Insert here your thesis' task.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

PUF Based IoT Device Over-the-air Update

Marek Kňazovický

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

June 23, 2022

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 school work under the provisions of Article 60(1) of the Act.

In Prague on June 23, 2022 ……

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Kňazovický, Marek. *PUF Based IoT Device Over-the-air Update*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Práca sa zaoberá prieskumom a navrhnutím zabezpečeného OTA update procesu s využitím SRAM PUF a nasadením na obecnom IoT zariadení. Je rozdelená na tri časti.

Prvá časť skúma štruktúru a fungovanie bezbežného OTA procesu a je ukončená navrhnutím obecnej varianty vhodnej pre zdrojovo obmedzené zariadenia. Druhá časť skúma možnosti využitia PUF v kryptografických technikách daného OTA procesu. A na záver bola vyvinutá knižnica obecne užívajuca SRAM PUF, integrovaná a následne nasadená do ESP32 proof of concept využitia, kde bol použitý v rámci diskutovaného návrh OTA procesu.

Kľúčové slová OTA, SRAM, PUF, IoT, zdrojovo obmedzené zariadenie, kryptografické techniky, ESP32, proof of concept

Abstract

The work deals with surveying and designing a secure OTA update process using an SRAM-based PUF and a typical IoT device deployment. It consists of three parts.

The first part examines the structure and the functionalities of a common OTA process. It is concluded with a proposal of a simple variant of the process suitable for resource-constrained devices. The second part studies the possibilities of using a PUF within the cryptographic techniques utilized by the proposed OTA process. And finally, a library for general use of SRAM PUF was designed, integrated and deployed on an ESP32 proof of concept demonstration, where it was used among the simplified OTA proposal.

Keywords OTA, SRAM, PUF, IoT, resource-constrained device, cryptographic techniques, ESP32, proof of concept

Contents

List of Figures

List of Tables

Introduction

As per the numerous forecasts predicting the enormous increase in the spending related to IoT solutions, e.g. [1], the everpresent need to keep the IoT devices safe and secure in a consistent, well administered, reliable and scalable manner grows even more, each year. Thus, the Over the Air update comes into play - a process to push updates from a centralised node remotely to possibly a large fleet of IoT devices while also, perhaps, attending to specific device' needs based on its specific state. However, these devices do not always possess the means to perform the required cryptographic techniques, which would make the process much safer. Thus a question arises of whether to somehow compromise on the security while updating such constrained devices or look for more unorthodox OTA process models and alternative means to acquire the required cryptographic assets.

Thesis goals

One of the goals of this thesis is to unravel the Over the Air update process and its pitfalls, more precisely which factors come into the play that make it as effective as a standard OTA process is, while also maintaining a good level of security. Afterwards, the threats and the types of attacks that commonly plague various aspects of the OTA process are examined, and typical security countermeasures to such concerns are brought up in contrast. Furthermore, a proposal for an OTA process suited for a resource-constrained IoT device is brought forward to summarise the first part of the work. Also, due to the constrained resources, even some cryptographic assets that play an essential part during the previously described security countermeasures might be missing. Therefore, it is then discussed which ones can be substituted with a PUF that does not impede the used cryptographic techniques.

The second primary goal is to look at the state of the PUF sphere and to explore how they could be implemented on a typical IoT device. The PUF variants that do not require physical intervention are brought up in more detail, and their possible participation during the proposed OTA process is discussed further. Afterwards, the one with the best prospects, SRAM PUF, is further analysed, and a way to implement its use on a resource-constrained device is outlined.

The last part of the thesis discusses the implementation details of the outlined general library incorporating the SRAM PUF as an asset for performing the desired cryptographic technique. The library's functionality is further tested on an ESP32 device test project, where the PUF properties, such as memory cell stability, reliability and bit uniformity, are examined more closely. Afterwards, a simplified version of the initial OTA update process proposal is applied to an ESP32 device. Finally, it is tested as a proof of concept while replacing a cryptographic asset that some CT would be using.

Chapter 1

Over the Air update analysis and model proposal

The need for secure, up-to-date software and thus safely operating hardware has been increasingly important in recent years. On most platforms, this goal can be achieved with a process called Over the Air (OTA) update. It can be a relatively straightforward procedure due to entirely unconstrained system resources or preexisting, tried frameworks and systems already in place, such as the ARM platform upon which the Android ecosystem is built [2]. However, currently, the most significant portion of devices connected to the internet is machine to machine, or more commonly, IoT - Internet of Things devices. Moreover, if the circumstances do not change radically, the proportion of IoT devices on the market will increase even more [3].

The massive number of these devices and its further increase also means that quite a few will be parts of critical infrastructures, making them possible attack vectors into large ecosystems or even targets themselves. Thus, keeping them exploit-free and safely operating is a critical concern, so they need to be updated regularly. Until a few decades ago, this was done manually on site almost exclusively. Nowadays, manual updates are still done in specific or very critical cases, where the transfer of the updated software with the help of network and automation is simply not acceptable due to numerous concerns, such as [4]:

- incredibly **small failure threshold** of critical applications as a lot can go wrong during an automated process such as an OTA update
- **update scheduling** - with either automated or massively done an update of an IoT product, it might be impossible to find a time slot where it can go offline, requiring it to be done manually
- **potential problems induced by the update** - simply updating the IoT device might cause it to be inaccessible from the connected network, making further off-site interaction impossible

- **scale** - the amount of IoT devices might be an order of magnitude smaller than the amount of effort required to implement OTA (no need to solve OTA security and similar concerns)

However, despite these concerns the OTA brings, it might be a great idea to use it as of the aforementioned increase in the IoT device amounts, during which the use of OTA brings the most benefit [4]. However, the fact that most of these devices are heavily resource-constrained poses a challenge in implementing this process safely and reliably.

1.1 OTA update scheme

Due to the nature of most IoT devices and their resource constraints, there might be a particular OTA scheme in place. However, with the rise of cloud usage [5] the need for at least partial uniformity of this mechanism becomes apparent. With this, there are generally accepted steps that must be adhered to and implemented to have a flexible and scalable IoT solution. To generalise the OTA update process - the IoT device, sometimes referred to as an edge node, will be specified as a client, and the other participant, the one identifying the client, choosing and administering the update, will be called the server (represented mainly by a cloud-like platform).

In general, the OTA update primarily consists of:

- correct **mutual identification** and subsequent authentication between a client and the server providing said update
- ascertain whether the client is **eligible** for the specific update
- **fetching** the updated binary
  - the process, generally, should be device *optimised*, caching a few pages at a time or even writing through in the worst-case scenario (in the case of a very small RAM)
  - the use of *compression* depends on the device's resources and capabilities
- the downloaded app should be **verified** (correct CA signing, checksum validation/hash comparison, etc.)
- after the update is in accordance with all requirements, the SSBL should branch to the new app on the next reset
- in case of unexpected failure, the old app should be designated as a fallback app, so SSBL can branch to it instead if needed

1.1.1 Provisioning

Not really a part of the OTA process, but it is still needed to have a functional network connection between the client and the server, which is achieved by provisioning. In most cases, it consists of configuring the device to access the local network properly (e.g. setting up WiFi credentials and certificates). For example, on the ESP32 platform, this is achieved by Unified Provisioning [6]*(go to /api-reference/provisioning/provisioning.html)*, which does the following:

1. User installs the configuration app, which connects to a target device in provisioning mode, which has some form of beaconing
2. The target device either host a WiFi Access Point with a running HTTP server or uses BLE to which the user can connect to
    - in the case of an Access Point, there should be security measures, such as WPA2 secured WiFi with a passcode provided with the device, e.g. in the form of a QR code on the device itself
        - however, the level of security is flexible and not enforced by the ESP32 framework
    - there might be additional configuration installed, or even certificates necessary for the OTA
3. After a successful configuration, the device turns off the provisioning interface so it can free resources (e.g. BLE can take up to a fifth of the ESP32's of memory [6]*(go to /api-reference/provisioning/provisioning.html)*) and be more stable (running the ESP32 in SoftAP+STA mode is costly in terms of power consumption and since both use the same radio stack and antenna it makes the communication less reliable)
4. The device connects to the local network with the provided configuration

1.1.2 Second stage boot loader

Generally speaking, the booting architecture is mainly based on the two (or more) stage bootloader. This originated on the PC platform, where by default, the first loaded sector was limited to a 512-byte size, thus narrowing down what the bootloader could do, for it must have been limited to this size. Thus naturally, a multi-stage bootloader was implemented, where the only limitation was placed on the first stage bootloader (FSBL), whose only job was to initialise necessary hardware and correctly load the second stage boot loader (SSBL) and execute it correctly. SSBL has no size limitations, so its logic and capabilities can be quite extensive.

On the IoT platforms and specifically the ESP32 platform, it might have been possible only to include a single-stage bootloader. However, it was disadvantageous to do so mainly as a result of FSBL residing in the read-only memory [7]:

- in the case of including the OTA capabilities to FSBL, it would not be possible to update them, thus limiting the lifespan of the device significantly
- if the OTA capabilities were to be granted to the user application, then that would result in
  - in the case of Real-Time Operating System, which ESP32 has, it would be problematic to grant the SSBL obligations to the user application, for simply branching to the reset handler of the new, updated application would induce serious issues due to RTOS presence - e.g. other tasks might be running in the background, etc.
  - if the branching to the current version of the application were to be done by the FSBL, it would solve the RTOS task issue, as it runs before the RTOS is initiated, but it might become unstable because of the interrupt vector table (IVT) which might be required to be relocated - if a power cycle occurs during this process the device might become inoperable

Having an SSBL solves the abovementioned problems since it resides at a constant address and always contains the current IVT table (an array of pointers to functions, handling faults, system requests, interrupt requests etc.[8]), is not in read-only memory, so that it can be patched later on. Since it is a non-RTOS "program", it can safely branch to whichever application is required [7]. Also, an SSBL that handles most of the OTA functionality can reduce application size by a large margin because neither the new, the fallback, nor any alternative version of the app are required to have the same duplicated OTA and other related code. However, the ESP32 platform has a minimalistic SSBL since the ESP32 OTA process cannot patch the SSBL directly. The reason behind this restriction is that if done and a power failure occurs, it might cause the device to become inoperable [7]. The SSBL patching can, therefore, be only done by uploading the new firmware locally [9].

Concerning the SSBL, the OTA proceeds as follows (with the ESP32 specifics [6]*(go to /api-reference/system/ota.html)*):

1. OTA/user portion of the user application detects an available update on the server.
2. It gets a new application and places it into the appropriate OTA partition.
   1. Establishes trust with the server (e.g. PKI usage)
   2. Downloads the update
   3. Verifies the update
   4. Updates the *ota_data* (similar to the Table of Contents in the figure 1.1)
   5. Grants control the SSBL by resetting the system.
3. After getting control from FSBL, the SSBL detects an updated application from the *ota_data* and partition table and acts accordingly:

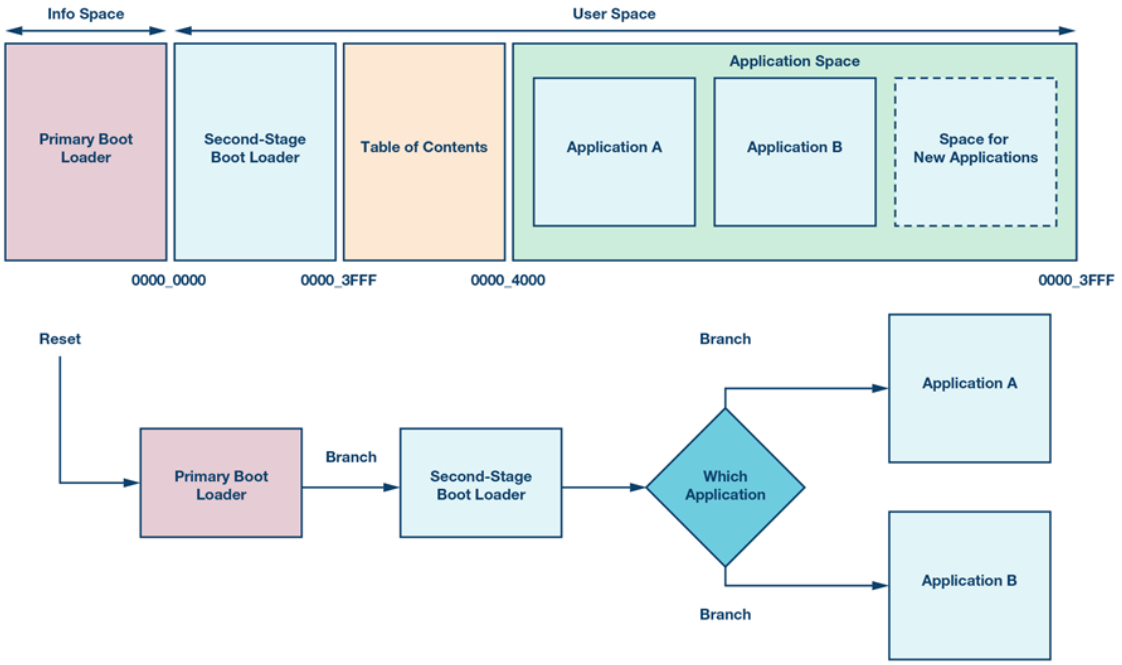

Figure 1.1: General bootloader scheme [7]

1. Initializes internal modules, among them, e.g. Flash encryption and Secure Boot, if enabled

2. Selects the updated application based on the *ota_data* partition (in case of the first boot of the new app, otherwise select the fallback app)

3. Relinquishes control to either the next stage bootloader or the application/RTOS

To summarise, having a part of OTA logic present during the SSBL initialisation operations can be exceedingly effective since it allows control over the selected application to load and run, thus allowing additional logic to be executed to determine the best course of action. Therefore, if the OTA logic running during the SSBL phase figures out that either the new application either has some internal inconsistency, a wrong version of a component or some other fatal problem, it could even suspend the loading of the said new application and switch to a previous or even the factory version.

Also, suppose it was somehow communicated to the device that the new application has some sort of dangerous behaviour, making it unsafe to use (unsafe operational logic, introduction of a security concern or the developing team becoming aware of an exploit). In that case, it can decide to use a fallback application too.

1.1.3 OTA update transfer, speed and compression

Nowadays, an average application has numerous external dependencies, which are normally not in the developer's control. All these dependencies pose a potential risk that often becomes real and needs to be monitored and acted upon if necessary. Suppose this is taken into account with the fact that a well-operating IoT service should also have regular user code security fixes, feature improvements or even new features frequently. In that case, all of this must be reflected in the frequency and volume of OTA updates. However, if such a system is using the OTA update often and in a large enough volume, then normal operation disruptions due to OTA updates must be taken into account and addressed.

Numerous factors affect the transfer process of an OTA update, some of which is:

- **speed** of the storage medium, in the case it is a bottleneck
- type of interface that provides the **network connectivity** to the device
- **reliability** of the network connectivity
  - *wireless connectivity* - the strength and quality of the signal
  - *wired connectivity* - the amount of noise present in the environment
- limited ability to use the available **network capacity** due to
  - device operations - e.g. critical action done by the device needing most resources
  - limited network bandwidth of the device
  - Limited amount of data available to transfer
- the **size** of the updated binary

These factors need to be considered, but most of them are out of scope during software development. However, the size of the update binaries is inherently tied to the development activity. Numerous factors do unnecessarily enlargen the application binary, which does, logically, by the same amount, increase the volume of data the OTA process has to transfer. Among these, the usual offenders are :

- the use of high-level constructs with needless **overhead**
- using a general solution for a problem instead of a minimal one
- forgetting to remove **unused components** and libraries
- inefficient choice of compiler and its configuration, etc.

However, these are still out of scope for the OTA mechanism to influence in any way. So in the case of the OTA process on the update server getting a binary executable on input, the most direct way for it to reduce the transmitted volume of data would be by compressing it [10]. There are multiple ways of doing so, which involve many quite simple and advanced algorithms. However, in the context of resource-constrained devices, the focus will be on its low hardware footprint and good **lossless** compression ratios.

Numerous compression algorithms offer very favourable compression ratios; however, the main concern in their case would be the system resource requirements and almost gigantic RAM prerequisites for their normal operation [10]. That might not be a problem automatically since lossless data compression is a two-way process, where decompressing is typically quite fast, sometimes even almost **100x** than the compression time [10]. Thus even libraries with unacceptable compression times might be usable on such devices since only decompression would be used, as it would receive already compressed binary data from the OTA update server. Though this is a feedback loop - to decompress data produced by a complex algorithm, it might be required to have the whole compression library, including the complicated compression methods, included on the device. Thus the library would have to be also included in all binary files that would be received through OTA, potentially negating some of the achieved size reduction.

Nonetheless, plenty of compression libraries are designed for resource-constrained devices that attain good performance during both compression and decompression while having a limited memory footprint. One of such solutions is an open-source specified **LZ4** compression algorithm. It is designed for very fast compression and decompression while retaining low resource consumption and favourable compression ratios on binary application data as high as **40%** [10]. It might even be possible to store the application binary compressed and only decompress it during SSBL loading the program from flash, thus saving extra device flash memory aside from saving OTA update time. The algorithm's performance is visualised in 1.2.

There are, of course, other ways of reducing binary application size by the OTA process, such as by not transferring the whole firmware at once. Such an example might be by introducing **binary difference data**, that after applying to the old firmware would result in a successful update, in place even, if the device is capable of it [11]. However, that might make using the fallback feature harder to implement and also might involve a lot of additional logic overhead which is never desirable.

So to summarise, depending on the exact hardware specification and the use case, applying a compression function on the sent OTA update might be very interesting. It also might be intriguing to evaluate the decompress costs, mainly time-wise, and potentially save the compressed firmware directly to flash.

1.1.4 General challenges and requirements

For the OTA process not to be just a simple fetch and persist tool that focuses on delivering the update as quickly as possible, it needs to conform to various functional expectations, guidelines and conditions. After fulfilling such requirements, the process should result

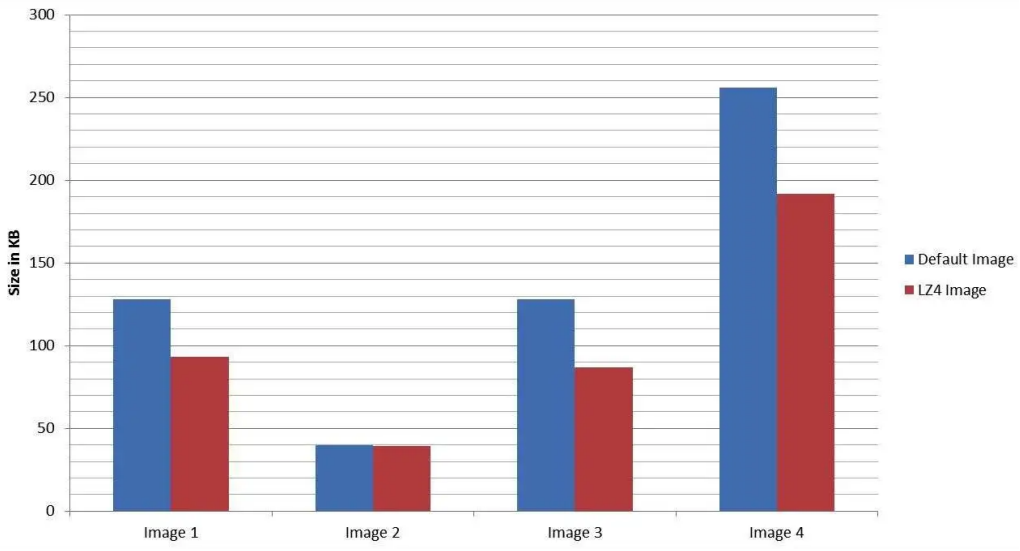

Figure 1.2: LZ4 compression examples [10]

in more predictable, robust and, most importantly, **secure** and deterministic behaviour. Out of those, the most important ones should be named specifically as[12] [13]:

- A **failsafe** mechanism after a failed update
  - frequent update deployment increases the likelihood of update failure
  - actual conditions of the working environment might seriously impact the ability to communicate with the server amidst an update
- A **guarantee** the device will receive the intended update
  - if the same server is serving multiple IoT devices with differing firmware branches of updates, it might result in the incorrect device receiving the incorrect firmware. Even with a failsafe mechanism, this would still result in a waste of device, server and network time. If not implemented, it might even disrupt normal functionalities of the devices since they would be receiving bogus updates not applicable to them
- An **assurance** that the update truly comes from the expected authority
  - if the process is not using methods through which it is cryptographically verifiable that the origin of the update is the intended authority, it could lead to losing control over the whole deployed ecosystem
- A means of update delivery which does **not expose its contents** to third parties

  - if the communication channel is unsecured, there is no guarantee that the concerned data is kept only by the intended parties, which also means that even if an initial connection might have been initiated with the trusted server, the data have no certainty that they have not been tampered with

Most of these are related to cryptographically secure needs of conduct of the OTA process. Therefore, that guarantees a more detailed look at the specific CTs involved while staying aware of the attacks and vulnerabilities that come hand in hand.

1.1.5 OTA update security

For a well-designed OTA process to function properly and also to tackle the already described elementary, functional and QoL requirements, it needs to take security seriously and implement it in a well thought out and systematic manner. Depending on the perspective, an OTA process can be seen as a trust-based process [14] since most security actions have a general purpose of trying to ascertain whether the party and the firmware are both genuine and keeping a **chain of trust** during the whole process in a verifiable manner. However, the OTA process, security-wise, should only be concerned with safeguarding the update process itself and immediately relevant system aspects, not the other facets of operations of either the server or the client[15], since it also needs to be as lightweight as possible.

1.1.5.1 OTA security concerns, threats and risks

There are numerous concerns and risks if one considers the security of an OTA update network, aside from more general security threats of a network-connected device. Therefore, threats related to the process are [16]:

- OTA server-side, where
  - the updates are stored and possibly also built, signed, encrypted and otherwise prepared for distribution among the wider population of the IoT device line.
  - various security details crucial to the device operations are being stored
  - server's private part of the PKI infrastructure used to verify itself and other cryptographic assets used
- IoT device side, the party that
  - normally is the one initiating the update process
  - is the subject of the update
  - the main subject running the OTA routines

Therefore, in order to somehow gain access, to gain classified information or otherwise damage the party operating the parties of whose the OTA process consists, the attacker could target one of those attack surfaces or anything in between them. Thus, the common attacks and threats are:

- **MiTM attack** - generally happens when an attacker either breaches the communication by attacking the protocol itself or using other exploits [17]. In other words, a third party to a normally two-party communication (in the OTA context, the IoT device and the server) inserts itself. The third party generally inserts itself into this communication by either placing a node under their control between the communicating parties or by executing malicious code on one the hardware of one of the parties [17]. The manner of malicious participation of such a third party differs immensely on the objective of the attack, visualised in 1.3. The main ones in the OTA context might be:
  - *impersonating* either the server or the device, compromising the whole singular OTA ecosystem
  - *eavesdropping* and stealing the firmware
  - *modifying* the firmware
  - *replacing* the firmware with a preprepared binary created by the attacker
- **Brute force attacks** - If preferring the speed, resources or fast development over security and using insufficient security measures, an attacker could try to break such a measure with a brute force attack. E.g. an OTA process using keys that are either too short or of a predictable type that in the end reduces their actual bits of security to a too low amount, guessable by a brute force attack within manageable time.
- **Denial of service** - a classic attack that makes the network resource unavailable to the intended group of users. It is normally achieved by flooding the resource with many requests so the resource cannot properly be used by the legitimate group of users[18]. In the context of an OTA update process, that would mean mimicking the IoT devices and flooding the OTA server with superfluous requests.
- **Side-channel attacks** - harnesses various signal emissions and other possible information leakages that the device might possess. There are multiple attack models, such as SPA, DPA, CPA and EM [19]; they normally require some knowledge of the system and the platform by the attacker
  - attacker must have access to the device
  - possible to retrieve virtually any resource, maintaining security with which the device works during runtime; requires a high degree of preparedness by the attacker

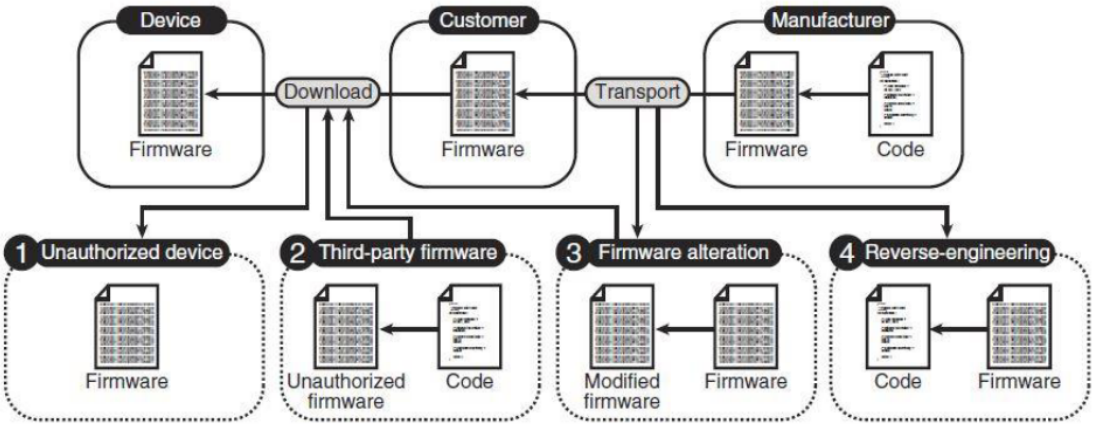

Figure 1.3: Common OTA security threats [16]

1.1.5.2 Security measure inquiry

For the sake of counteracting the described attack vectors and reducing the possible attack surfaces, there have been proposed and currently are in use numerous approaches, techniques and even philosophies of conduct. Among those, the ones relevant for use on the IoT device-side during the OTA process might be:

- **Secure boot** - In a well-protected IoT device, the user application code should be routinely verified and authenticated whether it has not been modified or hijacked by an attacker while stored in the flash memory. The best moment to do this would be during the booting process. More specifically, the SSBL should have a routine that does at least part of the process. This process is generally called **Secure boot** [14] and, as such, is a good addition to an OTA process since it might counter some threats and issues that come with using firmware from a remote source. In terms of the attack surface, it protects the IoT device directly in the case the firmware was somehow hijacked and redirects the attacker's attention to the SSBL or some other component doing the checks.

- **Poor security policy** - Using industry best practices for key lengths and employing valid cryptographic technique selection is also paramount. Not exactly conforming to the security standards and choosing an inadequate combination of cryptographic assets or just cheapening out on the bits of provided security by the chosen keys just to save some resources might [16] result in numerous vulnerabilities. To be specific, e.g. choosing triple DES might be a poor decision since, according to NIST [20], it is a deprecated standard with potential vulnerabilities to, among them, brute force attacks in the near future, since it has a 112 bits long key with only 80 bits of actual security due to its inadequacies. Therefore, giving up bits of security in favour of some performance increase is inadvisable and should be avoided.

- **Authentication** - The way of proving one's own identity is a powerful tool and quite necessary in the situation of deciding whether to trust the firmware provided by another party or not. By authenticating, if done correctly, it is being cryptographically proven that the party declaring itself in a certain role does, in reality, have the necessary verifiable cryptographic identification. Demanding successful server authentification as a prerequisite to subsequent firmware transfer during the OTA update negates some attack vectors, mainly a few MiTM attacks. As per authentication types, there are three main categories:

  - *Centralised three-way* - administrator registers the IoT devices with the main authority, assigning them valid certificates. After that, the authority facilitates secure handshakes between the registered parties [21]. As a plus, the devices do not have any certificates on them. Not relevant to most OTA processes.
  - *Distributed two-way* - i.e. mutual authentication. The device and the OTA server ought to have each other's public certificate installed, with the update commencing only when both the device and the server can authenticate each other [21]. However, managing that many more certificates and overhead on the device might not be feasible.
  - *Distributed one way* - only one party to the communicating pair is being authenticated; thus, in OTA, only the server has to prove to the device that it really is the OTA server authority. Therefore, only the public certificate of the OTA server needs to be distributed amongst the devices eligible for the updates [21]. However, a case could arise that an uneligible device might take hold of the certificate - in the case of unprotected firmware might mean a serious leak, therefore requiring further measures.

- **Securing of the firmware binary** - on an otherwise maximally secured ecosystem, where all parties are authenticated and communicate securely, the need for extra security just for the binary would not be extra useful outside of redundancy purposes. However, it is a powerful tool on, e.g. *Distributed one-way* authentication models, where the device receiving the update would not be authenticated. In this case, in order to not leak the firmware, it needs to be cryptographically ensured that only the right party can use the firmware. Also, just signing the firmware instead of doing full authentication of the server would not suffice since an attacker could hijack or otherwise mimick the OTA server role and perform a retransmission attack. Alternatively, even worse, send an older version of the firmware with a potentially unfixed exploit for it to harness later on.

- **Securing communication channel** - after authenticating the parties securely, the goal should be to agree on a session key in a similarly secure manner with whose help will the device and the server push their update-related communication through. A safely established, encrypted channel is paramount in keeping the updated firmware safe from the eavesdropping of other malicious parties. In this case, the TLS protocol is the main representation, which serves as a transparent layer

for other communication protocols [22]. DTLS can sometimes be found on really constrained devices, trying to provide a similar level of security as TLS but with a significantly lowered overhead.

- **Side-channel attack** - Another serious issue is the ability of an attacker to use a side channel to gain normally secure information or even compromise the device. This attack vector is a tough one to protect against since an IoT device is fundamentally a relatively cheap platform, making the application of any more advanced measures financially inadvisable for the solution to make sense still. However, there are still actions that can be undertaken[19]:
  - *software side*
    - device operation, ideally, should be independent of data - number of clock cycles randomised or uniform to not leak operation timings
    - avoid reusing the keys
    - power consumption not dependent on instructions executed
  - *hardware side*
    - use of Physical Unclonable Functions as CTs
    - correct shielding to mitigate leaking data through electromagnetic radiation
- **Flash encryption** - another way to counter various side-channel attacks is by safely encrypting the persistent flash storage. In the context of an IoT device, the only efficient way to do so is to use a symmetric key, and doing just that creates a problem of where the key would be stored. One such way might be having a hardware solution securely generating the said key and storing it in the form of an HSM, e.g. inside a chip on eFuses that is inaccessible to the CPU. Only the hardware-based encryption/decryption engine has the access, thus making the attacker have to use a more drastic invasive type of side-channel attack to access the eFuses and recover the key [23].

1.1.5.3 Cryptographic techniques in OTA security measures

When mentioning cryptography, nowadays, is commonly understood as a science centred around studying various techniques and methods with whose help it is possible to secure disparate types of data and communication. It is also closely related to cryptanalysis, whose study of methods of obtaining the protected information without the proper means of access, i.e. it is challenging the methods of cryptography. As for these cryptographic techniques, they are well-defined, deterministic algorithms whose operations should take adversarial behaviour into account, or their execution should be meant to counter it [24]. Ultimately, these CTs should have the pursuit of the following principles as their goal[24]:

- **Confidentiality** - only the authorised subjects should be able to view the protected information

- **Integrity** - the protected information should be either non-modifiable or easily detectable, e.g. during transit, storage etc.
- **Non-repudiation** - that no communicating party should be able to deny that they have sent or received the information in question
- **Authentication** - the identity of a communicating party can be confirmed

Therefore, these CTs use, in general, cryptographic methods/algorithms that in some way or form use cyphers, which is an algorithm used for encrypting and decrypting data. These techniques can be grouped into several categories:

- **Symmetric cryptography** - an encryption system whose purpose is to encrypt and decrypt data. The system uses only one key, which is relatively fast, but both parties need to have the knowledge of the said key in order for the system to work. A typical representative nowadays in the IoT sphere is the AES, with key sizes of 128, 192 and 256 bits [24].
  - commonly used as a session key of a TLS session - while solving the key exchange problem inherent to symmetric cryptography - more standardised way of establishing a safe OTA update communication channel
  - used as the main means of encryption in hybrid cryptography systems - therefore, OTA firmware encryption
  - also utilised by most flash encryption approaches to further secure the OTA update firmware
- **Asymmetric cryptography** - also an encryption system for data encryption purposes. However, it is based on key pairs, typically of which one is **private**, and one is **public**. Both can be used for encryption and decryption. It is quite slow but has solved the key exchange problem [24].
  - commonly used for safe establishing of a TLS session - solves the authentication of the server or both communicating parties
  - used as a basis of PKI cryptography, certificates and authentication techniques - the chain of trust building
  - use in hybrid cryptography, e.g. OTA firmware encryption
  - used for firmware signing
- **Cryptographic hash functions** - in contrast to normal encryption algorithms, proper hash functions should be a one-way function, where the original data should not be recoverable.
  - also utilises encryption, with the difference in not persisting the cypher texts of each block but just using it as an IV during the next round, with the hash output being the last cypher block
  - firmware signing, device verification and identification

1.2 Resource-efficient OTA update model

The chapter will describe the common feasible architectural approaches the OTA process can take while opting for one of them, will list the relevant functional obligations which a practical, robust, and secure OTA mechanism should strive to fulfil and also the strategies describing how to get these obligations fulfilled and finally, a description of the proposed OTA model in detail.

1.2.1 OTA architecture

There is no one size fits all type of OTA architecture, evermore due to the very different capabilities and granularities distinct IoT architectures have in the field. For example, in a use case where the amount of IoT devices per location is in the realm of tens or even hundreds with possibly very low computing power and resources, maybe even combined with a requirement for spending much time during deep sleep to conserve power, would have to approach the OTA update problem in a very contrasting manner than a use case with one deployed device per customer which is more autonomous resource-wise. Thus these dissimilar dilemmas could be categorised into three main architectural update classes [12] [25]:

1. **Edge to cloud OTA** - a single network-connected device with internet access that is solely responsible for itself. Its capabilities include the ability to receive and apply an update from a remote server by itself.
2. **Gateway to cloud OTA** - an internet-coupled gateway that manages a fleet of local edge devices. Capable of receiving the update from a remote server, same as the first architecture case. The update might be only intended for itself or the hosting environment.
    - Single point of failure in the gateway since the devices rely on the gateway being up to date.
3. **Edge to gateway to cloud OTA** - an internet-coupled gateway that manages a fleet of locally connected edge devices. Capable of receiving the update from a remote server, same as the previous architecture cases. However, the update might be either intended for itself, the hosting environment, the devices that are being managed or some kind of a subset of these devices, basically capable of attending to a heterogeneous group of devices and also acting as an update dispatcher for the said fleet.
    - Since the devices being managed are able to receive updates, having the gateway fail might not mean update failure overall.
    - Useful where the internet access is easier to set up for a single device, which would manage it for the other minor ones. It might be very useful in conditions where connecting the whole fleet directly to the internet would lead to a risk of deteriorating the connection or security concerns.

While each of these mentioned OTA update architectures has its uses, the proposed model will be based on an **Edge to Cloud** OTA architecture since it allows more time spent on the challenges of a specific device acquiring the update directly and its interactions with the remote server, having to solve all the security-related tasks too.

1.2.2 Requirements and implementaion strategies

As per the past section describing requirements, there are numerous qualitative features to an OTA update that need to be included in order to design a robust process.

Some sort of a **Failsafe mechanism** is practically a fundamental property since, over the course of operation, the OTA process is bound to enter a state where the update either cannot process or has been corrupted. Thus there needs to be a well-defined procedure describing how to approach such situations. Interestingly, OTA possessing a single attribute changes that - **transactionality**.

If the OTA process is transactional in nature and thus supports something of a commit and rollback operations, it:

- *improves* the update process reliability; the update is either "committed" as the currently active version or rolled back to the last good known one in case of any kind of an error
- *rollback feature* can also be used in other circumstances, i.e. in case of a new known exploit introduced in a new update - in this case, the central node administering the updates can mark the vulnerable version as faulty, and the devices will revert to a safe older version if possible as a quick fix
  - however, this might be a contested feature safety-wise - if an attacker finds a way to exploit it, e.g. the old version contains unfixed security exploit, it might lead to the device willingly setting up itself for a future attack

Henceforth, having the OTA process with these properties could be characterised as having ACID attributes. As a result, it can be considered robust and capable of enduring errors during multiple points of use while retaining the functioning state of the application.

Versioning is a helpful, inexpensive tool that helps with achieving numerous aims. Among them is:

- to be able to tell whether the device is up to date or runs an unsafe version - there needs to be some way to keep track of versions
- also, the versions should be able to track other aspects as well, such as whether or not the current version was successfully run, has a known exploit present, uses outdated or unavailable resources or is outright obsolete and should not be used at all

By itself, it might not exactly fulfil any requirements, but it might come into effect, e.g. during rollback operations while choosing the application with a version preference.

Due to OTA update being a resource and time-critical process, the inherent requirement is to use as little processing time as possible. Therefore, having the need to download and process a full update might be, in some cases, detrimental. So **differential update** approach might:

- *save* significant amounts of memory and reduce the time, power and bandwidth to fetch the update - desirable in deployments with low-powered devices with bad downlink connections.
- will require more *system resources* to maintain transactionality and keep versions up to date
- if the updates are sufficiently different, it mostly degrades to a normal approach while also retaining the differential overhead

Though using the differential approach implicitly might not be the way to go since it heavily depends on the specific use case and update data nature, therefore having it as a selectable feature would be preferred.

1.2.3 Proposed OTA process

After outlining the qualities a reliable and secure OTA update process should have in the preceding sections, the moment has come to choose and narrow down which properties and features should such a procedure, mainly focused on serving resource-constrained devices, possess. Due to this constrained nature, the proposal will move incrementally, only adding the essentials that will guarantee secure operations.

Therefore, the main execution steps, and the exact client obligations, of an Edge to cloud-type OTA are

- connecting to an AP
- establishing communication with an OTA server
- downloading the new firmware
- applying the received OTA update

This very basic OTA process, however, does not contain any security features. Thus firstly, both the two communicating parties and the firmware that is being transferred are not guaranteed to really be the requested parties/resource. This process is vulnerable to, amongst others, a MiTM attack due to a lack of server identity authentification and unsecured channel communication due to unsafe HTTP usage.

Furthermore, secondly, in the case of a malicious device of a third party, which is not the intended recipient of the update, it would be possible for an attacker to obtain the

updated firmware itself, potentially undermining the whole subject deploying it. Thus by neglecting the IoT device authentication, the OTA process becomes a potential victim of a spoofing attack.

In order to have a secure OTA process, it is vital to counter these risks. They can be countered by

- introducing public key cryptography through certificates, so it is possible to authenticate the server
  - private certificate is known only to the server. With its help, the server authenticates itself after a request
  - public certificate is known to everyone and used by the device requesting the download to prove the safety of using the offer by the said server
- establishing a secured connection with the authenticated server, e.g. through a transparent TLS layer
  - done only after the server had been authenticated
- encrypting the firmware sent by the server, if the OTA logic on the device is capable of decryption, of course

The authentication that would be done according to the preceding points would be only one way, that of the server to the device. Of course, the authentication of the IoT device could be useful too and could be achieved through another pair of public and private certificates, but that would introduce several problems such as

- having to store the private certificate on the device itself, either alongside or instead of the firmware decryption key
- safely generating and deploying a considerable amount of private certificates while storing the public certificates all on the server, which in a system where the amount of IoT devices is an order of magnitude larger than the OTA server participating hardware, could easily become unmanagable
- not necessarily replacing the guarantees provided by the firmware encryption

Therefore transferring the problem of authentication of the device out of the PKI cryptography realm to a simple requirement of knowledge of a secret necessary for the use of the firmware. This might probably be more resource-friendly as well, since the firmware would have to, in the case of failure of other components, be somehow cryptographically secured, either way, thus creating a kind of redundancy. Therefore writing off the PKI authentication of the IoT device to the server would have a twofold effect:

- *reduce* the workload of the OTA server, which would not be obliged to review each client's authenticity against the specific public certificate of the said device

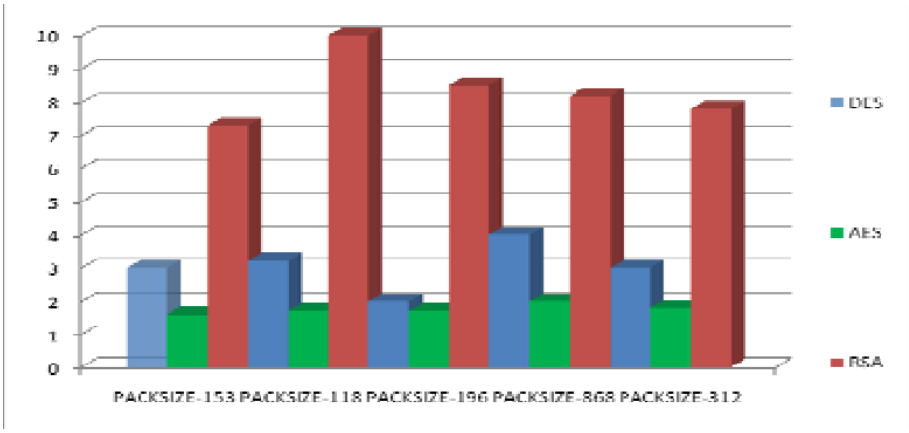

Figure 1.4: AES vs RSA encryption speed comparison [26]

- *increasing* the overall OTA process speed, for not depending on an extra exchange between the server and the device means that the update can be applied sooner

As for the mentioned **encryption of the firmware**, which is the subject of the whole OTA process, it is crucial to have an efficient scheme in place. By default, the simplest scheme would be to, once again, use public-private key pair, encrypting the firmware on the server by the public key and decrypting it by the private key on the device. However, asymmetric cryptography is terribly inefficient if working with any data larger than session information or of similar small data volume, as presented in the figure 1.4, with demonstrated key length of 56 bits for AES and 1024+ for RSA [26].

Since the application binary is in the realm of hundreds of thousands of bytes, this would practically make the whole OTA process unusable. Hence there are potentially two solutions to this problem [27]:

- *distribution and storage* of a single symmetric key per device
  - distribution is a nontrivial problem
  - needs to be securely stored on both ends
  - therefore, inherent problems with device scaling
- *hybrid cryptography* - both the symmetric and asymmetric keys used
  - only the asymmetric key needs to be distributed directly, for which asymmetric cryptography is designed
  - only the private asymmetric key needs to be stored

Even though in terms of performance, the solution with the symmetric key would have a slight edge since the hybrid solution ought to decrypt a small amount of data by an asymmetric key first, other than that, the symmetric key solution would provide no

Security Strength	Symmetric Key Algorithms	FFC (DSA, DH, MQV)	IFC* (RSA)	ECC* (ECDSA, EdDSA, DH, MQV)
128	AES-128	$L = 3072$ $N = 256$	$k = 3072$	$f = 256\text{-}383$
192	AES-192	$L = 7680$ $N = 384$	$k = 7680$	$f = 384\text{-}511$
256	AES-256	$L = 15360$ $N = 512$	$k = 15360$	$f = 512+$

Figure 1.5: AES vs RSA security strength[28]

real advantage, even to the contrary. Unless the device was using the same symmetric key during its whole lifetime, it would need to perform a symmetric key distribution, which by itself cannot be done and needs the help of asymmetrically based cryptography. Also, the need to store a key on the server's end might also prove fatal since the server would have to keep large records of deployed devices with corresponding symmetric keys safely stored in an HSM. Hybrid cryptography solution has none of those drawbacks.

Hybrid cryptography [27], where the practicality of an asymmetric key and the speed of a symmetric key are both retained, is critical for resource-constrained platforms. More specifically, the schema follows the model of public key cryptography, where the encryption can be done by anyone possessing the public key, i.e. the authenticated OTA server in this case, but the decryption can be only done by the party with the private asymmetric key, which is only the trusted device with the said key. However, the main difference if compared to a pure regular cryptographic model using public/private key pairs is that the only asymmetrically encrypted information is the symmetric key. After decrypting the symmetric key, the device is automatically authenticated and implicitly authorised to use the decrypted symmetric key in order to obtain the plaintext [27], in this case, the updated firmware.

However, to maintain security, the server needs to randomise the *initialisation vector* of the symmetric key so that no two binaries will be of the same encrypted value, even if they are equivalent in plaintext [27].

Finally, the question of secure storage of the private keys/secrets - in the current form, the proposed OTA update model uses only one private key on the side of the IoT device, and that is the asymmetric key used for retrieval of the symmetric key of the transferred binary. Therefore, the IoT device has to have some kind of HSM capable of safely storing the said asymmetric key. Also, according to NIST [28], the recommended RSA equivalent in terms of current perceived security strength to a commonly used AES-128 is an RSA of the length 3072 [28]. This is visualised in the table 1.5.

Storing such a key might be problematic for some IoT devices, so exploring alternative options of safely retaining and extracting such a key, such as **PUFs** using from some

unique device property, might be very convenient and useful. More specifically, the main secrets that have to be securely stored on the device during the proposed OTA process are the asymmetric key for firmware decryption and, in the case of mutual authentication, also the device's private certificate. If the device does not possess an HSM that could also operate safely in tandem with the normal device operations (an external HSM module might be unsafe for side-channel attacks), the SRAM PUF really might be the answer since it is already integrated into the typical device and it also provides good statistical properties to go ahead with either storing such a key or even deriving it. However, it all depends on the specific device, the SRAM quality, the manner of use and such, thus needing to be further investigated in such regard.

1.2.4 Usage of a PUF and CTs in OTA

As was already stated, there are numerous types of attacks against the cryptographic techniques employed by an IoT device during the process of updating its firmware. So therefore, in reference to the proposed OTA model, the following will be a listing of such CTs that are paramount to the process:

- **PKI certificates**, or more specifically, asymmetric cryptography
- **Firmware encryption**, therefore, hybrid cryptography

The listed CTs are the two main pillars of the proposed OTA process security. The first point guarantees the authentication of the OTA server and the subsequent creation of a secure communication channel. The second one guarantees the confidentiality of the subject of the communication, the OTA firmware. It also implicitly authenticates the device if it is able to decrypt it and consecutively authorises it for its storage and use.

1.2.4.1 Brief analysis and attacks on the CTs

As hybrid cryptography has been already looked upon in the previous chapter, let us examine the PKI certificate.

Firstly, **Public Key Infrastructure** is a framework, a two-key asymmetric cryptographic system, that does one main thing - it enables its users the use of secure information exchange through its certificates [29]. The reasoning behind the choice of the keyword of *Infrastructure* is mainly due to the fact that it encompasses all the numerous various factors that make this exchange possible. Among them, the following certainly belong to [29]:

- hardware solutions designed to support cryptographic operations
- software solutions harnessing the said hardware securely
- policies that are in place that, e.g. dictate how the specific operations should unfold
- methods and entities needed to make the whole system work. Their responsibilities might involve certificate

- distribution
- verification
- revoking

The certificate is an asymmetric key bundled with other information related to the subject so that the authentication process can take place. Also, the mentioned entities certainly include[29]:

- **Certification Authority** - a trusted member of the PKI. It is the root of trust of the specific certificate chain since it possesses the root certificate only for its use. The main service it provides is authentication.
- **Registration Authority** - permitted by the main CA to perform some of its duties to distribute the workload. Mainly issues specific certificates for specific groups.
- **Certificate store** and **database** - first used as a means to store certificates on an end-user, e.g. an IoT device; the other is used to track details about the issued certificates, also used for later certificate revoking.

1.2.4.2 Vulnerabilities to attacks

The main issues with the **PKI** are not related to the asymmetric cryptography directly (though it might come into question in the future, with the emergence of systems trivialising the underlying cryptographic algorithm) but more to the concept of the chain of trust. Therefore, an attacker would certainly find more success targetting the weakest link in this chain. After compromising the CA, every certificate issued by this CA becomes unsafe and exploitable by the attacker, thus dooming the whole specific PKI-based system [29]. So in the context of an OTA process, there is not much it can be done other than choose the CA issuing the certificates used for authentication during the update carefully. Moreover, again, in the IoT and the OTA context, using self-signed certificates might be actually a completely fine idea.

As for the issues with the *hybrid cryptography*- in the case of correct usage, they all depend on whether the asymmetric key used to encrypt the symmetric key has been kept safe by the device. Therefore, this degrades to the already described issues with safely storing a secret on a device, which might be a real problem if the device lacks an HSM or any other tools to keep the stored private key securely stored.

1.2.4.3 Viable CT to be replaced

As per the already discussed threats, only the securing of the private asymmetric key could be somewhat considered for replacement or implementation on devices without the needed cryptographic infrastructure.

More broadly, constrained devices often have low overall capabilities and hardware tools eligible for implementing such security measures, e.g. hardware support for AES encryption, no HSM present and etc. Nevertheless, what they almost always do have

is an SRAM - thus, one might be able to implement sufficient measures and CT with the help of the SRAM-based unclonable function. More specifically, since a device often needs to store an asymmetric private key, whether for authentication, data signing or firmware encryption purposes, placing such a crucial secret in unprotected flash is unacceptable. Thus having an SRAM-based PUF to either derive the key itself or enrol it after being provided by the admin and store an auxiliary noncritical structure in flash memory that only with it and the help of the PUF the key can be recovered. Therefore an SRAM might be the way to go in the case of storing private keys critical to the OTA update process.

1.2.5 Conclusion

So to finalise - a draft of a lightweight OTA update process was proposed. It highlights the necessary security features that such a process should harness, mechanisms like server authentication, firmware encryption and etc. Due to its focus on constrained devices, the process does not implement additional features discussed before, like compression, or it does not mention the use of transactionality and other reliability features. However, this is of little consequence since they could always be implemented afterwards, but to approach security in this manner would be a whole another issue. Also, as of the constraints such devices often have, usage of a PUF in some capacity as a CT might be very beneficial and allow the device to be more secure with more resources to spare.

Chapter 2

PUFs and IoT devices

Even if an average IoT device wants to operate as securely as possible during all times, it also has to maintain its operational requirements, thus, in most cases, sacrificing something security-wise. Therefore, in the case of devices that already are very resource strained or lack the assets to perform the needed cryptographic techniques, it might be very lucrative and potentially groundbreaking to look at the options a PUF can provide.

2.1 PUF principles, properties and definitions

Every valid PUF needs to possess some quality that is unique and is a source of entropy - no other device, even if being of the same manufacturing line or type, should have the same manner of such quality - it should behave like a fingerprint [30]. At least during normal circumstances, this quality should not be reproducible by any physical means, such as deliberately constructing a device imitating such quality or by modelling the behaviour artificially.

In order to somehow harness this quality, there ought to be some logic layer that, based on input, produces some output based on the specific quality.

In other words, let challenge **C** be a sequence of bits that the PUF circuit accepts as an input. The response **R** will be the output generated due to the underlying quality and that of the challenge. Such a pair is thusly defined as CRP - **Challenge Response Pair**. Also, even if two chips have similar PUF-like qualities and respond to the same challenge, their PUF responses cannot be the same. Therefore, there should not exist a pair of such devices where the response is equal, as it is visualised in the 2.1 figure [30] [31].

Thus, there are several properties a proper PUF has to possess, such as [31]:

- a CRP should be **random**, with no two differing challenges providing the same response
- modelling from a set of CRPs should not be easily achievable
- low attack multiplicity - actually extracting a CRP by an attacker should be impossible to repeat at more than one-time instance

Figure 2.1: PUF principle [30]

- has to satisfy SAC - Strict Avalanche Condition - the probability of the output bit being flipped should be 50% if the input bit has been flipped

If considering the amount and the manner of CRPs a PUF can actually provide, the PUFs can be divided into three categories[30] [31]:

- **weak PUF** - linear increase in PUF circuit size results in a linear gain in the volume of CRPs
- **strong PUF** - linear increase in PUF circuit size results in an exponential growth of the CRP amount
- **controlled PUF** - is based on a strong PUF in the background, with a front end consisting of control logic.

Due to the described nature of the PUFs, they are often used as authentication devices and in other related roles during secure communications [30].

Having a PUF primitive might be especially advantageous on resource-constrained devices, which might lead to an added ability to authenticate or securely communicate while not having any other cryptographic assets capable of doing so. So to satisfy such devices, there needs to be a specific authentication protocol based on the CRP phenomenon. Such a universal protocol is composed of [30]:

- **Enrolment phase**
  1. The server with a database is connected to the device with PUF through an interface directly - might be before device deployment in a controlled environment; therefore possible to focus on speed
  2. The server floods the device with all the possible challenge requests and records the relevant responses
  3. Finally, the database is populated by, ideally, all the CRPs the PUF potentially can produce.
- **Authentication phase**

1. The enrolment is a prerequisite - the server has a table or a database populated with all the CRPs.
2. To authenticate the device, the server sends a challenge for the device PUF.
3. If the response matches the specific one from the CRP pair in the DB, the device can be deemed authenticated.

The described authentication protocol favours the *strong PUFs* very heavily since, with their larger volume of CRPs, it does not matter whether a few are published publically or not. *Weak PUFs* on the other hand, are a lot less suitable for this since they often have only a few unique CRPs, sometimes even only one. They, however, might be a lot more suited for replacing other cryptographic techniques, such as deriving keys [31].

2.2 PUF threats and attack vectors

Since an essential feature of an IoT device is being able to operate in various conditions, it also means they have to be exposed to numerous threats too. Aside from direct physical attacks to communication disruptions and other malicious manipulations of the device's operating conditions, the device can face direct threats to the security provided by the PUF. Of course, a weak PUF faces quite different security threats than a strong PUF since they are mostly implemented to provide for or even implement different cryptographic techniques or assets. Thus in the context of strong PUFs, with CRPs being published between the two communicants [30]:

- **MiTM** attack - no physical access
  - malicious actor can hijack or otherwise get the knowledge of certain exchanged CRPs. Afterwards, the attacker can either perform replay attacks if the implementation is vulnerable or try to, piece by piece, reconstruct a model of the PUF with, e.g. ML
- **Side Channel** attack - requires physical access
  - Categorisation by the invasiveness
    - **invasive** - damaging the device in order to access crucial components requires precision and complex equipment - should result in PUF becoming inoperable
    - **non-invasive** - extract information by exploiting operational data of the device
  - Categorisation by type of activity
    - **passive** - purely observatory in nature
    - **active** - tampering with the device operations, e.g. supply voltage control by the attacker

To aggregate the most important PUF properties in regards to threat actors and the attack models - a good PUF needs to be:

- **Unique**
  - the main reason behind the good security properties and the validity of use during or in place of various cryptographic techniques
  - an ideal PUF should be physically reproducible by neither the manufacturer, designer, nor the attacker
- **Tamper-proof**
  - since the PUF employs the disorder introduced by a manufacturing process, then any attempt to break the package or access the components with the unique properties should result in a change, resulting in the PUF becoming altered or even nonfunctional
  - not all PUFs do possess a bulletproof resistance to tampering, but they are more likely to be resistant than a regular flash or other such hardware
- **Reliability**
  - for the authentication and other cryptographic techniques to work properly, the CRP pairs must be produced consistently and reliably across time. If some responses are to be found dependent on the conditions or other factors, they need to be processed so that on the final output, the reliability is maintained

2.3 Easily constructible PUFs on a common IoT device

Numerous types of PUFs could be considered, such as *arbiter PUF*, *Ring Oscillator PUF* etc.[30] However, most of these require the device to have some external or specially manufactured module that would contain the said PUF circuitry. Therefore, exploring PUF options that might be inherent to a device or its components and thus might be harnessed without any additional physical intervention, purely through a dedicated library. Since most IoT devices nowadays have some form of flash and also SRAM, the contenders are:

- **Flash**-based PUFs
- **SRAM**-based PUFs

There are multiple papers on the viability of **flash-based** PUF [32] with differing approaches and usefulness. However, for those that do satisfy the no modification required, the only thing staying in their way of being used on an average device is their focus on NAND type flash cells. In contrast, an average IoT device uses NOR-type flash cells, and

thus its usability and viability on such flash type seems to be less known. As per [32], the main sources of process variation in flash memories can be exposed through various means; here listing some of those that can be used for a PUF construction[32]:

- *Program Disturb* - by erasing a single block and repeatedly reprogramming a page in the block to affect nearby cell values. This disturbing process leads to unintended programming of nearby flash cells, thus can be used as a source of entropy for PUF
- *Program/Erase Interrupt* - by issuing a program/erase action. Since it normally takes more than one clock cycle, a reset signal can be sent between starting the process and finishing. Thus this results in partially programmed/erased flash cells, which can further be leveraged.
- *Random Telegraph Noise* - use of a type of noise found in semiconductors with specific parameters.

On the other hand, there are PUFs using the very common element of most IoT devices of the present use - the Static Random Access Memory. Unlike the flash memory type, SRAM has very exploitable statistical characteristics that can be harnessed for its PUF use with relative ease. The process that is being utilised by an SRAM PUF is [33]:

- Slight *property differences in semiconductors*
  - caused by the manufacturing process, due to deep sub-micron variations
  - outside of the control of the producer, unique to the built electronics
  - the direct effect is that after the SRAM start, this randomness causes the specific memory cells to initialise with a value influenced by these properties
    - not foolproof, might result in unstable cells with unpredictable startup values
    - the instability may be heavily influenceable by the environment

To summarise, both PUFs based on either the flash or SRAM can be implemented on most IoT devices without physical intervention. However, flash-based PUFs, maybe because of their nature or specific use conditions, are a lot less used on a common device. Thus from now on, this work will focus on the SRAM-based PUF.

2.4 SRAM-based PUF details

The basis of the SRAM-based PUF lies in harnessing the slight variations in the properties of the semiconductor that the SRAM is composed of.

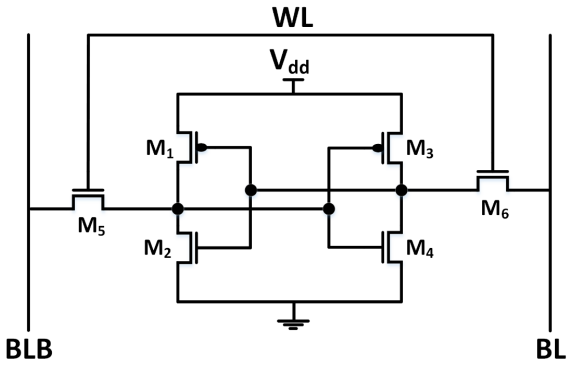

Figure 2.2: 6T SRAM cell [34]

2.4.1 SRAM PUF background

More specifically, a traditional SRAM cell[34] is that of a 6-Transistor type, as the 2.2 figure shows. Due to the already mentioned inherent differences present in the semiconductors, the cell, upon start, will, with a certain consistency, settle at either a logical 0 or 1 state. The exact cause of this behaviour is the mismatch in process variations in the cell transistors [34].

Therefore, such consistency with yet, random settlement states allow, at least in principle, to gather the initialisation states of these cells and use them for cryptographic techniques and procedures. However, the startup noise and other environmental factors will impact the exact settlement state of each of these cells. Thus, not all cells will be consistent, and therefore they can be categorised [34]:

- **low mismatch cells**, being easily influenceable by the noise and surrounding elements
  - these will be the bulk of cells highly inconsistent in assuming the same initial state, thus making their use in a fingerprint type identification of the device problematic
  - however, depending on the consistency of assuming the inconsistent states, it might be viable to consider these cells in the creation of PRNG or even a TRNG number generator [35]
- **high mismatch cells**, which produce sufficient differential drive to overcome the impact of noise
  - the drive might be only up to a point where if the noise gets up to a certain level, then some of these normally stable cells might start to behave unreliably

There are numerous happenings that might result in influence over some of these cells. Among them, there surely belong the ones as[34]:

- **supply voltage** levels
- **long-term transistor change** because of their ageing, or other parts of related hardware
- ambient conditions and other sources of **noise**
  - thermal noise
  - power supply noise
  - crosstalk
  - shot noise
  - telegraphic noise

Therefore, a PUF is considered unreliable if it produces different responses than the ideal ones, or if the errors (bits that are a result of cells that have been influenced by noise) are permitted during the operations, the ability to correct the errors has been neutered due to their sheer volume or severity.

As previously mentioned, with PUFs being divided mainly into strong and weak categories, the SRAM PUF is mostly classified as a weak PUF [34]. The main reason is that a reliable SRAM PUF should give somewhat reliable and very similar responses, thus limiting the amount of CRPs in the general PUF terminology. Also, in contrast to the strong PUFs, where a few incorrect responses do not impede operations, if they are kept under a certain amount, the weak PUFs are expected to provide extremely reliable and error-free responses due to their amount scarcity, as they are mostly used for key generation and such. Therefore, with SRAM PUF being prone to relatively high error rates, the raw value of the SRAM cells cannot be used directly and should be further processed. Due to the volume of most SRAM memories, however, that should not pose a problem since this can be counterbalanced simply by its size.

2.4.2 PUF response error rate improvement tools

In order to counter the said inherent problematic behaviour of the SRAM-based PUFs, there ought to be some procedure to extract the wanted information despite having an error-ridden PUF response. Thus, there have been brought up tools to achieve this purpose, the **fuzzy extractors** and **error-correcting codes**.

Fuzzy extractors - addresses error tolerance and nonuniformity. Functions by extracting a uniformly random string from the input string in an error-tolerant way. Thus, if the input had been changed, e.g. noise in an SRAM PUF response, but still stays otherwise similar to its intended form, then the extracted response will not contain such errors [36].

Error-correcting codes are algorithms that express, i.e. encode the input in such a way that any errors that happen to such an encoded input while it is in such a form will be correctable by the reverse operation, which, out of the encoded input, recreates the original input, i.e. it decodes it. However, if the error frequency is a number that the

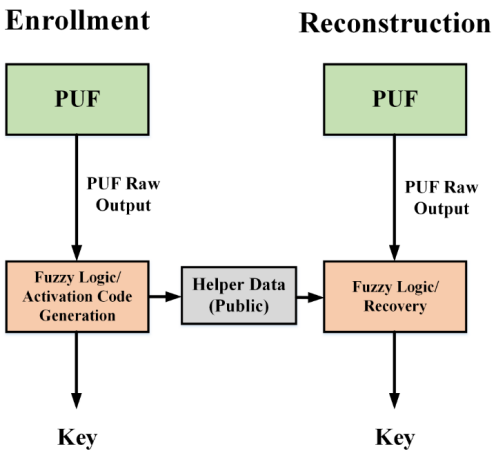

Figure 2.3: Enrolment procedure SRAM PUFs [34]

EEC cannot fix, then the error propagates from the encoded word to the output, thus losing the original input.

In the context of SRAM, PUFs fuzzy extractors and EECs are tools with whose help the PUF response can be made usable and appropriate for cryptographic operations and techniques as a representative of the weak PUF category [34]. With their help, a sort of helper data is produced, which can simplify the more correct and reliable PUF operations in security primitives. In principle, these tools utilise the fact that the length of the SRAM is mostly longer than what is really needed as a PUF response, and thus the extra length can be used as an overhead for these methods to perform their operations, e.g. the EECs typically encode the input in a redundant fashion, thus needing a bigger chunk of memory to store it. An exemplary figure 2.3 for this process [34]:

Chapter 3

SRAM PUF processing framework

Since the availability of hardware primitives that might possess some unique quality with enough entropy and without additional physical intervention are pretty scarce, the main point of dispute is - which one will the library be using. As it was already discussed, PUFs using either flash or SRAM are both possible to be implemented on a typical IoT device. However, aside from the increased complexity of a flash-based PUF, the thing is - that IoT devices typically use the NOR flash memory cell type. Thus, as per the preceding discussion, they are not compatible with some described concepts. In contrast, the SRAM-based PUF is worth the consideration since practically all the IoT devices use this resource. It might come to light that the amounts of the said SRAM will be insufficient or the quality will be inadequate, but that remains to be seen. Therefore, the PUF this library concept will be using is based on the static RAM. The process described in this work uses concepts described and used by the paper [37].

3.1 Enrolment

Enrolment is the first major phase of this PUF secret storage operation. In its conclusion, securely storing the secret, with the use of the SRAM PUF, is achieved and can be recovered afterwards at any point during any operation. The last state the device is let in after executing the enrolment is that a data structure is present in storage and with whose help the secret can be reconstructed. Also, the initially transmitted key must be securely removed. The whole enrolment process starts with:

3.1.1 PUF library provisioning

1. The secret is transmitted along with the firmware. It may or may not be in permanent or not secure storage at first, but after completing the secret enrolment, it must **not be recoverable**.

2. After the first startup, the newly uploaded firmware must start the process of assessing the SRAM.

In order to work correctly, the implementation needs to exclude nonsuitable or non-reliable parts of the memory. After selecting the part of the memory with expectedly good properties, there should be an obtaining of something like an initial snapshot of the SRAM on these selected addresses. This image, which is an unprocessed and error-ridden PUF response from the SRAM, will depict something of an "ideal", upon which the other mechanisms will be partially based and whose purpose is to improve the PUF response quality during the later steps of the enrolment. It would be better to use a fresh PUF response whenever needed in the future. However, this is where the notorious constrained resources of a standard IoT device take a toll - to obtain a different PUF response every time, where a snapshot would suffice, is simply not feasible since it would require:

- an **additional SRAM reset**, or in other words, the restoration of uninitialised values of the SRAM that the PUF response is using
  - depending on the type of SRAM the process is using for PUF generation, this might not mean much since some IoT devices might have some sort of a backup SRAM. For example, *STM32F407*[38] has a backup SRAM that can be powered on and of by will, making it very useful for PUF response generation since the main SRAM can be used independently, and therefore, the program execution does not have to be stopped. On the other side, the ESP32 does not have such options, so resetting the memory will be very costly time-wise and resource-wise.
- a means to save it to **non-volatile storage** since to get the suitable SRAM state, it needs to be purged of everything, even the previous PUF response the process is using
  - this might be a moot point since the initial PUF response is being saved either way. However, it should be noted that a replacement response would have to be saved whenever requiring two PUF responses, in contrast to only one initial saving in the case of the snapshot. So saving every time might be very wasteful, depending on the type of the non-volatile memory used.

After having an initial PUF response saved, a mask should be constructed, with which the same bits can be selected again. Initially having all bits included, it will have certain bits omitted depending on numerous conditions. All these conditions should lead to the processing the PUF response and whose output should be more reliable and cryptographically more secure than just the pure SRAM PUF response.

At this point, there are multiple routes the process could go. More specifically, there needs to be done:

- a debiasing where a substantial amount of bits is excluded
- exclusion of unstable bits

Since the resources are assumed to be constrained, the choice of debiasing method is almost set from the get-go - Von Neumann corrector (VNC) [39] is simple, fast and seemingly effective. Moreover, if the input is deemed as biased, having independent bit cells, the bits can be chained in pairs, rejecting any pairs with the same bit values for both bits and only leaving the first bit from a nonequal pair.

The second goal is trickier - unstable bits can be determined only after extracting the PUF response numerous times and comparing them to the initial image, which is resource and time-consuming. So the choice to debias the bits chosen by the PUF mask is based on:

- if the unstable bits were the priority, due to the VNC debiasing being fast and excluding a large number of bits, it might exclude a lot of them painstakingly calculated after numerous resets, wasting resources.
- the fact that initially, the mask is a string of solely ones, selecting a continuous stream of uninitialised PUF bits. Debiasing here means taking into account their physical placement on the chip, which might be a good idea.
- on the other hand, executing the debiasing after rejecting the unstable bits would mean pairing unrelated bits.

3.1.2 PUF mask and SRAM bits

From now on, the **PUF response** will refer to a continuous stream of bits of a predetermined length. This bit stream will be derived from the uninitialised SRAM bits, upon which a mask is applied. This mask also called the **PUF mask**, is a filtering tool whose goal is to improve upon the raw, uninitialised properties of the SRAM cells. This mask is of the same length as the whole region of the SRAM that is being processed, and its final *Hamming Weight* is of significantly lower magnitude than its initial form, containing every bit the targetted SRAM memory region contains. Thus the library needs to ensure that applying the mask results in a correct stream of bits, not using the bits excluded by the mask. Also, the mask has to be readily available after a shutdown of the device in addition to resilience to the SRAM reset since the library will use it during its whole operation.

3.1.3 Von Neumann Corrector debiasing

There are multiple routes describing how to retrieve a debiased response. If the SRAM quality might seem inadequate or has other problems, one might use more robust counters, such as the more advanced *pair output Von Neumann* (2O-VN) used in [37]. Currently, though, the used technique will be VNC. Also, regardless of when the VNC debiasing occurs, the implementation works as follows:

1. gets a new **PUF response** with the help of the mask. Considering no work has been done on it up to this point, the VNC algorithm can safely operate upon the initial SRAM image.

2. **iterates** over the whole mask where it:
    1. **selects a new bit pair** of the mask - selecting criteria is that they are set to one, and both are either consecutive or as close to each other as possible.
    2. selects a pair of PUF response bits **corresponding** to these two mask bits.
    3. if the response bits are the same, then both mask bits are **discarded** from the mask. If they are not, then only the **first bit is included** in the processed mask, the second one is still discarded.

3.1.4 Construction of a mask of stable bits

After having removed the potentially biased bits from the mask, there needs to be an improvement in the reliability of the responses too. At this point, it is falsely assumed that the initial SRAM snapshot will be consistent with all the responses the library will be getting, which is not valid. Thus, the unstable SRAM bits must be excluded.

The curating of the mask is as follows:

1. The SRAM is restarted, and a new PUF response with the help of the PUF mask is obtained.
2. An array of counters for the included PUF mask bits is constructed. This array must be stored in non-volatile memory since it needs to be recoverable after getting a new PUF response.
3. The initial snapshot is recovered, and the PUF mask is applied upon it, producing a sort of second PUF response.
4. The bits of the recent PUF response and the snapshot PUF response are compared one by one, and the ones that are not equal have their corresponding counter in the counter array increased.
5. After executing steps 1 to 4 as many times as necessary, the mask is after that brought up and cross-referenced with the counter array. Based on how many times the cycle was executed, an upper bound is determined, and the PUF mask bits whose counters do not fall into this approved range are removed from the mask.

After executing the aforementioned cycle, the current PUF mask is final and should be stored permanently, at least in terms of the current enrolment.

3.1.5 Helper data assembly

Having got the PUF mask available - the process can move towards the storage of the secret since, up until now, it was untouched in non-volatile memory. The goal now is to build a seemingly random structure with whose and a PUF response's help the secret can be recovered, thus imitating the function of an HSM.

The process is as follows:

1. A new PUF response is obtained after restarting the SRAM.

2. The secret is loaded into the main memory.

    a. The non-volatile storage where the secret was stored up until now has to be guaranteed not to be able to reconstruct it afterwards in any way from now on.

    b. In the case of using some kind of a file system

        - simply removing the file most probably will not be enough, since virtually always this means that the file is just no longer linked in the file system. It might get overwritten in the future with its blocks now marked as free, but that is of absolutely **no guarantee** that the file containing the secret is no longer accessible or recoverable.
        - overwriting the old file with zeros might not guarantee it has been written over in reality. This might be the case at least on some filesystems, since they might employ some kind of **wear levelling**, considering the flash memory used has often a relatively low amount of program/erase cycles. More specifically, the NOR flash memory, also used in, e.g. ESP32 [6]*(go to /api-reference/storage/wear-levelling.html)*, usually a NOR cell has a life expectancy of at most 100 000 [40], making the wear levelling still a bit of a necessity if one wants to use such a memory actively. So ideally, the whole NVS storage should be formatted securely, since there might be no way of deleting the specific file or object containing the secret in a secure manner.
        - the aforementioned overwriting problem might not be present only at file system scope, but at any NVS built on any flash type memory at all - as per [41] as confirmed by two Espressif employees, even the NVS does, in ESP-IDF at least, have some form of wear levelling in the background, so it should be considered the same as the file system problem and overwritten with zeros securely in its entirety, not just the specific supposed data placement.

3. The secret is encoded into an error-correcting code. The kind and manner of ideal encoding depend heavily on the kind of SRAM we are working with. Moreover, the PUF response should be equal in length to the encoded secret.

4. Perform a bit-by-bit logical exclusive or (XOR) operation between the acquired PUF response and the secret. The output is the helper data string.

In regards to wear levelling - it is a technique used to extend the lifetime of flash memory, be it a NAND, NOR, or any other type of memory with a limited amount of program/erase cycles. It commonly operates on a page level, and it might operate in several modes, such as[42], also displayed in the 3.1 figure:

- **dynamic wear levelling** - works on data block level and only during writing operations. Flash controller keeps track of the writes each block has had. When the controller is required to perform a write operation, it selects a free block which had the least amount of recorded writes. This block is then linked in the place of the old one, and it also marks the old data as invalid. After an indefinite amount of time, every block marked as invalid is freed up by the garbage collector, as illustrated by the image below. So in this manner, memory cells marked as invalid regarding dynamic wear levelling can still contain the secret for some time, even when it should have been overwritten by zeros since only the new active block has the zeros in it and not the invalid one which still has the old data.

- **static wear levelling** - also works on a data block level, but including also those not being written to. It works the same as dynamic wear levelling, but all least used data blocks are moved when certain conditions occur, so these blocks with lower write counts can share the burden. This fixes the issue of dynamic wear levelling where only the blocks currently processed participated, but the untouched blocks tended to keep their low write counts, sometimes even indefinitely.

- **global wear levelling** - uses both dynamic and static wear levelling, but its scope is all blocks on all chips governed by the flash controller. Thus if a particular chip has its data blocks worn out stops being used, and another one with minor wear takes its place.

In conclusion, towards wear levelling - all techniques practically never overwrite the old, formerly valid data with new during normal operations, thus producing some kind of invalid data constantly, which, unless overriding the wear levelling system, cannot be securely erased on demand.

Finally, the helper data generation - the main point of failure is that too many bits have been rejected from the mask, either by too much biased uninitialised SRAM data or too many unstable bits. Due to these factors, the Hamming weight of the mask might be smaller than what is needed to execute a bitwise XOR between the key and the PUF response. Also, after it has been generated, all temporary structures must be securely erased. Operations such as securely formatting the flash file system, overwriting the whole NVS or erasing the whole EEPROM might be necessary in order to avoid some kind of secret or related data leak. Among these temporary structures that have to be securely disposed of certainly belong:

- the **secret** itself for obvious reasons

- the **initial SRAM snapshot**, with whose help the whole PUF could be circumvented since now the attacker would only need the *helper structure*, the *PUF mask* and the *initial SRAM snapshot*. This would circumvent all the effort up until now since the security would not depend on the SRAM PUF at all, so even temporarily storing these three might be preferably avoidable.

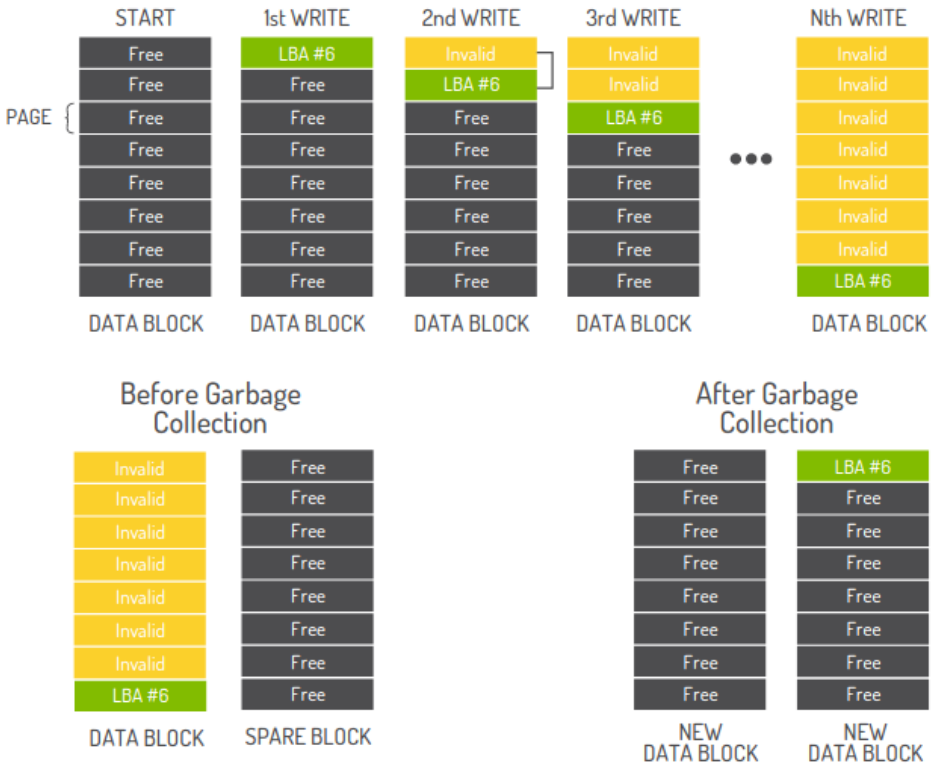

Figure 3.1: Dynamic wear levelling[42]

- the SRAM itself or any other working random access memory or cache which could have at one point had the secret, the initial snapshot or any other PUF response loaded

3.2 Secret extraction

The prerequisite for secret extraction to be possible is to have the program having undergone the enrolment phase and have the helper structure and PUF mask saved in non-volatile memory. Otherwise, it is just a reverse operation in regards to the helper structure generation:

1. With the help of the mask, a PUF response from the reset SRAM has to be acquired.
2. The helper structure should be recovered out of the NVS or similar storage.
3. Perform a bitwise XOR with the PUF response and the helper structure. The resulting bit string should be an encoded word in the error-correcting code.
4. Decode the error-correcting code, thus removing any bits that have been flipped. The result should be the stored secret.

The point of failure of this process could be sixfold:

a. the enrolment did not have **enough mask bits left** and could not generate a proper helper structure.

b. the chosen error-correcting code is **unsatisfactory and under-dimensioned**, thus not managing to repair all the errors caused by the SRAM's inherent properties.

c. not spending enough iterations of the **unstable bit detection** process during the enrolment phase, which would also result in more errors than the correcting code could handle.

d. the **operational conditions** in which the device itself finds itself operating in. It could affect some somewhat unstable bits which, during the enrolment phase, were included in the mask and thus the final PUF response.

e. catastrophic event which fundamentally **changes the SRAM properties**. It might be caused by long-term use or a single severe enough event.

f. **compromitation of one of the device's memories**, NVS storage, or other essential component taking part in the process. It could be caused either locally or remotely, depending on the attack vector.

A lot of these can be countered, sometimes even with the same tools:

1. *lack of mask bits obtained during the enrolment phase*

   - **decreasing the number of iterations** and/or increasing the upper bound of the counter limit during the stable bit finding process.
   - **increasing the volume of SRAM** memory participating during the enrolment - this might not always be possible.

2. *error-correcting code insufficient*

   - **increase the number** of errors the code can correct or change the manner the encoded information is stored - sometimes the errors happenings might be somewhat systematic or localised, thus dispersing or otherwise changing the manner in which the code is represented might help.

3. *too many unstable bits* in the PUF response, making it unreliable

   - **widening the amount** of unstable bit search iterations is ideal.
   - if further reducing the number of unstable bits is not possible, it might be viable to **increase the error count** the correcting code can handle.

4. *varied operational conditions* affecting SRAM performance

- **deepening the robustness** of the PUF responses is the best way to counter the unexpected behaviour of some bits.
  - *stepping up* the volume of used SRAM in the enrolment phase.
  - *mounting the number* of PUF responses which are taken into account during the bit stability determining phase
  - *toughening the requirements* on which bits are considered stable.
  - *building up* the number of errors the correcting code can take care of
  - using a *more robust* error-correcting code more suitable for such an application

5. *foundational changes* to the manner the SRAM behaves

   - the solution might be the same as the previous issue - **increasing the robustness** might be just enough to offset the damaging changes to the SRAM properties if they are small and workable enough
   - however, if they prove to be too severe and/or unmanageable, the only way to proceed might be to abandon the current enrolment, cease the normal operations of the device and undergo a **new enrolment**, adapting to these SRAM changes

6. *compromised device flash*, or other memory

   - cannot be solved by this framework alone; it needs to be addressed with the device's own tools. Instruments like flash encryption, flash download forbiddance and others should prove helpful in the effort.
   - with that being said, unless the attacker gains access either when the enrolment is ongoing and crucial structures are still stored in unsecured memory or right when the key extraction is occurring, the secret should remain safe since helper data and mask are both essentially public data

So to conclude the secret extraction - in order for the SRAM PUF to take on the role of an HSM, it needs to be reliable, robust and, most importantly, secure. These qualities can be achieved by adhering to the principles designed to dissipate the given risks. Of these risks, the *PUF reliability* and the prevention of the secret leakage into unsecured memory should be given extreme attention and scrutiny.

3.3 SRAM PUF framework summary

In the previous chapters, the basis of a framework for an SRAM PUF-based secret storage, suited for a resource-constrained IoT device, was laid. This framework describes

- how to go about the provisioning of the secret during deployment

- by what methods to undertake the refining of the PUF output of the device's SRAM
- in what manner to process auxiliary data structures and what to expect and exploit the given device's memory and storage
- in what way to approach the disposal of the secret the device was provisioned with
- how to extract the secret with the help of the SRAM and the enrolment data structures stored in NVS
- whereby risks should be approached

Chapter 4

Proof of concept implementation

This chapter will provide the context as to how and in what form the implementation details will be chosen. It will explain the choice of the IoT platform upon which the SRAM PUF will be built or the type of PUF that will be chosen. After building the general purpose library, the PUF will be integrated onto a common platform with sufficient testing. Lastly, on this platform, a proof of concept OTA application will be demonstrated using the said PUF during a critical process section.

4.1 IoT hardware platform comparison

In today's market, there is an exorbitant amount of numerous IoT platforms, solutions and frameworks. While choosing a physical platform might be straightforward purely based on capabilities and specifications on paper, the development and further use rely more heavily on the tools that the physical platform supports and delivers. Thus the choice of a solution that builds on all the required capabilities and has, effectively, a self-sufficient ecosystem or, in the case of open-source variants, an involved community.

Some of the more popular but also mature and common platforms might be done by:

- **Arduino** - Firstly, a well-known company in the field, offering solutions in multiple fields, ranging from IDE tools and libraries, and open-source development boards to a multitude of sensors. Also, the category relevant to this work - the range of IoT development products, is very different in terms of capabilities; the most reasonably priced boards mostly do not even offer full IoT connectivity or capabilities by default [43]. Most notably, their MKR family boards possess WiFi connectivity while also delivering a small form factor and low power consumption [43]

- **Espressif** - An established semiconductor company that is developing its own in-house IoT hardware solutions with various wireless connectivities. Like Arduino, their whole ecosystem comes with its development frameworks, easing the development cost[43].

- **Raspberry** - The most capable but also, generally, the most complex and expensive solution. More similar to a computer, Raspberry Pi products stand above others in computing capacity[43].

The platforms themselves are quite different; thus, to compare them for easy usage in the IoT sector depends on:

- open source system
- budget price
- availability
- development tools

Both Arduino and Espressif provide open-source software solutions, while Rasberry Pi does not. Regarding pricing, the highest amount of capabilities provides Espressif's ESP32, while Arduino MKR has a higher price tag and would mostly struggle to execute things as smoothly. Regarding the Raspberry - since it uses a fully-fledged OS, unlike the Arduino, which only has a simple main loop and Espressif, which uses FreeRTOS - it does provide a lot more, but at an incomparable price difference with also possibly being harder to use since the regular OS shields the hardware capabilities. Therefore, an open-source platform is financially accessible and provides community support and good development tools with access to low-level hardware features is exclusively the Espressif's ESP32.

4.2 Library implementation details on the ESP32 platform

There are several requirements placed on the development tools from the get-go. First would be the potential interoperability with numerous other platforms other than ESP32. Thus, this requires the library to be able to run independently of the chosen ESP32 toolkit. However, this is still very closely tied to the ESP platform itself since it is the platform of choice, and it needs to be able to run the library efficiently and seamlessly. So there must be a compromise between a high-level language, a platform suited for developing a library more rapidly with less time spent on low-level technical aspects and the fact that it needs to run on a resource-constrained system where such a choice could be impassable.

4.2.1 ESP32 developement tools

Currently, various development frameworks and platforms are built upon each other in varying states of upkeep, developer community engagement and maturity. Almost too many to count for this work, the most notable and prolific ones would probably be :

- **MicroPython**, a lightweight Python port for embedded devices
- **Arduino** framework, a higher level abstraction framework based on C++

- **ESP-IDF**, the Espressif's official framework based on C/C++

There are others, such as, e.g. TinyGo. However, they are mostly unusable for this project since they lack basic features like WiFi support [44], or Espruino, which seems pretty ambitious but lacks OTA support [45]. Alternatively, in turn, they are practically just proof of concept-like projects, comparable to various Rust use efforts [46], that endeavour needing users to, essentially, port their solutions from the official framework by themselves if they want to use something ESP32 resource-specific in the language that they are porting to.

Before the comparison starts, there need to be laid down some basic requirements placed on the tools:

- **Direct access to the SRAM** memory and the capability to use it as soon as possible since the bootup
- **WiFi capability**
- NVS or similar **storage capabilities**
- some means of an **OTA update process**

Firstly, the embedded Python port that is the most well is the **MicroPython**. Using a language implementing the key features of Python 3 brings the biggest blessing and the biggest curse into the fold, a dynamically typed, garbage collecting language on an embedded device.

The main feature of the MicroPython is the **REPL** [47], which stands for *Read Evaluate Print Loop*. This tool allows programmers to develop and improve software rapidly since all it takes is to upload the updated script file without requiring lengthy compilation since it is an interpreted language. Also, this way, the firmware does not get replaced along with the user code since the only thing transferred is the user script. Though it also allows the modification of the underlying C/C++ functions, since it is still built upon the ESP-IDF and FreeRTOS, at least the variant operable on the ESP platform [48].

Outside of the quick development cycle, it also features easier development because it has garbage collection and is dynamically typed. This comes with apparent costs since both are normally quite expensive, at least resource-wise - the MicroPython has at least taken the path of compromises, using less resource-demanding means to achieve these [49]. Also, due to it having taken over the role of managing memory, the developer has a lot more limited direct access to memory[47].

Furthermore, all these features have a high cost. The ESP platform might have enough resources to support running the REPL loop, the interpreter and other essential MicroPython components effectively; however, doing so impedes direct memory access, and the memory management of the MicroPython might seriously disrupt the functionalities of this library.

Arduino is an abstraction-level framework that allows users to reuse their codebase between numerous IoT platforms, regardless of their actual capabilities. It uses C++

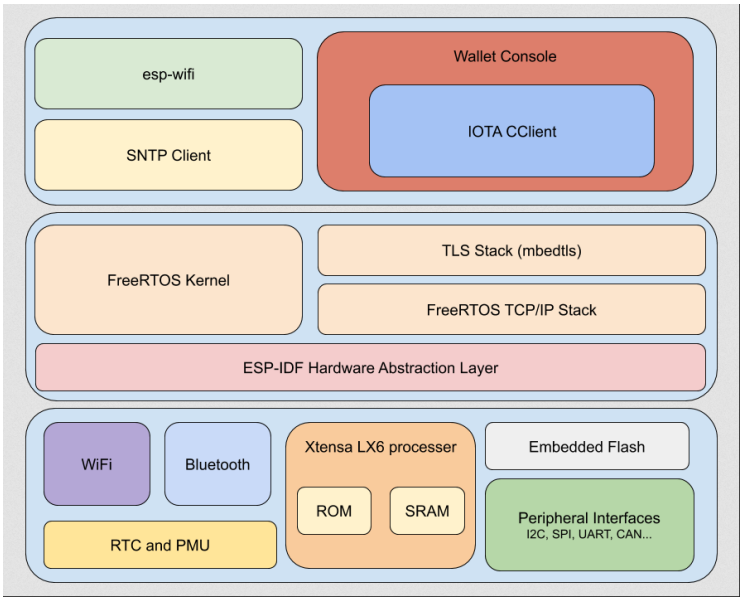

Figure 4.1: Illustration of an ESP32 architecture[51]

as the language of choice with numerous tools allowing rather effortless use of system resources and easier time during development. It exposes one unified *Arduino API*, where the specific platform that wishes to make use of it has to have its implementation fulfilling this API, the so-called **Arduino core**.

In the case of ESP32-specific Arduino core, this approach also has some negatives. Considering the ESP32 has a lot of configurable customisable settings and approaches to development, it has a tool called *menuconfig* within its ESP-IDF toolkit. This tool is a front end for the unified configuration system, which greatly affects how the compilation proceeds, what Espressif components are involved, and what form and mode. It is a powerful tool. However, the Arduino framework does not allow its use since it uses precompiled binaries of the ESP-IDF framework [50], thus limiting the development and use of the Arduino framework greatly. Theoretically, the binaries can be recompiled by the user on their own and replaced, but this approach is not organic and does not guarantee that something will not break. Also, since this framework depends on the precompiled versions of the ESP-IDF of a certain version and is also maintained by a separate team of people, it might lag behind the cutting edge branch, but also the organic releases of the ESP-IDF framework, practically **locking out** people out of features or potential security fixes for quite some time.

ESP-IDF is the official ESP32 development framework. It is a low-level framework, directly interacting with the hardware and thus is practically in its entirety written in plain C, with some exceptions. As per the 4.1 figure, it provides a hardware abstraction layer and thus, unless one wants to reimplement this, the FreeRTOS and other integration, then every tool will have to use ESP IDF in some form, as all the preceding solutions did as well.

Since it is the official solution, it has the largest community and support, with actual Espressif employees and developers present on their official forums, giving it an edge in this regard. And most of all, if the ESP32 has a capability, then ESP-IDF can provide the framework for using it. Also, if a new feature is implemented, one can simply clone the official main branch, which is supposed to be stable, and use it immediately and not wait for an official release that often has long-term support. The other two tools might take months to make the features accessible in their frameworks.

It has some cons; being written in plain C poses some developmental challenges to most long-term projects, but for this work, it provides almost hardware-level access with all the other required features, thus making it a bit of a non-choice.

4.2.2 SRAM PUF library

Due to the ESP32 framework choice of ESP-IDF, the library will have to call plain C functions at least in some form and use C-like interfaces. Also, since the library is intended for use on IoT devices which are very often heavily resource-constrained, it needs to use less resource-intensive primitives and tools more friendly toward such devices. Thus the library will be written in a hybrid C/C++ style, where some concepts like namespaces and classes for faster and more convenient use will be applied, but most primitives and concepts will be C-like, such as

- use of **pointers** instead of references
- adoption of **plain arrays** and not containers
- strict use of **C libraries** only, etc.

It is this way because, if necessary, the library could be switched to plain C rapidly, saving up much-needed resources that the application would need to have allocated elsewhere.

4.2.2.1 Development

As for the work on the library, the first step in the making of the library was to make a helper ESP32 application and use it to retrieve an SRAM snapshot image, hopefully getting some amount of uninitialised memory after isolating favourable chunk that seemed to behave, so it was decided that to simulate subsequent resets it was more cost and time effective to just simulate them by our own. This was done with a Python script with a user inputted amount of bits flipped in a block, not exactly emulating the real SRAM but being satisfactory enough for these purposes.

After analysing the obtained static memory and using its permutations during the process, it was also necessary to ensure that the processed PUF response would not obtain any kind of **bias or patterns** due to potential information leakage. Alas, during the first stages of the project, it was an inherent issue to produce a processed response without a bias. More specifically, the output was composed practically only of hexadecimal values

of 4,5,9 and a. This happened due to a faulty implementation of the Von Neumann corrector, where the issue was of the input passed; specifically, if the input bits were not the same, instead of only using one of them, the output did not do that, but it did use both of them. Thus, this had resulted in output only possessing data of permutations of bit pairs 01 and 10, thus the aforementioned hexadecimal values.

Another problem was to set a **unit of access** of the SRAM since some memory might be restricted to a specific bit length. And indeed, according to [6]*(go to /api-reference/system/mem_alloc.html)*, the IRAM is only 32-bit addressable; thus, setting a global bit length of the main word with which it is being worked would be the only means of directly accessing the uninitialised SRAM by the library.

Afterwards, the issue was more of determining the ideal form of **error-correcting code**, the manner of information storage and the technical parameters, like the number of iterations for the search of stable bits, the volume of initial memory considered to be needed to derive the needed helper structure that would help restore the secret etc. Most of these, though, would be questions for the next stage, where the library would get deployed and tested on actual hardware.

4.2.2.2 Getting SRAM uninitialized state

To even get the chance to perform any action, the library needs to somehow force the SRAM into its uninitialised state where the PUF properties observations can be done. Thus actually cutting the power to the SRAM would be the required operation. On some platforms, this line of action might be not exactly straightforward since a simple software solution might not do the trick, for after cutting the power, the device needs to restore it in a short interval and retain some sort of state to continue the process that had been ongoing (enrolment or execution). Thankfully, the ESPs support many sleep modes which might be able to fulfil these requirements. The supported modes are [52]:

- **Modem sleep** - only WiFi, Bluetooth and radio are cut from the power supply, with limited wakeups to maintain connections. No power supply changes to SRAM.
- **Light-sleep mode** - the peripherals, CPU, and most of the RAM are clock gated - the flip flops (of which the SRAM is also composed) do not switch states since that is the main power consumer. Thus the flip flop states are maintained.
- **Deep sleep** - only modules that retain power and operate normally are the RTC controller and its peripherals, the ULP co-processor and the RTC memories. Thus the SRAM, no longer maintained with a power supply, loses its state.
- **Hibernation mode** - ULP, RTC memory and everything else outside of one RTC timer and RTC GPIOs are inactive. SRAM gets into the uninitialised state.

Of these modes, only two perform the SRAM uninitialization, the *Deep sleep* and *Hibernation* modes. However, the Hibernation mode does not maintain any kind of memory that would be readily available on wakeup. This might be circumvented by persisting the

application state into the flash that does not need any kind of power to maintain information. Nevertheless, since the **Deep sleep** mode does maintain RTC memories on entering the sleep [52] and is accessible in the same state on wakeup, choosing the Hibernation mode does not make much sense in the context of the PUF library.

4.2.2.3 SRAM PUF library ESP32 integration

Before even adopting the library, the main point of interest was to determine the starting address of the SRAM that would be used in this project. After enabling system logging at the info level, the bootloader starts to provide numerous additional clues, such as SRAM addresses. After that, the average addresses of the SRAM used to look like this (note - actually used device was the ESP32 WROWER B):

Table 4.1: Example of SRAM address layout

Type	Address	Length	Size
DRAM	0x3FFAE6E0	0x00001920	6 KiB
DRAM	0x3FFB8578	0x00027A88	158 KiB
D/IRAM	0x3FFE0440	0x00003AE0	14 KiB
D/IRAM	0x3FFE4350	0x0001BCB0	111 KiB
IRAM	0x4009997C	0x00006684	25 KiB

However, using the biggest block or even chaining them up together and somehow making use of them in that manner was not the way to go since the objective was to use **uninitialised** SRAM data. Thus, the purpose has become twofold - to find the best moment to access the SRAM while it is in its uninitialised state or one that is as close as possible to it and the other to determine which SRAM type was the best to use.

Finding the ideal place to access the SRAM as PUF was not exactly straightforward. On paper, the best way to do so would be to utilise the **bootloader**. Since the ESP32 has the first stage bootloader ROM-based [6]*(go to /api-guides/bootloader.html)*, the next best chance would be the second stage bootloader. Furthermore, on the ESP32 platform, the SSBL is located in the flash on a predetermined address and is **user-modifiable**, at least in the form of hooks. These hooks are of two kinds, one launched before the bootloader itself and the other right after. The first kind would be ideal since it allows the user to access practically unaltered uninitialised memory, only touched by FSBL. The main point of the FSBL is to load the SSBL into memory, and because the SSBL is located at a given address with a well-known and modifiable size, the effects of the FSBL are very predictable. Even more so, since the default maximum SSBL size is 0x8000 bytes[6]*(go to /api-guides/bootloader.html)*. Furthermore, the Secure Boot V2 feature limits the size at absolute 0xC000 bytes, narrowing the unpredictability of safe applications.

However, using SSBL to capture the SRAM PUF response is not exactly viable for many reasons. Firstly, the init hook is executed before all the module and system resources have been loaded properly, limiting the features greatly. Thus, if the library were to be executed fully in the init SSBL hook, it would not be able to store the helper and mask

structures since no NVS has been initialised. Even in the case of somehow getting the flash initialised before the SSBL runs - it might be possible to write at an address directly, but that would complicate any manipulation with such data afterwards and also bypass wear levelling that would normally be present on a file system or a configuration type NVS. Another option might be the SRAM itself, but this approach would be highly unsafe and potentially not even feasible: There is no user-friendly way, or as far as is known, no way how to pass a properly allocated main SRAM data from the bootloader to the main application. Writing the data into unallocated regions would probably result in an application crash, thus making such an application unusable even due to this mere possibility. Using the post SSBL hook does not make much sense since after the SSBL executes its purpose, which is to initialise and load the main application, it grants control to the main app, at any rate, thus making it more convenient to avoid the SSBL hooks altogether.

4.2.2.4 Usage of a DRAM block for PUF

With the SSBL out of the picture, the question now is - how much of the SRAM is still uninitialised? Surprisingly, depending on the kind of program, it might be quite a lot. If we look at SRAM types specifically, even though their size and designation vary quite a bit, it still designates most of them to the second DRAM block. The amounts and addresses vary from time to time, but it mainly depends on either the ESP-IDF version, the application size/layout, selected flash size or other build factors. For example, sometimes, the second DRAM block would get even as much as 180 KiB, but it never got less than 158 KiB. Also, the latter SRAM blocks with the IRAM designation would, which is evident by their name, contain loaded instructions; thus, even accessing such memory for reading purposes sometimes leads to failure. In all this, the most stable and suitable one would be the 158KiB block.

The largest DRAM block would practically never be at an address different than the one mentioned. However, it might still vary, but it was never far off during countless attempts to access the said memory. The current range of assignment was determined to be from 0x3FFB2C88 to 0x3FFB8578, thus making a safe offset of around 0x6200 of the lowest recorded address to be the best course of action to use as a starting point.

On the matter of using the said block during the PUF response extraction process - there are some caveats. Namely:

- it needs to be guaranteed that no action by the library would result in modifying the to-be-processed DRAM region of the SRAM
- that no action before or during the moment when the library gains control would result in similar modifying action

To counter the reliance on chance - the best bet would be to exclude the said memory region from being a part of the DRAM, which sadly is impossible. Nonetheless, this need could be reduced to just being protected from getting used by the system `malloc` function or, in other words, redirect dynamic memory allocation elsewhere. The simplest

solution would surely be to just use static allocation to avoid using the DRAM if possible. However, the sizes needed for such arrays would go far beyond what the ESP-IDF can allow and would probably result in a non-compiling program or a stack smashing runtime error. So the next best thing would be not to use the SRAM at all if possible and use another RAM if possible, which it is. Thus, using flash as an SPI-connected RAM for all dynamic memory allocations done by the library and other runtime purposes could be the most sensible solution in this scenario. It could be even possible to embed some sections of the binary that would normally be loaded into the main memory, such as the .bss or .noinit segments [6]*(go to /api-guides/external-ram.html)*. This does not come without a cost, though, since the SPI interface is shared between other modules, and allocating memory blocks there dynamically might make it inaccessible during some situations, resulting in failure [6]*(go to /api-guides/external-ram.html)*.

As for the issue of persistence and data storage of the initial snapshot and the helping structure, there is practically only one built-in solution that allows allocating enough space to hold large enough structures - the flash. And again, the simplest and probably the most secure solution - to write at a static flash address and null it after the validity of the data has expired - is not possible due to the innate limited flash characteristics, requiring wear levelling. There are multiple solutions to this, as there is the ESP32 support for both the FATFS and SPIFFS, and also the NVS might be acceptable. However, the last-mentioned has a fatal drawback - it is designed to store key-value pairs, which is not inherently bad, but seriously limits the general implementation that the library is focused on and the fact that it already uses file API, and this would require a connector style interface built upon the NVS to circumvent the storage difference.

Otherwise, the two main embedded filesystem implementations available to be used on flash with the ESP32 officially are quite different. The main differences between the FATFS and SPIFFS file systems are [53]:

- **SPIFFS**
  - light weight
  - reliable, e.g. safe failure on power loss
  - designed with wear levelling in mind
  - no encryption support
  - no true folder support. Everything in root folder `/` as per official ESP documentation.
- **FATFS**
  - not designed with flash in mind, not reliably safe during a power loss
  - ad hoc wear levelling, just in the layer between the FAT library and the flash itself
  - flash encryption support

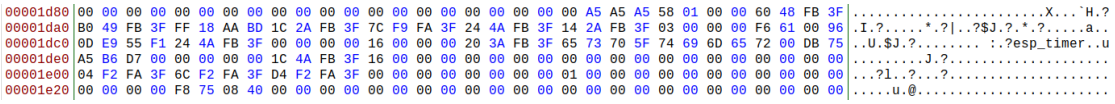

Figure 4.2: Esp_timer API/function name

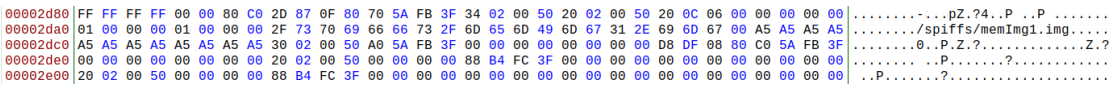

Figure 4.3: Snapshot saving pathname

– true folder support, since it is a FAT implementation

Overall, the SPIFFS seems like a better-suited file system since it is designed for flash-based storage. The encryption support might be significant, but considering the SPIFFS has a lower memory footprint compared to FATFS, which is probably due to more complex implementation because of the folder support, it might be better to use the SPIFFS. Even more so because the folder support is not needed at all, the reliability issues and finally, the bottom line is that the main need is for the PUF library to have access to as much unaltered SRAM as possible, and the lower memory footprint makes it more appealing.

And lastly, regarding snapshotted SRAM state of the said DRAM region, it depends on what kind of code was executed before. With actually limiting what could have been run to a bare minimum before taking a snapshot of the SRAM, it can be reduced to a few items, like the SPIFFS initialisation, a few function calls with integer-sized arguments, using minimal dynamic allocation on SRAM and mainly on SPIRAM pretty favourable results can be achieved. More specifically, after analysing several snapshots in differing configurations, it seems that at most around 0x5000 bytes but no more than 0x6000 bytes from the targetted DRAM block starting address are used for stack or similar memory allocation purposes by the program.

This is reinforced by the fact that after analysing the SRAM snapshots, whose similar state that is being dissected here would be used for the PUF response construction, they do contain things like

- **function/API** names like *esp_timer* 4.2 (possibly somehow used internally by either *esp_sleep_get_wakeup_cause*)
- **the presence of strings** used by the SPIFFS library like pathname of the saved snapshot, snapshot name itself and the spiffs volume name. Displayed by figures 4.3, 4.6 and 4.5
- **partition table** labels with addresses 4.4

The specific snapshot described here and in these images is saved and located in `attachments/measured_data/memImg1.img`.

This hypothesis is also supported by the way the FreeRTOS works memory allocation-wise and the way the ESP-IDF is used. Furthermore, in reality, FreeRTOS is pretty

```
00003fe0 F4 13 FB 3F 01 00 00 00 02 00 00 00 00 90 01 00 00 60 00 00 6E 76 73 00 00 00 00 00 00 00 00 |...?.............`..nvs.........|
00004000 00 00 00 00 00 00 00 00 00 00 00 00 9C 6C FB 3F 30 00 00 00 F4 13 FB 3F 01 00 00 00 01 00 00 00 |.............l.?0......?........|
00004020 00 F0 01 00 00 10 00 00 70 68 79 5F 69 6E 69 74 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |........phy_init................|
00004040 D0 6C FB 3F 30 00 00 00 F4 13 FB 3F 00 00 00 00 00 00 00 00 00 00 02 00 00 00 10 00 66 61 63 74 |.l.?0......?..............fact|
00004060 6F 72 79 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04 6D FB 3F 30 00 00 00 F4 13 FB 3F |ory..................m.?0......?|
00004080 01 00 00 00 82 00 00 00 00 00 12 00 00 00 20 00 73 74 6F 72 61 67 65 00 00 00 00 00 00 00 00 00 |.............. .storage........|
000040a0 00 00 00 00 00 00 00 00 00 00 00 00 54 00 00 00 5C 76 FB 3F 90 6D FB 3F 04 6D FB 3F 2F 73 70 69 |............T...\v.?.m.?.m.?/spi|
000040c0 66 66 73 00 00 00 00 00 00 00 00 00 00 00 00 00 14 B9 0D 40 4C B9 0D 40 84 B9 0D 40 00 00 20 00 |ffs..............@L..@...@.. .|
000040e0 00 00 00 00 00 10 00 00 00 10 00 00 00 01 00 00 58 74 FB 3F E8 6D FB 3F F0 00 00 00 DC 6E FB 3F |................Xt.?.m.?.....n.?|
00004100 78 05 00 00 54 00 00 00 00 00 00 00 00 90 6D FB 3F 84 5B FB 3F 00 00 00 00 00 00 00 00 A8 6D FB 3F |x...T.........m.?.[.?.........m.?|
00004120 FF FF FF FF A8 6D FB 3F A8 6D FB 3F 00 00 00 00 BC 6D FB 3F FF FF FF FF BC 6D FB 3F BC 6D FB 3F |.....m.?.m.?.....m.?.....m.?.m.?|
00004140 00 00 00 00 01 00 00 00 00 00 00 00 FF FF 00 65 00 00 00 00 FF FF 3F B3 00 00 00 00 F0 00 00 00 |...............e......?.........|
00004160 5C 76 FB 3F 01 00 1F 80 FF FF FF FF 0F 00 0F 00 00 00 00 00 00 00 00 00 00 00 00 00 14 00 00 00 |\v.?............................|
```

Figure 4.4: Partition table with addresses

```
000047c0 FF FF FF FF FF FF FF FF D2 05 FF FF 00 02 00 00 1F 80 00 00 F8 00 00 00 98 46 00 00 01 2F 6D 65 |.........................F.../me|
000047e0 6D 49 6D 67 31 2E 69 6D 67 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0A 00 00 |mImg1.img.......................|
00004800 00 11 00 12 00 13 00 14 00 15 00 16 00 17 00 18 00 19 00 1A 00 1B 00 1C 00 1D 00 1E 00 1F 00 21 |...............................!|
00004820 00 22 00 23 00 24 00 25 00 26 00 27 00 28 00 29 00 2A 00 2B 00 2C 00 2D 00 2E 00 2F 00 31 00 32 |."#.$.%.&.'.(.).*.+.,.-./.1.2|
00004840 00 33 00 34 00 35 00 36 00 37 00 38 00 39 00 3A 00 3B 00 3C 00 3D 00 3E 00 3F 00 41 00 42 00 43 |.3.4.5.6.7.8.9.:.;.<.=.>.?.A.B.C|
00004860 00 44 00 45 00 46 00 47 00 48 00 49 00 4A 00 4B 00 4C 00 4D 00 4E 00 4F 00 51 00 52 00 53 00 54 |.D.E.F.G.H.I.J.K.L.M.N.O.Q.R.S.T|
00004880 00 55 00 56 00 57 00 58 00 59 00 5A 00 5B 00 5C 00 5D 00 5E 00 FF FF FF FF FF FF FF FF FF FF |.U.V.W.X.Y.Z.[.\.].^...........|
```

Figure 4.5: Image name

```
00004ac0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................................|
00004ae0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2F 73 70 69 66 66 73 00 10 B9 B1 5E 4F B7 88 AA |.............../spiffs....^O...|
00004b00 07 00 00 00 38 6D FB 3F 02 00 00 00 54 00 00 00 00 00 00 00 98 77 FB 3F 84 5B FB 3F 01 00 00 00 |....8m.?....T........w.?.[.?....|
00004b20 00 00 00 00 B0 77 FB 3F FF FF FF FF B0 77 FB 3F B0 77 FB 3F 00 00 00 00 C4 77 FB 3F FF FF FF FF |.....w.?.....w.?.w.?.....w.?....|
```

Figure 4.6: Spiffs volume name

much like most applications in managing their memory, with the stack having the highest address, which decreases as it grows and the heap with a lower address with increasing address as it grows, thus both growing against each other. However, since the FreeRTOS is designed around tasks and is not a simple application, there are differences, such as that each *xTaskCreate()* call allocates a memory block within the heap, even if the application does not use any kind of multitasking or multithreading [54]. Also, this allocated task space has its own stack, which is important. The whole mechanism is illustrated in 4.7.

So due to this task-related caveat, the observed artefacts in memory might not, in fact, be related to the global stack but to a local, task-owned stack placed inside a Task Control block, thus located on the global heap. This might be possible if the address location and growth directions are reversed between the stack and the heap as described before, which is plenty possible. Also, this consideration has taken into account only the largest DRAM block; the situation in others would probably shed more light on the subject.

So, in summary, the selected DRAM block seems to be a safe pick for uninitialised SRAM collection during the run of the main application. This is possible due to call as little unnecessary unrelated code as possible, reducing the size of the dynamic memory structure overlaying it and thus, by adding a simple offset, the nature of the uninitialised SRAM cells selected by such an offset should be preserved. And just a note - there might be ways actually to reserve the DRAM region of the SRAM, thus making sure the memory stays uninitialised [6] (/api-guides/memory-types.html#noinit-dram).

4.2.2.5 Library PUF metrics module

In order to determine how much the proposed SRAM region selection and the PUF response processing were effective, there needs to be some sort of a benchmark. Being a part

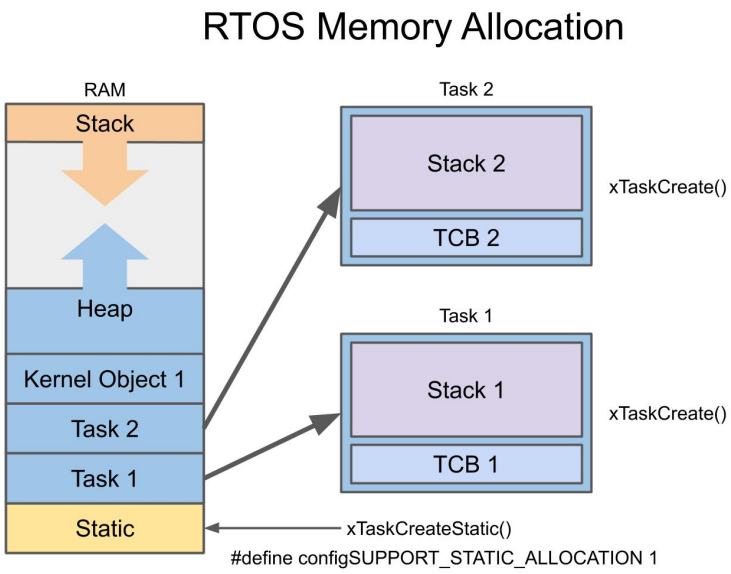

Figure 4.7: FreeRTOS memory management [54]

of the library, the tool should provide feedback on how the other tools have been effective and further determine the execution parameters and help with how to approach real deployment.

Some multiple metrics and qualities might be very useful in determining the efficiency of the SRAM PUF; among them, the noteworthy ones might be [55]:

- reliability
- PUF uniqueness across multiple devices
- bit uniformity across the response
- bit uniformity across multiple devices

Some of these will be quite impossible to use during the development, notably the uniqueness and bit uniformity for a single cell across multiple SRAM PUFs - this would require a lot more focus on this subject since it would involve acquiring multiple devices with independent SRAM PUFs, preferable from both the similar batches and completely unrelated ones to determine real-life usefulness. Thus this restricts us to the reliability of the PUF response and bit uniformity.

Reliability is the prime metric of a PUF, regardless of whether it is based on PUF, flash or another physical component. It mainly affects how useful it would be in real use and how many resources it would require to adopt for secret storage, key generation and other applications. The main goal of reliability testing is to determine how robust and

noise resistant the PUF really is. It is done by comparing the difference between two responses divided by the replies' bit length [55]. The formula specifically is

$$Rel(x_i) = 1 - \frac{HD(x_i, x_0)}{len}$$

Where x_i is the specific PUF response, x_0 is the initial snapshot, len is the reply bit length, and HD is the Hamming distance. Hamming distance is implemented as Hamming weight of a bit string that was obtained by bitwise logical XOR between the x_i and x_0 PUF responses.

The output value of this function will be between 0 and 1, e.g. if the x_i and x_0 are the same, they would have the HD of 0, and otherwise, if one response were a negation of the other, then each bit would be different and this would result in a value of 1. Thus division by len brings the value out of 1 to bit length into a ratio dimension 0 to 1. Furthermore, since the most reliable PUF in this context would be the one giving out as similar responses as possible, then subtracting the result from 1 gets a reliability ratio, or multiplying by 100 a percentage. Getting the value as close to 100% should be the library PUF response processing goal.

Bit uniformity is a second metric determining how suitable the PUF would be for various cryptographic use cases. The reason is - if the distribution of ones and zeros in the PUF response is not equal or uniform, it would mean there is a certain bias for ones or zeros, which, depending on the amount of such a bias, could be exploited by an attacker. It is calculated as the Hamming weight of a reply divided by the bit length [55]:

$$BU(x_i) = \frac{HW(x_i)}{len}$$

The formula basically counts the number of ones in a PUF response and divides it by the bit len of the response. Thus, the output is between 0 and 1, where only if the response is composed of only ones gets assigned one on output and vice versa with zeros. Thus, for a good bit uniformity, the number of ones and zeros should be roughly of the same order, thus having the output of 0.5, or 50% bit uniformity.

This testing, however, does not determine how well the bit ones and zeros are dispersed in the reply, just their overall amount. Also, even though the reliability should be aimed at as high as possible, getting the same response would take the unpredictability factor out of the equation entirely, which might not be exactly wanted.

4.2.2.6 Library integration and use on ESP32

After solving all the preceding problems - which SRAM block to use as a source of PUF, at what point during execution to start working with the SRAM data to avoid overwriting, how to persist the library structures and how to determine whether the PUF responses have appropriate qualities - it has come to incorporate them into a testing deployment, where the specific PUF processing configurations could be fine-tuned while also receiving feedback on its performance.

Thus, the main goal of this step was to produce a single entry point of the application, where the action would be selected, such as:

- **secret enrolment**, where the algorithm would try to make do with the provided SRAM range and, in the case it was not enough, would print how much more was missing

- subsequent **extraction** with a form of feedback on whether the extracted key is correct and, in the case it is not printing the number of faulty bits

- a tool for creating **snapshots of multiple uninitialised states** of the SRAM memory, which can, after its conclusion, be uploaded from the ESP32 for further analysis

- a tool for **analysing the effectiveness** of the enrolment and for determining how do the PUF responses improve after being processed with the library

- a tool for **testing performing** the quality testing metrics

The most straightforward implementation of securing these requirements is a single prompt during the start after flashing the firmware. Hence the complete deployment can be tested momentarily, with the secret enrolment procedure with the following key extraction in tow. As a consequence, since the regular operation would be a one-time upload of firmware and the secret, performing the enrolment and finally storing the helper data with the finalisation being the secret removal, there does not need to be anything well defined after executing the enrolment and a test extraction, since the key would have to be reuploaded.

4.2.2.7 Library integration testing on ESP32

The library tests and discussion will focus on three main aspects - the optimal parameter configuration, execution speed related to the secret being enrolled and the quality metrics.

Firstly, the parameter configuration. Outside of the choice of the used SRAM memory region, the practically only other parameters directly influencing the type and effectiveness of the library operations are macros.

- `STB_BITS_SRAM_RESETS`, defines the number of iterations of the SRAM memory mask refining algorithm, where each cycle, the SRAM gets uninitialised and compared to the initial snapshot, and any flipped bits are excluded. Very time expensive, one SRAM reset with a flash write costs around 5 to 7 seconds, depending on the type of ESP configuration and optimisation

- `REP_CODE_MAGNITUDE`, defines the order of the simple error-correcting code that is used to encode the secret before bitwise XOR-ing it to the PUF response. The code is `REP_CODE_MAGNITUDE`/2 error detecting and `REP_CODE_MAGNITUDE`/2 - 1(if a rep is even), just a simple error-correcting code. It is very resource expensive, mainly memory-wise, and significantly raises SRAM volume requirements.

- `EXPECTED_SRAM_TO_PUF_BIT_INFFICIENCY`, defines the expected amount of SRAM bits to acquire one PUF output bit. This significant reduction results from the unstable bit exclusion by the memory mask and the debiasing done by the VNC.

Out of these, the first two are the main culprits behind the library's performance and memory requirements - both flash and SRAM for PUF. However, it also directly affects whether the secret will be able to be extracted with the PUF's help. Thus there needs to be some sort of a balance since reducing the process of unstable bit reduction would mean that the ECC would need to be used in a higher order. Also, using a longer ECC would mean larger secret extraction times. Furthermore, the reverse is true as well. Suppose most of the enrolment time is invested into the search of the stable bits, and it was compensated by a smaller required SRAM region due to the cutback to the order of the ECC. In that case, it might still mean a failure if some localised errors in one code word occur. Therefore, both these parameters need to be fine-tuned together.

As per memory requirements on the SRAM themself, they are defined as

$$SRAMReq = SecretSize * ECCBits * coeff$$

Where the SecretSize is the size of the stored secret in bytes, ECCBits is the number of bits the code word needs to encode one bit, and the *coeff* is the `EXPECTED_SRAM_TO_PUF_BIT_INFFICIENCY`, directly affected by how many `STB_BITS_SRAM_RESETS` have occurred. It is defined as a parameter due to the SRAM unpredictability and might need the highest amount of fine-tuning if one wants to avoid the state when the mask provides significantly more bits than the secret actually needs to be encoded; therefore, unnecessarily wasting processing time.

The other issue at hand is the time complexity. There are numerous sources of drastic time complexity hikes; the most notable of them might be

- the Deep sleep cycles
- SPIFFS initiation
- flash read and write operations
- malloc overriding, where the allocations occur on SPIRAM on flash
- long binary blob manipulation

Some of these have larger time complexity than constant, mainly all the binary blob manipulations. However, they are limited by the size of the SRAM itself, or more specifically, by the usable region selected. So, in the end, the highest time cost incurred is by the Deep sleep reset cycles, which take something in the realms of tens of seconds. Moreover, as the experimental data suggests, further visualised in the figure 4.8, the resets really are the main offender. The used data is stored in *attachments/measured_data/execution_times.log*

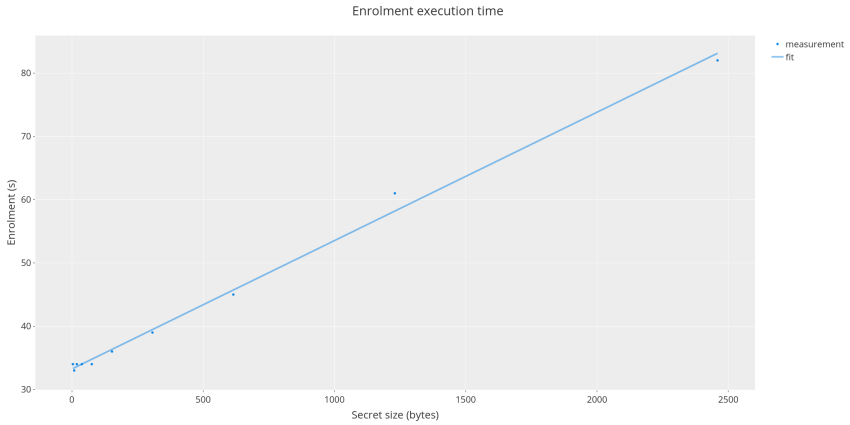

Figure 4.8: Enrolment time consumption

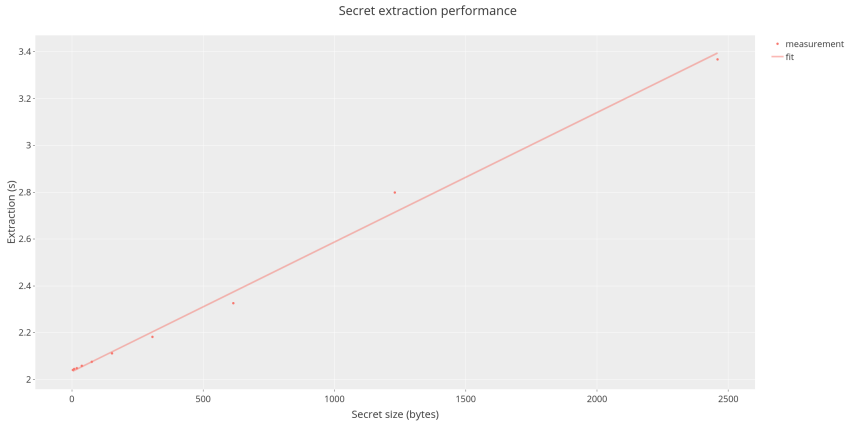

Figure 4.9: Extraction time consumption

As per the graph, while gradually doubling the secret length, the perceived time consumed during enrolment rose pretty favourably. Mainly while considering the starting point where the secret length was the length 4 B, the enrolment was done after 34 seconds.

Without relying on Deep sleep restarts, the secret extraction time, visualised by the figure 4.9, seems a lot more reasonable for normal usage as an HSM replacement. While the code itself is of a proof-of-concept nature, it surely features numerous bottlenecks where the time could be reduced further.

However, all these measurements had been done on an unoptimised code, without various tweaks that might be possible, such as passing -O2 to the compiler for speed optimisation, selecting Quad mode for flash operating mode and increasing the flash operating frequency to 80 MHz, as per the [6] *(/api-guides/performance/speed.html)* ESP32 documentation. This has some quite tangible results as seen in:

Table 4.2: SRAM PUF library performance comparison on ESP32

Optimization type	Enroll time (s)	Extract time (s)	Secret length (B)	SRAM used (B)	Total Resets
no optimization	82	3.36	2459	82136	13
-O2	77	2.84	2459	82136	13
-O2 + QIO flashmode	76	2.80	2459	82136	13
-O2 + QIO flashmode + 80 MHz	76	2.78	2459	82136	13

Thus if one has the choice, these optimisations should be applied since they have quite an adverse effect on both the enrolment and extraction time. However, these optimisations do induce some problems in some circumstances since the -O2 directive for the compiler performs the optimisations with the end goal being the execution speed; thus, sometimes, it might result in a larger binary, which, depending on the type of deployment, might be unacceptable. Also, the other two settings are closely tied to the hardware, and not always it might be possible to enable them.

And lastly, the **quality testing**. Again, the whole testing batch had been executed on an **ESP32 Wrover B board** rev.1, and only the tests aimed at determining the quality of a singular PUF have been deployed, the reliability and the bit uniformity tests.

First off, the concern of whether the library processing the SRAM PUF output is efficient at all needs to be looked at. Hence, the following testing was conducted with the help of the implementation of the library-based metrics.

- Starting from a “cold” status, where the device was unpowered for at least an hour before testing.
- Testing during stable conditions.
- Comparison of an unprocessed PUF response with library processed PUF response.
- Executed two days in a row, during similar times, within room temperatures and with minimal environmental noises and influences.
- The whole testing took 1 hour and 40 minutes, with a measurement made every circa 7 seconds.

After conducting the said test, with a visualisations 4.10 and 4.11, it can be safely concluded that the library does have a nonnegligible effect on the SRAM states, improving the quality of the SRAM PUF responses, at least reliability-wise. In the case of bit uniformity, the case is less clear; the difference here is negligible. Thus, regarding the PUF, it can probably be said that it by itself has a good inherent bit uniformity, at least the specific one used. However, the reliability of the unprocessed SRAM PUF is not acceptable, and it really needs a response processing library in order to use the responses

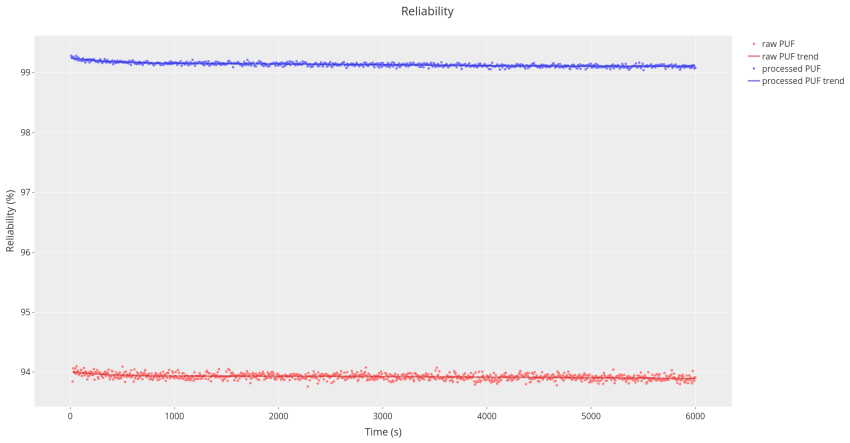

Figure 4.10: Reliability comparison

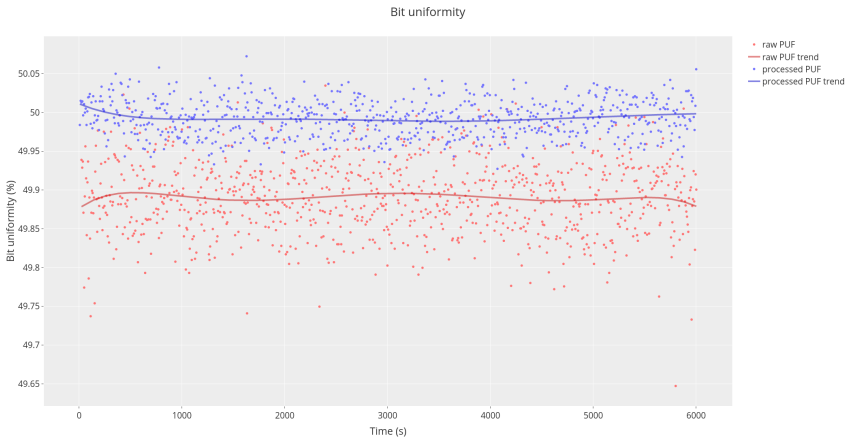

Figure 4.11: Bit uniformity comparison

further on. The measured data is stored in *attachments/measured_data/pufSym_comp.log* and *attachments/measured_data/pufSym_raw_comp.log*.

After confirming the positive effects of the library on the PUF, the next interesting question would be to determine how the library process the PUF responses during extended periods of time. Thus, the following testing was conducted in the same conditions as the first one but with longer runtime. The test had a runtime of 3 hours and 15 minutes. However, the test was conducted multiple times, and each time it was done, it resulted in a similar pattern, as per the visualisations 4.12 and 4.13. The measured data is stored in *attachments/measured_data/pufSym_long.log*

Being more specific regarding the reliability data, it can be observed that the reliability during the first initial measurements was of the highest values coming close to the 99.3% mark. Only afterwards did the average values gradually fall to around the 99.15% mark. This can be attributed that the enrolment being executed right after the device was started

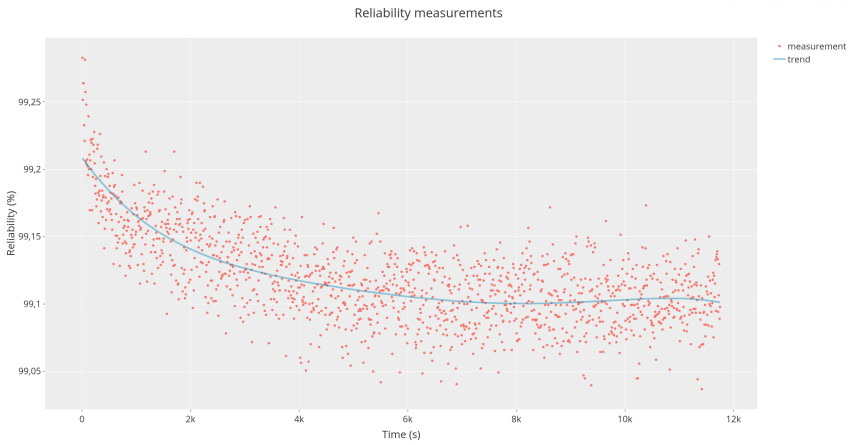

Figure 4.12: Long reliability testing

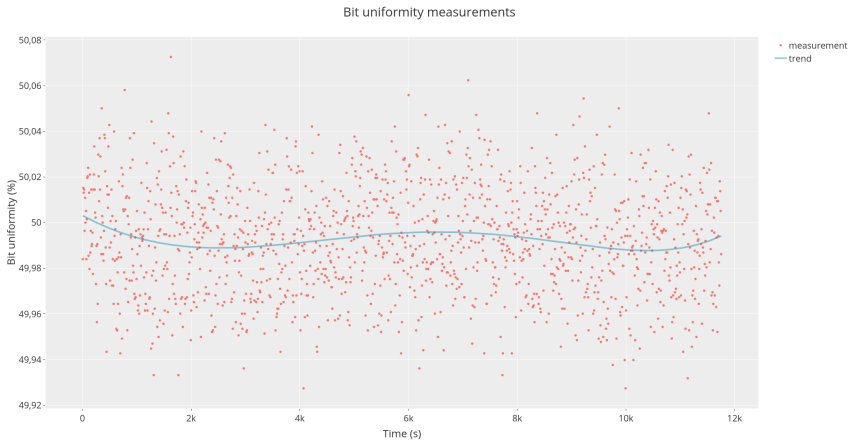

Figure 4.13: Long bit uniformity testing

up, thus not giving it the time to start operating in a normal operational state. However, the difference is quite small and, in the case of it causing secret extraction failures, and simply the ECC could be set up to a higher error correcting order.

4.2.3 PUF-based OTA update process

In general, an OTA update process needs to be robust but lightweight. The latter is even more emphasised due to it not being a daily driver of most real-world applications, and taking up a lot of already stretched system resources would only dissuade its secure use or use at all.

The bare minimum an OTA update process should provide in terms of security is the **authentification** of the server of the updated firmware to the client (in this case, the IoT device) and a **secure retrieval** of the update binary file by an authorised client.

Authentification of the OTA server can be achieved through public key cryptography, where the recipient of the OTA update has the public key. If the server can properly respond to a request from a client encrypted with the help of the public key, it can be assumed the server really is the trusted party since the knowledge of the private key to decrypt the message is required. Furthermore, the public key infrastructure can be further used to agree upon a common symmetric key with which the communication from then on would be encrypted, thus speeding up the OTA transfer and further guaranteeing that everything received from the server truly is from a trusted OTA server authority.

However, at this point, a simple sending of the raw binary would not be very wise since anyone could obtain the public key used to establish the connection - the IoT device is still unauthenticated at this point. So to achieve a **secure retrieval** of the updated firmware by an authorised IoT client, it could be done analogously as the server authentification, by the device proving the knowledge of a secret only a trusted party would know. Thus it might be quite convenient to simply encrypt the target binary with another public key the server knows (and now that it has been authenticated, it is now known that even though others could have the key that this usage can be trusted by the device). The device is implicitly authorised when it displays the knowledge of the secret, in this case, the private key, by simply decrypting the binary. Otherwise, if the device were not the trusted party, they would only receive some seemingly nonsensical string of bits, not really learning anything of substance. Furthermore, this is the point where the developed **SRAM PUF library** comes into place, as it can serve as an alternative to an HSM for storing the secret used to decrypt the firmware.

4.2.3.1 OTA process on ESP32

After having tested the PUF library, the final step of this work is to integrate this PUF-based secret storage and retrieval library into a functioning OTA update showcase, based on the already described OTA update proposal.

The ESP-IDF toolkit does provide the basic tools necessary for the operation of an OTA process [56]. The implementing modules of each step are, with the figure 4.14 showcasing it further:

- **connecting to an AP** - wireless networking implementation
- **establishing communication** with an HTTP server - TCP/IP stack and an HTTP client
- **downloading the new firmware**
- **applying the received OTA update** - main OTA logic - mainly flash manipulation and SSBL logic

While being able to perform the basic unsecured OTA process, the ESP-IDF also supports implementing the described security features within their tools. The usage of a certificate for server authentication is practically required to be used since the referential

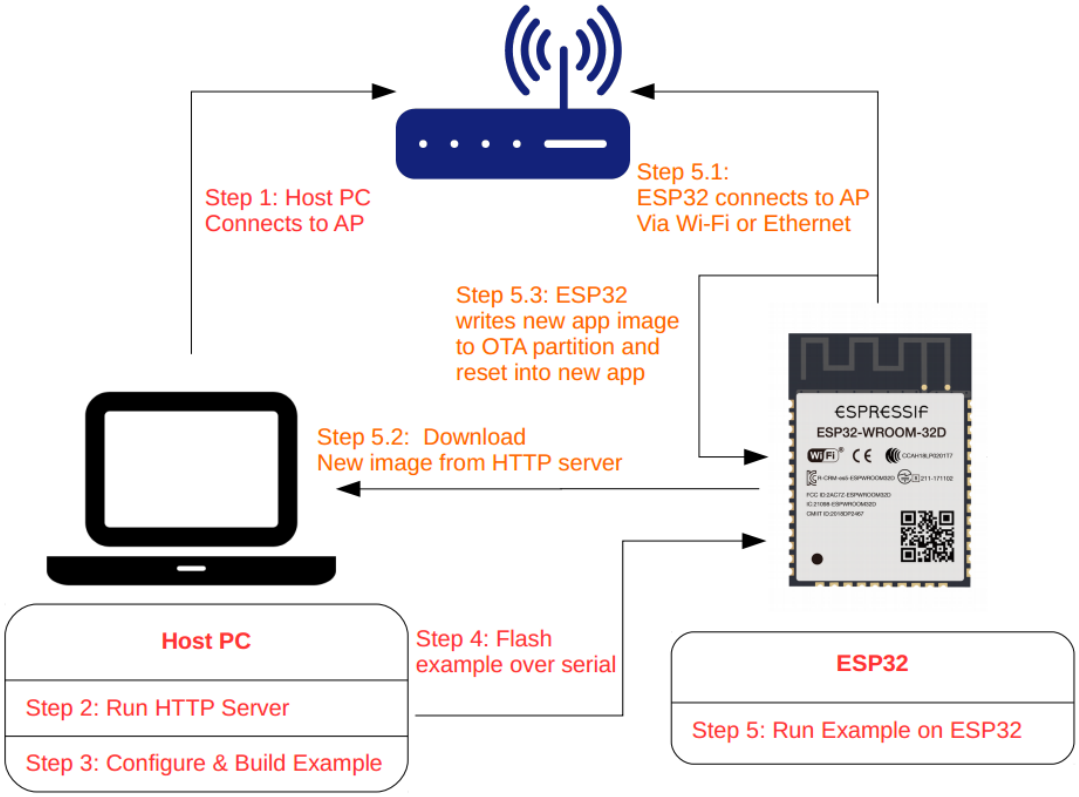

Figure 4.14: Basic OTA process [6]

usage of the OTA process does implement them. Furthermore, removing them would require a conscious effort. As for the matter of the firmware encryption, an ESP Encrypted Image Abstraction Layer, or EIAL, is available [57].

The EIAL makes the use of encryption in a twofold manner.

- implements an asymmetric key that should be stored safely in an HSM module
- a symmetric key that is stored encrypted in the image header

This schema is a case of hybrid cryptography [58], already described during the proposal, is a type of encryption where the practicality of an asymmetric key and the speed of a symmetric key are both retained, which is critical for a platform like an ESP32. Therefore, the ESP EIAL works with a 3072 RSA asymmetric key and an AES-GCM key. Also, to save as many resources as possible only, the AES key is asymmetrically encrypted, leaving the auxiliary data necessary for the symmetric encryption in plaintext [57], as per the following image 4.15. This might cause some information leakage since the initialisation vector is accessible. However, since the whole EIAL image should be

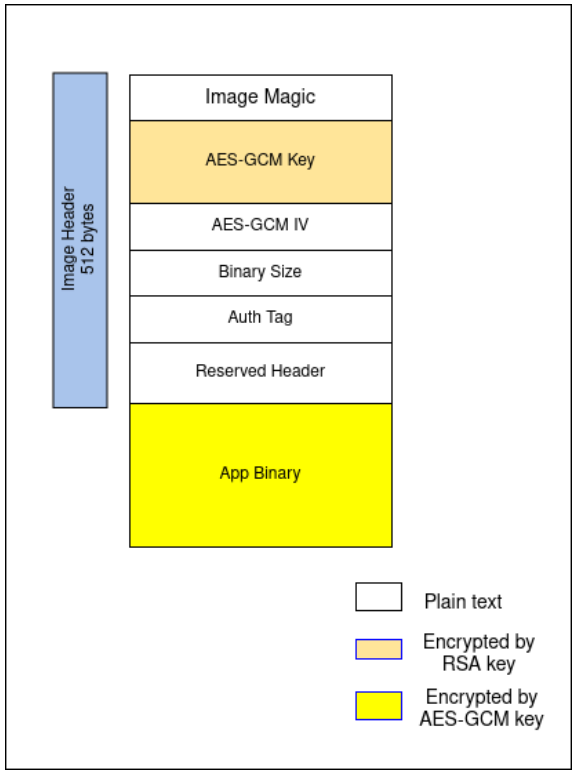

Figure 4.15: ESP EIAL firmware format [58]

transported over a secured channel, it should not matter much. A showcase of the EIAL image format is displayed in figure 4.15.

Also, to maintain security, the project needs to randomise the initialisation vector of the AES so that no two binaries will be of the same encrypted value, even if they are the same in plaintext [58].

4.2.3.2 SRAM PUF library integration into an OTA project

In order to make use of the PUF library, the project needs to somehow employ it to store an essential secret that will be getting used during the OTA update process. In this case, that would either be a private certificate used for the device's authentication or a private key used for firmware decryption. In the ESP32's case, the latter is the obvious use case since it has a whole abstraction layer over an encrypted image format built.

The proof of concept project will employ the storage of the key in the following manner - along with the initial local firmware deployment, the two main security elements are, likewise, placed into an unencrypted memory. In the case of the server's public key, it does not matter; it is public information.

On the other hand, the case of the private RSA key used for firmware decryption is different. After deployment, it needs to be stored safely as quickly as possible, so the PUF library enrolment should begin right after the SSBL grants control to the main

program after the initial start. Here, the already described secret enrolment takes place. However, in contrast to the regular testing deployment on the testing ESP32 project, all the debugging and performance metric structures must strictly be erased. So due to wear levelling already explained before, both the SPIFFS filesystem and all the unused flash blocks should be safely erased. This is not done strictly in a correct manner in the actual project since it would be resource and time-demanding for a showcase, but it needs to be implemented in a release.

After successful enrolment, the key extraction takes place, which is provided to the OTA process on the client.

Note - as the project is still not a release-worthy implementation but a proof of concept, the implementation does conduct an enrolled key check against the plaintext key provided by the deployment in order to determine the success or the number of bits that had been falsely decoded. Also, it does not provide a probable use case presentation - after the initial enrolment, the OTA update would typically occur quite sometime later, which was not practical to be done in this project; hence it all is streamlined.

After obtaining the private RSA key through the PUF library, the OTA process itself starts [56]:

1. The project uses the provided WiFi credentials from the *menuconfig* tool to connect to an AP. In the case of an unstable connection, the amount of retries and reconnects is not specified.

2. In the case of deployment on a local server (laptop with a hotspot type AP) and device, the server's address is entered into the *menuconfig* tool as the IP address of the AP.

3. After establishing a connection, the device tries to start a TLS session with authentication of the server through the public key in the flash. If the certificates match, the HTTP over TLS connection is established.

4. Sends an HTTP request for the download to begin, which is specified in the *menuconfig* tool, specifically the filename of the encrypted binary

5. After receiving the firmware, it verifies the image magic and ensures that the binary's length is valid, along with other checks.

6. Uses the recovered asymmetric key through the PUF library extracted from an SRAM PUF response done before.

7. In the case of successful decryption of the symmetric key, it should be possible to decrypt the new firmware efficiently.

8. If the decrypted firmware is valid and passes the necessary checks, it is stored in the OTA partition.

Also, the ESP-IDF implementation of the OTA process allows one to use a fallback to the factory firmware, which, if enabled, allows a continuous operation. It even has

multiple fallback firmware versions [6]*(go to /api-reference/system/ota.html and /api-guides/partition-tables.html)*, all stored in their own partitions. That is the reason for having both the *factory* partition alongside the other *OTA* partitions, of whose there can be multiple instances. Having such partitions might be costly in terms of flash capacity; however, numerous ESP32s, like the Wrover B used in this work, provide 16 MB of flash, which is more than enough in most cases.

The **PUF library**, while being quite reliable according to quality testing done, needs to be configured for the most robust operation possible, with as much SRAM memory available to it, since a single bit wrongly extracted from the helper function and the PUF response means that the decryption of the symmetric key is impossible and the whole process, all the way from enrolment, needs to be restarted.

4.3 Integration conclusion

To summarise - after implementing the SRAM PUF library based on the highlighted principles, it was, if considering the reliability tests, able to work on real hardware, an ESP32 Wrover B, while also improving the inherent SRAM PUF responses. Therefore proving the SRAM of an ESP32 and its tools are viable enough to implement the storage of a secret and use in various CTs. Afterwards, the tuned library was integrated into a proof of concept OTA update project, where, as an example, the private key was securely stored with the help of the PUF library and, afterwards, used to decrypt the retrieved firmware update from an OTA server. As a finishing touch, the update is applied. After resetting, the device can demonstrate that it has, indeed, been able to securely acquire the updated firmware while replacing the HSM storage with a PUF SRAM.

Conclusion

The goal of the work was to propose an OTA update process suitable for constrained IoT devices with the possibility of using a PUF to provide security, closing the work with a proof of concept project.

The work has mainly dealt with listing the components of an OTA process, where its bottlenecks lie, which aspects need to be especially secure and how to use the process in a simple, device-to-server architecture. After analyzing the threat vectors and surfaces, the needed CT techniques to ensure the protection is well in place, even in the face of adversaries, were listed. To finalize this step, the whole set of information was brought together, and a proposal for a simple and secure OTA process was brought up.

After identifying the possible CTs which could use the SRAM PUF, cursory research was conducted into the PUF types and, most importantly, the possible applications on a constrained IoT device.

In the face of scarce competition that would have been viable to implement relatively simply, the SRAM PUF had been chosen to serve as the basis for the library. This library was designed to use the PUF properties to store a secret in such a way that the only data that was persisted into the flash would not have been of any use to any attacker. Unless, of course, he would have somehow got his hands on a PUF response as well, which would have been quite impossible since the uninitialized values of the SRAM last only so long. Afterwards, by using this data, the device can extract the key with the help of the SRAM, thus making the SRAM PUF serve as a limited HSM.

Lastly, the said library was integrated into two ESP32 projects. The first one had served as a testing ground for assessing whether the SRAM PUF of the said platform would be usable for further use. After it became apparent that it was valid, the library was integrated into another project, this time a simplified OTA update process using some of the described principles. After making sure that the device was able to decipher the encrypted firmware provided by the OTA server after extracting the private key, with the help of the SRAM PUF library successfully, the project was declared a success.

References

1. *IoT market size worldwide 2017-2025* [online] [visited on 2022-06-19]. Available from: https://www.statista.com/statistics/976313/global-iot-market-size/.
2. GOOGLE, LLC. *OTA Updates | Android Open Source Project* [online] [visited on 2022-06-19]. Available from: https://source.android.com/devices/tech/ota.
3. CISCO. *Cisco Annual Internet Report - Cisco Annual Internet Report (2018–2023) White Paper* [online] [visited on 2022-06-19]. Available from: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.
4. VERY. *Manual vs. OTA Firmware Updates for IoT* [online] [visited on 2022-06-19]. Available from: https://www.verypossible.com/insights/manual-vs-ota-firmware-updates-for-iot.
5. TICLO, Isabel. *3 IaaS Cloud Computing Trends to Watch* [online] [visited on 2022-06-19]. Available from: https://www.insight.com/en_US/content-and-resources/2017/02132017-3-iaas-cloud-computing-trends-to-watch.html.
6. ESPRESSIF SYSTEMS (Shanghai) Co., Ltd. *ESP-IDF Programming Guide latest documentation* [online] [visited on 2022-06-19]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32.
7. BROWN, Benjamin Bucklin. *Over-the-Air (OTA) Updates in Embedded Microcontroller Applications: Design Trade-Offs and Lessons Learned | Analog Devices* [online] [visited on 2022-06-19]. Available from: https://www.analog.com/en/analog-dialogue/articles/over-the-air-ota-updates-in-embedded-microcontroller-applications.html#.
8. CACAMERA, Daniele. *12ft | The interrupt vector table - Embedded Systems Architecture [Book]* [online] [visited on 2022-06-20]. Available from: https://www.oreilly.com/library/view/embedded-systems-architecture/9781788832502/2cfb60b5-d366-4fa0-a750-b942e022665d.xhtml.

9. ESP_ANGUS, Espressif employee. *[closed] updating the bootloader via OTA – is it possible? - ESP32 Forum* [online] [visited on 2022-06-19]. Available from: https://www.esp32.com/viewtopic.php?t=6080.

10. STAFF, Embedded. *Speeding over-the-air latency for IoT applications with compression* [online]. 2017 [visited on 2022-06-19]. Available from: https://www.embedded.com/speeding-over-the-air-latency-for-iot-applications-with-compression/.

11. SUZUKI, Naoki; HAYASHI, Toshiki; KIYOHARA, Ryozo. Data Compression for Software Updating of ECUs. In: *2019 IEEE 23rd International Symposium on Consumer Technologies (ISCT)*. 2019, pp. 304–307. Available from DOI: 10.1109/ISCE.2019.8901008. ISSN: 2159-1423.

12. TEMBOO. *How to Approach OTA Updates for IoT* [online]. 2019 [visited on 2022-06-19]. Available from: https://medium.com/@temboo/how-to-approach-ota-updates-for-iot-d088c217b31c.

13. RADOVICI, Alexandru; CULIC, Ioana; ROSNER, Daniel; OPREA, Flavia. A Model for the Remote Deployment, Update, and Safe Recovery for Commercial Sensor-Based IoT Systems. *Sensors*. 2020, vol. 20, pp. 4393. Available from DOI: 10.3390/s20164393.

14. BOEHM, Ellen. *What is Secure Boot? It's Where IoT Security Starts* [online] [visited on 2022-06-19]. Available from: https://www.keyfactor.com/blog/what-is-secure-boot-its-where-iot-security-starts/.

15. MOSELEY, Drew. *Security considerations for OTA software updates for IoT gateway devices* [online]. 2020 [visited on 2022-06-19]. Available from: https://stackoverflow.blog/2020/12/14/security-considerations-for-ota-software-updates-for-iot-gateway-devices/.

16. SCHMIDT, Silvie. *Secure Firmware Updates in the IoT* [online] [visited on 2022-06-19]. Available from: https://sec4dev.io/assets/uploads/slides/Secure-Firmware-Updates-OTA-in-the-IoT_2.pdf.

17. SIVASANKARI, N.; KAMALAKKANNAN, S. Detection and prevention of man-in-the-middle attack in iot network using regression modeling. *Advances in Engineering Software* [online]. 2022, vol. 169, pp. 103126 [visited on 2022-06-19]. ISSN 0965-9978. Available from DOI: 10.1016/j.advengsoft.2022.103126.

18. CISA. *Understanding Denial-of-Service Attacks | CISA* [online] [visited on 2022-06-19]. Available from: https://www.cisa.gov/uscert/ncas/tips/ST04-015.

19. JHA, Asmita. *Log4j-Vulnerability* [online] [visited on 2022-06-19]. Available from: https://payatu.com/blog/asmita-jha/side-channel-attack-basics.

20. BARKER, Elaine; MOUHA, Nicky. *Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher* [online]. 2017 [visited on 2022-06-19]. Available from DOI: 10.6028/NIST.SP.800-67r2. Technical report. National Institute of Standards and Technology.

21. BORGINI, Julia. *How to use IoT authentication and authorization for security* [online] [visited on 2022-06-19]. Available from: https://www.techtarget.com/iotagenda/feature/How-to-use-IoT-authentication-and-authorization-for-security.

22. CLOUDFLARE. *What is Transport Layer Security? | TLS protocol* [online] [visited on 2022-06-19]. Available from: https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/.

23. ESP_SPRITE, Espressif employee. *How is the flash encryption key stored? - ESP32 Forum* [online] [visited on 2022-06-19]. Available from: https://www.esp32.com/viewtopic.php?t=16110.

24. RICHARDS, Kathleen. *What is Cryptography? Definition from SearchSecurity* [online] [visited on 2022-06-19]. Available from: https://www.techtarget.com/searchsecurity/definition/cryptography.

25. *3 OTA architectures for IoT devices* [online] [visited on 2022-06-19]. Available from: https://www.techtarget.com/iotagenda/feature/3-OTA-architectures-for-IoT-devices.

26. PADMAVATHI, B; KUMARI, S Ranjitha. A Survey on Performance Analysis of DES; AES and RSA Algorithm along with LSB Substitution Technique. 2013, vol. 2, no. 4, pp. 5.

27. *Hybrid encryption | Tink* [online] [visited on 2022-06-19]. Available from: https://developers.google.com/tink/hybrid.

28. BARKER, Elaine. *Recommendation for Key Management Part 1: General* [online]. 2016 [visited on 2022-06-19]. Available from DOI: 10.6028/NIST.SP.800-57pt1r4. Technical report. National Institute of Standards and Technology.

29. TECHTARGET. *What is PKI (public key infrastructure)* [online] [visited on 2022-06-20]. Available from: https://www.techtarget.com/searchsecurity/definition/PKI.

30. BABAEI, Armin; SCHIELE, Gregor. Physical Unclonable Functions in the Internet of Things: State of the Art and Open Challenges. *Sensors* [online]. 2019, vol. 19, no. 14, pp. 3208 [visited on 2022-06-19]. ISSN 1424-8220. Available from DOI: 10.3390/s19143208.

31. JOSHI, Shital; MOHANTY, Saraju P.; KOUGIANOS, Elias. Everything You Wanted to Know About PUFs. *IEEE Potentials* [online]. 2017, vol. 36, no. 6, pp. 38–46 [visited on 2022-06-18]. ISSN 0278-6648. Available from DOI: 10.1109/MPOT.2015.2490261.

32. GORDON, Holden; EDMONDS, Jack; GHANDALI, Soroor; YAN, Wei; KARIMIAN, Nima; TEHRANIPOOR, Fatemeh. Flash-Based Security Primitives: Evolution, Challenges and Future Directions. *Cryptography* [online]. 2021, vol. 5, no. 1, pp. 7 [visited on 2022-06-19]. ISSN 2410-387X. Available from DOI: 10.3390/cryptography5010007.

33. DAHAD, Nitin. *Basics of SRAM PUF and how to deploy it for IoT security* [online]. 2021 [visited on 2022-06-19]. Available from: https://www.embedded.com/basics-of-sram-puf-and-how-to-deploy-it-for-iot-security/.

34. VIJAYAKUMAR, Arunkumar; PATIL, Vinay; KUNDU, Sandip. On Improving Reliability of SRAM-Based Physically Unclonable Functions. *Journal of Low Power Electronics and Applications* [online]. 2017, vol. 7, no. 1, pp. 2 [visited on 2022-06-19]. ISSN 2079-9268. Available from DOI: 10.3390/jlpea7010002.

35. WANG, Wendong; GUIN, Ujjwal; SINGH, Adit. Aging-Resilient SRAM-based True Random Number Generator for Lightweight Devices. *Journal of Electronic Testing* [online]. 2020, vol. 36, no. 3, pp. 301–311 [visited on 2022-06-21]. ISSN 0923-8174, 1573-0727. ISSN 0923-8174, 1573-0727. Available from DOI: 10.1007/s10836-020-05881-6.

36. DODIS, Yevgeniy; REYZIN, Leonid; SMITH, Adam. Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. In: CACHIN, Christian; CAMENISCH, Jan L. (eds.). *Advances in Cryptology - EUROCRYPT 2004*. Berlin, Heidelberg: Springer, 2004, pp. 523–540. Lecture Notes in Computer Science. ISBN 978-3-540-24676-3. Available from DOI: 10.1007/978-3-540-24676-3_31.

37. ARJONA, Rosario; PRADA-DELGADO, Miguel; ARCENEGUI, Javier; BATURONE, I. Trusted Cameras on Mobile Devices Based on SRAM Physically Unclonable Functions. *Sensors*. 2018, vol. 18, pp. 3352. Available from DOI: 10.3390/s18103352.

38. STMICROELECTRONICS. *STM32 Reference manual* [online] [visited on 2022-06-19]. Available from: https://www.st.com/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf#%5B%7B%22num%22%3A153%2C%22gen%22%3A0%7D%2C%7B%22name%22%3A%22XYZ%22%7D%2C124%2C762%2Cnull%5D.

39. *ciphergoth.org: Generating random binary data from Geiger counters* [online] [visited on 2022-06-20]. Available from: http://www.ciphergoth.org/crypto/unbiasing/.

40. MICRON TECHNOLOGY, Inc. TN-12-30: NOR Flash Cycling Endurance and Data Retention. 2013, pp. 12. Available also from: https://media-www.micron.com/-/media/client/global/documents/products/technical-note/nor-flash/tn1230_nor_flash_cycling_endurance_data_retention.pdf?rev=e499d40bf03a4e18842e05890c18ee59.

41. ESP_SPRITE, Espressif employee. *Simple Pre-Purchase Questions (RTC/memory/flash/NVS) - ESP32 Forum* [online] [visited on 2022-06-19]. Available from: https://www.esp32.com/viewtopic.php?p=29525#p29525.

42. LIMITED, Cactus Technologies. *Wear Leveling - Static, Dynamic and Global* [online] [visited on 2022-06-19]. Available from: https://www.cactus-tech.com/wp-content/uploads/2019/03/Wear-Leveling-Static-Dynamic-Global.pdf.

43. LEE, Jeffrey. *The 6 Best IoT Hardware Platforms (2021 Update)* [online]. 2021 [visited on 2022-06-20]. Available from: https://blog.particle.io/iot-hardware-comparison-guide/.

44. AUTHORS, The TinyGo. *ESP32 - mini32* [online] [visited on 2022-06-19]. Available from: https://tinygo.org/docs/reference/microcontrollers/esp32-mini32/. Section: docs.

45. LTD, Pur3. *Espruino on ESP32* [online] [visited on 2022-06-19]. Available from: https://www.espruino.com/ESP32.

46. *Rust ESP32 Example* [online]. Espressif, 2022 [visited on 2022-06-19]. Available from: https://github.com/espressif/rust-esp32-example. original-date: 2021-06-08T15:39:17Z.

47. TRANTER, Jeff. *MicroPython Optimizes Python for Microcontrollers* [online] [visited on 2022-06-19]. Available from: https://www.ics.com/blog/micropython-optimizes-python-microcontrollers.

48. DAMIEN P. GEORGE, Paul Sokolovsky; CONTRIBUTORS. *General information about the ESP32 port — MicroPython 1.19.1 documentation* [online] [visited on 2022-06-19]. Available from: https://docs.micropython.org/en/latest/esp32/general.html#technical-specifications-and-soc-datasheets.

49. BOLTON, David. *Exploring the Benefits of MicroPython* [online]. 2017 [visited on 2022-06-19]. Available from: https://insights.dice.com/2017/08/03/exploring-benefits-micropython/.

50. KOLBAN, Neil. *Menuconfig options for ESP32 Arduino - ESP32 Forum* [online] [visited on 2022-06-19]. Available from: https://esp32.com/viewtopic.php?t=5225.

51. FOUNDATION, IOTA. *IOTA ESP32 Wallet* [online] [visited on 2022-06-19]. Available from: https://blog.iota.org/iota-esp32-wallet-1b12b45d8a5/.

52. ENGINEERS, Last Minute. *Insight Into ESP32 Sleep Modes & Their Power Consumption* [online]. 2018 [visited on 2022-06-19]. Available from: https://lastminuteengineers.com/esp32-sleep-modes-power-consumption/.

53. TARMO. *Answer to "in ESP32 / ESP-IDF - when to use EEPROM vs NVS vs SPIFFS?"* [online]. 2022 [visited on 2022-06-19]. Available from: https://stackoverflow.com/a/70826781.

54. HYMEL, Shawn. *Introduction to RTOS - Solution to Part 4 (Memory Management)* [online] [visited on 2022-06-19]. Available from: https://www.digikey.es/en/maker/projects/6d4dfcaa1ff84f57a2098da8e6401d9c.

55. BARBARESCHI, Mario; BATTISTA, Ermanno; MAZZEO, Antonino; MAZZOCCA, Nicola. Testing 90 nm microcontroller SRAM PUF quality. In: *2015 10th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. 2015, pp. 1–6. Available from DOI: 10.1109/DTIS.2015.7127360.

56. *Espressif IoT Development Framework* [online]. Espressif Systems, 2022 [visited on 2022-06-21]. Available from: `https://github.com/espressif/esp-idf`. original-date: 2016-08-17T10:40:35Z.

57. ESPRESSIF. *Espressif IDF Extra Components* [online]. Espressif, 2022 [visited on 2022-06-19]. Available from: `https://github.com/espressif/idf-extra-components/blob/45d872e6a6e9954ad5a584a4881b139887eb5d91/esp_encrypted_img/README.md`. original-date: 2021-11-16T17:59:26Z.

58. GOOGLE, LLC. *Hybrid encryption | Tink* [online] [visited on 2022-06-19]. Available from: `https://developers.google.com/tink/hybrid`.

Appendix A

Acronyms

PUF Physical Unclonable Function

OTA Over-the-Air

ROM Read Only Memory

SRAM Static Random-Access Memory

IRAM Instruction Random-Access Memory

DRAM Data Random-Access Memory

SPIRAM Serial Peripheral Interface Random-Access Memory

CT Cryptographic technique

REPL Read Evaluate Print Loop

SDK Software development kit

CA Certification Authority

HSM Hardware Security Module

ACID Atomicity, Consistency, Isolation, Durability

PKI Public Key Infrastructure

NVS Non-Volatile Storage

EEPROM Electrically Erasable Programmable Read-Only Memory

IVT Interrupt Vector Table

MiTM Man in the middle

SPA Simple Power Analysis

DPA Differential Power Analysis

CPA Correlation Power Analysis

EM Electromagnetic

CRP Challenge Response Pair

SAC Strict Avalanche Condition

ML Machine Learning

ECC Error Correcting Code

NIST National Institute of Standards and Technology

AES Advanced Encryption Standard

GCM Galois counter mode

DES Data Encryption Standard

RSA Rivest-Shamir-Adleman

IV Initialisation Vector

EIAL Encrypted Image Abstraction Layer

VNC Von Neumann Corrector

PRNG Pseudorandom Number Generator

TRNG True Random Number Generator

EEV Error Correcting Code

BLE Bluetooth Low Energy

AP Access point

STA Station

RTC Real Time Clock

OS Operating system

HW hardware

PC Personnal Computer

CPU Central Processing Unit

ULP Ultra Low Power

IoT Internet of Things

API Application programming interface

TCP Transmission Control Protocol

HTTP Hyper Text Transfer Protocol

HTTPS HTTP Secure

IP Internet Protocol

TLS Transport Layer Security

DTLS Datagram Transport Layer Security

RTOS Real-Time operating system

FSBL First Stage Bootloader

SSBL Second Stage Bootloader

GPIO General Purpose Input Output

SPI Serial Peripheral Interface

FATFS File Allocation Table File System

SPIFFS Serial Peripheral Interface Flash File System

Appendix B

Library usage

B.1 PufSymtesting project

The main point of this project was to analyze obtained SRAM PUF responses from an ESP32, which can be done practically instantly if compared to the ESP itself. Thus the main development was contained here, in the folder `src/pufSymtesting`.

B.1.1 Prerequisites of the pufSym library

In order to use the library in this form, there are several extra steps that need to be followed.

1. Place the secret, currently named as **ca_cert.pem**, into the project's root (just an example, in reality, it would not make much sense to treat a public certificate as a secret).
2. Provide the snapshots of the actual examined SRAM into the *memImg* folder, currently named **memImgX.img**, where X is an unsigned number.
3. Double check the macros in **pufSym.h** and make sure that they represent what has been provided in previous steps, mainly the
   - the key length `KEY_BYTE_SIZE` is the same or lower than the **ca_cert.pem**
   - the expected length and offset of the snapshots are within the limits of the provided memImg files
   - the amount of SRAM resets is **lower** than the number of actual snapshots provided. Also, keep in mind that enrollment requires four additional snapshots.
4. Provide macro **NOT_ESP32** to the compiler to exclude ESP32 exclusive functions; otherwise, it cannot be compiled normally.

The project already contains 42 snapshots and several more of varying sizes in the *temp_bins_analysis* folder.

B.1.2 Use of pufSym library

In order to simulate a deployment on an ESP32, use:

```
cp ca_cert.pem memImg/ca_cert.pem && g++ -D "NOT_ESP32" -Wall -pedantic
-O -ggdb3 pufSym.cpp pufSym_testing.cpp  &&  ./a.out
```

Depending on the provided macros, either **DEBUG_MODE** or **DEBUG**, there will be more output and processed files and data that normally is not left behind. Accessible in *memImg* folder.

In order to clean the project, use `./clean_project.sh`.

B.2 PufSym library testing project

This ESP32 project serves as a testing base for the pufSym library on actual hardware in the folder `src/pufSym_device_test`.

B.2.1 Requirements

1. Have the esp-idf toolchain installed and correctly configured. The version used during the development was the master branch of the GitHub repository, more specifically, this commit

2. Have the various supporting tools installed and used during the deployment, some of which are required on the official website, at [6] *(go to /get-started/linux-macos-setup.html)*, some of which are used in this project, such as **mkspiffs** tool. Be sure to use the esp-idf binary variant, the one used here was *mkspiffs-0.2.3-esp-idf-linux64.tar.gz*; otherwise, it might not work.

3. Provide the directory **./spiffs_image** with the secret, currently named *ca_cert.pem*.

If the build folder was deleted, menuconfig needs to be run and reenable flash usage as a SPIRAM and possibly other settings. Accessing menuconfig: `idf.py -p /dev/ttyUSB0 menuconfig`.

The relevant settings are located at:

- menuconfig -> comp config -> HW settings -> support for external SPI RAM enable;
- menuconfig -> comp config -> HW settings -> SPI RAM config -> SPI RAM access method

B.2.2 Deployment

In order to successfully compile and deploy, use the following command `idf.py -p /dev/ttyUSB0 build flash monitor`.

The command compiles the application, flashes it into the memory and runs a serial console which allows monitoring and interaction with the application.

If, for some reason, the flash has to be erased manually, do it with `idf.py -p /dev/ttyUSB0 erase_flash`.

B.2.3 Usage

After flashing the application and using the idf monitor function, the application will be waiting for input, where the user can choose whether to start.

0. (key extraction)
1. (PUF reliability test)
2. (SRAM bit stability test)
3. (SRAM image collection)

After selecting the relevant option, the system will ask the number of tests/images that need to capture. Be careful not to go above the 2MB limit the SPIFFS currently has in place if using the image collection. It could be enlargened through menufonfig if the device supports more than 4 MB of SPIRAM. Also, if not resolved hardware-wise, the FSBL will print an initialization message on each reset. Otherwise, the outputs in the serial console are the same as in the pufSym standalone project in the *../pufSymtesting* folder.

B.2.4 Data collection

After running a successful testing session to get the data from the flash, one can run the **import_spiffs.sh**, if one has not changed the partition table, of course. Otherwise, the addresses and memory size must be updated. The command also unpacks the downloaded flash image into a folder.

B.3 Encrypted Binary OTA update with SRAM PUF usage

A demonstration of the use of the SRAM PUF library within an OTA process, constructed as a part of this thesis in the folder `src/ota_pufSym`.

B.3.1 Requirements

1. All the requirements of the preceding project are included in this work.
2. Strict requirement for the 5.0 version of the esp-idf.
3. Generated RSA key pair that will be used for server authentication needs to have the server address in the CN field during generation, as per the official example, located in *examples/system/ota* at the official repository [56]. If operated locally, it

needs to be the IP address of the AP behind which the server operates. The pair should be placed into the *ota_server* folder, and the public key should be placed into the *server_certs*.

4. The Generated private RSA key used to decrypt the firmware binary should be placed into the *rsa_key* folder.

5. Configured the correct wifi access details, the server IP and the encrypted file name in the menuconfig tool

B.3.2 Usage

1. Run the HTTPS server with the *run_https_server.sh* script from the *ota_server* folder while having AP or similar connectivity with the device.

2. Flash the firmware, the public and private key with the *idf.py -p /dev/ttyUSB0 build flash monitor* command, where */dev/ttyUSB0* is the serial port.

B.3.3 Monitoring

The enrollment phase should take around 1 to 2 minutes, during which the system will reboot multiple times. Afterwards, it will recover the key from SRAM, connect to the wifi, establish an HTTPS connection with the server, obtain the firmware, decipher it and apply the update. If the update was successful, the new firmware should be the Hello World application from the esp-idf example applications [56].

B.3.4 Glitch/bug notice

After building the project and replacing a certificate/private key, the change might not propagate, mainly observed in the current main branch as of 7.6.2022. The solution was to complete a clean with *idf.py fullclean* or even delete the build folder. Note - deleting the build folder results in a loss of menufoconfig settings, which must be redone.

Also, if the path of the current IDF project changes, it might be required to do fullclean regardless.

Appendix C

Contents of enclosed DVD

- `readme.md` ..... brief description of DVD contents
- `text` ..... thesis text
  - `thesis.pdf` ..... thesis text in PDF format
- `attachments` ..... folder containing attachments
  - `measured_data` ..... measured and collected data used in thesis
- `src`
  - `ota_pufSym` ..... sources of the SRAM PUF and OTA update showcase
  - `pufSym_device_test` ..... sources of the PufSym lib ESP32 integration project
  - `pufSymtesting` ..... sources of the PufSym development project