



**Budapest University of Technology and Economics**  
Faculty of Electrical Engineering and Informatics  
Department of Networked Systems and Services

# Secure boot and firmware update on a microcontroller-based embedded board

BACHELOR'S THESIS

*Author*

András Sándor Gedeon

*Advisors*

Dr. Levente Buttyán  
Dorottya Futóné Papp

December 10, 2020

# Contents

<b>Kivonat</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background Technology</b>	<b>3</b>
2.1 Secure boot . . . . .	3
2.2 Flash Encryption . . . . .	4
2.3 Remote Firmware Update . . . . .	5
<b>3 Design</b>	<b>6</b>
3.1 OTA self-update application . . . . .	6
3.1.1 Security considerations regarding the update server . . . . .	6
3.1.2 Versioning . . . . .	7
3.1.3 Summary of requirements . . . . .	8
<b>4 Implementation</b>	<b>10</b>
4.1 ESP32 . . . . .	10
4.1.1 Overview . . . . .	10
4.1.2 Security of ESP32 . . . . .	10
4.2 Development environment for ESP32 . . . . .	11
4.2.1 ESP-IDF . . . . .	11
4.2.2 Visual Studio Code . . . . .	13
4.3 Secure Boot . . . . .	13
4.3.1 Cryptography used . . . . .	15

4.3.2	Reflashable and One-time flash methods . . . . .	15
4.3.3	Setup for Reflashable method . . . . .	16
4.3.4	Verifying signature . . . . .	17
4.4	Flash Encryption . . . . .	18
4.4.1	Cryptography used . . . . .	18
4.4.2	Development and Release modes . . . . .	19
4.4.3	Setup in Development mode . . . . .	19
4.4.4	Content of the flash before and after . . . . .	21
4.5	OTA self-update application . . . . .	22
4.5.1	OTA with ESP32 . . . . .	22
4.5.2	Simple OTA update example application . . . . .	22
4.5.3	The update server . . . . .	23
4.5.4	Changing keys . . . . .	25
4.5.5	Sub-application . . . . .	25
4.5.6	Versioning . . . . .	25
4.5.7	Updating firmware OTA . . . . .	27
4.6	Testing OTA application . . . . .	28
4.6.1	Configuration . . . . .	28
4.6.2	During run . . . . .	29
4.6.3	The updated application . . . . .	29
4.6.4	Possible errors in OTA application . . . . .	32
<b>5</b>	<b>Security analysis</b>	<b>33</b>
5.1	Secure boot . . . . .	33
5.1.1	Basic functioning . . . . .	33
5.1.2	Security . . . . .	33
5.2	Flash encryption . . . . .	33
5.2.1	Basic functioning . . . . .	33
5.2.2	Security . . . . .	34
5.3	OTA self-update . . . . .	34
5.3.1	Basic functioning . . . . .	34

5.3.2 Security . . . . .	34
<b>6 Related Work</b>	<b>35</b>
<b>7 Conclusion</b>	<b>37</b>
<b>Acknowledgments</b>	<b>38</b>
<b>List of Figures</b>	<b>40</b>
<b>Bibliography</b>	<b>40</b>
<b>Appendix</b>	<b>43</b>

## HALLGATÓI NYILATKOZAT

Alulírott *Gedeon András Sándor*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2020. december 10.

---

*Gedeon András Sándor*  
hallgató

# Kivonat

Napjainkban egyre több és több elektronikus eszköz kapcsolódik az Internethez és egymáshoz, ezáltal létrehozva a Dolgok Internetét (angolul Internet of Things, vagy röviden IoT). Ugyan az IoT rengeteg új ajtót nyit ki és számos lehetőséget teremt távoli kommunikáció terén, az IoT eszközök gyakran súlyos biztonsági sebezhetőségekkel rendelkeznek, melyek gyakran kibertámadásokhoz vezetnek.

Dolgozatomban különféle megoldásokat mutatok be, melyekkel elérhető, hogy egy mikrokontroller alapú beágyazott eszköz erős biztonsággal rendelkezzen. Megoldásaim közt szerepel a biztonságos boot folyamat megvalósítása, az eszköz háttértárának titkosítása, valamint egy implementáció a távoli firmware frissítéshez. A biztonságos boot folyamat használatával biztosítható, hogy csak a tulajdonos futtathassa a kódját az eszközön, a titkosítással pedig, hogy ne is lehessen kiolvasni annak a tartalmát. A távoli frissítés kapcsán egy olyan alkalmazást valósítok meg, ami rendszeresen ellenőrzi, hogy található-e frissítés egy adott kiszolgálón, és ha igen, akkor letölti és az alapján frissíti magát. Az alkalmazás továbbá lehetővé teszi, hogy a kiszolgálóval biztonságos csatornán kommunikáljunk és tényleg csak megfelelő frissítés kerüljön telepítésre.

# Abstract

Nowadays, more and more electronic devices connect to the Internet and other devices, creating the Internet of Things, or IoT for short. Although IoT opens many new doors and creates many remote communication opportunities, IoT devices often have severe security vulnerabilities by default, which leads to cyber-attacks.

In this thesis, we present solutions that provide strong security for a microcontroller-based embedded board. We implement secure boot, encrypt the device's persistent storage, and implement an application for a remote firmware update. Using a secure boot process, we can ensure that only the owner's code can run on the device, and with encryption, its storage's content is not readable in any way. As for the remote firmware update, our application periodically checks whether a new firmware can be found on a given server. If so, the device downloads this new firmware and updates itself. The application also allows secure communication with the update server and ensures that only an appropriate update is installed.

# Chapter 1

## Introduction

Although not so long ago, mostly only PCs had the ability to connect to the Internet, today, this situation is entirely different. We use many other types of devices in our everyday lives that use the Internet. We have smartphones, smart refrigerators, smartwatches, smart speakers, smart thermostats, WiFi cameras, etc. These electronic devices together can make a network, this is called the Internet of Things, or IoT for short, and these devices are IoT devices. Even though the expression contains the word ‘Internet,’ IoT devices are not necessarily connected to the Internet. In many cases, they can also communicate, for example, via Bluetooth, NFC, and using a local network.

IoT devices can make our lives easier because they can save us time and money, give us automation, remote control, easy communication opportunities, among others. However, several IoT devices have critical security issues because they have very weak or missing security mechanisms. This is mostly because of the simplicity and low manufacturing cost of their hardware. Suppose an attacker is able to take control over many IoT devices. In that case, those devices can be used as an extensive network, for example, as a botnet, and the attacker can perform denial-of-service attacks with them, as was the case with the Mirai botnet in 2016 [1]. However, there are solutions for protecting IoT devices, for example, by using secure boot, encryption mechanisms, and a secure firmware update process.

With having secure boot functionality enabled, the device only loads and executes code that is digitally signed by trusted entities, e.g., the vendor of the device. This functionality protects against loading and executing malicious software (malware) developed by attackers.

Security can be further increased by encrypting the content stored on the device using cryptographical methods so that physical readout is not sufficient for gathering information about the running firmware. Nowadays, as most embedded devices use non-volatile flash memories, the encryption process used for them can be called flash encryption.



In many real-world scenarios, after deploying, IoT devices can only be accessed remotely. As IoT devices can communicate using the Internet, they can download new firmware updates Over The Air (OTA). It is critical that the connection between the device and the update server is secure. Otherwise, an attacker can eavesdrop on the communication, get information about the firmware or spoof the update server and send its own code to the device.

In this thesis, we show how a specific microcontroller-based IoT device, ESP32, can be made secure, enabling security functions: secure boot and flash encryption. Also, we implement an application (app) that can perform secure firmware updates by downloading new updates from a server using a secure connection. In order to achieve this, the device periodically checks for new version and verifies the identity of the update server.

The rest of this paper is organized in the following way. In Chapter 2, there is a description of the technologies upon which our solution is built. In Chapter 3, we discuss the design steps for our OTA application. In Chapter 4, we take an overview of the device used, the development environment, and present our exact solutions for secure boot, flash encryption, and OTA update. In Chapter 5, we analyze our solutions from the perspective of basic functioning and security. In Chapter 6, we mention works that are related to ours. Finally, Chapter 7 gives a conclusion of our work. To collect information about the technologies discussed in this thesis, we used many online references that are mentioned in footnotes. We accessed all the links on December 5, 2020.

## Chapter 2

# Background Technology

In this chapter, we describe the technologies upon which our solution is built. For understanding secure boot, an IoT Security Foundation article<sup>1</sup> is beneficial as it explains the main concepts very clearly. Also, a document by Zimmer and Krau [11] helps to understand the root of trust with different implementations described. Moreover, Espressif's documentation,<sup>2</sup> which is the manufacturer of the ESP32 device we use in this project (Section 4.1), also sums up these technologies very well.

In case of microcontroller-based embedded devices, there is often no clear distinction between firmware and an operating system (OS). In addition, applications are often packed together with the firmware/OS in a single image that is flashed on the device. Therefore, in this document, we do not use the term operating system. Also, we use the terms firmware image and application image alternately.<sup>3</sup> We assume that they mean the same: an image file that contains both the firmware and the set of applications to be executed on the embedded device.

### 2.1 Secure boot

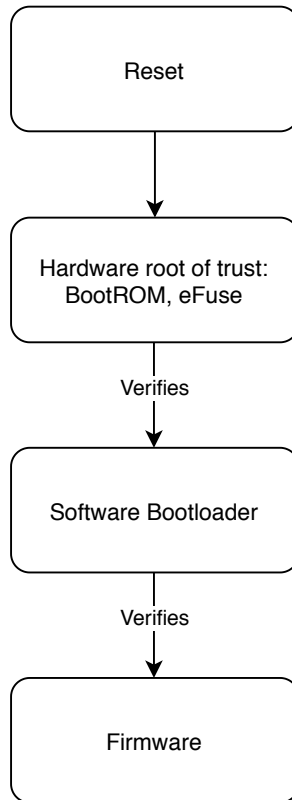
To enable secure boot functionality, the concept of 'chain of trust' needs to be implemented (Figure 2.1). In the chain of trust, there are different stages, and each must be verified by the previous one. This verification is done using digital signature schemes. Obviously, there has to be a first element that can be trusted entirely by itself; it is called the 'root of trust.' To make the root of trust unmodifiable, we should use a hardware component for it. For example, we can use Read-Only Memory (ROM) for write-protection or One Time Programmable (OTP) Memory (e.g., eFuse) to store a secure key. This key can be

---

<sup>1</sup><https://www.iotsecurityfoundation.org/best-practice-guide-articles/device-secure-boot>

<sup>2</sup><https://docs.espressif.com/projects/esp-idf/en/latest/esp32/>

<sup>3</sup>For simplicity and readability, we mostly omit the term 'image' and refer to them simply as firmware and application.



**Figure 2.1:** Stages in the chain of trust during the secure boot process

used for verifying the next stage, the software bootloader, as the software bootloader has to be digitally signed by the root of trust's key.

After the bootloader is verified, the process can continue, and the software bootloader must verify the firmware. The firmware also has to be digitally signed, so the already verified bootloader can verify it with its public key.

If we want to extend the chain of trust, we can continue in the same manner. The current component can verify the next element by checking its digital signature. This way, by using the chain of trust, the boot process is not only secure but is also modular; any stage can be updated without hardware modification.

If the verification fails on any of these stages, the boot process needs to be stopped. We can define fallback mechanisms to load a previously valid state or the simplest, and maybe a more secure solution is to reset the device again and again until a valid bootloader and firmware are flashed.

## 2.2 Flash Encryption

With flash encryption, the flash memory's content is encrypted so that only the device can decrypt it while running. To achieve the encryption, a key is used, for example, a

cryptographically secure symmetric key. This key needs to be stored on the device and Read-Write-protected (R/W-protected). As with the secure boot, we should use an OTP Memory, e.g., eFuse, to store this key in hardware.

## 2.3 Remote Firmware Update

Remote firmware update or OTA update is based on the idea that a device can request a server for a new firmware image, download it, then update its firmware with the new one. For this functionality, first, the device has to connect to the update server securely. To establish a secure connection, we can use Transport Layer Security (TLS)<sup>4</sup> between the device and the server, as it is a well-known cryptographic protocol used to provide security over a computer network.<sup>5</sup> TLS relies on asymmetric cryptography, and its security includes encryption, authentication, and integrity between the communicating applications. After the device accessed the server, it needs to store the newly downloaded firmware in its memory as a binary file. Then, it verifies whether the file downloaded is really a valid firmware. If so, typically, the device needs to reboot, and during reboot, it updates its firmware to the new one.

---

<sup>4</sup>Current approved version 1.3 is specified in RFC 8446.

<sup>5</sup>HTTP over TLS is also referred to as HTTPS.

# Chapter 3

## Design

In Chapter 2, we summarized how secure boot and flash encryption work. Because the implementation is specific for the device used, we do not have to take design steps in connection with them, and we explain everything in Section 4, where we discuss the implementation. Therefore, in this chapter, we exclusively focus on the design questions of the secure firmware update mechanism.

### 3.1 OTA self-update application

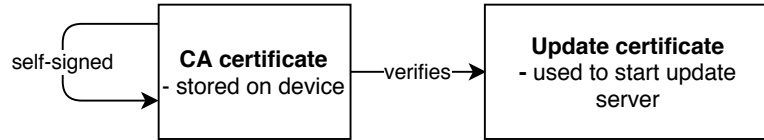
As briefly mentioned in Chapter 1, our goal is to create an application able to securely update itself with the help of an update server that stores new firmware. This process raises some questions:

- How can we establish a secure connection with the update server?
- What kind of versioning should we use, and how should we detect if a new version is available?
- How do we know whether the new firmware is valid?

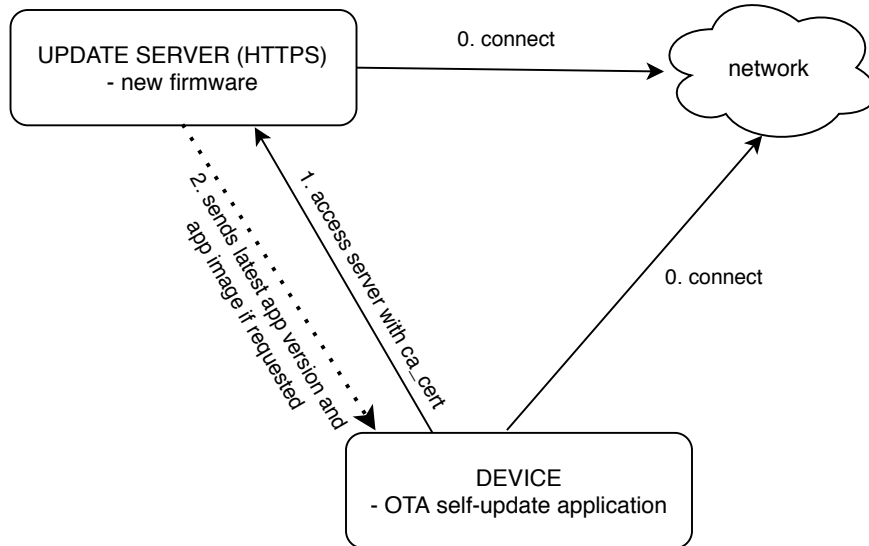
We discuss these questions in the following subsections.

#### 3.1.1 Security considerations regarding the update server

As discussed in Section 2.3, we can use TLS between the device and the server. As TLS relies on asymmetric cryptography and digital signature scheme, we need to generate a public-private key pair for the update server. We have to consider that when we want to access the running server from the device, we need to verify the identity of the update server. We can do this by creating a certificate chain, as can be seen in Figure 3.1. Firstly, we generate a self-signed key pair that provides the root certificate. We call this key pair



**Figure 3.1:** Certificate chain



**Figure 3.2:** Relationship between the server and the device

Certificate Authority (CA)<sup>1</sup> key pair. Next, we generate another key pair for the update server and a certificate that is digitally signed by the CA’s private key. Thus, we can store the CA’s public key or certificate on the device, and every time we try to reach the update server, we can verify its certificate with CA’s key. The exact relation between the device and the update server can be seen in Figure 3.2.

### 3.1.2 Versioning

For versioning, there are lots of known methods. In our project, for the sake of simplicity, we use a four-digit<sup>2</sup> number as the version number. We start from 1000 and increment with each update. But how does the device know the latest version? The answer is simple: we can download it from the update server, if we store the version there as a `latest.html` file.<sup>3</sup> Every time we connect to the update server, we first get the `latest.html` and save the version number. While we do this, we can check if it is really a four-digit number. Next, we compare this number with the actual one on the device. If the latest value is higher, we request the firmware, and if the firmware is valid, we perform the self-update.

<sup>1</sup>An entity that issues digital certificates.

<sup>2</sup>If we would run out of 4-digit numbers, we can change it to a bigger number.

<sup>3</sup>It does not necessarily need to be a `.html` file, it is only our choice. It could be `.txt`, or anything, as we store the version number as plaintext.

These steps, together with the server verification part, are shown in a sequence diagram in Figure 3.3.

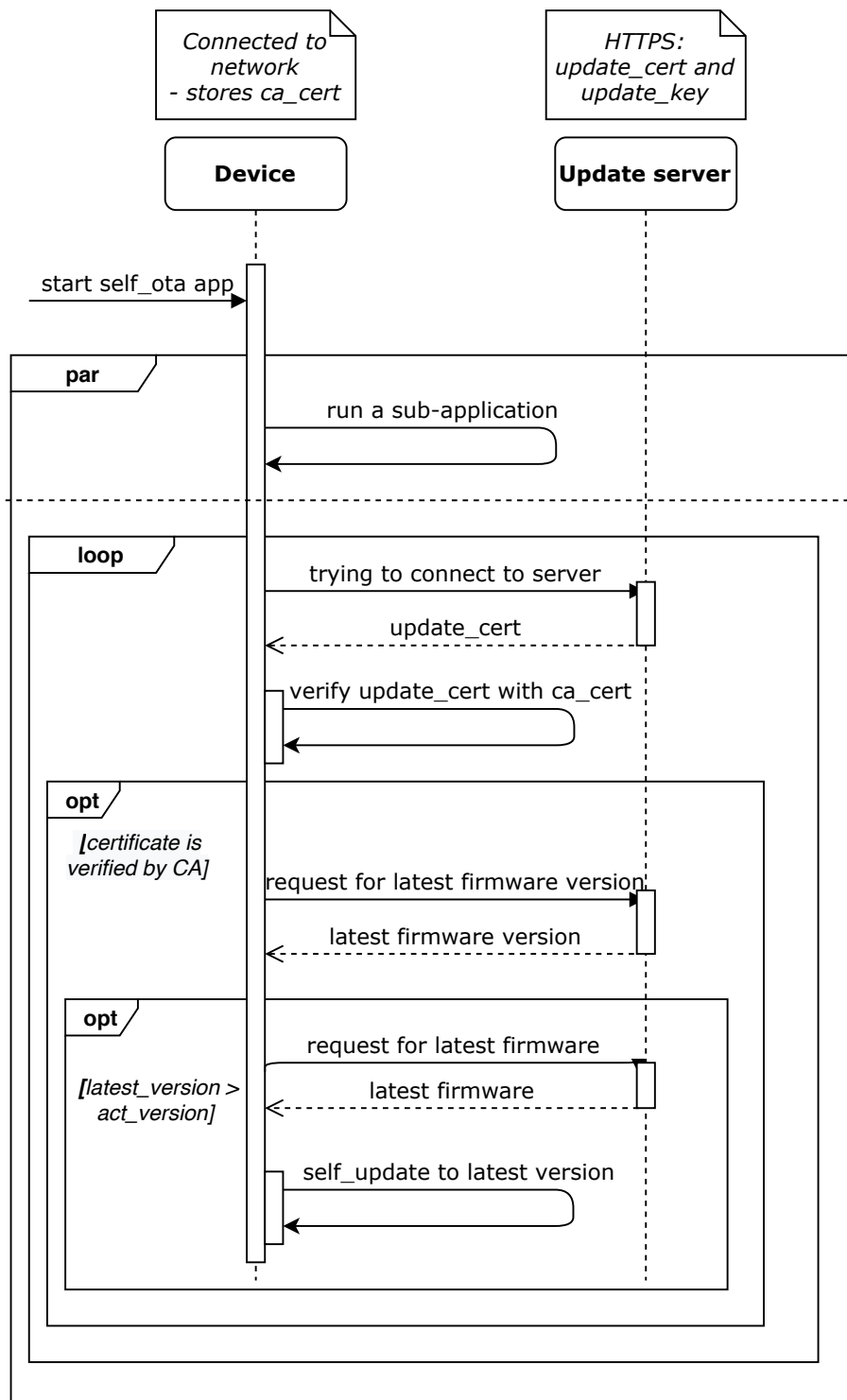
### 3.1.3 Summary of requirements

Based on the previous discussion, we can summarize the requirements for our application. The application for OTA self-update needs to store the CA's certificate and has to be able to:

- connect to WiFi network,
- reach the update server (which can potentially be remote),
- verify the update server's certificate with the CA's certificate stored on the device,
- obtain the number of the latest version via a secure connection established with the help of the server's certificate,
- check whether the latest version is newer than the actual one, and
- download the latest firmware from the update server via a secure connection again established with the help of the server's certificate.

We need a server from where the device can download the new firmware. This server needs to

- be accessible through HTTPS connection (created with a certificate signed by CA's certificate),
- store the version number of the latest firmware, and
- store the latest firmware.



**Figure 3.3:** High-level sequence diagram of the update process



# Chapter 4

## Implementation

### 4.1 ESP32

#### 4.1.1 Overview

For this project, we use a low-cost microcontroller manufactured by Espressif Systems, ESP32-WROOM.<sup>1</sup> This device has a dual-core CPU, 4 MiB flash memory, a small module that allows it to connect to a WiFi or Bluetooth network, and many other useful functionalities. These functionalities include many security functions realized by hardware; we concentrate on those.

#### 4.1.2 Security of ESP32

According to the device's datasheet,<sup>2</sup> security components and functions of ESP32 are:

- Secure boot
- Flash encryption
- 1024-bit One Time Programmable (OTP) Memory
- Cryptographic hardware acceleration:
  - Advanced Encryption Standard (AES) [10]
  - Secure Hash Algorithm (SHA) [4]
  - RSA cryptosystem [8]
  - Elliptic Curve Cryptography (ECC) [5]
  - RNG (Random Number Generator)

---

<sup>1</sup><https://www.espressif.com/en/products/socs/esp32>

<sup>2</sup>[https://www.espressif.com/sites/default/files/documentation/esp32\\_datasheet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf)

According to the documentation of eFuse Manager:<sup>3</sup>

The 1024-bit OTP Memory contains four eFuse blocks with the sizes of 256 bits:

- EFUSE\_BLK0 is used entirely for system purposes
- EFUSE\_BLK1 is used to store the flash encryption key
- EFUSE\_BLK2 is used to store the secure boot key
- EFUSE\_BLK3 can be partially reserved or used for user application

This structure is clearly visible in Figure 4.1. In this project, BLK1 and BLK2 are used to store flash encryption and secure boot keys.

BLK_0	BLK_1	BLK_2	BLK_3
Reserved	Flash Encryption Key	Secure Boot Key	Reserved / Application

**Figure 4.1:** Structure of eFuse

As the used cryptography methods are based on strong, reliable algorithms, we can ensure real strong protection for the device using all the above. We discuss the exact implementation in Sections 4.3 and 4.4.

## 4.2 Development environment for ESP32

There are different development environments we can use our device with. For example, we can use Arduino IDE,<sup>4</sup> PlatformIO,<sup>5</sup> and ESP-IDF.<sup>6</sup> Although we can write simpler codes with the first two, the manufacturer recommends ESP-IDF, and only this gives us full configurability, this is why we use ESP-IDF in the project.

### 4.2.1 ESP-IDF

Espressif IoT Development Framework, or ESP-IDF for short, is the official development framework for ESP32 devices. It is entirely open-source; we can download the source code with examples written in C from Espressif's GitHub repository.<sup>7</sup> ESP-IDF is a complex but very powerful tool that makes ESP32 fully configurable. It uses a configuration mechanism

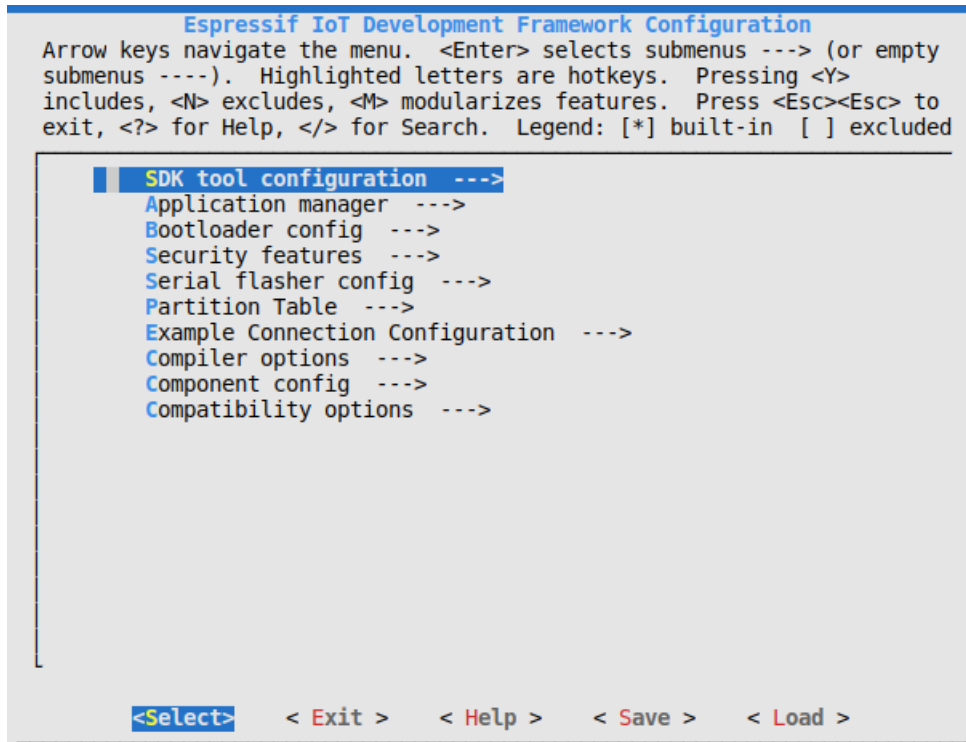
<sup>3</sup><https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/efuse.html>

<sup>4</sup><https://github.com/espressif/arduino-esp32>

<sup>5</sup><https://docs.platformio.org/en/latest/platforms/espressif32.html>

<sup>6</sup><https://github.com/espressif/esp-idf>

<sup>7</sup><https://github.com/espressif/esp-idf>



**Figure 4.2:** ESP-IDF’s menuconfig

based on Kconfig system<sup>8</sup> that gives us a handy terminal-based configuration menu, seen in Figure 4.2. The configuration made with menuconfig later is saved in a file, called `sdkconfig`, that is later used in the building procedure.

With ESP-IDF, we can also use many command-line tools and scripts with various parameters for different purposes. We use most of the commands with the `idf.py` command-line tool that helps us in the build process<sup>9</sup> and enables us to do many things. The commands we use with `idf.py` in this project are the following (we give their original definition as in the documentation):

- `idf.py menuconfig` runs the “menuconfig” tool to configure the project
- `idf.py build` builds the project found in the current directory
- `idf.py flash` automatically builds the project if necessary, and then flashes it to the target
- `idf.py encrypted-flash` updates all partitions in an encrypted format
- `idf.py monitor` displays serial output from the target

In connection with flashing, `esptool.py`<sup>10</sup> commands can be used to read and write flash:

<sup>8</sup><https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/kconfig.html>

<sup>9</sup>ESP-IDF Build System: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/build-system.html>

<sup>10</sup><https://github.com/espressif/esptool>

- `esptool.py write_flash [where] [file.bin]` command writes a specific binary at a given location in memory
- `esptool.py read_flash [from] [to] [output.bin]` command allows reading back the contents of flash and save to a specified binary

Another useful tool is `espefuse.py`<sup>11</sup> that is made for reading and burning in eFuse values:

- `espefuse.py summary` displays a summary of eFuse's content
- `espefuse.py burn_efuse [SPECIFIC FUSE] [VALUE]` is used to burn an eFuse to a new value
- `espefuse.py burn_key` loads a key (stored as a raw binary file) and burns it to a key block (BLK1, BLK2, or BLK3)

With the commands above, we can fully configure our device from the terminal. For development, we can use some official plugins with different environments, such as Eclipse<sup>12</sup> or Visual Studio Code.<sup>13</sup> For this project, we use Visual Studio Code (Figure 4.3 shows its code editor).

## 4.2.2 Visual Studio Code

Visual Studio Code<sup>14</sup>, or VS Code for short, is a free, open-source-based, multi-platform source-code editor application made by Microsoft. When using with ESP-IDF plugin, it works very similarly to an IDE; we can configure and compile our project. Besides VS Code's default capabilities, there are some ESP-IDF-specific features we can use, for example, a graphical SDK configuration editor. Also, as Figure 4.4 shows, many of the commands can be performed automatically. In this project, we take advantage of VS Code's auto-completion and syntax highlighting functions. However, we perform building and flashing commands in ESP-IDF terminal as it gives us the maximum flexibility in configuration.

## 4.3 Secure Boot

The way how secure boot can be enabled on ESP32 is well-documented on Espressif's site.<sup>15</sup> Therefore, in this section, every technical detail is based on this documentation.

<sup>11</sup><https://github.com/espressif/esptool/wiki/espefuse>

<sup>12</sup>Eclipse ESP-IDF plugin: <https://github.com/espressif/idf-eclipse-plugin>

<sup>13</sup>Visual Studio ESP-IDF plugin: <https://github.com/espressif/idf-eclipse-plugin>

<sup>14</sup>Official website: <https://code.visualstudio.com/>

<sup>15</sup><https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/secure-boot-v1.html>

```

C simple_ota_example.c ●
C-Implementation > ota_self-update_sb > main > C simple_ota_example.c > set_latest_version(esp_http_client_event_t*)
38
39 #define VERSION_LENGTH 4 // Version is a 4-digit number
40 #define IMAGE_URL_SIZE 50
41
42 #if CONFIG_EXAMPLE_CONNECT_WIFI
43 #include "esp_wifi.h"
44 #endif
45
46 extern const uint8_t ca_cert_file[] asm("_binary_ca_cert_pem_start");
47 int latest_version = VERSION;
48 char latest_version_str[5] = VERSION_STR;
49 bool version_check = false; // True if we are checking the version right now
50
51 /* Sets latest version variables to the number arrived from latest.html*/
52 void set_latest_version(esp_http_client_event_t *evt)
53 {
54     /* VERSION_LENGTH-digit number is required */
55     if (evt->data_len != VERSION_LENGTH)
56     {
57         ESP_LOGE(OTA_TAG, "latest.html's content should be a %d-digit number",
58                 VERSION_LENGTH);
59         return;
60     }
61
62     char *data = (char *)evt->data;
63     // You, a month ago • download latest certificate from another server
64     /* Check whether data really contains only digits */

```

Figure 4.3: Code editor in VSCode

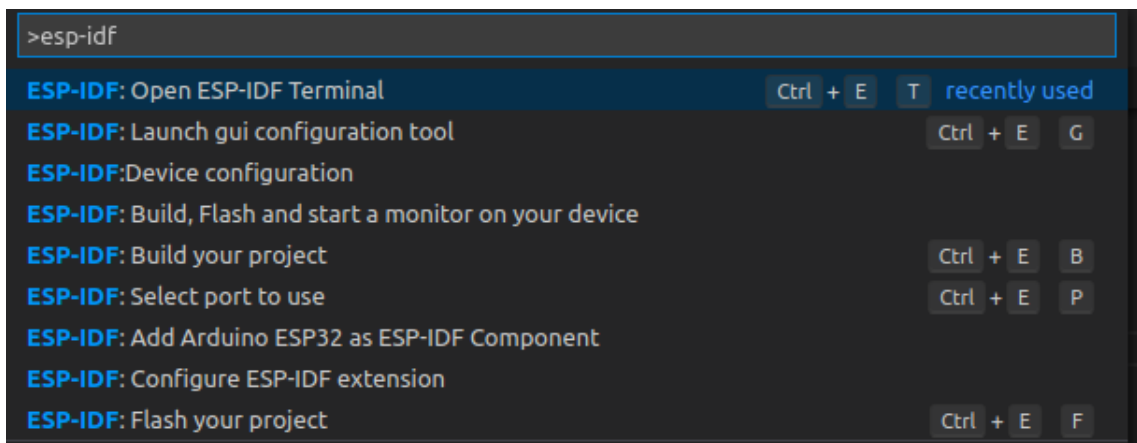


Figure 4.4: Some ESP-IDF-specific functions in VS Code

However, simply following the steps below without a clear understanding of the underlying concepts can lead to problems. This section describes how we enable secure boot using the official documentation and by adding our own pictures. We also show that by having secure boot enabled, incorrectly signed applications do not run on the device.

### 4.3.1 Cryptography used

As already mentioned in Section 4.1.2, ESP32 has many built-in security functions realized by hardware. For secure boot, the most essential hardware component used is the eFuse. After enabling secure boot, the device stores an R/W-protected 256-bit key in eFuse Block2 that makes the hardware component the root of trust, as discussed in Section 2.1. The value stored in eFuse is an AES256 key that can be generated in two different ways. These methods are explained in Section 4.3.2. Besides this AES256 key, ECDSA keys are used for signing application images. These keys need to be permanent and cryptographically secure. Therefore, for this project, the ECDSA signing key was generated using OpenSSL<sup>16</sup> that can generate cryptographically secure pseudo-random numbers. The key is generated using the following command:

```
openssl ecparam -name prime256v1 -genkey -noout -out mykey.pem
```

After secure boot is enabled, and both AES and ECDSA keys are generated, a secure boot digest is flashed to 0x0 in the flash, derived from the AES key, an Initialization Vector, and the bootloader image contents. Every time the device boots, there is a comparison made by hardware, whether the saved digest at 0x0 matches the newly calculated one. If it does not, the boot process does not continue. In addition, the bootloader stores the ECDSA public key that can be used to verify the signature of the application image.

### 4.3.2 Reflashable and One-time flash methods

In ESP32, there are two different approaches to enable secure boot. The main difference is in how the AES256 key in eFuse being generated. In One-time flash mode, the user does not need to supply this key; the internal hardware random number generator generates it. So, as eFuse is R/W-protected after burning, there is no way to access the key externally. However, in Reflashable mode, the AES256 key is equivalent to the SHA-256 hash of the ECDSA private key used for app signing. Thus, the user can generate a new bootloader digest using this ECDSA key and reflash the new bootloader and digest together on the device. From the security point of view, One-time flash is more secure than Reflashable method because there is no way to externally gain access to the AES256 key created by the hardware RNG. Also, using Reflashable method, we can flash the same bootloader to multiple devices. That is why Espressif recommends Reflashable method only for development purposes. Still, as this project is not part of any production environment, we use Reflashable method. It makes no significant difference, especially with using flash encryption.

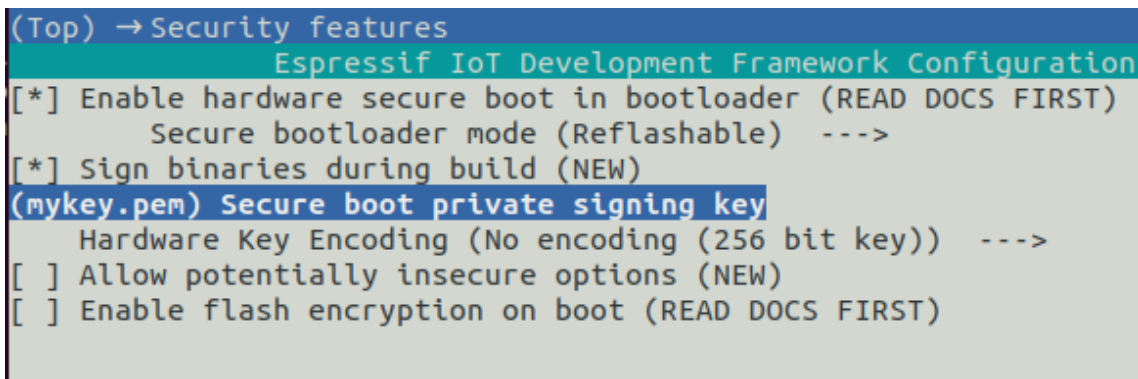
---

<sup>16</sup>A cryptography and SSL/TLS Toolkit: <https://www.openssl.org/>

### 4.3.3 Setup for Reflashable method

Now, as for the exact setup of secure boot in Reflashable mode, the original documentation describes the exact steps that we can follow; we mention them only briefly the way the project use them. Enabling secure boot is a configuration step enabled in menuconfig; it does not appear in C code.

1. Generate ECDSA signing key with OpenSSL
2. In menuconfig, enable secure boot in Reflashable mode, and supply our generated private key as secure boot signing key, as seen in Figure 4.5

A screenshot of the menuconfig interface. The title bar reads "(Top) → Security features" and "Espressif IoT Development Framework Configuration". The configuration list includes: "[\*] Enable hardware secure boot in bootloader (READ DOCS FIRST) Secure bootloader mode (Reflashable) --->", "[\*] Sign binaries during build (NEW)", "(mykey.pem) Secure boot private signing key" (highlighted in blue), "Hardware Key Encoding (No encoding (256 bit key)) --->", "[ ] Allow potentially insecure options (NEW)", and "[ ] Enable flash encryption on boot (READ DOCS FIRST)".

```
(Top) → Security features
Espressif IoT Development Framework Configuration
[*] Enable hardware secure boot in bootloader (READ DOCS FIRST)
    Secure bootloader mode (Reflashable) --->
[*] Sign binaries during build (NEW)
(mykey.pem) Secure boot private signing key
    Hardware Key Encoding (No encoding (256 bit key)) --->
[ ] Allow potentially insecure options (NEW)
[ ] Enable flash encryption on boot (READ DOCS FIRST)
```

**Figure 4.5:** Configuration for enabling secure boot in Reflashable mode

3. Flash the bootloader to the device with command `idf.py bootloader`. It results in a longer text that contains commands we need to execute next.
4. To burn the secure boot key in eFuse, execute the following command:

```
espefuse.py burn_key secure_boot /build/bootloader/secure-
bootloader-key-256.bin
```

After that, we write BURN, and if we check the content of eFuse with command `espefuse.py summary`, we see that the secure boot key is successfully burnt in eFuse, and has R/W-protection. Also, `ABS_DONE_0`'s value is 1, indicating that secure boot is enabled (Figure 4.6).

5. Flash the bootloader from 0x1000:

```
esptool.py write_flash 0x1000 build/bootloader/bootloader.bin
```

6. Build and flash our application with `idf.py flash` command (Figure 4.7)
7. Check the working application with running `idf.py monitor` (Figure 4.8 and 4.9)

```

Security fuses:
FLASH_CRYPT_CNT      Flash encryption mode counter            = 0 R/W (0x0)
FLASH_CRYPT_CONFIG   Flash encryption config (key tweak bits) = 0 R/W (0x0)
CONSOLE_DEBUG_DISABLE Disable ROM BASIC interpreter fallback   = 1 R/W (0x1)
ABS_DONE_0           secure boot enabled for bootloader   = 1 R/W (0x1)
ABS_DONE_1           secure boot abstract 1 locked     = 0 R/W (0x0)
JTAG_DISABLE         Disable JTAG                           = 0 R/W (0x0)
DISABLE_DL_ENCRYPT    Disable flash encryption in UART bootloader = 0 R/W (0x0)
DISABLE_DL_DECRYPT    Disable flash decryption in UART bootloader = 0 R/W (0x0)
DISABLE_DL_CACHE     Disable flash cache in UART bootloader     = 0 R/W (0x0)
BLK1                 Flash encryption key
= 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 R/W
BLK2                 Secure boot key
= ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? -/-
BLK3                 Variable Block 3
= 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 R/W

```

**Figure 4.6:** Content of eFuse after burning secure boot key

```

[5/7] Generating binary image from built executable
esptool.py v3.0-dev
Generated /home/linuxman/esp/hello_world/build/hello-world-unsigned.bin
[6/7] Generating signed binary image
espsign.py v3.0-dev
Signed 196528 bytes of data from /home/linuxman/esp/hello_world/build/hello-world-unsigned.bin with key /home/linuxman/esp/hello_world/mykey.pem
Generated signed binary image /home/linuxman/esp/hello_world/build/hello-world.bin from /home/linuxman/esp/hello_world/build/hello-world-unsigned.bin
[6/7] cd /home/linuxman/esp/esp-idf/components/esptool...man/esp/esp-idf/components/esptool_py/run_esptool.cmake
esptool.py --chip esp32 -p /dev/ttyUSB0 -b 460800 --before=default_reset --after=no_reset write_flash --flash_mode dio --flash_freq 40m --flash_size 2MB 0x10000 partition_table/partition-table.bin 0x20000 hello-world.bin
esptool.py v3.0-dev
Serial port /dev/ttyUSB0
Connecting.....

```

**Figure 4.7:** The application is being signed and flashed to the device

```

I (193) esp_image: segment 6: paddr=0x00044634 vaddr=0x00000000 size=0x0b94c ( 47436)
I (215) esp_image: Verifying image signature...
I (563) boot: Loaded app from partition at offset 0x20000
I (563) secure_boot_v1: bootloader secure boot is already enabled. No need to generate digest
I (568) boot: Checking secure boot...
I (572) secure_boot_v1: bootloader secure boot is already enabled, continuing..

```

**Figure 4.8:** We can see through monitoring our application that secure boot is enabled

```

I (701) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.
Hello world!
This is esp32 chip with 2 CPU cores, WiFi/BT/BLE, silicon revision 1, 2MB external flash
Restarting in 10 seconds...
Restarting in 9 seconds...
Restarting in 8 seconds...

```

**Figure 4.9:** Working basic Hello World application with Secure Boot

### 4.3.4 Verifying signature

We presented secure boot with the desired operation in the previous section. Let’s check two cases where secure boot protects the device from an unverified code run on it. The first case is when we try to flash an application signed with a key that differs from the



```

I (458) esp_image: Verifying image signature...
E (458) secure_boot: image has invalid signature version field 0xffffffff
E (459) esp_image: Secure boot signature verification failed
I (465) esp_image: Calculating simple hash to check for corruption...
W (694) esp_image: image valid, signature bad
E (695) boot: Factory app partition is not bootable
E (695) boot: No bootable app partitions in the partition table
ets Jun  8 2016 00:22:57

rst:0x3 (SW RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DI0, clock div:2
load:0x3fff0030,len:4
load:0x3fff0034,len:11144
load:0x40078000,len:21668
load:0x40080400,len:3984
entry 0x40080688
I (56) boot: ESP-IDF v4.2-dev-910-g4fe04f115 2nd stage bootloader

```

**Figure 4.10:** Error when the application is not correctly signed

valid ECDSA signing key. The second case is when we try to do the same but without any signature. In both cases, we can see the output as in Figure 4.10. The bootloader starts, but when it tries to verify the application image, the verification fails because the signature is wrong.

## 4.4 Flash Encryption

Similarly to secure boot, ESP32's flash encryption is also well-documented on Espressif's site,<sup>17</sup> so every technical detail mentioned here is based on the original documentation.

### 4.4.1 Cryptography used

As secure boot, flash encryption also uses the eFuse. We have to store a permanent AES-256 key in Block 1 and burn some eFuse bits to enable this functionality. As for the eFuse key, we can choose between using a key generated by the hardware RNG or supply it ourselves. Although with secure boot, we supplied the key ourselves, right now, it does not have any particular advantage, so we generate a private key with RNG. The device can encrypt data during the flash process using this key. In ESP32, there are two different approaches to enable flash encryption: Development and Release mode. When using flash encryption, some other functions are disabled depending on the mode we use, as discussed in the following section.

<sup>17</sup><https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/flash-encryption.html>

## 4.4.2 Development and Release modes

What the two methods have in common is that they encrypt the content of the flash the same way, using the AES-256 key stored in eFuse. To prevent someone from decrypting the content of the flash via the UART,<sup>18</sup> UART decryption is disabled. Also, flash cache in UART bootloader, and JTAG<sup>19</sup> is disabled for more security in both modes.

The biggest difference between the two modes is that after enabling flash encryption in Development mode, we are limited to disable it only three times. As the eFuse key obviously cannot be modified, it is done by modifying an 8-bit long eFuse value, called flash encryption mode counter (FLASH\_CRYPT\_CNT). Initially, its value is 0x0, and whenever switching between enabling and disabling flash encryption, a bit with value 1 is being burnt. So, if the parity of 1 bits in this 8-bit value is odd, encryption is enabled. If it is even, encryption is disabled. Because of these, we can disable the already enabled encryption only three times. In this project, we enable it only once.

Another significant difference is that in Release Mode, also UART encryption is disabled. UART encryption means that after enabling flash encryption in Release mode, we cannot flash encrypted applications or bootloader on the device via UART connection. The only option to update is using OTA method. As in this project, we want to flash and test our base OTA application by monitoring it, we use UART connection, so we choose Development mode.

## 4.4.3 Setup in Development mode

Now, again, original documentation describes exact steps to enable flash encryption; many steps are performed automatically, we mention the important steps only briefly. First, in menuconfig, we enable flash encryption in Development mode (Figure 4.11). Next, we build and flash the application with `idf.py flash` command on the device. As a result, the device generates a new key, sets appropriate eFuse values (Figure 4.12), then reboots with enabled encryption (Figure 4.13). Figure 4.14 shows flash encryption is really enabled, and as we are in Development mode, we could disable encryption functionality three times. Because everything went well, we can see that our application is running as expected (Figure 4.15).

---

<sup>18</sup>Universal Asynchronous Receiver Transmitter: hardware used for asynchronous serial communication

<sup>19</sup>Joint Test Action Group: standard for testing integrated circuit boards

```

(Top) → Security features
Espressif IoT Development Framework Configuration
App Signing Scheme (ECDSA) --->
[*] Enable hardware Secure Boot in bootloader (READ DOCS FIRST)
    Select secure boot version (Enable Secure Boot version 1
    Secure bootloader mode (Reflashable) --->
[*] Sign binaries during build
(mykey.pem) Secure boot private signing key
    Hardware Key Encoding (No encoding (256 bit key)) --->
[ ] Allow potentially insecure options
[*] Enable flash encryption on boot (READ DOCS FIRST)
    Enable usage mode (Development(NOT SECURE)) --->
    Potentially insecure options --->

```

Figure 4.11: Configuration for enabling flash encryption in Development mode

```

I (560) boot: Checking flash encryption...
I (564) flash_encrypt: Generating new flash encryption key...
I (583) flash_encrypt: Read & write protecting new key...
I (594) flash_encrypt: Setting CRYPT_CONFIG efuse to 0xF
W (605) flash_encrypt: Not disabling UART bootloader encryption
I (605) flash_encrypt: Disable UART bootloader decryption...
I (607) flash_encrypt: Disable UART bootloader MMU cache...
I (613) flash_encrypt: Disable JTAG...
I (618) flash_encrypt: Disable ROM BASIC interpreter fallback...

```

Figure 4.12: Appropriate values are written in eFuse automatically

```

I (13861) secure_boot_v1: bootloader secure boot is already enabled, con
I (13868) boot: Resetting with flash encryption enabled...
ets Jun  8 2016 00:22:57

rst:0x3 (SW_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0030,len:4
load:0x3fff0034,len:11144
load:0x40078000,len:21668
load:0x40080400,len:3984
0x40080400: _init at ???

entry 0x40080688
I (56) boot: ESP-IDF v4.2-dev-910-g4fe04f115 2nd stage bootloader
I (57) boot: compile time 16:41:20
I (57) boot: chip revision: 1

```

Figure 4.13: The device reboots with enabled flash encryption

```
I (582) boot: Checking flash encryption...
I (587) flash_encrypt: flash encryption is enabled (3 plaintext flashes left)
I (594) boot: Checking secure boot...
```

**Figure 4.14:** Flash encryption mode is enabled in Development mode

```
I (734) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.
Hello world!
This is esp32 chip with 2 CPU cores, WiFi/BT/BLE, silicon revision 1, 2MB external flash
Restarting in 10 seconds...
Restarting in 9 seconds...
Restarting in 8 seconds...
Restarting in 7 seconds...
```

**Figure 4.15:** Encrypted application successfully starts

#### 4.4.4 Content of the flash before and after

When encrypting the flash, it can be interesting to see how its content changed. As with `idf.py read_flash` command, we can read the flash content and store it in a binary file; we can check the differences between unencrypted and encrypted flash. We could go through every part of the flash to compare, but here, we make this comparison only with the beginning of the bootloader at 0x1000. Figure 4.16 shows that before encryption, we have readable strings stored in the flash (which can be dangerous with sensitive data). On the contrary, after performing encryption, the same text is becoming gibberish because of the encryption (Figure 4.17).

```
00001000 E9 04 02 10 88 06 08 40 EE 00 00 00 00 00 00 00 .....@.....
00001010 00 00 00 00 00 00 00 01 30 00 FF 3F 04 00 00 00 .....0..?....
00001020 FF FF FF FF 34 00 FF 3F 8C 25 00 00 EE EE EE EE ...4..?.%.....
00001030 EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE .....
00001040 EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE 01 01 01 01 .....
00001050 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
00001060 01 01 01 01 01 01 01 01 01 01 01 01 00 10 00 00 .....
00001070 00 F0 00 00 01 00 00 00 00 F0 F5 3F 00 50 04 FF .....?.P..
00001080 AC 00 00 00 00 00 00 00 04 00 00 00 05 00 00 00 .....
00001090 06 00 00 00 07 00 00 00 1B 5B 30 3B 33 31 6D 45 .....[0;31mE
000010A0 20 28 25 64 29 20 25 73 3A 20 6C 6F 61 64 20 70 (%d) %s: load p
000010B0 61 72 74 69 74 69 6F 6E 20 74 61 62 6C 65 20 65 artilion table e
000010C0 72 72 6F 72 21 1B 5B 30 6D 0A 00 65 72 72 00 6E rror!.[0m..err.n
000010D0 6F 00 79 65 73 00 62 6F 6F 74 5F 63 6F 6D 6D 00 o.yes.boot_comm.
000010E0 1B 5B 30 3B 33 31 6D 45 20 28 25 64 29 20 25 73 .[0;31mE (%d) %s
000010F0 3A 20 62 6F 6F 74 6C 6F 61 64 65 72 5F 6D 6D 61 : bootloader_mma
00001100 70 28 30 78 25 78 2C 20 30 78 25 78 29 20 66 61 p(0x%x, 0x%x) fa
00001110 69 6C 65 64 1B 5B 30 6D 0A 00 1B 5B 30 3B 33 31 iled.[0m...[0;31
00001120 6D 45 20 28 25 64 29 20 25 73 3A 20 46 61 69 6C mE (%d) %s: Fail
00001130 65 64 20 74 6F 20 76 65 72 69 66 79 20 70 61 72 ed to verify par
00001140 74 69 74 69 6F 6E 20 74 61 62 6C 65 1B 5B 30 6D tion table.[0m
00001150 0A 00 1B 5B 30 3B 33 32 6D 49 20 28 25 64 29 20 ...[0;32mI (%d)
00001160 25 73 3A 20 23 23 20 4C 61 62 65 6C 20 20 20 20 %s: ## Label
00001170 20 20 20 20 20 20 20 20 55 73 61 67 65 20 4F 66 Usage Of
00001180 66 73 65 74 20 20 20 4C 65 6E 67 74 68 20 20 20 fset Length
00001190 43 6C 65 61 6E 65 64 1B 5B 30 6D 0A 00 1B 5B 30 Cleaned.[0m...[0
```

**Figure 4.16:** Unencrypted flash

```

00001000 47 1C 59 D9 93 72 42 2A 48 0E AB 03 5B EF F6 18 G.V..rB*H...[...
00001010 12 96 B3 08 E0 94 7B 50 32 30 48 DF 76 6B 7A 67 .....{P20H.vkzg
00001020 A0 44 31 D0 B6 49 23 00 33 9A C2 6F 61 B4 2D 89 .D1..I#.3..oa.-.
00001030 2E 72 D9 09 FB E7 1F A3 2C 5D DE 67 24 B9 BF 22 .r.....,].g$.
00001040 87 53 09 16 52 09 B0 C7 84 E6 4F 81 83 01 82 44 .S..R.....O...D
00001050 79 FC 24 FB 87 45 16 19 F0 BC BE CC 2D 2B 0C D0 y.$..E.....-+..
00001060 2F 37 9E EF 96 7E 8C 06 CE 4F 35 4A C2 50 C4 E6 /7....~...05J.P..
00001070 85 BC C6 CF BB 7F 52 2E FE 3B 44 D1 77 38 F8 F7 .....R...;D.w8..
00001080 21 1B A3 6C 1A AB 56 86 17 06 B1 34 F7 E5 C1 74 !..l..V....4...t
00001090 52 E6 3B 34 A5 F4 1F 3A C7 04 EA AC CB 7F E2 96 R.;4...:.....
000010A0 C7 C6 6F EA 48 36 AA 58 16 11 0C F0 1D C0 E1 8C ..o.H6.X.....
000010B0 B5 F0 9C BF D4 F7 6B E8 61 3D A3 85 25 72 FE 6B .....k..a=.%r.k
000010C0 55 F9 AA B0 7E 0C 9B 01 EC A8 B1 94 26 E8 54 14 U....~.....&.T.
000010D0 24 2F AC 2B C5 05 5C CA 2C A5 DF DC 23 3B 51 81 $/.+..\.,...#;Q.
000010E0 C0 70 84 CC 45 67 41 66 8A E2 18 8C 23 40 50 CB .p..EgAf....#@P.
000010F0 3A 49 6F 10 60 31 36 E2 E7 89 D6 5C 2B 47 2C DA :Io.`16....\+G,.
00001100 25 65 F9 B7 BA F2 D4 7F C4 12 7B 4D B3 A4 1A 44 %e.....{M...D
00001110 51 7A 7F 38 5C A4 24 C0 D3 FA DF 31 35 D3 D5 F5 Qz;8\.$....15...
00001120 B0 3C 9A 1F A3 D4 E0 F6 41 C2 9F 1E 07 36 91 D3 .<.....A....6..
00001130 86 98 66 9B 7C 99 E3 B7 E4 69 21 12 C7 5A E4 FB ..f.|....i!..Z..
00001140 74 5D BC BA 56 59 74 38 7B BD 06 EA 84 A8 9E 35 t]..VYt8{.....5
00001150 03 87 E4 90 70 58 39 72 5F E6 22 C4 93 99 4E C6 ....pX9r_."...N.
00001160 F5 5C 75 DF 8C F3 55 09 28 0C 65 C8 4B A4 05 06 .\u...U.(.e.K...

```

Figure 4.17: Encrypted flash

## 4.5 OTA self-update application

### 4.5.1 OTA with ESP32

The OTA process is also well-documented on Espressif’s site.<sup>20</sup> As this project concentrates on security considerations in the software, we do not go into many details in connection with the hardware part. Maybe the only thing we need to know that there are multiple OTA partitions. As quoted from the documentation:

OTA requires configuring the Partition Table of the device with at least two “OTA app slot” partitions (i.e. ota\_0 and ota\_1) and an “OTA Data Partition”.

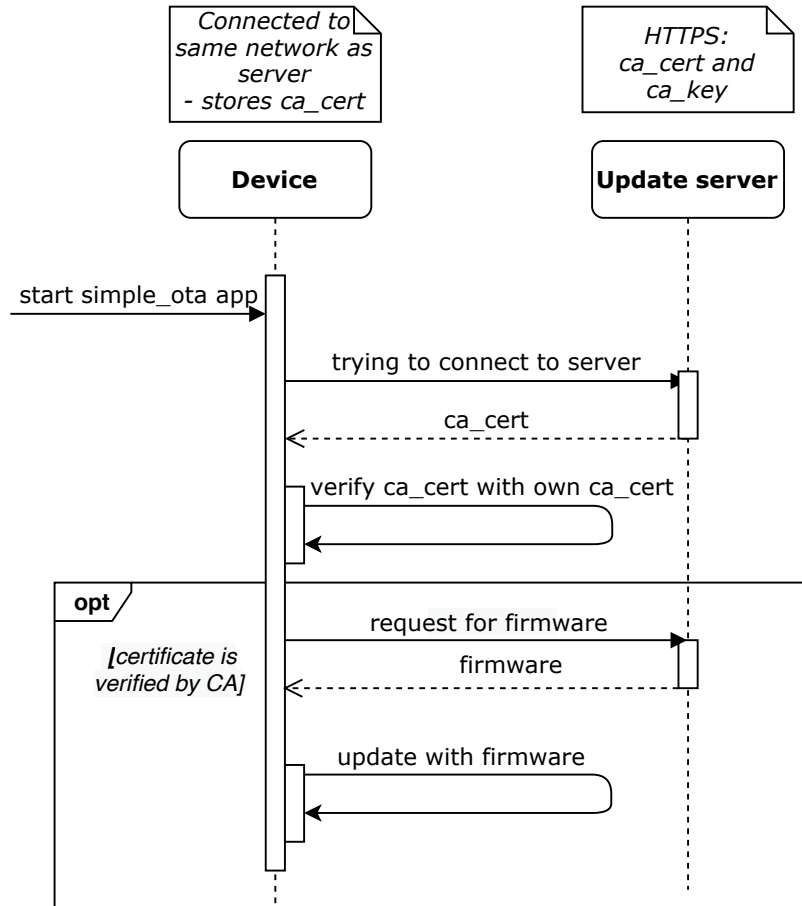
The OTA operation functions write a new app firmware image to whichever OTA app slot is not currently being used for booting. Once the image is verified, the OTA Data partition is updated to specify that this image should be used for the next boot.

### 4.5.2 Simple OTA update example application

As a starting point, we use an example application from GitHub, Simple OTA example.<sup>21</sup> The goal of this example application is to download new firmware, then perform an update with the help of this. The application firstly connects to a specified network, then, with the

<sup>20</sup><https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/ota.html>

<sup>21</sup>ESP-IDF OTA examples on GitHub: <https://github.com/espressif/esp-idf/tree/master/examples/system/ota>



**Figure 4.18:** Sequence diagram of Simple OTA update example

help of a pre-given certificate, verifies the HTTPS server. If this verification is successful, the application downloads the new firmware, checks whether it is valid, then performs the update. If this update is successful, the system reboots and starts with the new firmware. We can compare this example's sequence diagram (Figure 4.18) with the one we created for ours (Figure 3.3). We can see that the two sequence diagrams are very similar, so our application can be based on this one. In both cases, we have to create a server that stores the files we need. But obviously, we must implement additional elements, as well. We need to

- have a sub-application that can run parallel with the OTA update part,
- ensure opportunity for key exchange as discussed in Section 3.1.1, and
- use versioning as described in Section 3.1.2.

### 4.5.3 The update server

As an update server, we use a server created locally by OpenSSL. As mentioned in Section 3.1.3, we store the new firmware and its version number on this server. It is the server

owner's responsibility to set `latest.html`'s value to the same value as the latest firmware's actual version number.

As discussed in Section 3.1.1, to provide a secure connection, we need to generate a self-signed CA certificate and an update server certificate that is signed by CA's private key. We generate the CA key and certificate with the following command:

```
openssl req -x509 -newkey rsa:2048 -days 365 -nodes -keyout ca_key.pem
-out ca_cert.pem
```

This command generates a 2048-bit RSA private key with the name of `ca_key`, and a self-signed X.509<sup>22</sup>-type certificate valid until 365 days. Both files are stored as PEM<sup>23</sup> files.

Next, we generate a key and a certificate for the update server with the following commands:

```
openssl req -newkey rsa:2048 -days 60 -nodes -keyout update_key.pem
-out update_req.pem
openssl x509 -req -in update_req.pem -CA ca_cert.pem -CAkey ca_key.pem
-CAcreateserial -out update_cert.pem
```

The first command, similar to the previous one, generates a 2048-bit RSA private key, but here, it creates a certificate signing request (CSR) instead of a certificate. Using this CSR file, CA certificate and CA private key, with the second command OpenSSL generates a certificate that is signed by the CA. In our case, we use our PC's IP address as Common Name; so that if ESP32 is connected to the same network, it can connect to our server.

With command `openssl verify`, we can verify the generated update server's certificate with CA's certificate. As Figure 4.19 shows, we get an OK message meaning that everything went well, update certificate is verified by CA.

```
D:\http_server\update>openssl verify -CAfile ..\CA\ca_cert.pem .\update_cert.pem
.\update_cert.pem: OK
```

**Figure 4.19:** Verifying update server's certificate with CA certificate with OpenSSL

Last, we start the update server on a specific port, 8060, in our case, with the following command:

```
openssl s_server -WWW -key update_key.pem -cert update_cert.pem -port
8060
```

Figure 4.20 shows that the server successfully started. If we later put our `latest.html` and the latest firmware to the folder where the server started, we can access them.

<sup>22</sup>Defined by RFC5280.

<sup>23</sup>Privacy-Enhanced Mail defined by RFC 1421 and 1424.



```
D:\http_server\update\https_update_server>openssl s_server -www -key
..\update_key.pem -cert ..\update_cert.pem -port 8060
Loading 'screen' into random state - done
Using default temp DH parameters
ACCEPT
```

**Figure 4.20:** OpenSSL server successfully started on port 8060

#### 4.5.4 Changing keys

As discussed in Section 4.5.3, we can use a self-signed CA certificate to verify the update server's certificate. ESP-IDF gives us a simple way to flash the CA certificate to the device and to use it in the code. We need to store the certificate inside the project directory and register it inside makefiles. Then, we can use its content as a constant value in our code:

```
extern const uint8_t ca_cert_file[] asm("_binary_ca_cert_pem_start");
```

This means, every time we build a new version that we want to flash, we also have to put the CA certificate inside the firmware. This certificate is permanent until its expiration day, so until then, we can flash this same certificate on the device. However, there can be cases when we want to replace this certificate with a new one. For example, when the expiration date is close, as after expiration, we could not verify the update server's certificate with the CA certificate on the device. Also, when we suspect that the CA private key has been compromised, we definitely want to generate a new private key with a new certificate, as soon as possible. As if an attacker has our private key causes a huge compromise, we can only hope that we can change the key faster than the attacker uses it for exploitation.

#### 4.5.5 Sub-application

As during OTA firmware update, we want to update an existing application; we must have a sub-application on the device beside the OTA part that can run parallel. As this project focuses on the technical details of the OTA update, we use a very simple blink application as sub-application (Code snippet 4.1). This application periodically turns the device's LED on and off, while printing the actual version number to the console. As we want to flash the base application via serial port, we can also monitor the serial output for testing. This gives us an easy way to check the actual running version.

#### 4.5.6 Versioning

As already discussed in Section 3.1.2, for the sake of simplicity, we use 4-digit numbers as version numbers in this project stored in a `latest.html` file. To prevent security issues, after getting this version number from the update server, we do some security checks. First, we check whether the string we got is really exactly four characters long. Second,



```

const int VERSION = 1000;

void blink_task(void *pvParameter){
    gpio_pad_select_gpio(BLINK_GPIO);
    gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);
    while (1)
    {
        printf("Turning off the LED in version %d\n", VERSION);
        gpio_set_level(BLINK_GPIO, 0);
        vTaskDelay(2500 / portTICK_PERIOD_MS);
        printf("Turning on the LED in version %d\n", VERSION);
        gpio_set_level(BLINK_GPIO, 1);
        vTaskDelay(2500 / portTICK_PERIOD_MS);
    }
}

```

**Code snippet 4.1:** Blink application for ESP32 written in C

we check whether all the four characters contain a decimal digit. If both conditions are met, we can assume that the downloaded string is really a valid version number. Code snippet 4.2 shows this process. After that, we can compare this version number with the actual one, and depending on the comparison's result; we download the latest image.

```

void set_latest_version(esp_http_client_event_t *evt){
    if (evt->data_len != VERSION_LENGTH){
        ESP_LOGE(OTA_TAG, "latest.html's content should be a %d-digit number ",
        VERSION_LENGTH);
        return;
    }

    char *data = (char *)evt->data;

    for (int i = 0; i < VERSION_LENGTH; ++i){
        if (!isdigit(data[i])){
            ESP_LOGE(OTA_TAG, "latest.html contains non-digit character ");
            return;
        }
    }
    strncpy(latest_version_str, data, VERSION_LENGTH);
    latest_version = atoi(latest_version_str);
}

```

**Code snippet 4.2:** Setting the latest version

## 4.5.7 Updating firmware OTA

In this project, we use `esp_http_client`<sup>24</sup> API to make an HTTPS request to download `latest.html` and `esp_https_ota`<sup>25</sup> API to perform the OTA update. Although `esp_http_client` says HTTP on its name, if we configure it with a certificate, it can work as an HTTPS client, too. With the help of `mbedtls`<sup>26</sup>(a C cryptographic library created for embedded systems), both API can verify the certificate chain itself. We only need to pass the CA certificate as a configuration parameter the following way:

```
esp_http_client_config_t config_image = {
    .cert_pem = (char *)ca_cert_file
}
```

Every time something goes wrong, e.g., we cannot reach the server, certificate verification fails, or something is not right with the connection, we wait for a specific amount of time,<sup>27</sup> then try again. Code snippet 4.3 shows this described process. First, we configure our client, and we try to download the version. If it succeeds, and the latest version is higher than the actual one, we download the latest firmware and perform OTA update.

```
esp_http_client_config_t config_image = {
    .event_handler = _http_event_handler,
    .cert_pem = (char *)ca_cert_file
};

esp_http_client_handle_t client_image;
char image_url[IMAGE_URL_SIZE];

while (1){
    config_image.url = LATEST_URL;

    client_image = esp_http_client_init(&config_image);

    if (esp_http_client_perform(client_image) == 0){
        if (latest_version > VERSION){
            snprintf(image_url, IMAGE_URL_SIZE, "%s%d.bin",
IMAGE_URL_BASE, latest_version);
            config_image.url = image_url;
        }
    }
}
```

<sup>24</sup>[https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/protocols/esp\\_http\\_client.html](https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/protocols/esp_http_client.html)

<sup>25</sup>[https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/esp\\_https\\_ota.html](https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/esp_https_ota.html)

<sup>26</sup><https://www.trustedfirmware.org/projects/mbedtls/>

<sup>27</sup>In Code snippet 4.3, it is five seconds, but it can be much longer.

```

        esp_err_t ret_image = esp_https_ota(&config_image);
        if (ret_image == ESP_OK){
            esp_restart();
        }
        else{
            ESP_LOGE(OTA_TAG, "Firmware upgrade failed");
        }
    }
    else ESP_LOGE(OTA_TAG, "No newer version found");
}
vTaskDelay(5000 / portTICK_PERIOD_MS);
}

```

**Code snippet 4.3:** Using ESP HTTPS Client and ESP HTTPS OTA APIs in our OTA application

## 4.6 Testing OTA application

### 4.6.1 Configuration

After we started the HTTPS server and everything is implemented, let's configure, then flash the application on the device and check whether everything works well. As we do this via serial connection, we can watch the serial output. As we enabled secure boot and flash encryption on the device, we can use these functionalities when flashing the application for more security. Figure 4.21 shows the setup for this configuration. We also have to configure our connection to have WiFi connection (Figure 4.22). Next, we build our application, then flash it to the device with encryption enabled. We can perform these steps with executing `idf.py encrypted-flash` command. After successfully built, the application is being flashed encrypted(Figure 4.23). After successful flashing, we can check the running application's console output with command `idf.py monitor`.

```

[*] Enable hardware secure boot in bootloader (READ DOCS FIRST)
    Secure bootloader mode (Reflashable) --->
[*] Sign binaries during build
(mykey.pem) Secure boot private signing key
    Hardware Key Encoding (No encoding (256 bit key)) --->
[ ] Allow potentially insecure options
[*] Enable flash encryption on boot (READ DOCS FIRST)
    Enable usage mode (Development(NOT SECURE)) --->
Potentially insecure options --->

```

**Figure 4.21:** Menuconfig setup for security functions

```

Example Connection Configuration
ects submenu ---> (or empty submenu ----). Highligh
> to exit, <?> for Help, </> for Search. Legend: [*]

Connect using (Wi-Fi) --->
(ssid) WiFi SSID
(pass) WiFi Password
[*] obtain IPv6 link-local address

```

**Figure 4.22:** Menuconfig setup for WiFi connection

```

WARNING: - compress and encrypt options are mutually exclusive
Will flash uncompressed
Wrote 16384 bytes at 0x00010000 in 0.5 seconds (241.6 kbit/s)...
Wrote 16384 bytes at 0x00015000 in 0.4 seconds (359.2 kbit/s)...
Writing at 0x00094000... (62 %)

```

**Figure 4.23:** Uncompressed encrypted flash

## 4.6.2 During run

After booting, the device tries, then successfully connects to the Access Point, and gets an IPv4 and an IPv6 address (Figure 4.24). As the blink sub-application and OTA part runs parallel, they are independent of each other. That is why we see there is a ‘Turning on the LED in version 1000’ line before the connection, as the latter is part of the OTA update only. After the device established connection, it tries to download latest.html first. It reaches the server, verifies its certificate, then receives the latest version number from the latest.html file. Then, the device successfully downloads the latest firmware itself and starts performing OTA update. Figure 4.25 shows that we really have the latest version; it is 1013, then OTA update is performed with 1013.bin. In Figure 4.26, we can see that also OpenSSL’s console shows the server accepted the requests and responded with the two files.

## 4.6.3 The updated application

After OTA successfully saved the new firmware as a binary file to partition at offset 0x120000, there are some checks whether this image is valid and has a valid signature (Figure 4.27). Because everything goes well, the device reboots with the new updated version. Figure 4.28 shows that now version 1013 is running, so the update was successful. When now the application is downloading latest.html, it sees that the latest version is still 1013, so no newer version is found. It can continue its normal run, then check for updates later (Figure 4.29).

```

I (1111) wifi:Init static rx buffer num: 10
I (1121) wifi:Init dynamic rx buffer num: 32
I (1121) example_connect: Connecting to [REDACTED]...
I (1221) phy: phy_version: 4370, 4e803b3, Aug 11 2020, 14:18:07, 0, 0
I (1221) wifi:mode : sta (cc:50:e3:b6:54:c4)
I (1951) wifi:new:<6,0>, old:<1,0>, ap:<255,255>, sta:<6,0>, prof:1
I (2821) wifi:state: init -> auth (b0)
I (2821) wifi:state: auth -> assoc (0)
I (2831) wifi:state: assoc -> run (10)
I (2861) wifi:connected with [REDACTED], aid = 11, channel 6, BW20, bssid = 40:ee:dd:f2:46:e8
I (2861) wifi:security: WPA2-PSK, phy: bgn, rssi: -66
I (2861) wifi:pm start, type: 1

I (2951) wifi:AP's beacon interval = 100352 us, DTIM period = 1
Turning on the LED in version 1000
I (5551) tcpip_adapter: sta ip: 192.168.100.24, mask: 255.255.255.0, gw: 192.168.100.1
I (5551) example_connect: Connected to [REDACTED]
I (5551) example_connect: IPv4 address: 192.168.100.24
I (5561) example_connect: IPv6 address: fe80:0000:0000:0000:ce50:e3ff:feb6:54c4
I (5571) wifi:Set ps type: 0

I (5571) ota_self-update: I'm version 1000

Turning off the LED in version 1000
Turning on the LED in version 1000

```

**Figure 4.24:** The device is successfully connected to WiFi

```

Turning off the LED in version 1000
Turning on the LED in version 1000
Turning off the LED in version 1000
I (11085) ota_self-update: Latest Version is:1013
I (11095) ota_self-update: Downloading image from URL: https://192.168.100.38:8060/1013.bin
Turning on the LED in version 1000
Turning off the LED in version 1000
I (16815) esp_https_ota: Starting OTA...
I (16815) esp_https_ota: Writing to partition subtype 16 at offset 0x120000
Turning on the LED in version 1000
Turning off the LED in version 1000

```

**Figure 4.25:** The device downloads version.html and the latest firmware

```

D:\http_server\update\https_update_server>openssl s_server
-WWW -key ..\update_key.pem -cert ..\update_cert.pem -port
8060
Loading 'screen' into random state - done
Using default temp DH parameters
ACCEPT
bad gethostbyaddr
FILE:latest.html
ACCEPT
bad gethostbyaddr
FILE:1013.bin
ACCEPT

```

**Figure 4.26:** Update server accepts requests, then responds with the files asked

```

Turning off the LED in version 1000
Turning on the LED in version 1000
Turning off the LED in version 1000
Turning on the LED in version 1000
D (34335) HTTP_CLIENT: esp_transport_read returned:-1 and errno:128
I (34345) esp_https_ota: Connection closed
I (34345) boot_comm: chip revision: 1, min. application chip revision: 0
I (34345) esp_image: segment 0: paddr=0x00120020 vaddr=0x3f400020 size=0x1db8c (121740) map
I (34395) esp_image: segment 1: paddr=0x0013dbb4 vaddr=0x3ffb0000 size=0x0245c ( 9308)
I (34405) esp_image: segment 2: paddr=0x00140018 vaddr=0x400d0018 size=0x81f5c (532316) map
0x400d0018: _stext at ???

I (34585) esp_image: segment 3: paddr=0x001c1f7c vaddr=0x3ffb245c size=0x01214 ( 4628)
I (34595) esp_image: segment 4: paddr=0x001c3198 vaddr=0x40080000 size=0x00400 ( 1024)
0x40080000: _WindowOverflow4 at /home/linuxman/esp/esp-idf/components/freertos/xtensa_vectors.S:1778

I (34595) esp_image: segment 5: paddr=0x001c35a0 vaddr=0x40080400 size=0x145d4 ( 83412)
I (34635) esp_image: segment 6: paddr=0x001d7b7c vaddr=0x00000000 size=0x08404 ( 33796)
I (34645) esp_image: Verifying image signature...

```

Figure 4.27: Application checks OTA partition

```

Turning on the LED in version 1013
I (5547) tcpip_adapter: sta ip: 192.168.100.24, mask: 255.255.255.0, gw: 192.168.100.1
I (5547) example_connect: Connected to [REDACTED]
I (5547) example_connect: IPv4 address: 192.168.100.24
I (5557) example_connect: IPv6 address: fe80:0000:0000:0000:ce50:e3ff:feb6:54c4
I (5567) wifi:Set ps type: 0

I (5567) ota_self-update: I'm version 1013
Turning off the LED in version 1013

```

Figure 4.28: Application successfully updated to version 1013

```

Turning on the LED in version 1013
Turning off the LED in version 1013
I (1103517) ota_self-update: Latest Version is:1013
E (1103527) ota_self-update: No newer version found
Turning on the LED in version 1013
Turning off the LED in version 1013

```

Figure 4.29: The actual version on the device is the latest one

#### 4.6.4 Possible errors in OTA application

Although we showed the expected operation, there can be errors which we ran into during our work. If we assume that we implemented everything correctly and a valid CA certificate is really stored in our device, these errors can mainly occur because of connection failure. As discussed in Section 3.1.1, the update server’s certificate is signed by the CA certificate, so CA verification can fail if the update server’s certificate is not signed by CA. In other words, verification fails when the CA certificate on the device is not the one we used for signing the update server’s certificate (Figure 4.30). Also, when the device cannot reach the server, it logs “Failed to connect to host” error (4.31). Furthermore, if the WiFi is disconnected, we get an error seen in Figure 4.32.

```
Turning off the LED in version 1013
E (401457) esp-tls: mbedtls ssl handshake returned -0x2700
I (401457) esp-tls: Failed to verify peer certificate!
I (401457) esp-tls: verification info: ! The certificate is not
correctly signed by the trusted CA

E (401467) esp-tls: Failed to open new connection
E (401477) TRANS_SSL: Failed to open a new connection
E (401477) HTTP_CLIENT: Connection failed, sock < 0
Turning on the LED in version 1013
```

Figure 4.30: Certificate signature error

```
Turning on the LED in version 1013
E (520277) esp-tls: Failed to connect to host (errno 113)
E (520277) esp-tls: Failed to open new connection
E (520277) TRANS_SSL: Failed to open a new connection
E (520277) HTTP_CLIENT: Connection failed, sock < 0
Turning off the LED in version 1013
```

Figure 4.31: Failed to connect to host error

```
I (366746) example_connect: Wi-Fi disconnected, trying to reconnect...
Turning on the LED in version 1000
I (368796) example_connect: Wi-Fi disconnected, trying to reconnect...
I (370846) example_connect: Wi-Fi disconnected, trying to reconnect...
Turning off the LED in version 1000
```

Figure 4.32: Cannot connect to WiFi error

# Chapter 5

## Security analysis

### 5.1 Secure boot

#### 5.1.1 Basic functioning

By enabling secure boot, we successfully achieved that only our code can run on the device, as only we have the signing key. In Section 4.3.4, we showed that an incorrectly signed application makes the device reboot again and again.

#### 5.1.2 Security

The security of secure boot on ESP32 depends on the strength of the keys and on whether we keep these keys in secret. As in Reflashable mode,<sup>1</sup> AES256 key derives from the ECDSA key, if an attacker gains access to this ECDSA key, it can use that to generate a new bootloader or flash its own code on the device. Moreover, a section of the documentation draws attention to the fact that without flash encryption, the device is vulnerable to ‘time-of-check to time-of-use’ attacks.<sup>2</sup>

### 5.2 Flash encryption

#### 5.2.1 Basic functioning

By enabling flash encryption, we successfully achieved that the flash’s physical readout is not enough to recover most of the flash contents. Without having the AES256 key stored in eFuse, decrypting is not possible.

---

<sup>1</sup>Also, with One-time flash mode, the ECDSA key has to be permanent, as it is used in the permanent bootloader for verifying.

<sup>2</sup><https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/secure-boot-v1.html#secure-boot-and-flash-encr>



## 5.2.2 Security

The security of flash encryption depends on the strength of the only key, the AES256 key. As this key is generated by a hardware RNG and stored in the R/W protected eFuse, it is theoretically impossible for an attacker to access this key.

## 5.3 OTA self-update

### 5.3.1 Basic functioning

If we check the application requirements discussed in Section 3.1, we see that every requirement is met. The device can connect to the update server via a secure connection, and it can verify the certificate chain created. Also, the device can compare versions and download new firmware only if it is newer than the actual one.

### 5.3.2 Security

As for the security of our application, we can mention two parts. First, the security of the flashed app itself, and second, the OTA process's security. As we enabled secure boot and flash encryption, and our application is signed, it has all the attributes coming with these functionalities. As for the second consideration, as we accept new firmware images only from the update server verified by the CA certificate stored on the device, we have the same security as any other HTTPS client. The responsibility is at the server. It is essential that the private key we used for starting the server is cryptographically strong and kept in secret.

# Chapter 6

## Related Work

Looking up for available implementations and concepts related to our work, we can find plenty of them. Here we mention some briefly.

In an article published in *Advances in Electrical and Electronic Engineering* journal [7], there is an implementation for secure remote firmware update on a UHF RFID<sup>1</sup> reader. This article concentrates not only on the update mechanism itself but also the cryptography used by the device. Similar to our work, the authors implement secure boot and enable image file encryption on the device. However, as the device originally had no security features built-in, the authors had to implement them. In contrast with the opportunities of ESP32, which uses hardware secure boot, this implementation uses software-based secure boot.

An ICC Workshops paper [6] discusses IoT's major challenges with having security considerations in prime focus. The paper mentions all the three processes we implemented, secure boot, encryption methods, and secure remote firmware update.

The Chromium Projects<sup>2</sup> shows a solution for an auto-update system that is used for their own Chromium OS file system. Similar to our implementation, this process does not require any user interaction; the update is performed automatically with maximum security in mind. However, Chromium's auto-update process also has some specific goals, such as speed and small update size.

Work with the fantasy name ASSURED [2] designs a secure and scalable framework for large-scale IoT while providing end-to-end security between manufacturers and devices. As proof of concept, ASSURED implements its concepts for two architectures: HYDRA[3] and ARM TrustZone-M.<sup>3</sup>

---

<sup>1</sup>Ultra-High Frequency Radio Frequency Identification system.

<sup>2</sup><https://www.chromium.org/chromium-os/chromiumos-design-docs/filesystem-autoupdate>

<sup>3</sup>[https://static.docs.arm.com/100690\\_0101/00/armv8\\_m\\_architecture\\_trustzone\\_technology\\_100690\\_0101\\_00\\_en.pdf](https://static.docs.arm.com/100690_0101/00/armv8_m_architecture_trustzone_technology_100690_0101_00_en.pdf)

Shade [9] writes about many considerations when designing secure firmware update methods in general. He discusses the architecture, requirements, and authentication and validation logic for a secure remote firmware update.

## Chapter 7

# Conclusion

In this thesis, we presented solutions to make a specific IoT device (an ESP32) secure. Although the device has hardware security support, in the beginning, security functionalities were not enabled. Thus, anyone could flash their code on it, read out its content also if it contained sensitive data, and there was no working method for securely updating the device remotely. After designing and implementing security mechanisms and a remote update process, we achieved that

- only the owner's code can run on the device because of the enabled secure boot process,
- with physical readout of the flash, the attacker cannot obtain real information about the flash's content because of enabled flash encryption, and
- the device can periodically check for firmware updates via secure TLS connection and update itself whenever a new version is found because of our secure remote firmware update implementation.

Although the main concepts behind these three implemented functions are not difficult to understand, the security of IoT devices can be drastically increased with the usage of them. Obviously, the benefits of these security mechanisms depend on the actual project, but it is definitely useful to have them in a production environment or case of critical applications. Many cyberattacks could be avoided this way.

# Acknowledgments

I would like to thank my thesis supervisors, Dr. Levente Buttyán and Dorottya Futóné Papp. Levente's professionalism and Dorottya's precision truly helped me to get the most out of this project and learn a lot.

The presented work was carried out within the SETIT Project (2018-1.2.1-NKP-2018-00004), which has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.

# List of Figures

2.1	Stages in the chain of trust during the secure boot process . . . . .	4
3.1	Certificate chain . . . . .	7
3.2	Relationship between the server and the device . . . . .	7
3.3	High-level sequence diagram of the update process . . . . .	9
4.1	Structure of eFuse . . . . .	11
4.2	ESP-IDF's menuconfig . . . . .	12
4.3	Code editor in VSCode . . . . .	14
4.4	Some ESP-IDF-specific functions in VS Code . . . . .	14
4.5	Configuration for enabling secure boot in Reflashable mode . . . . .	16
4.6	Content of eFuse after burning secure boot key . . . . .	17
4.7	The application is being signed and flashed to the device . . . . .	17
4.8	We can see through monitoring our application that secure boot is enabled	17
4.9	Working basic Hello World application with Secure Boot . . . . .	17
4.10	Error when the application is not correctly signed . . . . .	18
4.11	Configuration for enabling flash encryption in Development mode . . . . .	20
4.12	Appropriate values are written in eFuse automatically . . . . .	20
4.13	The device reboots with enabled flash encryption . . . . .	20
4.14	Flash encryption mode is enabled in Development mode . . . . .	21
4.15	Encrypted application successfully starts . . . . .	21
4.16	Unencrypted flash . . . . .	21
4.17	Encrypted flash . . . . .	22
4.18	Sequence diagram of Simple OTA update example . . . . .	23
4.19	Verifying update server's certificate with CA certificate with OpenSSL . . . .	24

4.20	OpenSSL server successfully started on port 8060 . . . . .	25
4.21	Menuconfig setup for security functions . . . . .	28
4.22	Menuconfig setup for WiFi connection . . . . .	29
4.23	Uncompressed encrypted flash . . . . .	29
4.24	The device is successfully connected to WiFi . . . . .	30
4.25	The device downloads version.html and the latest firmware . . . . .	30
4.26	Update server accepts requests, then responses with the files asked . . . . .	30
4.27	Application checks OTA partition . . . . .	31
4.28	Application successfully updated to version 1013 . . . . .	31
4.29	The actual version on the device is the latest one . . . . .	31
4.30	Certificate signature error . . . . .	32
4.31	Failed to connect to host error . . . . .	32
4.32	Cannot connect to WiFi error . . . . .	32

# Bibliography

- [1] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th {USENIX} security symposium ({USENIX} Security 17)*, pages 1093–1110, 2017.
- [2] N Asokan, Thomas Nyman, Norrathep Rattanaivanon, Ahmad-Reza Sadeghi, and Gene Tsudik. Assured: Architecture for secure software update of realistic embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2290–2300, 2018.
- [3] Karim Eldefrawy, Norrathep Rattanaivanon, and Gene Tsudik. Hydra: hybrid design for remote attestation (using a formally verified microkernel). In *Proceedings of the 10th ACM Conference on Security and Privacy in wireless and Mobile Networks*, pages 99–110, 2017.
- [4] Patrick Gallagher and Acting Director. Secure hash standard (shs). *FIPS PUB*, 180: 183, 1995.
- [5] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.
- [6] Bilal Javed, Mian Waseem Iqbal, and Haider Abbas. Internet of things (iot) design considerations for developers and manufacturers. In *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 834–839. IEEE, 2017.
- [7] Lukas Kvarda, Pavel Hnyk, Lukas Vojtech, Zdenek Lokaj, Marek Neruda, and Tomas Zitta. Software implementation of a secure firmware update solution in an iot context. *Advances in Electrical and Electronic Engineering*, 14(4):389–396, 2016.
- [8] Ronald L Rivest, Adi Shamir, and Leonard M Adleman. Cryptographic communications system and method, September 20 1983. US Patent 4,405,829.
- [9] Loren K Shade. Implementing secure remote firmware updates. In *Embedded Systems Conference*, 2011.



- [10] NIST-FIPS Standard. Announcing the advanced encryption standard (aes). *Federal Information Processing Standards Publication*, 197(1-51):3–3, 2001.
- [11] Vincent Zimmer and Michael Krau. Establishing the root of trust. [https://uefi.org/sites/default/files/resources/UEFI%20RoT%20white%20paper\\_Final%208%208%2016%20\(003\).pdf](https://uefi.org/sites/default/files/resources/UEFI%20RoT%20white%20paper_Final%208%208%2016%20(003).pdf), 2016.

# Appendix

## The full application for secure firmware update

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_system.h"
#include "esp_event.h"
#include "esp_event_loop.h"
#include "esp_log.h"
#include "esp_ota_ops.h"
#include "esp_http_client.h"
#include "esp_https_ota.h"
#include "protocol_examples_common.h"
#include "string.h"
#include "esp_tls.h"
#include "mbedtls/pk.h"
#include "nvs.h"
#include "nvs_flash.h"
#include "driver/gpio.h"
#include "tcpip_adapter.h"
#include <esp_http_server.h>
#include <esp_wifi.h>
#include <sys/param.h>
#include <stdio.h>
#include <stdbool.h>
#include <ctype.h>

#define VERSION 1000
#define VERSION_STR "1000"

#define BLINK_GPIO 2

#define OTA_TAG "ota_self-update"
#define HTTP_EVENT_TAG "http-event"

#define IMAGE_URL_BASE "https://192.168.100.38:8060/"
#define LATEST_URL "https://192.168.100.38:8060/latest.html"

#define VERSION_LENGTH 4
#define IMAGE_URL_SIZE 50
```

```

extern const uint8_t ca_cert_file[] asm("_binary_ca_cert_pem_start");
int latest_version = VERSION;
char latest_version_str[VERSION_LENGTH+1] = VERSION_STR;
bool version_check = false;

void set_latest_version(esp_http_client_event_t *evt)
{
    if (evt->data_len != VERSION_LENGTH)
    {
        ESP_LOGE(OTA_TAG, "latest.html's content should be a %d-digit number",
VERSION_LENGTH);
        return;
    }

    char *data = (char *)evt->data;

    for (int i = 0; i < VERSION_LENGTH; ++i)
    {
        if (!isdigit(data[i]))
        {
            ESP_LOGE(OTA_TAG, "latest.html contains character that is not a digit");
            return;
        }
    }
    strncpy(latest_version_str, data, VERSION_LENGTH);
    latest_version = atoi(latest_version_str);

    ESP_LOGI(OTA_TAG, "Latest Version is:%s", latest_version_str);
}

esp_err_t _http_event_handler(esp_http_client_event_t *evt)
{
    switch (evt->event_id)
    {
        case HTTP_EVENT_ERROR:
            ESP_LOGD(HTTP_EVENT_TAG, "HTTP_EVENT_ERROR");
            break;
        case HTTP_EVENT_ON_CONNECTED:
            ESP_LOGD(HTTP_EVENT_TAG, "HTTP_EVENT_ON_CONNECTED");
            break;
        case HTTP_EVENT_HEADER_SENT:
            ESP_LOGD(HTTP_EVENT_TAG, "HTTP_EVENT_HEADER_SENT");
            break;
        case HTTP_EVENT_ON_HEADER:
            ESP_LOGD(HTTP_EVENT_TAG, "HTTP_EVENT_ON_HEADER, key=%s, value=%s", evt->
header_key, evt->header_value);
            break;
        case HTTP_EVENT_ON_DATA:
            if (version_check)
            {

```

```

        set_latest_version(evt);
    }
    break;
case HTTP_EVENT_ON_FINISH:
    ESP_LOGD(HTTP_EVENT_TAG, "HTTP_EVENT_ON_FINISH");
    break;
case HTTP_EVENT_DISCONNECTED:
    ESP_LOGD(HTTP_EVENT_TAG, "HTTP_EVENT_DISCONNECTED");
    break;
}
return ESP_OK;
}

void self_ota_task(void *pvParameter)
{
    ESP_LOGI(OTA_TAG, "I'm version %d\n", VERSION);

    esp_http_client_config_t config_image = {
        .event_handler = _http_event_handler,
        .cert_pem = (char *)ca_cert_file,
        .transport_type = HTTP_TRANSPORT_OVER_SSL};

    esp_http_client_handle_t client_image;
    char image_url[IMAGE_URL_SIZE];
    esp_err_t ret_image = -1;

    while (1)
    {
        version_check = true;
        config_image.url = LATEST_URL;

        client_image = esp_http_client_init(&config_image);
        ret_image = esp_http_client_perform(client_image);
        version_check = false;

        if (ret_image == 0)
        {
            if (latest_version > VERSION)
            {
                snprintf(image_url, IMAGE_URL_SIZE, "%s%d.bin",
IMAGE_URL_BASE, latest_version);
                ESP_LOGI(OTA_TAG, "Downloading image from URL: %s", image_url);
                config_image.url = image_url;

                ret_image = esp_https_ota(&config_image);
                printf("\nHTTPS OTA RETURNS WITH %d", ret_image);
                if (ret_image == ESP_OK)
                {
                    esp_restart();
                }
            }
            else

```

```

        {
            ESP_LOGE(OTA_TAG, "Firmware upgrade failed");
        }
    }
    else
    {
        ESP_LOGE(OTA_TAG, "No newer version found");
    }
}

vTaskDelay(5000 / portTICK_PERIOD_MS);
}
}

void blink_task(void *pvParameter)
{
    gpio_pad_select_gpio(BLINK_GPIO);
    gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);
    while (1)
    {
        printf("Turning off the LED in version %d\n", VERSION);
        gpio_set_level(BLINK_GPIO, 0);
        vTaskDelay(2500 / portTICK_PERIOD_MS);
        printf("Turning on the LED in version %d\n", VERSION);
        gpio_set_level(BLINK_GPIO, 1);
        vTaskDelay(2500 / portTICK_PERIOD_MS);
    }
}

void app_main()
{
    esp_err_t err = nvs_flash_init();
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err ==
    ESP_ERR_NVS_NEW_VERSION_FOUND)
    {
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK(err);

    xTaskCreate(&blink_task, "blink_task", 2048, NULL, 1, NULL);

    tcpip_adapter_init();
    ESP_ERROR_CHECK(esp_event_loop_create_default());

    ESP_ERROR_CHECK(example_connect());

#ifdef CONFIG_EXAMPLE_CONNECT_WIFI
    esp_wifi_set_ps(WIFI_PS_NONE);
#endif
}

```

```
xTaskCreate(&self_ota_task, "self_ota_task", 8192, NULL, 2, NULL);  
}
```