all the steps leading up to here and the public key check were completed successfully, the device sends last header "-----END ENROLLMENT-----" to signalize a successful enrollment.

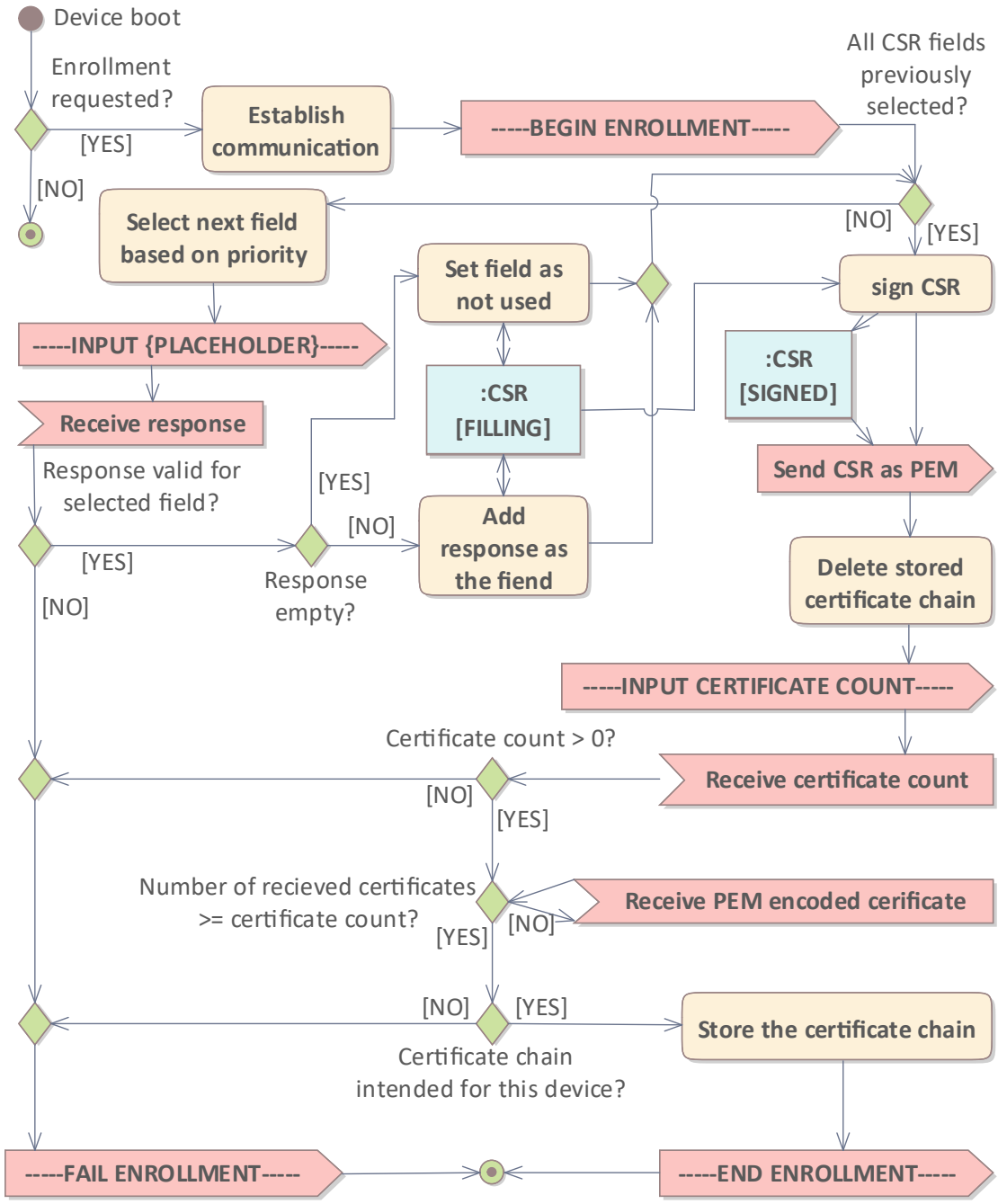

Figure 6.1 Enrollment activity diagram

Chapter 7
Implementation

In this chapter, we demonstrate the use of physical unclonable functions in Transport Layer Security on ESP32. We constructed a sample project that demonstrates the usability of such solution.

We are creating a proof of concept application to test the feasibility of using physical unclonable functions in Transport Layer Security on ESP32. As a TLS library we selected, in section 5.4, wolfSSL.

In this section, we sometimes include code listings. It should be noted that the code is usually without any error checking. It is also simplified, and some variable names are changed to be more readable in printed form.

7.1 ESP-IDF

For the development of the application, we decided to use Espressif IoT Development Framework [50] (ESP-IDF) provided by Espressif, the manufacturer of the ESP32 family of chips. ISP-IDF is the official framework for developing C (and C++) applications.

ESP-IDF is released under Apache 2.0 license and offers an Application Programming Interface (API) and a build system that allows to build, flash, and monitor the application. Numerous components are provided to simplify the development by providing functionality such as a Wi-Fi driver, networking protocols, file systems, and nonvolatile storage. [50, 51, 52]

The ESP-IDF is available as a command line tool, or it can be integrated with several of Integrated Development Environments (IDE) using a CMake suite. [52]

7.1.1 ESP-IDF Version

ESP-IDF currently has two major versions. The v4.4.1 is a stable version, and v5.0 is in development. [53]

We decided to use the newest possible version (v5.0-dev-1730-g229ed08484) at the time. This decision was made because we wanted it to be possible to use this application in the future with minimal effort. And since the new major version brings some API changes [54], we decided to use it even though it is still in development.

7.1.2 Installing WolfSSL

The application was developed using wolfSSL version 5.2.0. The library can be integrated as a component into ESP-IDF. [55]

When you download the library and enter `IDE/Espressif/ESP-IDF/` folder, you have a `setup.sh` and `setup_win.bat` scripts that transforms the library into an ESP-IDF component and moves it into standard component location. [55]

It should now be usable as the rest of the components. Unfortunately, it is not that simple. WolfSSL manages its build configuration via a `user_settings.h` header file [56].

For this configuration to work you need to [56]:

- place the header inside the include path of the application;
- include `wolfssl/wolfcrypt/settings.h` everywhere wolfSSL is used;
- define `WOLFSSL_USER_SETTINGS` preprocessor macro.

This, however presents some challenges when using wolfSSL as an ESP-IDF component. Because the component is stored with the rest of the components, it is outside of the application structure. However, we would like to keep the configuration as part of the project.

The simplest solution is to move the component inside the application structure. Now you can have the configuration versioned with the rest of the project. This, however requires you to modify the `CMakeList.txt` build configuration to work from its new location. It is simply a matter of replacing all relative single up directory (`..`) paths with paths that originate from ESP-IDF root (`${IDF_PATH}/components`).

It does not end there. Since every component is compiled separately, and the available wolfSSL features are configuration dependent, you need to make the same configuration file available inside every component. This is best achieved by modifying every component's `CMakeList.txt` to manually include the `user_settings.h` header.

7.2 Additional Components

While the main aim of this application is to demonstrate the use of physical unclonable functions, we also require some additional components that allow us to demonstrate it.

7.2.1 Access Point

We need a way to communicate with the device to establish a TLS connection. The ESP32 platform supports multiple networking options [57]. However, the most straightforward option is Wi-Fi.

The Wi-Fi component supports multiple modes. In a production environment, the device would most likely be connected to an already existing Wi-Fi network.

We decided to use the device as an access point. This allows us to create a local Wi-Fi network that other devices can connect to, where the TLS connection can be demonstrated and tested.

We created a new component called `accessPoint`. It allows us to use the configuration element of ESP-IDF, whereby using "menuconfig" the developer can choose the SSID name, password, and channel for the access point.

As you can see in Code listing 7.1[1], the process of creating an AP is quite simple. Keep in mind that this snippet does not contain any error checking, and some minor configuration options were omitted.

[1] snippet from `src/ESP32/components/accessPoint/accessPoint.c`

Code listing 7.1 Access point on ESP32

```
#include <esp_wifi.h>
...
void accessPointInit() {
    wifi_init_config_t initConfig = WIFI_INIT_CONFIG_DEFAULT();
    esp_wifi_init(&initConfig);      // Allocate resources for WiFi
    esp_wifi_set_mode(WIFI_MODE_AP); // Set WiFi to Soft AP

    wifi_config_t config = {
            .ap = {
                    .ssid = "SSID",
                    .password = "PSK",
                    .channel = 1,
                    .authmode = WIFI_AUTH_WPA2_PSK,
                    ...
            }
    };

    esp_wifi_set_config(WIFI_IF_AP, &config); // Configure settings
    esp_wifi_start(); // Start AP
}
```

7.2.2 Domain Name System

Now that we have networking capabilities, we can use the default IP address (192.168.4.1), to connect to the device.

But since we would like to demonstrate the TLS connection using a web browser over HTTPS, we would encounter a security warnings when connecting. The web browsers comply with RFC 2818 [58], which means that they compare information stored inside the certificate with the domain name of the server.

If present, the subject alternative name field of the certificate must be used to validate. If this extension is not present, then the common name is compared. [58]

Since wolfSSL does not support creating a certificate signing requests with subject alternative name extension [59], we will be using the common name for our purposes.

The only way to connect to the device is by using the IP address. While it would be possible to set the common name to the IP address, it is not the most elegant solution. We would like to use a domain name (e.g. pufintls.cz) instead of the IP address of the device. But how do we tell the browser to contact the device when we request the domain?

The solution to this is a protocol known as Domain Name System (DNS), which translates domains to IP addresses. ESP-IDF offers a component that implements multicast DNS (mDNS) [60]. However, mDNS is not supported on all platforms.

We decided to create a simple DNS server that emulates an RFC 1035 [61]. It is implemented inside the `accessPoint` component. It listens on port 53 and parses incoming messages. If the incoming message is a DNS IPv4 request for a domain that matches the common name of a certificate created in the last successful enrollment procedure, it responds with a device's IP address. Otherwise, a *not implemented* (for unsupported requests) or *refused* (for unknown domain names) DNS response is returned to signalize the requesting device that another DNS server should be contacted.

While this system should not be used in a production environment, it provides a simple mechanism for this proof of concept application.

7.2.3 Web Server

As we stated previously, we wanted to demonstrate the use inside HTTPS (where TLS is the underlying protocol), this meant that we needed a simple web server.

We decided to implement a simple server (inside `webserver_WolfSSL` component) that receives information from the TLS connection and responds with a simple web page. All the communication passes through a secure connection, establish with the help of the TLS library, that we later describe in section 7.5.

7.2.4 Mock Library Providing PUF

We needed a library that would simulate a physical unclonable function. We created a component `puflibMock` that behaves as an ideal PUF. At least in terms of evaluability and reproducibility. It simply returns a predefined response every time.

The library interface is copied from the library created by Ondřej Staníček [1] to facilitate drop in support at a later time.

7.3 Creating a Key

Based on our recommendations in section 4.4 we decided to use elliptic curve cryptography without any additional processing on the responses. Specially we decided on secp256r1[2] curve since is one of the supported in TLS 1.3 [28].

There is one main format for ECC keys in wolfSSL, and that is `ecc_key` object. A function `wc_ecc_import_private_key` offers the ability to import raw byte strings as a private part of the key. [62]

This function, unfortunately does not provide the ability to select a curve. If we, however, dive into the source code, we can see that it simply calls `wc_ecc_import_private_key_ex` with a preselected curve. [63]

We can therefore use this function with an error corrected response, provided by our PUF library, to generate a private-public key pair, as you can see in Code listing 7.2[3].

Code listing 7.2 Creating a key from response

```
ecc_key key;         // The key structure
wc_ecc_init(&key);

// Calculates how long the key needs to be
int len = wc_ecc_get_curve_size_from_id(ECC_SECP256R1);

... // Generate RESPONSE from PUF + validate that it is long enough

// Import the private key only
wc_ecc_import_private_key_ex(RESPONSE, len, NULL, 0, key, ECC_SECP256R1);

wc_ecc_make_pub(&key, NULL); // Generate a public key from private
if (wc_ecc_check_key(&key))  // Validate that keypair matches
    // KEY IS VALID
```

This `ecc_key` object can be used to generate the key and to generate a certificate signing request. Unfortunately wolfSSL can not use this key when creating a TLS connection. When

[2] also known as P-256 or prime256v1

[3] snippet from `src/ESP32/components/certificateHandler/certificateHandler.c`

establishing a TLS connection, only keys read from file and/or buffer are supported. Both case use either Distinguished Encoding Rules (DER) or Privacy-Enhanced Mail (PEM) encoding. [64]

Since the PEM format is simply a wrapper around the DER format, it makes sense to use the DER format internally.

The conversion from `ecc_key` to a DER encoded buffer is a simple matter of calling one function. `wc_EccKeyToDer(&key, KEY_DER, KEY_DER_SIZE)`. [59]

And now, we can create cryptographic keys that can be used inside TLS in ESP32.

7.4 Key Enrollment

In this section, we take a look at the implementation of the enrollment procedure we designed in chapter 6. The key enrollment is implemented inside `certificateHandler` component.

7.4.1 Communication Media

The ESP32 offers three UART ports using the ESP-IDF API [65]. We decided to use `UART_NUM_0`. This UART port uses GPIO pins 1 and 3 for communication [65]. They are usually connected to the onboard USB to serial adapter on most development boards. This allows us to enroll the device over the same connection used to flash it.

This however, presents some challenges if ESP-IDF logging library [66] is used. The library uses the same UART interface by default. This can prove a challenge if an automated script is used on the computer side.

Since the enrollment protocol messages, that are sent from the device, are quite unique and easy to filter, we did not consider this to be a problem for our application. The only significant problem might arise, when an enrollment message is interrupted by a log. This is mainly a problem for the message that sends the CSR. However, since the enrollment process is done at the start of the device, when only a limited number of processes is running, the probability of actual interrupted message is quite low.

The library supports multiple priority settings for each log entry. It is possible to disable all logging output using the *menuconfig* from ESP-IDF [66]. For production application, the logging would most likely be fully disabled or redirected to a log file.

Another logging output originates from the devices boot loader at startup. It is possible to disable this output by externally pulling pin GPIO 15 low. [67]

Once the UART connection is established, we can start communication with the other device. Unfortunately, the provided read and write functions are rather simple. When reading from the UART port, the function retrieves anything that is available. This means that multiple messages could be read with a single read function. [65]

A temporary buffer, that you can see in Code listing 7.3[4], is created to store the excess messages for later processing. We also created functions to read a line and send a message over the UART connection.

Code listing 7.3 UART buffer

```
typedef struct SUART_BUFFER {
    char * const m_buffer; // pointer of the buffer
    const int m_size; // total size of buffer
    int m_used; // current number of characters in buffer
} SUART_BUFFER;
```

[4] snippet from `src/ESP32/components/certificateHandler/include/certificateCommunication.h`

7.4.2 Triggering Key Enrollment

The first step in enrollment is to signal the device that this process should start. It can only be done right after the device booted.

A pin GPIO 16 was selected as an input for this purpose. Mainly because it was unused and because it does not offer any advanced features that could be used for something else. If a high signal (3.3V) is connected to this port at startup, the enrollment procedure starts.

As you can see in Code listing 7.4[5], we first need to designate this pin as an input. We can then use the internal pull down resistor to guarantee stable measurements when nothing is connected.

After that, it is a simple matter of checking the state of the pin. And then optionally starting the UART communication and enrollment process.

Code listing 7.4 A check for an enrollment trigger

```
gpio_set_direction(BUTTON_PIN, GPIO_MODE_INPUT); // set pin as output
// enable integrated pulldown resistor to combat noise
gpio_set_pull_mode(BUTTON_PIN, GPIO_PULLDOWN_ONLY);

if (gpio_get_level(BUTTON_PIN)) { // 1 => pin is HIGH (3.3V)
    configureUART()
    handleEnrollment();
    freeUART()
}

gpio_reset_pin(BUTTON_PIN); // reset pin to default settings
```

7.4.3 Creating Certificate Signing Request

WolfSSL provides an interface that can be used to create a certificate signing requests. [59] It requires us, as you can see in Code listing 7.5, to create a certificate object, fill the supported fields with information provided by the computer, and finally sign it.

Code listing 7.5 A check for an enrollment trigger

```
Cert CSR;
wc_InitCert(&CSR); // Create a empty certificate/CSR

// Fill fields with information provided by computer over UART
strcpy(CSR.subject.commonName, "COMMON NAME FROM UART");
CSR.subject.commonNameEnc = CTC_PRINTABLE; // Set the correct encoding
...

uint8_t DERBuffer[DER_SIZE];
// Creates a DER encoded CSR with public key from "key"
wc_MakeCertReq(&CSR, DERBuffer, DER_SIZE, NULL, &key);

WC_RNG rng; // RNG is required to generate a random seed for sign
wc_InitRng(&rng);
// And lastly we need to sign the CSR
wc_SignCert(CSR.bodySz, CTC_SHA256wECDSA, DERBuffer, DER_SIZE,
            NULL, &key, &rng);
```

[5] snippet from `src/ESP32/main/main.c`

The `DERBuffer` now contains a fully prepared CSR in DER format. We only need to transform it to PEM (with the help of wolfSSL function), and transport it over UART to the computer, where it will be signed.

7.4.4 Receiving Certificate Chain

Now that we successfully created and transmitted the CSR to the computer, we need to wait until the computer is ready to send back the certificate chain.

The first message we need to receive is the number (n) of certificates that are in the chain. This allows up to know when to stop reading. Since every PEM encoded certificate contains precisely 20 dashes, we can stop reading after we read $20 \cdot n$ dashes.

We need to be able to store those certificates even after the device turns off, ESP-IDF offers a non-volatile storage library [68] that stores a key-data pairs in flash memory. Unfortunately, the length of data is currently limited to 4000 bytes.

A better option is to use a SPIFFS file system [69] also provided by ESP-IDF. First, we need to create a partition table that allows us to assign a part of the device's flash memory to be used with the file system. In the Code listing 7.6[6] you can see the configuration. The relevant partition is called `storage` and is just under 1MB in size. This size was not chosen for any particular reason, it simply provides adequate space to store all the certificates.

Code listing 7.6 Partition table allocating space on ESP32

```
# Name,   Type, SubType, Offset,  Size,    Flags
nvs,      data, nvs,     0x9000,  0x50000,
phy_init, data, phy,            ,  0x1000,
factory,  app,  factory,        ,  2M,
storage,  data, spiffs,         ,  0xF0000,
```

Now that we have a space where the partition can be stored, we need to initialize it before use. The initialization you can see in Code listing 7.7[7] is simple. After that, we can simply use a standard C library function for working with files. [69]

Code listing 7.7 SPIFFS initialization

```
esp_vfs_spiffs_conf_t conf = {
  .base_path = "/spiffs",         // base path to partition
  // "spiffs" SubType from partition table will be used as a storage
  .partition_label = NULL,
  .max_files = 1
  .format_if_mount_failed = true // Format at first use (and on errors)
};
esp_vfs_spiffs_register(&conf);
```

7.4.5 Validating Certificate Chain

As the last part of the enrollment procedure, we check that the certificate matches our private key. WolfSSL provides a `wolfSSL_CTX_check_private_key(const WOLFSSL_CTX *ctx)` that can validate a private key on TLS context.

[6] snippet from `src/ESP32/partitions.csv`
[7] snippet from `src/ESP32/components/certificateHandler/certificateHandler.c`

7.4.6 Helper Script

We also created a Linux Bash script to communicate with the device using a USB to serial adapter. It uses the `stty` command [70] to configure the port, it then parses and forwards the communication.

It can be used to automatically sign the CSR, using a minimal CA application provided by OpenSSL [71], or manually via any certificate authority. It could also be modified to automate the signing process with another service.

7.5 Establishing TLS Connection

In figure 7.1 you can see the startup process for the application. It shows how our components cooperate when creating a TLS connection.

ESP-IDF offers implementation of TCP/IP stack [72] that we used in conjuncture with wolfSSL library to implement a TLS connection. We used a Berkeley socket interface to establish a base connection on which we built the TLS tunnel.

We implemented this inside procedure as part of `tlsAwaitConnect` component. All snippets in this section originate from `src/ESP32/components/tlsAwaitConnect/tlsAwaitConnect.c`.

As a first step of the process, we initialize and set up a *context* (`ctx`) object that wolfSSL uses to store a general configuration for the TLC connection. You can see the process in Code listing 7.8. We need to specify which TLS version to use and whether we will be a server or client. We can then import the certificate chain stored as part of the enrollment procedure in subsection 7.4.4. The server will transmit the certificate as part of the handshake process in an attempt to authorize itself.

Some additional configurations of the server can be performed here as well. We decided to limit the number of cipher suites – combinations of ciphers that secure the connection – to two of our favorites. This might be useful in the future if some exploit is discovered in supported ciphers. We can simply disable the affected cipher to completely neutralize the thread.

And last part of the configuration is to import the private key. WolfSSL supports two key inputs, from file and from DER encoded buffer [64]. Since we can retrieve the key from our PUF, it does not make much sense to save it to a file. We, therefore, generate the key and encode it into the DER format.

Code listing 7.8 Configuring a TLS context

```
// CTX stores general settings for multiple connections, we use TLS 1.3
WOLFSSL_CTX *ctx = wolfSSL_CTX_new(wolfTLSv1_3_server_method());

// Certificate chain from our SPIFFS file is used as authenticator
wolfSSL_CTX_use_certificate_chain_file(ctx, getCertificateFilename());
// We can also explicitly enable only some ciphers
wolfSSL_CTX_set_cipher_list(ctx,
                "TLS_CHACHA20_POLY1305_SHA256:TLS_AES_128_GCM_SHA256");

int bufLen; // key from PUF is imported to be used as part of handshake
const unsigned char *key = getKeyDER(&bufLen);
wolfSSL_CTX_use_PrivateKey_buffer(ctx, key, bufLen, SSL_FILETYPE_ASN1);
```

Now that our general configuration is done, we can work on creating a connection. As you can see in Code listing 7.9, we use Berkeley sockets to create a listening endpoint that accepts Transmission Control Protocol (TCP) connections on a specified port. This allows anyone to use a TCP connection on the port to reach our application.

Code listing 7.9 Creating a Berkeley socket

```
// create a endpoint for connections over IPv4 using TCP
int socketFD = socket(AF_INET, SOCK_STREAM, 0);

// This is a server side configuration
struct sockaddr_in serverAddress;
//initialize all the fields to 0
memset(&serverAddress, 0, sizeof(serverAddress));
serverAddress.sin_family = AF_INET;
// HostTONetwork Short converts endianness
serverAddress.sin_port = htons(port); // port to listen on
serverAddress.sin_addr.s_addr = INADDR_ANY; // use all interfaces

// configures the socket with the configuration
bind(socketFD, (struct sockaddr *)&serverAddress, sizeof(serverAddress));
listen(socketFD, maxConnections); // start listening for connections
```

Now the only thing remaining is to wait for some connection. In Code listing 7.10 we wait for a client to establish a TCP connection, we can then create a wolfSSL object (`ssl`) from the configuration we set up in context. This object can then be assigned to the TCP connection, and the TLS handshake can begin.

As soon as the TLS connection (on top of the TCP layer) is established, we can pass the TLS connection object to any application. It can be used by the application to securely communicate with the other side.

Code listing 7.10 Establishing TLS connection with key generated using PUF

```
// waits for incoming connection, then starts communication over TCP
int connectionFD = accept(socketFD, NULL, NULL);

// create a local TLS handler from context config
WOLFSSL *ssl = wolfSSL_new(ctx);
// use the handler on the accepted connection
wolfSSL_set_fd(ssl, connectionFD);
wolfSSL_accept(ssl); // TLS handshake and establish secure channel

callback(ssl); // start the application that uses secure connection

wolfSSL_shutdown(ssl); // we ended, send terminate
wolfSSL_free(ssl); // free all allocated resources
close(connectionFD); // close TCP connection
```

7.6 Test of Authentication and Connection Establishment

In this version, we describe what tests we performed to validate that the TLS connection was successfully completed. We firstly enrolled the device with a certificate signed by our helper script (it was signed using a local OpenSSL CA). Based on the test described below, we can confirm that the TLS connection was established successfully and that the device can be authenticated using certificates.

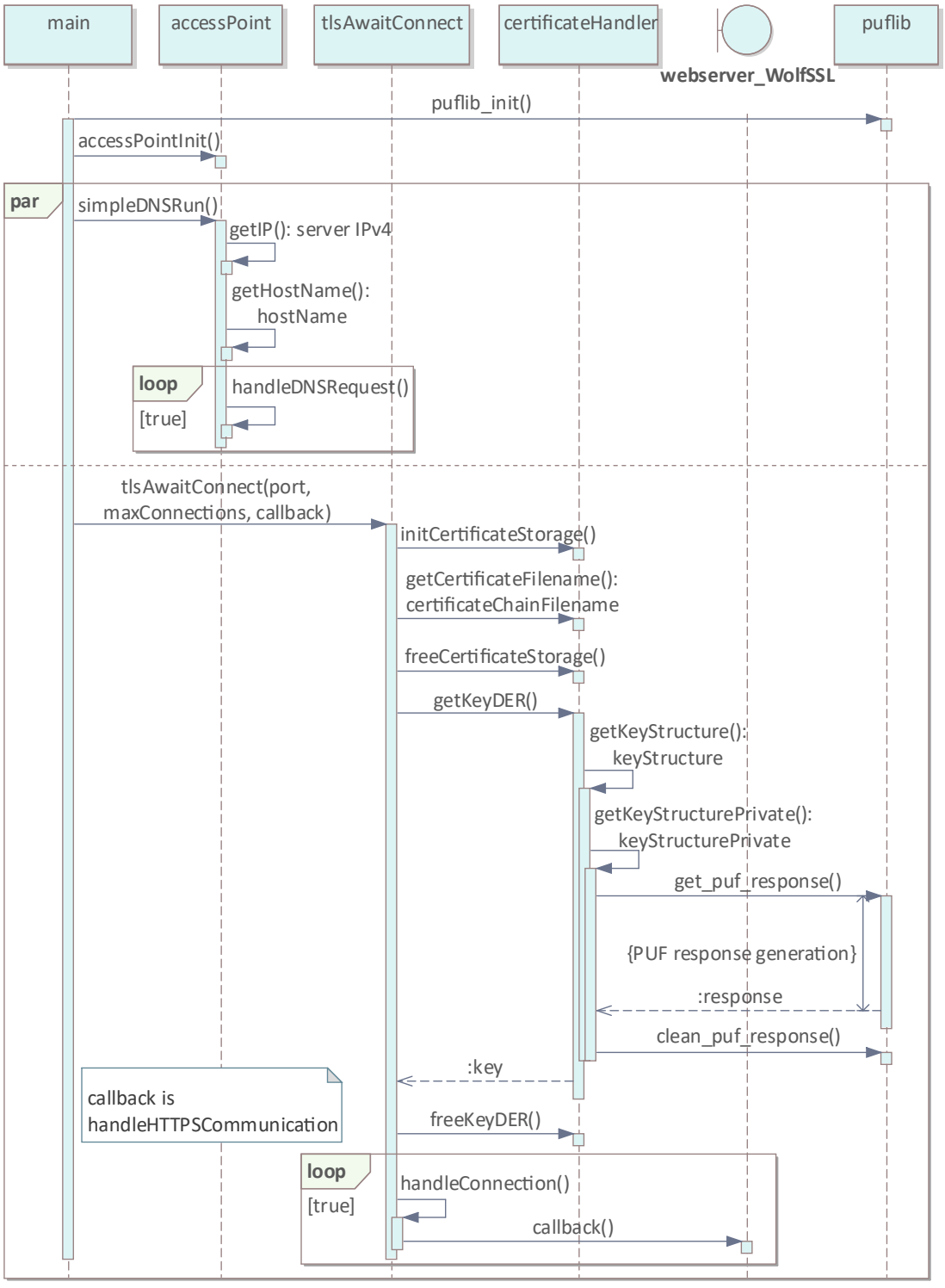

Figure 7.1 Sequence diagram of the standard startup

7.6.1 Web Browsers

This test was chosen specifically because it is simple to demonstrate. A web browser is present on most devices. It therefore allowed us to perform multiple tests from different devices.

As we have discussed previously in subsection 7.2.2, web browsers provide a simple demonstration. We tested multiple browsers (Mozilla Firefox, Google Chrome, Chromium, Safari) on multiple platforms (Linux, Windows, Android, iPad OS).

All the tested browsers (on all platforms) behaved similarly. Some privacy warnings were triggered occasionally

Bad certificate domain warning was displayed every time the browser connected using host name that did not match the common name of the certificate. This warning was not present when using the simple DNS implementation.

Unknown issuer warning is displayed because the browser does not trust the CA that issued the certificate. In figure 7.2 you can see, that we imported the root certificate, as a trusted CA, into the browser. This makes the warning disappear.

Subject alternative name mismatch was encountered only in Chromium. Certificates without alternative names are officially considered deprecated [58]. However, other browsers do not seem to have this warning.

Certificate validity too long was only encountered on iPad version of Chrome. We used a certificate with roughly 10 year validity.

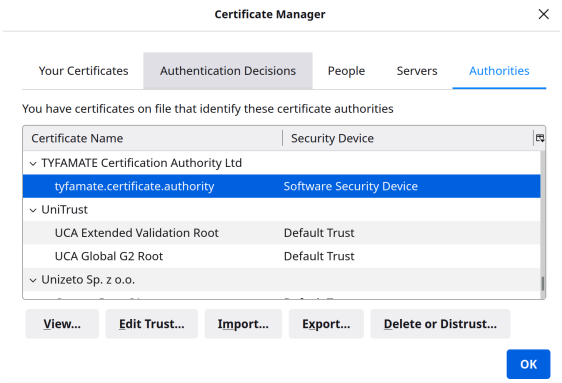

Figure 7.2 Managing trusted CAs in Firefox

Once we imported the root certificate into the browser as a trusted CA and started using the DNS, most browsers worked flawlessly. As you can see in figure 7.3, the connection was established successfully using TLS 1.3. All certificates in the chain were also transmitted.

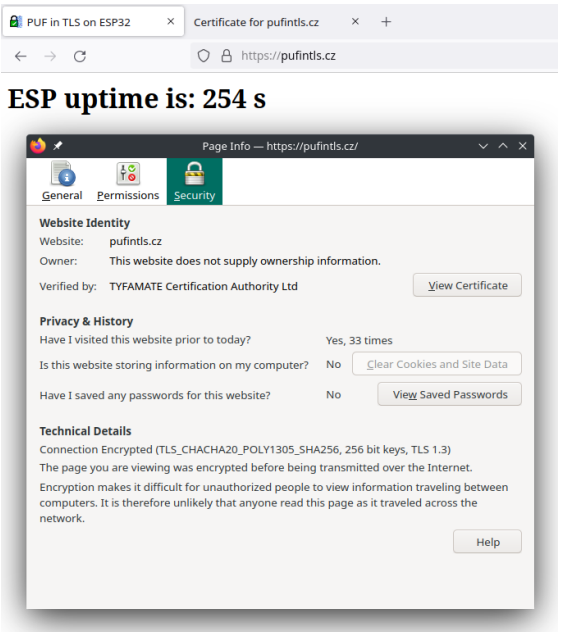

Figure 7.3 Firefox established a TLS connection

7.6.2 OpenSSL s_client

OpenSSL offers a s_client [73] application. It can be used to connect to a remote TLS server. The full output can be found in Appendix B. The application confirmed that the connection was established successfully and the root CA issued the provided certificate.

7.6.3 sslscan

A sslscan [74] is an application that scans a TLS server and displays supported versions, ciphers, and other configurations. You can find the full output in Appendix C.

The test confirmed that only TLS 1.3 was supported; it also confirmed our selection for supported cipher suites.

Chapter 8

Using an Actual PUF

In this chapter, we describe the process and obstacles to using an actual Physical Unclonable Function instead of mock-up.

As the last part of our application, we integrated a library created by Ondřej Staníček [75] that implements a SRAM PUF on ESP32.

The library is available on GitHub as an ESP-IDF component [75]; this allowed us to simply drop it it as another of our components.

The library uses non-volatile storage library we mentioned in subsection 7.4.4 to store the helper data. Since the amount of the data is not small, a modification to the partition table is required. We need to enlarge the *nvs* partition to roughly $300kB$. [75]

The library also requires that helper data are initialized when it is first used on a device. We included this initialization as a first step after the enrollment process is triggered.

8.1 Challenges with PUF Library Integrations

The library requires that an initialization function is called as the first function inside the main function. It also requires a modification to a wake up callback function. You can find the modifications inside `src/ESP32/main/main.c` file.

8.1.1 Incompatibility with ESP-IDF Version 5.0

We found out that the library does not include the correct header files. It attempts to use `ets_delay_us` function, that is included inside `rom/ets_sys.h` header on our IDF-ESP version.

We created a GitHub issue [76] to notify the developer of this issue. We fixed this issue for our component by modifying the library source code to include the correct header file.

8.1.2 Response not Available

In order to receive a PUF response the library requires that `get_puf_response` function is called. If the function returns successfully, it signalizes that the response was generated and is available.

It also contains `PUF_STATE` global state that should signalize `RESPONSE_READY` any time the response is generated.

Our testing discovered that the function would return success, but the global state would still signalize that the response is not ready. We believe that this is due to a `const` type specifier

inside the library header file. The compiler might have optimised away since the state should always remain the same.

We remedied this issue by removing the specifier from the header file and opening a GitHub issue [77].

8.1.3 Deep Sleep

Generally, the library can retrieve a response without a problem. However, in some circumstances, it is not possible to generate stable response with the device running. For those instances a `get_puf_response_reset` function is provided. It uses a deep sleep to generate the response. [75]

The problem comes from the deep sleep requirement. When the device wakes up from the deep sleep, it starts again in the main function. Most of the device's operation memory is also reset. A limited amount of memory can be retained after a deep sleep using a `RTC_DATA_ATTR` attribute. [78]

As you can see in Code listing 8.1[1], we use this attribute to store a state, that we can later use to restore the application after the deep sleep.

Code listing 8.1 Establishing TLS connection with key generated using PUF

```
enum EWakeupTargets { NONE, ENROLL, VALIDATE, WEBSERVER_INIT };
enum EWakeupTargets RTC_DATA_ATTR g_WakeupTarget;
```

Every time we generate a response using the library, we set the state so we can later resume where we left off. This, however, presents quite a few challenges.

Firstly all initialization needs to be redone. This means that it takes additional time to resume the operation. It also means that a complicated system of conditions needs to be implemented to initialized the right components for the corresponding wake up targets.

Secondly, any application data will be lost. We encountered this problem when generating a CSR in subsection 7.4.3. We wanted to generate the response as late as possible to limit the amount of time the response and key stay in memory. This, however meant that the CSR would be already filled when the response would be generated. We ended up saving the DER encoded CSR in a buffer[2] with the `RTC_DATA_ATTR` attribute.

8.2 Test of Authentication and Connection Establishment

We repeated our testing in section 7.6 with the real PUF library. The tests did not reveal any differences – apart from the used certificate and private key – between our mock and real physical unclonable function.

[1] snippet from `src/ESP32/globalWakeupState/globalWakeupState.h` and `src/ESP32/main/main.c`
[2] you can find the implementation inside `src/ESP32/components/certificateCommunication.c` file

Chapter 9

Usability of PUF in TLS on ESP32

The general aspect of using physical unclonable functions to generate keys for Transport Layer Security on the ESP32 platform is possible, as we demonstrated with our proof of concept application in chapter 7.

For an actual widespread deployment, one unknown remains. The construction of PUF plays a huge role in the usability of such solutions. While the library created by Staníček [75] is definitely usable, the deep sleep requirement make the process very complicated.

9.1 Speed of Response Generation

One difference between the speed of mock and real PUF was the amount of time the library required to create the helper data during provisioning. The real library required up to 30 seconds, in contrast to 0 seconds required for mock, to generate the helper data. Nevertheless, it is not an impassable obstacle since this is done only once per enrollment.

Another factor is the time it takes to generate a response. This is critical if we would like to use keys on demand. From our testing, the amount of time required to generate the response using the real PUF is in milliseconds. While this increase might be substantial in some circumstances, it does not mean much for our purposes. The post-processing we require to convert the key into something wolfSSL can use inside TLS takes so long that this slight increase is negligible.

9.2 Future Improvements

We wanted to address some possible future improvements – that we did not have space to cover in this work – but are interesting.

9.2.1 Another PUF Construction

In chapter 8 we used a SRAM based physical unclonable function. It is possible that other types of constructions would not be that complex to integrate.

The main integration complexity originated in the deep sleep requirement. If another construction, or even the same construction with some improvements, would not have the same requirements, it would be quite simple to integrate it.

9.2.2 Generating Private key on Demand in TLS

As you can see in Code listing 7.8, we decided to load the private key at server startup. This is not the most secure way of storing the private key. Ideally, the key would spend as little time in memory as possible. Since with that we add another barrier (triggering the key generation) before the attacker can even attempt to read the key from memory.

This decision was made for three reasons:

1. The key generation might take too long. The opposite device might think that the server is not responding, while it is only generating the keys, and close the connection. Or the delay might simply cause the application too unpleasant to use.
2. WolfSSL does not currently support an easy to use way of importing keys on demand. We found two possible solutions to achieve the same result. Using a callback to import the key prior to its use and another callback to later remove the key after it is used [79]. Another possibility is to load the key into every connection handler instead of the context [1] [64].
3. The deep sleep would require significantly more complex recovery. It would need to pause and resume every connection and every internal state of the application that is used over the connection. It would also possibly cause a denial of service for already connected clients.

[1] see section 7.5 for details on how context and connection handler are used

Chapter 10
Conclusion

The use of physical unclonable functions on low-power and low-cost systems on chip as ESP32 was not widely researched. The aim of this thesis was to define what is a physical unclonable function in the context of key generation and to implement a proof of concept application to demonstrate the use of PUF in Transport Layer Security on ESP32. The thesis also aimed to analyze the usability of real physical unclonable functions in those applications.

While precisely defining physical unclonable function might be almost impossible, we believe that our definition applies to PUF in the context of key generation.

The proposed enrollment procedure might not be optimized for automatic/batch enrollment. However, we feel that the disadvantages are outweighed by the ability to function over a standard serial interface in a human readable format without any additional software required.

Transport Layer Security is a highly utilized protocol. Both the defining standards and available libraries are still in active development. Because of this, the most suitable library is likely going to change. When implementing a TLS support in your application, it is recommended to analyze the available libraries according to your requirements.

Our development started with an ideal physical unclonable function that provided perfect responses in a minimal amount of time. This was done to demonstrate the best case scenario. The application utilized the generated private-public key pair in a TLS connection. We then demonstrated that a secure connection was established and that such applications could successfully developed and used.

Lastly, we adapted an actual physical unclonable function to use in our application. To analyze how the real PUF influenced the usability of our application with its less than ideal characteristics.

Our proof of concept application lives up to its name. We successfully integrated – both mock and real – physical unclonable function into Transport Layer Security on ESP32. With this, we proved that it is possible to utilize PUF for such purposes.

However, the challenges we encountered when integrating a real physical unclonable function into our proof of concept application caused a significant problem. No perfect solution exists to resume the application after a deep sleep was performed to generate the required key. This, in our opinion, makes it quite hard to fully utilize the potential of this library in TLS on ESP32.

Suppose more (hopefully easy to use) implementations of physical unclonable functions are developed, and TLS libraries improve their support for on demand key loading. In that case, it is feasible that such solutions can see widespread adoption.

Appendix A

Example of the Enrollment Procedure

This is an example of enrollment procedure where the device (on the left) communicates with a computer (on the right). As you can see the enrollment procedure was completed successfully.

```
 MESSAGES FROM DEVICE:                              :MESSAGES FROM COMPUTER
=========================================================================
-----BEGIN ENROLLMENT-----
-----INPUT COMMONNAME-----
                                                                pufintls.cz
-----INPUT SERIALNUMBER-----
                                                             DEADBEEFCAFE
-----INPUT C-----
                                                                       CZ
-----INPUT L-----
                                                                 Prague 6
-----INPUT ST-----
                                                  Capital City of Prague
-----INPUT STREET-----
                                                             Thákurova 9
-----INPUT O-----
                                                                FIT CVUT
-----INPUT OU-----
                                                                  BI-BIT
-----INPUT POSTALCODE-----
                                                                  160 00
-----INPUT MAIL-----
                                                 tyfamate@fit.cvut.cz
-----INPUT CACHALLENGE-----
                                                             CA_challange
-----BEGIN CERTIFICATE REQUEST-----
MIIBuDCCAV8CAQIwgd8xCzAJBgNVBAYTAkNaMR8wHQYDVQQIExZDYXBpdGFsIENp
dHkgb2YgUHJhZ3V1MRUwEwYDVQQJDAxUaMOha3Vyb3ZhIDkxETAPBgNVBAcTCFBy
YWd1ZSA2MREwDwYDVQQKEwhGSVQgQ1ZVVDEPMA0GA1UECxMGQkktQklUMRQwEgYD
VQQDEwtwdWZpbnRscy5jejEVMBMGA1UEBRMMREVBREJFRUZDQUZFMQ8wDQYDVQQR
EwYxNjAgMDAxIzAhBgkqhkiG9w0BCQEWFHR5ZmFtYXR1QGZpdC5jdnV0LmN6MFkw
EwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEgXiYVB6gNXesoEdqsFVedeAt58PN89Tz
FbKdnIhpVRD8oUAVJ6F9/x4WSBXcFDteixr9evTaAMJsDbUbe/fxoqAdMBsGCSqG
SIb3DQEJBzEODAxDQV9jaGFsbGFuZ2UwCgYIKoZIzj0EAwIDRwAwRAIgG3+rHPz7
```

```
evGiFJCIIHJDFqjNRO5HxMqFkl5XbBzChiUCIGDZY3oEnIpMbnLcsdFwYb4lp17n
HGH4rJA2oEqSUg5y
-----END CERTIFICATE REQUEST-----
-----INPUT CERTIFICATE COUNT-----
                                                                1
-----INPUT CERTIFICATE CHAIN-----
                                          -----BEGIN CERTIFICATE-----
        MIIEGzCCAgMCAhACMA0GCSqGSIb3DQEBCwUAMIHPMQswCQYDVQQGEwJDWjEPMA0G
        A1UECAwGUHJhZ3VlMREwDwYDVQQHDAhQcmFndWUgNjEtMCsGA1UECgwkVFlGQU1B
        VEUgQ2VydGlmaWNhdGlvbiBBdXRob3JpdHkgTHRkMRUwEwYDVQQLDAxDQSBSb290
        IHVuaXQxJzAlBgNVBAMMHnR5ZmFtYXRlLmNlcnRpZmljYXRlLmF1dGhvcml0eTEt
        MCsGCSqGSIb3DQEJARYedHlmYW1hdGVAY2VydGlmaWNhdGUuYXV0aG9yaXR5MB4X
        DTIyMDUwODIxMjMzMVoXDTMyMDUwNTIxMjMzMVowgaAxCzAJBgNVBAYTAkNaMR8w
        HQYDVQQIExZDYXBpdGFsIENpdHkgb2YgUHJhZ3VlMREwDwYDVQQHEwhQcmFndWUg
        NjERMA8GA1UEChMIRklUIENWVVQxDzANBgNVBAsTBkJJLUJJVDEUMBIGA1UEAxML
        cHVmaW50bHMuY3oxIzAhBgkqhkiG9w0BCQEWFHR5ZmFtYXRlQGZpdC5jdnV0LmN6
        MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEgXiYVB6gNXesoEdqsFVedeAt58PN
        89TzFbKdnIhpVRD8oUAVJ6F9/x4WSBXcFDteixr9evTaAMJsDbUbe/fxojANBgkq
        hkiG9w0BAQsFAAOCAgEAyiVG8+asEdiWj4HmyEVMpiTE7MAs0CCqNmdaBIK2q6MO
        NgXbcYUW2fRCk7gzfhFD536jPGa+rmVIwc4i9HMeznexTinakpYnLQ+MklIAFg2U
        zB9TSr38YRaCyOHIKiIwTxc1l6BDwd6t06k8uWjmfDzzAuzks2OSmLWGTb3rUi++
        m+1ThALLWEOMJx0QikS6rY5pqaNo6KHPpQy0AiOewGgCGZXyaSeM61qt8Ec8Zvpa
        RBVpFTv+94chqfvSYJiAixgd6RWRWqs+z51XD1oFhSp48RLBUmFQb+7yVWxJJZPW
        +nnCJ1hXcNoRyutEOHwL+Chmt2l7fY1CxOO5c1xX1V+9t7tTosVmeItvoJ6i1BCx
        TFn8svOrQJrP9hFu8zbnxN3HKhGUZn9PbNgKSC+C3m9x1Dt8og7eXLb8sAEMJLeD
        W74kVnQ/VrP/YerVAPOSauDSyQWpqN0VyY5bAHpLjuh0zosQexa7Ah6B8+k3tsOy
        chdA2g3y5mZcniBNwmpzbprTih0JF6ulzSrx3Yk7sPYNztYl6AN+PnAF/kXe49kn
        hUip1hDkGf4tEtR5p0uU+dBEIrSuwz3H1JAyHlha2Rajmk4nVyjUb2M/MMzdbnwg
        HsrDYqp4Fu/8rlhkyws1SQGOwQ7H83DOIVBvkwNtB10hpxsilVQshe+EduSsoBI=
                                          -----END CERTIFICATE-----
-----END ENROLLMENT-----
```

Appendix B

Testing TLS Using the s_client Application

Here you can see the output provided by s_client [73] application. The application was instructed to connect to the device on port 443, it also was provided with a root certificate to validate the certificate provided by the device.

```
> openssl s_client --connect pufintls.cz:443 -CAfile CAcert.pem

CONNECTED(00000003)
depth=1 C = CZ, ST = Prague, L = Prague 6, O = TYFAMATE Certification
    Authority Ltd, OU = CA Root unit, CN = tyfamate.certificate.authority
    , emailAddress = tyfamate@certificate.authority
verify return:1
depth=0 C = CZ, ST = Capital City of Prague, L = Prague 6, O = FIT CVUT,
     OU = BI-BIT, CN = pufintls.cz, emailAddress = tyfamate@fit.cvut.cz
verify return:1
---
Certificate chain
 0 s:C = CZ, ST = Capital City of Prague, L = Prague 6, O = FIT CVUT, OU
      = BI-BIT, CN = pufintls.cz, emailAddress = tyfamate@fit.cvut.cz
   i:C = CZ, ST = Prague, L = Prague 6, O = TYFAMATE Certification
       Authority Ltd, OU = CA Root unit, CN = tyfamate.certificate.
       authority, emailAddress = tyfamate@certificate.authority
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIEGzCCAgMCAhACMA0GCSqGSIb3DQEBCwUAMIHPMQswCQYDVQQGEwJDWjEPMA0G
A1UECAwGUHJhZ3VlMREwDwYDVQQHDAhQcmFndWUgNjEtMCsGA1UECgwkVFlGQU1B
VEUgQ2VydGlmaWNhdGlvbiBBdXRob3JpdHkgTHRkMRUwEwYDVQQLDAxDQSBSb290
IHVuaXQxJzAlBgNVBAMMHnR5ZmFtYXRlLmNlcnRpZmljYXRlLmF1dGhvcml0eTEt
MCsGCSqGSIb3DQEJARYedHlmYW1hdGVAY2VydGlmaWNhdGUuYXV0aG9yaXR5MB4X
DTIyMDUwODIxMjMzMVoXDTMyMDUwNTIxMjMzMVowgaAxCzAJBgNVBAYTAkNaMR8w
HQYDVQQIExZDYXBpdGFsIENpdHkgb2YgUHJhZ3VlMREwDwYDVQQHEwhQcmFndWUg
NjERMA8GA1UEChMIRklUIENWVVQxDzANBgNVBAsTBkJJLUJJVDEUMBIGA1UEAxML
cHVmaW50bHMuY3oxIzAhBgkqhkiG9w0BCQEWFHR5ZmFtYXRlQGZpdC5jdnV0LmN6
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEgXiYVB6gNXesoEdqsFVedeAt58PN
89TzFbKdnIhpVRD8oUAVJ6F9/x4WSBXcFDteixr9evTaAMJsDbUbe/fxojANBgkq
hkiG9w0BAQsFAAOCAgEAyiVG8+asEdiWj4HmyEVMpiTE7MAs0CCqNmdaBIK2q6MO
NgXbcYUW2fRCk7gzfhFD536jPGa+rmVIwc4i9HMeznexTinakpYnLQ+MklIAFg2U
```

```
zB9TSr38YRaCyOHIKiIwTxc1l6BDwd6t06k8uWjmfDzzAuzks2OSmLWGTb3rUi++
m+1ThALLWEOMJx0QikS6rY5pqaNo6KHPpQy0AiOewGgCGZXyaSeM61qt8Ec8Zvpa
RBVpFTv+94chqfvSYJiAixgd6RWRWqs+z51XD1oFhSp48RLBUmFQb+7yVWxJJZPW
+nnCJ1hXcNoRyutEOHwL+Chmt2l7fY1CxOO5c1xX1V+9t7tTosVmeItvoJ6i1BCx
TFn8svOrQJrP9hFu8zbnxN3HKhGUZn9PbNgKSC+C3m9x1Dt8og7eXLb8sAEMJLeD
W74kVnQ/VrP/YerVAPOSauDSyQWpqN0VyY5bAHpLjuh0zosQexa7Ah6B8+k3tsOy
chdA2g3y5mZcniBNwmpzbprTih0JF6ulzSrx3Yk7sPYNztYl6AN+PnAF/kXe49kn
hUip1hDkGf4tEtR5p0uU+dBEIrSuwz3H1JAyHlha2Rajmk4nVyjUb2M/MMzdbnwg
HsrDYqp4Fu/8rlhkyws1SQGOwQ7H83DOIVBvkwNtB10hpxsilVQshe+EduSsoBI=
-----END CERTIFICATE-----
subject=C = CZ, ST = Capital City of Prague, L = Prague 6, O = FIT CVUT,
     OU = BI-BIT, CN = pufintls.cz, emailAddress = tyfamate@fit.cvut.cz

issuer=C = CZ, ST = Prague, L = Prague 6, O = TYFAMATE Certification
    Authority Ltd, OU = CA Root unit, CN = tyfamate.certificate.authority
    , emailAddress = tyfamate@certificate.authority

---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: ECDSA
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 1404 bytes and written 377 bytes
Verification: OK
---
New, TLSv1.3, Cipher is TLS_CHACHA20_POLY1305_SHA256
Server public key is 256 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
---
DONE
```

Appendix C

Testing TLS using the sslscan application

Here you can see the output provided by sslscan (version 2.0.13) [74] application. The application was instructed to connect using IPv4, print used certificates, and show times of handshakes.

```
> sslscan -4 --show-certificate --show-times pufintls.cz

Version: 2.0.13
OpenSSL 1.1.1n  15 Mar 2022

Connected to 192.168.4.1

Testing SSL server pufintls.cz on port 443 using SNI name pufintls.cz

  SSL/TLS Protocols:
SSLv2     disabled
SSLv3     disabled
TLSv1.0   disabled
TLSv1.1   disabled
TLSv1.2   disabled
TLSv1.3   enabled

  TLS Fallback SCSV:
Server supports TLS Fallback SCSV

  TLS renegotiation:
Session renegotiation not supported

  TLS Compression:
OpenSSL version does not support compression
Rebuild with zlib1g-dev package for zlib support

  Heartbleed:
TLSv1.3 not vulnerable to heartbleed

  Supported Server Cipher(s):
Preferred TLSv1.3  256 bits  TLS_CHACHA20_POLY1305_SHA256  Curve 25519
    DHE 253 1911ms
Accepted  TLSv1.3  128 bits  TLS_AES_128_GCM_SHA256        Curve 25519
```

```
    DHE 253 1924ms

  Server Key Exchange Group(s):
TLSv1.3  81 bits  sect163k1
TLSv1.3  81 bits  sect163r1
TLSv1.3  81 bits  sect163r2
TLSv1.3  96 bits  sect193r1
TLSv1.3  96 bits  sect193r2
TLSv1.3  116 bits  sect233k1
TLSv1.3  116 bits  sect233r1
TLSv1.3  119 bits  sect239k1
TLSv1.3  141 bits  sect283k1
TLSv1.3  141 bits  sect283r1
TLSv1.3  204 bits  sect409k1
TLSv1.3  204 bits  sect409r1
TLSv1.3  285 bits  sect571k1
TLSv1.3  285 bits  sect571r1
TLSv1.3  80 bits  secp160k1
TLSv1.3  80 bits  secp160r1
TLSv1.3  80 bits  secp160r2
TLSv1.3  96 bits  secp192k1
TLSv1.3  96 bits  secp192r1
TLSv1.3  112 bits  secp224k1
TLSv1.3  112 bits  secp224r1
TLSv1.3  128 bits  secp256k1
TLSv1.3  128 bits  secp256r1 (NIST P-256)
TLSv1.3  192 bits  secp384r1 (NIST P-384)
TLSv1.3  260 bits  secp521r1 (NIST P-521)
TLSv1.3  256 bits  brainpoolP512r1
TLSv1.3  128 bits  x25519
TLSv1.3  224 bits  x448
TLSv1.3  112 bits  ffdhe2048
TLSv1.3  128 bits  ffdhe3072
TLSv1.3  150 bits  ffdhe4096
TLSv1.3  175 bits  ffdhe6144
TLSv1.3  192 bits  ffdhe8192

  SSL Certificate:
    Certificate blob:
-----BEGIN CERTIFICATE-----
MIIEGzCCAgMCAhACMA0GCSqGSIb3DQEBCwUAMIHPMQswCQYDVQQGEwJDWjEPMA0G
A1UECAwGUHJhZ3VlMREwDwYDVQQHDAhQcmFndWUgNjEtMCsGA1UECgwkVFlGQU1B
VEUgQ2VydGlmaWNhdGlvbiBBdXRob3JpdHkgTHRkMRUwEwYDVQQLDAxDQSBSb290
IHVuaXQxJzAlBgNVBAMMHnR5ZmFtYXRlLmNlcnRpZmljYXRlLmF1dGhvcml0eTEt
MCsGCSqGSIb3DQEJARYedHlmYW1hdGVAY2VydGlmaWNhdGUuYXV0aG9yaXR5MB4X
DTIyMDUwODIxMjMzMVoXDTMyMDUwNTIxMjMzMVowgaAxCzAJBgNVBAYTAkNaMR8w
HQYDVQQIExZDYXBpdGFsIENpdHkgb2YgUHJhZ3VlMREwDwYDVQQHEwhQcmFndWUg
NjERMA8GA1UEChMIRklUIENWVVQxDzANBgNVBAsTBkJJLUJJVDEUMBIGA1UEAxML
cHVmaW50bHMuY3oxIzAhBgkqhkiG9w0BCQEWFHR5ZmFtYXRlQGZpdC5jdnV0LmN6
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEgXiYVB6gNXesoEdqsFVedeAt58PN
89TzFbKdnIhpVRD8oUAVJ6F9/x4WSBXcFDteixr9evTaAMJsDbUbe/fxojANBgkq
hkiG9w0BAQsFAAOCAgEAyiVG8+asEdiWj4HmyEVMpiTE7MAs0CCqNmdaBIK2q6MO
NgXbcYUW2fRCk7gzfhFD536jPGa+rmVIwc4i9HMeznexTinakpYnLQ+MklIAFg2U
zB9TSr38YRaCyOHIKiIwTxc1l6BDwd6t06k8uWjmfDzzAuzks2OSmLWGTb3rUi++
m+1ThALLWEOMJx0QikS6rY5pqaNo6KHPpQy0AiOewGgCGZXyaSeM61qt8Ec8Zvpa
RBVpFTv+94chqfvSYJiAixgd6RWRWqs+z51XD1oFhSp48RLBUmFQb+7yVWxJJZPW
+nnCJ1hXcNoRyutEOHwL+Chmt2l7fY1CxOO5c1xX1V+9t7tTosVmeItvoJ6i1BCx
```

```
TFn8svOrQJrP9hFu8zbnxN3HKhGUZn9PbNgKSC+C3m9x1Dt8og7eXLb8sAEMJLeD
W74kVnQ/VrP/YerVAPOSauDSyQWpqN0VyY5bAHpLjuh0zosQexa7Ah6B8+k3tsOy
chdA2g3y5mZcniBNwmpzbprTih0JF6ulzSrx3Yk7sPYNztYl6AN+PnAF/kXe49kn
hUip1hDkGf4tEtR5p0uU+dBEIrSuwz3H1JAyHlha2Rajmk4nVyjUb2M/MMzdbnwg
HsrDYqp4Fu/8rlhkyws1SQGOwQ7H83DOIVBvkwNtB10hpxsilVQshe+EduSsoBI=
-----END CERTIFICATE-----
    Version: 0
    Serial Number: 4098 (0x1002)
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: /C=CZ/ST=Prague/L=Prague 6/O=TYFAMATE Certification
        Authority Ltd/OU=CA Root unit/CN=tyfamate.certificate.authority/
        emailAddress=tyfamate@certificate.authority
    Not valid before: May  8 21:23:31 2022 GMT
    Not valid after: May  5 21:23:31 2032 GMT
    Subject: /C=CZ/ST=Capital City of Prague/L=Prague 6/O=FIT CVUT/OU=BI
        -BIT/CN=pufintls.cz/emailAddress=tyfamate@fit.cvut.cz
    Public Key Algorithm: NULL
    EC Public Key:
      Public-Key: (256 bit)
      pub:
          04:81:78:98:54:1e:a0:35:77:ac:a0:47:6a:b0:55:
          5e:75:e0:2d:e7:c3:cd:f3:d4:f3:15:b2:9d:9c:88:
          69:55:10:fc:a1:40:15:27:a1:7d:ff:1e:16:48:15:
          dc:14:3b:5e:8b:1a:fd:7a:f4:da:00:c2:6c:0d:b5:
          1b:7b:f7:f1:a2
      ASN1 OID: prime256v1
      NIST CURVE: P-256
  Verify Certificate:
    unable to get local issuer certificate

  SSL Certificate:
Signature Algorithm: sha256WithRSAEncryption
ECC Curve Name:      prime256v1
ECC Key Strength:    128

Subject:  pufintls.cz
Issuer:   tyfamate.certificate.authority

Not valid before: May  8 21:23:31 2022 GMT
Not valid after:  May  5 21:23:31 2032 GMT
```

Bibliography

1. STANÍČEK, O. *Physical unclonable functions on ESP32*. Prague, 2022. Bachelor's Thesis. Czech Technical University in Prague, Faculty of Information Technology, Department of Computer Systems. to appear.
2. MAES, R. *Physically Unclonable Functions*. Springer Berlin Heidelberg, 2013. Available from DOI: 10.1007/978-3-642-41395-7.
3. BHUNIA, S.; TEHRANIPOOR, M. *Hardware Security: A Hands-on Learning Approach*. 1st ed. Katey Birtcher, 2018. ISBN 978-0-12-812477-2.
4. GASSEND, B. *Physical random functions*. Cambridge, 2003. Masters's Thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science.
5. KODÝTEK, F. *HW Bezpečnost: Fyzicky neklonovatlené funkce* [online]. 2017 [visited on 2022-04-22]. Available from: https://courses.fit.cvut.cz/BI-HWB/lectures/files/prednaska8.pdf. accessible after authorization with Czech Technical University account.
6. RÜHRMAIR, U.; BUSCH, H.; KATZENBEISSER, S. Strong PUFs: Models, Constructions, and Security Proofs. In: *Information Security and Cryptography*. Springer Berlin Heidelberg, 2010, pp. 79–96. Available from DOI: 10.1007/978-3-642-14452-3_4.
7. HAMMING, R. W. Error detecting and error correcting codes. *The Bell System Technical Journal*. 1950, vol. 29, no. 2, pp. 154–155. Available from DOI: 10.1002/j.1538-7305.1950.tb00463.x.
8. MAITI, A.; GUNREDDY, V.; SCHAUMONT, P. A Systematic Method to Evaluate and Compare the Performance of Physical Unclonable Functions. In: *Embedded Systems Design with FPGAs*. Springer New York, 2012, pp. 245–267. Available from DOI: 10.1007/978-1-4614-1362-2_11.
9. MAES, R.; VERBAUWHEDE, I. Physically Unclonable Functions: A Study on the State of the Art and Future Research Directions. In: *Towards Hardware-Intrinsic Security: Foundations and Practice*. Ed. by SADEGHI, A.-R.; NACCACHE, D. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 3–37. ISBN 978-3-642-14452-3. Available from DOI: 10.1007/978-3-642-14452-3_1.
10. HORI, Y.; YOSHIDA, T.; KATASHITA, T.; SATOH, A. Quantitative and Statistical Performance Evaluation of Arbiter Physical Unclonable Functions on FPGAs. In: *2010 International Conference on Reconfigurable Computing and FPGAs*. 2010, pp. 298–303. Available from DOI: 10.1109/ReConFig.2010.24.
11. MAITI, A.; CASARONA, J.; MCHALE, L.; SCHAUMONT, P. A large scale characterization of RO-PUF. In: *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 2010, pp. 94–99. Available from DOI: 10.1109/HST.2010.5513108.

12. MCGRATH, T.; BAGCI, I. E.; WANG, Z. M.; ROEDIG, U.; YOUNG, R. J. A PUF taxonomy. *Applied Physics Reviews*. 2019, vol. 6, no. 1, p. 011303. Available from DOI: 10.1063/1.5079407.

13. PAPPU, R.; RECHT, B.; TAYLOR, J.; GERSHENFELD, N. Physical One-Way Functions. *Science*. 2002, vol. 297, no. 5589, pp. 2026–2030. Available from DOI: 10.1126/science.1074376.

14. HOLCOMB, D. E.; BURLESON, W. P.; FU, K. Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers. *IEEE Transactions on Computers*. 2009, vol. 58, no. 9, pp. 1198–1210. Available from DOI: 10.1109/TC.2008.212.

15. HELFMEIER, C.; BOIT, C.; NEDOSPASOV, D.; SEIFERT, J.-P. Cloning Physically Unclonable Functions. In: *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 2013, pp. 1–6. Available from DOI: 10.1109/HST.2013.6581556.

16. CHANDRA, S.; PAIRA, S.; ALAM, S. S.; SANYAL, G. A comparative survey of Symmetric and Asymmetric Key Cryptography. In: *2014 International Conference on Electronics, Communication and Computational Engineering (ICECCE)*. 2014, pp. 83–93. Available from DOI: 10.1109/ICECCE.2014.7086640.

17. BÖHM, C.; HOFER, M. Error Correction Codes. In: *Physical Unclonable Functions in Theory and Practice*. Springer New York, 2012, pp. 87–102. Available from DOI: 10.1007/978-1-4614-5040-5_5.

18. DODIS, Y.; OSTROVSKY, R.; REYZIN, L.; SMITH, A. Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. *SIAM Journal on Computing*. 2008, vol. 38, no. 1, pp. 97–139. Available from DOI: 10.1137/060651380.

19. PRICE, N. E.; SHERMAN, A. T. *How to Generate Repeatable Keys Using Physical Unclonable Functions Correcting PUF Errors with Iteratively Broadening and Prioritized Search* [online]. 2014 [visited on 2022-04-30]. Available from eprint: https://ia.cr/2014/1023.

20. EUROPEAN UNION AGENCY FOR CYBERSECURITY. *Algorithms, key size and parameters: report – 2014* [online]. Ed. by SMART, N. LU: Publications Office, 2014 [visited on 2022-04-27]. Available from DOI: 10.2824/36822.

21. KRAWCZYK, H. Cryptographic extraction and key derivation: The HKDF scheme. In: *Advances in Cryptology – CRYPTO 2010*. Springer Berlin Heidelberg, 2010, pp. 631–648. ISBN 9783642146220. ISSN 0302-9743. Available from DOI: 10.1007/978-3-642-14623-7_34.

22. BONEH, D. Twenty Years of Attacks on the RSA Cryptosystem. *Notices of the American Mathematical Society*. 1999, vol. 46, pp. 203–213.

23. NAKOV, S. *Practical Cryptography for Developers: Elliptic Curve Cryptography (ECC)* [online]. 2021 [visited on 2022-05-10]. ISBN 978-619-00-0870-5. Available from: https://cryptobook.nakov.com/asymmetric-key-ciphers/elliptic-curve-cryptography-ecc.

24. CLOUDFLARE, INC. *What is TLS (Transport Layer Security)?* [Online]. © 2022 [visited on 2022-05-05]. Available from: https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/.

25. SULLIVAN, N. *A Detailed Look at RFC 8446 (a.k.a. TLS 1.3)* [online]. 2018-08-11 [visited on 2022-05-05]. Available from: https://blog.cloudflare.com/rfc-8446-aka-tls-1-3/.

26. CLOUDFLARE, INC. *What is an SSL certificate?* [Online]. © 2022 [visited on 2022-05-05]. Available from: https://www.cloudflare.com/learning/ssl/what-is-an-ssl-certificate/.

27. VEHENT, J.; DESTUYNDER, G.; MEIHM, A.; KING, A. *Security/Server Side TLS* [online]. 2022-04-01 [visited on 2022-05-05]. Available from: https://wiki.mozilla.org/Security/Server_Side_TLS.
28. RESCORLA, E. *The Transport Layer Security (TLS) Protocol Version 1.3* [RFC 8446]. RFC Editor, 2018-08. Tech. rep., 8446. Available from DOI: 10.17487/RFC8446.
29. OPENSSL FOUNDATION, INC. *Vulnerabilities* [online]. © 2021 [visited on 2022-05-05]. Available from: https://www.openssl.org/news/vulnerabilities.html.
30. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *ESP-TLS* [online]. © 2022 [visited on 2022-05-05]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/protocols/esp_tls.html.
31. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *Copyrights and Licenses* [online]. © 2022 [visited on 2022-05-05]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/COPYRIGHT.html.
32. WOLFSSL INC. *wolfSSL Embedded SSL/TLS Library* [online]. © 2022 [visited on 2022-05-05]. Available from: https://www.wolfssl.com/products/wolfssl/.
33. WOLFSSL INC. *Documentation* [online]. © 2022 [visited on 2022-05-05]. Available from: https://www.wolfssl.com/docs/.
34. TRUSTED FIRMWARE. *Mbed TLS* [online]. 2022-02-28 [visited on 2022-05-05]. Available from: https://developer.trustedfirmware.org/w/mbed-tls/.
35. WEBWIRE. *Hafnium, MbedTLS, PSA Crypto join the Trusted Firmware Project* [online]. 2020-07-17 [visited on 2022-05-05]. Available from: https://www.webwire.com/ViewPressRel.asp?aId=261668.
36. TRUSTED FIRMWARE. *Releases · Mbed-TLS* [online]. 2021-12-17 [visited on 2022-05-05]. Available from: https://github.com/Mbed-TLS/mbedtls/releases.
37. TRUSTED FIRMWARE. *TLS 1.3 support* [online]. 2022-03-31 [visited on 2022-05-05]. Available from: https://github.com/Mbed-TLS/mbedtls/blob/development/docs/architecture/tls13-support.md.
38. ARM LIMITED. *mbed TLS v2.16.1 source code documentation* [online]. © 2015 [visited on 2022-05-05]. Available from: https://tls.mbed.org/api/.
39. TRUSTED FIRMWARE. *README for Mbed TLS* [online]. 2022-03-31 [visited on 2022-05-05]. Available from: https://github.com/Mbed-TLS/mbedtls/blob/development/README.md.
40. PORNIN, T. *BearSSL* [online]. © 2018 [visited on 2022-05-05]. Available from: https://bearssl.org/.
41. PORNIN, T. *TLS 1.3 Status* [online]. © 2018 [visited on 2022-05-05]. Available from: https://bearssl.org/tls13.html.
42. ORYX EMBEDDED. *CycloneSSL: Embedded TLS / DTLS Library* [online]. © 2021 [visited on 2022-05-05]. Available from: https://oryx-embedded.com/products/CycloneSSL.
43. WOLFSSL INC. *Releases · wolfSSL* [online]. 2022-05-03 [visited on 2022-05-05]. Available from: https://github.com/wolfSSL/wolfssl/releases.
44. ORYX EMBEDDED. *Download: Here you can find our source code by version* [online]. © 2021 [visited on 2022-05-05]. Available from: https://www.oryx-embedded.com/download/.
45. WOLFSSL INC. *First DO-178 SOI Audits* [online]. 2020-06-30 [visited on 2022-05-05]. Available from: https://www.wolfssl.com/first-178-soi-audits/.
46. BAKKER, P. *Is mbed TLS FIPS certified?* [Online]. 2019-12-12 [visited on 2022-05-05]. Available from: https://tls.mbed.org/kb/generic/is-mbedtls-fips-certified.

47. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *ESP32 Series: Datasheet* [online]. 2022-03 [visited on 2022-05-05]. Available from: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

48. JOSEFSSON, S.; LEONARD, S. *Textual Encodings of PKIX, PKCS, and CMS Structures* [RFC 7468]. RFC Editor, 2015-04. Tech. rep., 7468. Available from DOI: 10.17487/RFC7468.

49. TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU. *Information technology - Open Systems Interconnection - The Directory: Selected attribute types* [online]. 2019-10-14 [visited on 2022-05-05]. Tech. rep. International Telecommunications Union. Available from: https://handle.itu.int/11.1002/1000/14037.

50. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *ESP-IDF* [online]. © 2022 [visited on 2022-05-09]. Available from: https://www.espressif.com/en/products/sdks/esp-idf.

51. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *Get Started* [online]. © 2022 [visited on 2022-05-09]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html.

52. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *Build System* [online]. © 2022 [visited on 2022-05-09]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/build-system.html.

53. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *ESP-IDF Versions* [online]. © 2022 [visited on 2022-05-09]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/versions.html.

54. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *ESP-IDF 5.0 Migration Guides* [online]. © 2022 [visited on 2022-05-09]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/migration-guides/index.html.

55. WOLFSSL INC. *ESP-IDF port* [online]. 2022-03-30 [visited on 2022-05-09]. Available from: https://github.com/wolfSSL/wolfssl/tree/master/IDE/Espressif/ESP-IDF.

56. WOLFSSL INC. *Building wolfSSL* [online]. © 2022 [visited on 2022-05-09]. Available from: https://www.wolfssl.com/documentation/wolfssl-manual/chapter02.html.

57. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *Networking APIs* [online]. © 2022 [visited on 2022-05-09]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/index.html.

58. RESCORLA, E. *HTTP Over TLS* [RFC 2818]. RFC Editor, 2000-05. Tech. rep., 2818. Available from DOI: 10.17487/rfc2818.

59. WOLFSSL INC. *ASN.1* [online]. © 2021 [visited on 2022-05-09]. Available from: https://www.wolfssl.com/doxygen/group__ASN.html.

60. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *mDNS Service* [online]. © 2022 [visited on 2022-05-09]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/protocols/mdns.html.

61. MOCKAPETRIS, P. V. *Domain names - implementation and specification* [RFC 1035]. RFC Editor, 1987-11. Tech. rep., 1035. Available from DOI: 10.17487/RFC1035.

62. WOLFSSL INC. *Algorithms - ECC* [online]. © 2021 [visited on 2022-05-09]. Available from: https://www.wolfssl.com/doxygen/group__ECC.html.

63. WOLFSSL INC. *wolfssl/ecc.c* [online]. 2022-04-27 [visited on 2022-05-09]. Available from: https://github.com/wolfSSL/wolfssl/blob/master/wolfcrypt/src/ecc.c.

64. WOLFSSL INC. *wolfSSL Certificates and Keys* [online]. © 2021 [visited on 2022-05-09]. Available from: https://www.wolfssl.com/doxygen/group__CertsKeys.html.

65. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *Universal Asynchronous Receiver/Transmitter (UART)* [online]. © 2022 [visited on 2022-05-09]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/uart.html.

66. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *Logging library* [online]. © 2022 [visited on 2022-05-09]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/log.html.

67. ESP_IGRR. *Re: ESP32 How to turn off automatic log of system output* [online]. 2018-01-17 [visited on 2022-05-09]. Available from: https://www.esp32.com/viewtopic.php?t=4249.

68. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *Non-volatile storage library* [online]. © 2022 [visited on 2022-05-09]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/nvs_flash.html.

69. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *SPIFFS Filesystem* [online]. © 2022 [visited on 2022-05-09]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/spiffs.html.

70. MACKENZIE, D. *stty(1) — Linux manual page* [online]. .328th ed. 2020 [visited on 2022-05-09]. Available from: https://www.man7.org/linux/man-pages/man1/stty.1.html.

71. OPENSSL FOUNDATION, INC. *openssl-ca* [online]. © 2021 [visited on 2022-05-09]. Available from: https://www.openssl.org/docs/manmaster/man1/openssl-ca.html.

72. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *lwIP* [online]. © 2022 [visited on 2022-05-09]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/lwip.html.

73. OPENSSL FOUNDATION, INC. *s_client* [online]. © 2021 [visited on 2022-05-09]. Available from: https://www.openssl.org/docs/man1.0.2/man1/openssl-s_client.html.

74. RBSEC. *sslscan2* [online]. 2022-04-10 [visited on 2022-05-09]. Available from: https://github.com/rbsec/sslscan.

75. STANÍČEK, O. *esp32_puflib* [online]. 2022-03-22 [visited on 2022-04-01]. Available from: https://github.com/Cpt-Hook/esp32_puflib/tree/main.

76. TÝFA, M. *ets_delay_us(...) is not defined in puf_measurement.c* [online]. 2022-04-14 [visited on 2022-04-14]. Available from: https://github.com/Cpt-Hook/esp32_puflib/issues/1.

77. TÝFA, M. *PUF_STATE is RESPONSE_CLEAN after successful get_puf_response* [online]. 2022-05-11 [visited on 2022-05-11]. Available from: https://github.com/Cpt-Hook/esp32_puflib/issues/3.

78. ESPRESSIF SYSTEMS (SHANGHAI) CO., LTD. *Deep Sleep Wake Stubs* [online]. © 2022 [visited on 2022-05-09]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/deep-sleep-stub.html.

79. WOLFSSL INC. *Callbacks* [online]. © 2022 [visited on 2022-05-09]. Available from: https://www.wolfssl.com/documentation/wolfssl-manual/chapter06.html.

Contents of the enclosed CD

- `readme.txt` ..... introduction and information about the media
- `src`
  - `ESP32` ..... source code for the proof of concept application
    - `components`
      - `puflib` ..... esp32_puflib developed by Ondřej Staníček
      - `wolfssl` ..... wolfSSL library developed by wolfSSL Inc.
    - `main` ..... application starting point
    - `wolfSSLConfiguration` ..... configuration for the wolfSSL library
    - `sdkconfig.defaults` ..... default configuration for the application
  - `PC` ..... enrollment helper script
  - `thesis` ..... LaTeX source code for this thesis
- `generated`
  - `thesis.pdf` ..... PDF version of this thesis