FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Secure Over the Air Update of ESP32
Student:	Marek Kočí
Supervisor:	Ing. Jiří Dostál, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2021/2022

Instructions

Based on the previous thesis 'Security of IoT Devices Based on ESP32,' analyze threats related to the Firmware Over the Air (OTA) update of IoT devices.
Describe related ESP32 platform security features. Analyze remote server authentication options and the client authenticity verification. Analyze the possibility to upgrade the authentication process in the future. Create an ESP32 application that will securely download and apply a new firmware from a remote server. The update process will verify the authenticity of both the server and firmware and the integrity of the firmware. Test and evaluate the result in terms of cybersecurity.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 30 January 2021 in Prague.

Bachelor thesis

SECURE OVER THE AIR UPDATE OF ESP32

Marek Kočí

Faculty of Information Technology CTU in Prague
Department of Information Security
Supervisor: Ing. Jiří Dostál, Ph.D.
May 11, 2022

Contents

List of Figures

List of Tables

List of code listings

I would like to thank my student colleagues for support, discussions and help during my studies, my wife for endless patience and understanding, my colleagues and manager for encouragement and support, and my supervisor for the flexibility, advisory and his empathy.

Declaration

Abstract

This bachelor thesis deals with the cybersecurity concerns and vulnerabilities related to the IoT device Over the Air firmware update.

The analytical part of the thesis describes the firmware update mechanism, the related cybersecurity technologies and the ESP32 platform and its hardware and software cybersecurity features such as Secure Boot, Anti-rollback, or Flash Encryption.

The thesis describes two versions of the ESP32 firmware. A basic firmware version without any cybersecurity measures and a firmware version which includes verification of the firmware integrity, the firmware image encryption and the secure firmware image transfer using the Transport Layer Security (TLS).

The performed tests confirmed that the listed cybersecurity features are supported by the ESP32 platform. It also confirmed that the TLS 1.2 is fully supported and found that the version 1.3 has only limited support and cannot be used.

The evaluation of the security measures discussed in this thesis helps the IoT community to choose the proper solutions for the development of the OTA firmware for IoT devices.

The thesis is complemented by the firmware source code implementing the mitigation described in it.

Keywords ESP32, over the air, firmware update, IoT security, server authentication, firmware integrity, IoT device

Abstrakt

Obsahem této bakalářské práce je analýza bezpečnostních rizik a zranitelností souvisejících s Firmware Over the Air aktualizací IoT zařízení.

Analytická část práce popisuje princip aktualizace firmware, související bezpečnostní technologie a platformu ESP32 včetně jejich bezpečnostních prvků jako Secure Boot, Anti-rollback a šifrování paměti flash, pro které platforma poskytuje hardware a software podporu.

Práce popisuje dvě verze firmware pro platformu ESP32. Základní firmware, který neobsahuje žádná bezpečnostní opatření, a dále verzi, která zahrnuje ověření integrity firmware, jeho zašifrování a zabezpečený přenos nového firmware ze serveru za využití protokolu Transport Layer Security (TLS).

Provedené testy potvrdily, že navrhovaná bezpečnostní opatření mají na platformě ESP32 podporu. Dále potvrdily, že protokol TLS verze 1.2 je plně podporován, zatímco verze 1.3 je ve fázi implementace a není prozatím vhodná k použití v praxi.

Vyhodnocení bezpečnostních opatření diskutovaných v této práci pomůže IoT komunitě zvolit správná řešení při návrhu a vývoji nových IoT zařízení.

Přílohu práce tvoří zdrojové kódy, v kterých jsou implementovány diskutovaná opatření.

Klíčová slova ESP32, over the air, aktualizace firmware, IoT bezpečnost, autentizace serveru, integrita firmware, IoT zařízení

Introduction

In the early stage of IoT (Internet of Things) devices' firmware development the cybersecurity concerns were usually not considered a very high priority. Nowadays with millions of IoT devices surrounding us and being a part of our daily lives the cybersecurity has become a crucial requirement for the firmware development. It protects us from turning each of those devices into an open door for potential attackers. It helps with a prevention of data breach, attackers gaining access to our private data or attackers gaining control of our devices and computers. Ensuring the secure firmware update has become one of the key aspects for the IoT devices' development.

The thesis is focused on the two main topics: to define and summarize general suggestions for the secure firmware over the air update principles, which will be beneficial for a wide community and for the firmware development of the IoT devices. Secondly the thesis will provide an example of an implementation of those principles on the ESP32 platform and therefore will become the source of inspiration for the community leveraging the ESP32 platform.

The choice of the topic for the bachelor thesis was inspired by the importance of the IoT and related issues in general and their importance in everyone's life. Furthermore the author has passion for embedded devices, which is his original background and experience. Most importantly the topic can be beneficial for both commercial and hobby IoT application developers building IoT devices.

The thesis provides an introduction to the cybersecurity topics, describes the firmware over the air update principle and provides an overview of the ESP32 and its security features. The practical part of the thesis explains a proof of concept firmware application.

This bachelor thesis is a follow up of the diploma thesis *Security of IoT Devices Based on ESP32* authored at Faculty of Informatics by Bc. Michal Vácha in February 13, 2020, which describes the general cybersecurity concerns of IoT devices and how they are addressed on the ESP32 platform. The thesis is focused on the over the air firmware update part of the previous thesis and related cybersecurity concerns. The thesis proposes mitigation of those concerns independently of the platform as well as addressing those concerns on the ESP32 platform.

Goal

The first goal of the theoretical part is to describe mechanism of the IoT (Internet of Things) device Firmware Over The Air (FOTA) update. The second goal is to describe terms and technologies used for the FOTA update of IoT devices. The third goal is to analyse threats related to the FOTA update of IoT devices. The fourth goal is to analyse remote server authentication options and the IoT device authenticity verification. The fourth goal is to analyse the possibility to upgrade the authentication process in the future. The last goal is to describe security features of the ESP32 platform.

The main goal of the practical part is to create an ESP32 application that will securely download and apply a new firmware from a remote server. Related goal is to verify the authenticity of both the server and the firmware as well as the integrity of the firmware. The last goal is to test and evaluate the result in terms of the cybersecurity.

Acronyms

2FA	Two-Factor Authentication
3-DES	Tripple Data Encryption Standard
AES	Advanced Encryption Standard
AIoT	Artificial Intelligence of Things
ALPN Negotiation	Application-Layer Protocol Negotiation (extension of TLS)
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
Bluetooth (LE)	Bluetooth Low Energy
CA	Certification Authority
CISA	Cybersecurity and Infrastructure Security Agency
CLI	Command Line Interface
CN	Common Name
CoAP	Constrained Application Protocol
CRC	Cyclic Redundancy Check
CSR	Certificate Signing Request
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
DDS	Data Distribution Service
ECDSA	Elliptic Curve Digital Signature Algorithm
ESP	Espressif
ESP-IDF	Espressif Integrated Development Framework
FOTA	Firmware Over The Air
FQDN	Fully Qualified Domain Name
FTP	File Transfer Protocol
GCM	Galois/Counter Mode
GPIO	General Purpose Input Output
HMAC	Hash-based Message Authentication Code
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IIoT	Industrial Internet of Things
IoMT	Internet of Military Things
IoT	Internet of Things
IP	Internet Protocol
IPEX	miniature surface-mount coaxial connectors
M2M	Machine to Machine
MCU	Microcontroller Unit
MD5	Message Digest
MQTT	Message Queuing Telemetry Transport
NIST	National Institute of Standards and Technology
NVD	National Vulnerability Database
NVS	Non-Volatile Storage
OTA	Over The Air
PCB	Printed Circuit Board
PKI	Public Key Infrastructure

PSRAM	Psuedostatic Random Access Memory
RF Balun	Radio Frequency Balun (transformer between balanced and unbalanced)
RFC	Request for Comments
RISC	Reduced Instruction Set Computer
RNG	Random Number Generator
RoT	Root of Trust
RSA	Cipher developed by Rivest, Shamir, Adleman
RSA-PSS	RSA Probabilistic Signature Scheme
SHA	Secure Hash Algorithm
SNI	Server Name Indication
SNOW	Word-based Synchronous Stream Ciphers
SoC	System on Chip
SPIRAM	Serial Peripheral Interface Random Access Memory
SRAM	Static Random Access Memory
SSL	Secure Sockets Layer
TEE	Trusted Environment Execution
TLS	Transport Layer Security
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus

Chapter 1

Internet of Things and Firmware Update

This chapter describes the basic information and principles of the Internet of Things (IoT). It gives example of IoT devices and explains what is firmware update and its importance.

1.1 Definition of IoT

Alexander S. Gillis in his article "*What is the internet of things (IoT)?*" defines the Internet of Things as a "*system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction*". [1]

1.2 How Does IoT Work?

The IoT solutions typically consist of multiple devices collecting data. These devices are often directly accessible locally for a purpose of installation, configuration, firmware updates or to read a status and measures of the device. The devices can integrate into a bigger infrastructure with a cloud[1] solutions, which provide users comprehensive overview of all the data received from a single device or multiple devices. The cloud application can also provide more complex analysis of the data. It can compare or merge the data from multiple devices. [1]

In some applications it is convenient to use a gateway[2], which can pre-process data from multiple devices and send them merged or consolidated to the cloud. This solution can be beneficial to reduce the amount of the data sent to the cloud or to connect the gateway to the devices locally using a low-consumption technology, which prolongs the life-time of the battery in the device and reduces the manufacturing costs of the IoT devices.

An example of how IoT system works is shown in the figure 1.1.

1.3 Examples of IoT Applications

There are many types of IoT applications based on their usage. These are some of the most common ones:

[1]Cloud or Cloud Computing can host services such as storage, email or document and data processing. [2]

[2]A device used to connect two different networks, especially a connection to the internet.

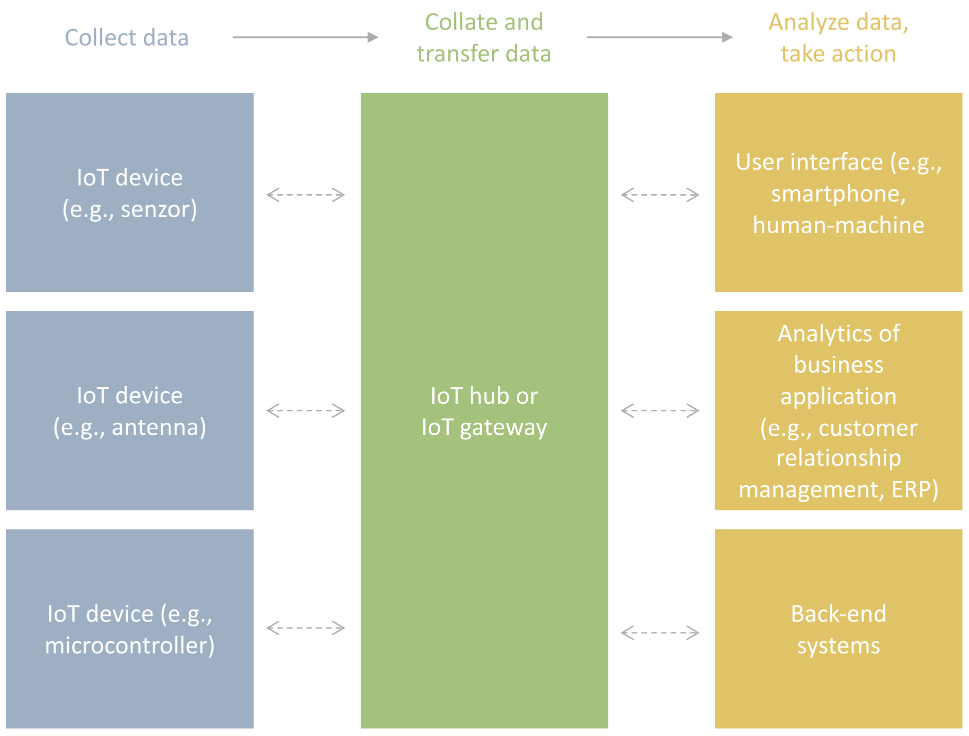

Collect data
Collate and transfer data
Analyze data, take action
IoT device (e.g., senzor)
IoT device (e.g., antenna)
IoT device (e.g., microcontroller)
IoT hub or IoT gateway
User interface (e.g., smartphone, human-machine
Analytics of business application (e.g., customer relationship management, ERP)
Back-end systems

Figure 1.1 An example of how an IoT system works from collecting data to taking action [1]

Consumer IoT for our everyday use (home appliances, voice assistance, and light fixtures),

Commercial IoT used in the healthcare and transport industries (smart pacemakers and monitoring systems),

Military Things (IoMT) used for the application of IoT technologies in the military field (surveillance robots and human-wearable biometrics for combat),

Industrial Internet of Things (IIoT) used with industrial applications, such as in the manufacturing and energy sectors (Digital control systems, smart agriculture and industrial big data),

Infrastructure IoT used for connectivity in smart cities (infrastructure sensors and management systems). [3]

1.4 Connecting IoT Devices

The IoT devices can be connected via several different physical protocols. Although ethernet connection via cable is in many cases possible, most of the today IoT devices are connected via one of the wireless technologies (especially when the device is portable). To transfer the data on the physical layer, there are multiple standard data protocols defined for the link/application layer. [4]

1.4.1 IoT Data Protocols

IoT data protocols are used to connect low-power IoT devices. The connectivity in the IoT data protocols and standards is done through a wired or cellular network. Some examples of the IoT data protocols are:

- MQTT (Message Queuing Telemetry Transport),
- CoAP (Constrained Application Protocol),
- AMQP (Advanced Message Queuing Protocol),
- DDS (Data Distribution Service),
- HTTP (HyperText Transfer Protocol),
- WebSocket. [4]

1.4.2 IoT Network Protocols

IoT network protocols are used to connect devices over a network, typically used over the internet. Here are some examples of various IoT network protocols:

- Wifi,
- Bluetooth,
- ZigBee,
- Z-Wave,
- LoRaWan. [4]

1.5 Over the Air Firmware Update

This sections describes what is the firmware update in general and introduces some of the best practices used to upgrade the firmware. It also describes the basic Over The Air update performed locally on one device and variants of the centralized firmware update, when the user wants to update multiple devices at the same time without the need to connect directly to each of the devices.

1.5.1 Device Firmware

The following text describes the definition of the firmware and information which is common to all the variants of the firmware update.

1.5.1.1 Firmware Definition

Firmware is a type of software. Compared to any other software people use in their computers (games, internet browser, email client, calculator, . . .) the firmware runs directly on a dedicated piece of hardware (typically a small device with a simple and specific purpose). [5]

The firmware is usually linked to the *embedded system* or *embedded device* terms. ”*An embedded system is a stand alone, intelligent system dedicated to running a set of tasks from the moment it is powered on.*” These devices are usually connected to a variety of sensors to process various inputs and to a multiple outputs to provide a control or show a status. [6]

A washing machine is one of many examples where the embedded device and the firmware are used. The firmware collects various inputs (water level, desired washing program, length of washing or speed of drying), it constantly processes all the mentioned inputs and based on the values received it controls the outputs (displays remaining time, tells the motor how fast and when to rotate, locks the door of the machine, and many more). [6]

Embedded devices surround us everywhere and they have become inseparable part of everyone's life. Examples of the embedded devices are: keyboard, wireless printer, smart power socket, smart bulb, remote control, smart TV, elevator control, wrist watch, auto-adjusting lights in a car, bike speedometer and many others.

1.5.1.2 Firmware Life-cycle

Firmware is always uploaded into a device during the manufacturing process. The firmware which is installed on a device should be updated by a newer version of the firmware regularly depending on a use case (and therefore complexity of the firmware). In some cases the updates are necessary every few weeks, in other cases only once a year or never. The life-cycle of the device can be long. After the device is manufactured it can be stored for several months in a stock and at the time when it is being commissioned by the customer, there are very likely newer versions of the firmware already available and the firmware needs to be updated. [7]

1.5.1.3 Firmware Update Need

There are three main triggers for a new version of the firmware to be developed and updated to a device:

- addition of a new features (requested by the customers, suggested by the manufacturer or enforced by a new standards),
- fixing a cybersecurity vulnerability[3],
- fixing a bug[4] found after the firmware has been released to the public. [7]

1.5.1.4 Firmware Structure

Performing the firmware update should not require a trained person and it needs to be done quickly and reliably. At every step of the update process there must be a safety mechanisms to allow a recovery with no or minimal intervention from the end user.

The firmware to be launched on the device is loaded from a flash[5] into the a Random Access Memory (RAM)[6] during the start up of the device. The program is then executed from the RAM and therefore the new firmware can be written into the flash instead of the original one. This would be a very risky step in case that the new firmware contains an issue or the device is restarted during the update process and it is very probable that the device would not be able to recover from this state. [8]

The solution to prevent this situation is keeping at least two versions of the firmware in the device – the active one and the one which is getting uploaded. Most of the embedded devices contain a bootloader, which is typically a small read-only program located in the flash, which is executed as first after the device start up. Bootloader decides which of the two firmware images in the flash should be launched as illustrated in the figure 1.2. After the firmware update is completed, the update mechanism will switch the configuration flag so that during the next startup the bootloader runs the new version of the firmware. [8]

[3]A flaw in a system related to the internet and network security.
[4]Malfunction of the firmware or its unexpected behavior.
[5]Non-volitile permanent storage.
[6]Volatile memory, which contains random date in case of device reboot.

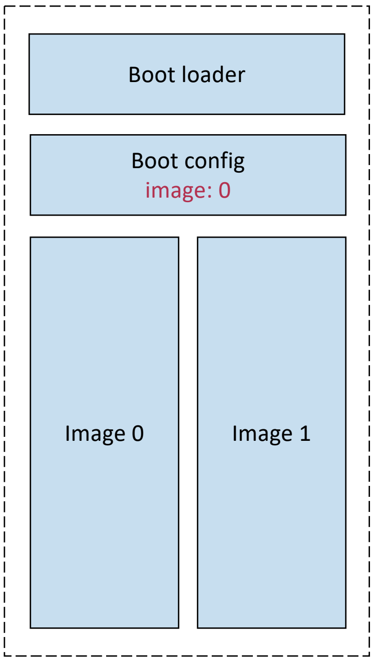


Figure 1.2 Firmware structure (bootloader and firmware images) [8].

Depending on the size of the firmware images and the size of the memory on the device, more then two partitions can be used. There is also an option to use a factory default firmware, which is never overwritten by the update. The usage of this partition can be used either when by a user decides to restore the factory default settings and behavior or as an automatic roll-back option in case that the firmware update fails and there are no other valid partitions present.

1.5.2 Local Firmware Update

Based on the author's experience, there are multiple ways how to update the device firmware. Some of the options are used mainly during the development phase of the firmware – prior the release is provided to the users. The other means are common for the development phase and can be also used by end users. And lastly there are means which are mainly used by the users. Some of the local firmware options are:

JTAG is a programming and debugging device connected via USB to the programmers computer. It is a mean which is only used by the developers while working actively on the new firmware development. It is never used by the end users. The JTAG connects directly to the pins of the microcontroller. [9]

Serial Line is previously RS232 a nowadays USB cable connection, which is used for the upload of the firmware to the device mainly by the developers. In the past this was one of the most common way to perform the firmware update also by the users. The same serial connection is typically used also for the logging of the device messages (such as errors, warnings or informative) or even to configure the device.

USB Stick or SD Card has benefit, that user when performing the update, doesn't need to have computer connected to the device. There is also no need to connect any special devices or tools or install any related software on the computer to perform the update. Files downloaded from the internet are copied to the USB/SD card, which is then inserted to the device and the device will perform the update typically while it's restarted.

Both the serial connection and USB/SD card options can be used as a recovery solution performed by a user in case that a remote update described in the next sections fails and the device becomes inaccessible remotely.

1.5.3 Remote Firmware Update

In this section, we assume that the device is connected to a network and accessible from a computer via the HTTP, SSH or FTP protocol.

Comparing to the section 1.5.2 the mechanism is very similar. The main difference is that the firmware file is transferred to the device via the network connection using one of the following protocols:

HTTP request will make the device receive the firmware file and the device will then perform the needed actions to parse and apply the new firmware and will typically restart the device automatically,

File Transfer such as via FTP protocol or similar can be used to send a new firmware file to the device and the update mechanism can be launched by other action (automatic or for example by user connecting to the device via the HTTP or SSH and running a script to start the update itself).

The firmware can be either fully updated or only partially updated. In the IoT environment, partial updates should be possible in particular for the IoT devices that do not have enough capabilities. This option would decrease the bandwidth consumption and on-device processing time since less code is downloaded and processed. It would also reduce the total update time and make it easier to update even an isolated and rarely awake devices when the update is properly scheduled. [10]

1.5.4 Centralized Firmware Update

The firmware update can be performed remotely without a user directly connecting to the IoT device (Push Mode). The update can also be triggered by the device and download the firmware from the mentioned centralized storage (Pull Mode). Some implementations combine the two modes to have more flexibility for updating different types of IoT devices. [10]

Pull Mode "*is practical for capable devices, such as gateways, that can manage other devices and has the support of a wide range of communication protocol stacks. Moreover, they are most likely always powered and directly connected to the Internet. It is a client-initiated mode, where the client queries the update server for new updates periodically, and when an update is available the client downloads, verifies and then installs the new firmware. In this case, the remote firmware server sends either a URI of the firmware image repository via a data structure or directly the new firmware image binary.*" [10]

Push Mode "*is mainly applicable for resource-constrained IoT devices with limited protocol stack support. It can also be seen as a server-initiated mode, where the update server pushes the updates to the client when there is a new patch available. The same procedure of downloading, verifying and installing the new image is then launched.*" [10]

There are multiple options to deploy centralized firmware update. Either via in house created solution with the servers deployed in the company infrastructure or in a public cloud. An alternative is to use cloud solutions offered for example by Microsoft, Amazon or Google providers.

Table 1.1 Pros and cons of the OTA modes [10]

Mode	Strength	Weaknesses
Push	• Efficient for small-sized updates. • Allows automatic scheduling for pushing updates when there is a zero-day for instance or a new version. • Several IoT devices (e.g. from the same brand and version) could be updated in a controlled manner.	• IoT devices must be registered into the update server and must be uniquely identified. • For security concerns, a secure channel must be established, which is not simple for constrained IoT devices. • The scheduling of the updates is not also easy to keep the availability of all the devices and additionally real-time can be an issue for the last-scheduled devices. • The firmware server needs to securely store and remember all the IoT devices. • Spike of update transmission and load in the firmware server, which is caused by in case of a tight scheduling of FoTA update for each IoT device (or with the corresponding gateways) with the corresponding secure channels.
Pull	• A good alternative when a gateway is deployed. • Each gateway can schedule the locally managed network.	• Pull is not done in real-time, which can be an issue for urgent updates. • A significant scale of pull may apply a significant resource consumption on the firmware server side.

Chapter 2

Cybersecurity and FOTA

This chapter describes the known threats related to the IoT devices and FOTA. It also explains the security mechanisms, that can be used to mitigate the threats. Some the mechanisms are described in the following sections.

2.1 Cybersecurity Principles and Terms

This section describes the cybersecurity terms and common practices used in the computer and IoT security.

2.1.1 Encryption Algorithms

Encryption is a way of scrambling data so that only authorized parties can understand the information. Encryption takes readable data and alters it so that it appears random. It requires use of a cryptographic key or key-pair which the sender and the recipient of an encrypted message use to encrypt and decrypt the text, while anyone else is unable to decrypt the content. [11]

The two main kinds of encryption are symmetric and asymmetric encryption:

Symmetric encryption only uses one key and all communicating parties use the same (secret) key for both encryption and decryption.

Asymmetric encryption is also known as the public key encryption. It uses a pair of keys (public and private). One key is used for encryption, and a different key is used for decryption. The decryption key is kept private, while the encryption key is shared publicly, for anyone to use. Asymmetric encryption is a foundational technology for Transport Layer Security (TLS) often called as Secure Socket Layer (SSL) which is a predecessor of TLS. [11]

An encryption algorithm is the method used to transform data into ciphertext and back. Commonly used algorithms for symmetric encryption are:

- AES,
- 3-DES,
- SNOW,

commonly used algorithms for asymmetric encryption:

- RSA,
- Elliptic curve cryptography. [11]

2.1.2 Hash Functions

Hash is a tool used in cryptographic to securely store passwords in databases to ensure data integrity, and to make secure authentication possible. [12]

Hash function is defined as a unique identifier for a given content. It can also convert a plain-text on the input into a unique cipher text of a specific length. It is a one-way cryptographic algorithm, which means that thre is no possibility to convert the cipher text back to the original plaintext. The output of the hash function is called hash digest, hash value or hash code. Strong hash algorithm should have the following properties:

determinism ensures that the same input we will always result in the same hash on the output,

pre-image resistance makes it impossible to reverse the hash back into the original plaintext,

collision resistance ensures that every hash is matching only one original plaintext (the other words two different plaintexts on the input will never result into the same hash),

avalanche effect means that minor input text change will result in a completely different hash,

hash speed says that the calculation of the hash should be done in a reasonable time. [12]

One of the use cases for the hash is to ensure the data integrity. A simple example is to create the hash of a file, which is stored on the internet and which user downloads to their computer. In case that the owner of the file created and attached also hash for this file, the user can then calculate the hash of the file after it's downloaded and compare it to the hash created by the file author. In case that the hash is different it means that either content of the file has changed or is corrupted or less likely case, that the downloaded hash code was not downloaded correctly. In case that the file will be modified by its owner they will also have to generate a new hash. [12]

Commonly used hash functions are:

- MD5 (deprecated),
- SHA1 (deprecated),
- SHA2 (SHA-224, SHA-256 - most common, SHA-384, SHA-512),
- SHA3 (SHA3-224, SHA3-256, SHA3-384, SHA3-512). [12]

2.1.3 Public Key Infrastructure (PKI)

PKI is based on the public-private key pair mentioned in the section 1.2. PKI is the basic authentication and identity verification process used for secure communication (for example with the internet websites), for document signing and many other applications where security is involved. The public key is a very long sequence of numbers, which is embedded into a certificate. [13]

The certificate (referred to as X.509 certificate) contains private key mentioned in the previous chapter as well as many other information. The certificate is assigned to one entity, which wants to use the PKI-secured communication. It also includes information about it's authenticity. To obtain the trusted certificate it must be issued by a trusted source called Certification Authority (CA). The certification authority is an entity, which is generally trusted. If anyone wants the CA to issue the certificate, it will require them to proof their identity and provide their public key to be included in the certificate. The CA will then sign(explained in 2.1.4 the new certificate. The certificate which is to be signed is passed to the CA via Certificate Signing Request (CSR) as illustrated in 2.1. [13]

When using the PKI to establish secure communication, the client/user/device will get the certificate of the server and will verify its authenticity using the trusted signature by the CA.

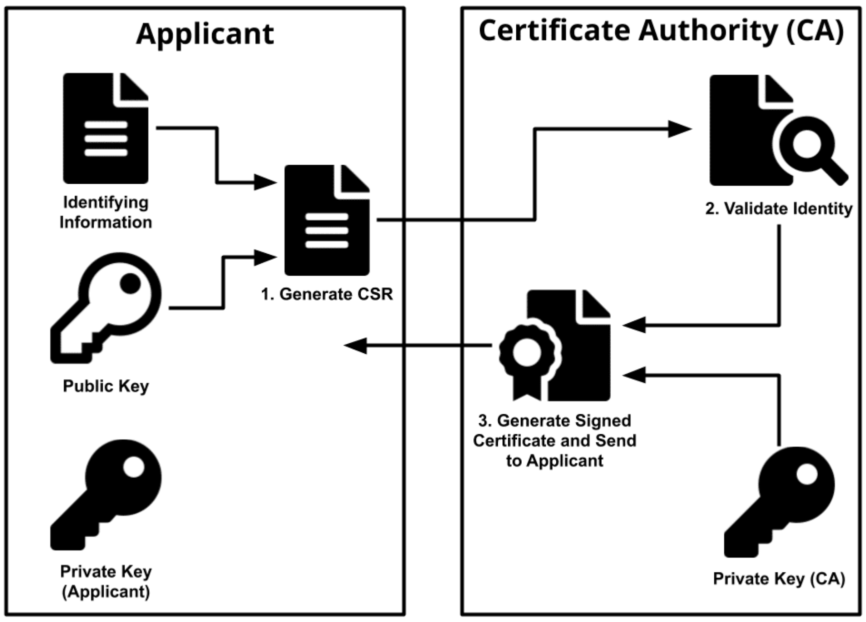


Figure 2.1 Signing of PKI certificate [14]

The root CA might issue so-called intermediate certificate, which is used to sign the applicant's certificate. In this case the client must validate the complete chain of trust to make sure all the certificates used are trusted and authentic. [13]

For the test purpose developers often use self-signed certificate. That means, that the developer will create a Certification Authority locally (on their computer or withing their network) and will use this CA to sign their certificates. Such a certificate will appear as a signed one, but if the CA certificate itself isn't signed by a trusted authority, then the self-signed certificate should neither be trusted.

2.1.4 Digital Signature

The figure 2.2 illustrates the process of signing data by the author of the data and verification of the authenticity and integrity of the data by the consumer.

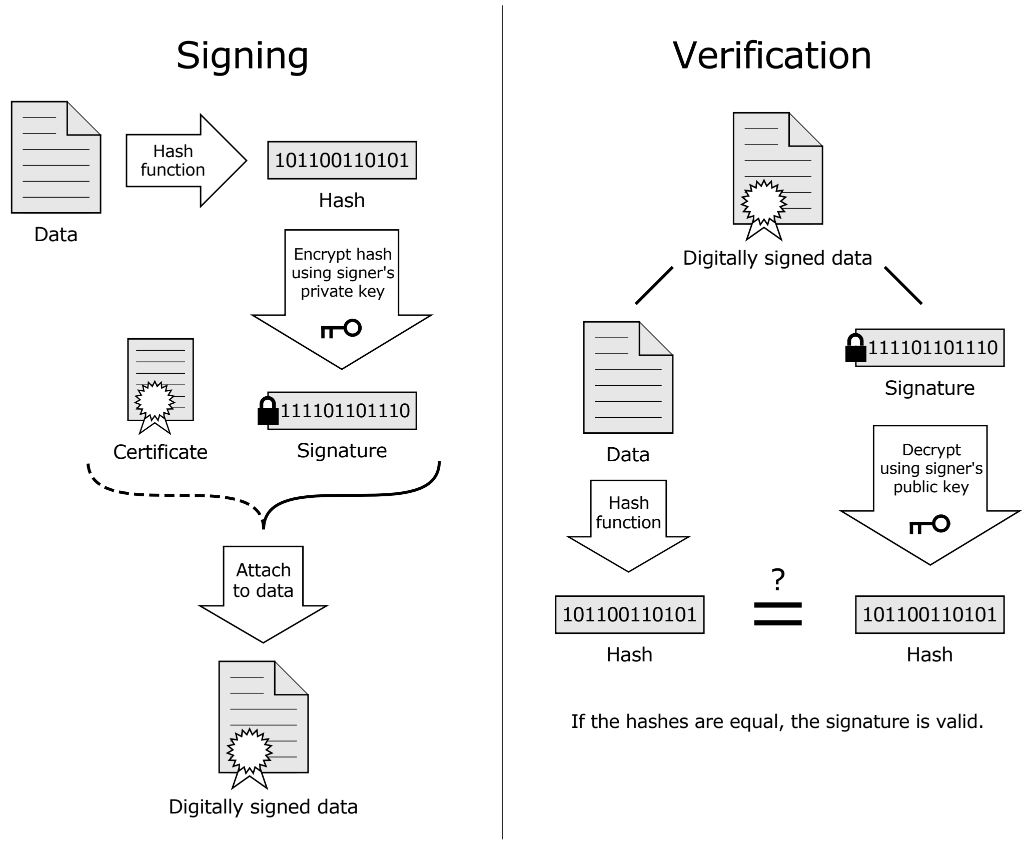
The signing process will firstly calculate hash of the data, which is then encrypted by the private key of the server. Because the key is private no-one else should be able to modify the data and them sign them using the same key. The signed hash is published together with the certificate of the author (which contains the public key of the author). [15]

In this example, the data themselves are not encrypted, but the signing process is in place to ensure that no-one has modified the data during the download and that they were not corrupted. It means that anyone can read the content of the data unless it is encrypted before or after the signing.

To verify the integrity and authenticity of the data anyone can calculate the hash of the data. As the certificate contains the public key, it means that anyone can decrypt the hash, which has been sent together with the data. The last step of this process is to compare the calculated hash against the decrypted hash received from the author. If they are equal then the data are considered valid and authentic. If any part of the data or the encrypted hash were corrupted or modified during the transfer the final comparison would fail. [15]

Digital signature can be used to sign email, document, bank transaction and also electronic certificates by the certification authority.

Figure 2.2 Digital Signature diagram [16]



2.1.5 Transport Layer Security

"Transport Layer Security (TLS) is one of the most important and widely used security protocols. It protects a significant proportion of the data that gets transmitted online. It's most prominently used to secure the data that travels between a web browser and website via HTTPS, but it can also be used to secure email and a host of other protocols." [17]

TLS is a successor of the previous Secure Sockets Layer (SSL) protocol and even today these terms are being used interchangeably. Like its predecessor it is valuable because it ensures authenticity of the other party, helps with the data integrity and provides confidentiality to the data transferred. TLS is used to secure many network protocols such as HTTP, SMTP, FTP, XMPP or NNTP. [17]

The first version of the TLS was introduced in 1999 in the RFC 2246 document. The most recent version of TLS (1.3) was introduced in 2018. Since 2019 the protocol went through multiple improvements such as adding of a new cryptographic ciphers[1] or removal of the ciphers, which have been proven weak or obsolete and also changes to the utilization of the hash functions. [17]

The main part of the TLS protocol is called record protocol, which is the underlying protocol for all further steps. The record protocol contains five separate sub-protocols (Handshake, Application, Alert, Change CIpher Spec and Heartbeat). [17]

The TLS 1.2 handshake protocol is used to establish the connection between the entities in a secure way.

[1] Cipher is an algorithm for performing the encryption or decryption.

There are three sub-types of the handshake:

basic handshake uses a sequence described in the diagram 2.3. It only authenticates the server but not the client. In the hello messages the server and the client agree on the cipher suite and compression to be used. In the next step the server sends its certificate for the server authentication. Lastly the server and the client will agree on a common secret (typically using Diffie-Hellman key exchanges) which will be used for the further encrypted communication.

Client-authenticated handshake also verifies the client authenticity,

abbreviated handshake can restart the communication between the server and client using previously agreed common secret and parameters. [17]

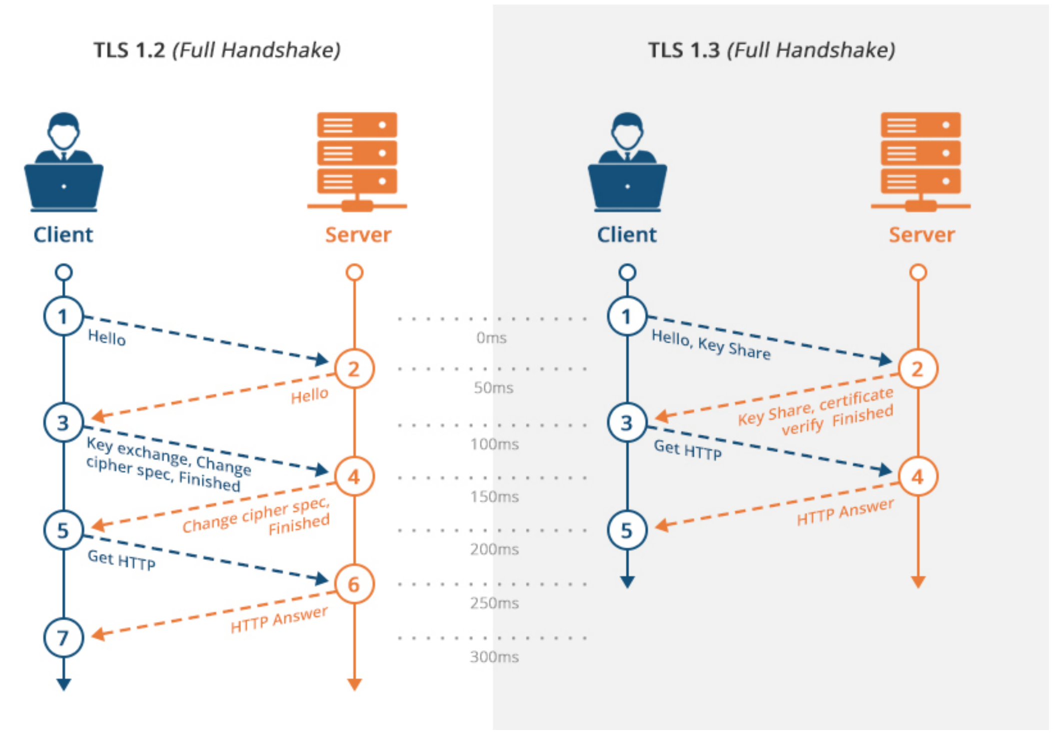


Figure 2.3 TLS 1.2 and 1.3 handshake [18]

The transferred data are encrypted using a symmetric encryption algorithm (for example AES) and the data are signed using authentication algorithms based on hash (for example HMAC-MD5, HMAC-SHA2 and others). [17]

The newest version TLS 1.3 (introduced in August 2018) is becoming dominant in the IT environments. It's main advantages are simplified handshake protocol (Client Hello message assumes that the server will accept the suggested key exchange parameters) and it supports less cipher suites, which makes it more straightforward to use. [17]

2.2 Cybersecurity Threats Related to FOTA

This section speaks about cybersecurity threats related to the firmware over the air update. It also shows examples of the FOTA related vulnerabilities.

2.2.1 Common Vulnerabilities and Exposures

This term is more known as a CVE. It is a list of publicly disclosed computer security flaws. Most of the public vulnerabilities are tracked by *Mitre corporation*[2] with support from the Cybersecurity and Infrastructure Security Agency (CISA) being part of the U.S. Department of Homeland Security. CVEs can also be found in other databases such as National Vulnerability Database (NVD) maintained by the National Institute of Standards and Technology (NIST). The purpose of CVEs is to help professionals coordinate their efforts to prioritize and address these vulnerabilities. [19]

Each CVE is given a unique ID and every year researchers, users and vendors submit thousands of new CVEs. To rate and prioritize the CVEs the Common Vulnerability Scoring System (CVSS) has been established to rate the CVE from 0.0 to 10.0 based on several criteria such as impact or probability of the flaw. The CVE ID has the following format: "CVE-year-number" (for example CVE-2021-22909). [19]

2.2.1.1 CVE-2021-22909 – Ubiquiti Firmware Update Bug

In February 2021, Ubiquiti released a new firmware update, to fix the CVE-2021-22909. The vulnerability described in this CVE is about a vulnerability in the firmware update procedure itself allowing a man-in-the-middle attacker to execute a code as the root on the device by replacing the official firmware by a malicious firmware created by the attacker. This CVE has been rated 7.5 (high). [20]

The devices created by the Ubiquiti allow user to log in via Command Line Interface (CLI) providing limited access to the device. One of the commands is used to trigger the remote update. The script will check the availability of a new firmware on the server. If there is a new firmware image available, the device will download it and perform the actual firmware update. [20]

The developers have used parameter -k when performing the curl operation, which disables the certificate verification for the TLS connections. That means the attacker can create a malicious firmware, upload it to his server and redirect the firmware update request to his server which might use a self-signed certificate. [20]

Code listing 2.1 Snippet of the get_tr_by_url() function [20]

```
get_tar_by_url ()
{
    mkdir $TMP_DIR
    if [ "$NOPROMPT" -eq 0 ]; then
        echo "Trying to get upgrade file from $TAR"
    fi

    if [ -n "$USERNAME" ]; then
        auth="-u $USERNAME:$PASSWORD"
    else
        auth=""
    fi

    filename="${TMP_DIR}/${TAR##*/}"
    if [ "$NOPROMPT" -eq 0 ]; then
        curl -k $auth -f -L -o $filename $TAR      # <-----
    else
        curl -k $auth -f -s -L -o $filename $TAR   # <-----
    fi
```

[2]https://cve.mitre.org

2.2.2 OWASP Internet of Things Project

The Open Web Application Security Project (OWASP) is a nonprofit foundation that works on improving the security of a software. Through the community-led open-source software projects, hundreds of local chapters worldwide, tens of thousands of members, and leading educational and training conferences, the OWASP Foundation is the source for developers and technologists to secure the web.

Owasp has dedicated suggestion for the IoT devices and selected Top 10 most common vulnerabilities and weaknesses in the IoT devices defined in 2018 and most of them relate to the firmware update itself. Sorted by the priority:

1. Weak Guessable, or Hard-coded Passwords
2. Insecure Network Services
3. Insecure Ecosystem Interfaces
4. Lack of Secure Update Mechanism
5. Use of Insecure or Outdated Components
6. Insufficient Privacy Protection
7. Insecure Data Transfer and Storage
8. Lack of Device Management
9. Insecure Default Settings
10. Lack of Physical Hardening [21]

2.3 Firmware Image Encryption

This section describes what the firmware image encryption is, how it increases the security of the FOTA update and how it can be implemented into the IoT Device and integrated to the new firmware release process.

2.3.1 Importance of Image Encryption

Encrypting firmware image, rather than sending it as a plain-text, improves the protection of the image confidentiality and intellectual property. Performing the device update without encryption increases risk of the firmware source code exposure. This could allow an attacker to subvert other protections in the device through reverse engineering thus allowing all similar devices in the field to become exploited. Alternatively a competitor could extract valuable algorithms or techniques in the software and use them to their advantage. [22]

The encryption of the firmware image before its transmission and then decryption of the image after it is downloaded to the device can reduce the risk of exposure during the transmission regardless of the communications paths of the image. [22]

2.3.2 Image Encryption Implementation

The image encryption is handled in the application layer of the ISO-OSI model[3]. [22]

[3]https://www.itu.int/rec/T-REC-X.200-199407-I/en

In a resource restricted environment it should be considered to use a Lightweight Encryption[4] described by the NIST in their report. [22]

Further security measures should be considered such as making sure the firmware image cannot be obtained by a physical attack on the device during or after the firmware update. These measures are described in the section 2.7. [22]

It is also suggested to use a secure data transfer (for example TLS) as a secondary level of the encryption to further mitigate the risk of exposure while the encrypted update deliverable is being distributed to the device. The communications path encryption only prevents exposure over the communications path. Any intermediate holding locations may result into exposure of the information. [22]

2.4 Firmware Integrity

This section describes what is the firmware integrity, why the integrity verification should be part of the firmware update process and describes the existing mechanisms to implement it.

2.4.1 Importance of the Firmware Integrity

The integrity check is commonly used mechanism to validate, that the data transferred or downloaded are complete and not corrupted (for example when downloading a file form a network shared drive or a web page).

Without integrity check, there is no way of verifying what the device received is what it was supposed to receive. Changing the code on the device could lead to any number of deviations from the intended functionality including preventing expected operations, adding new, undesired functions, changing how the data flow from the machine, or weaponizing the device to attack any other targets. [22]

Traditionally the file integrity can be achieved by usage of non-cryptographic hash functions such as a Cyclic Redundancy Check (CRC) or the checksum. The non-cryptographic hashes can validate the integrity against naturally occurring corruption of the payload, but can be easily subverted by bad actors. [22]

A more common mean of the integrity check is the use of cryptographic hash function to ensure the firmware file integrity, including undetected intentional modification by a bad actor as well as its authenticity (discussed in the next chapters). Even the cryptographic hash or other functions might fail to completely mitigate the risks. For the older, weaker hash functions, an attacker with a sufficient motivation and resources could generate a malicious update that generated the same hash as the legitimate update. [22]

2.4.2 Firmware Integrity Implementation

The suggested practice to ensure the firmware integrity is using the cryptographic signature of the firmware image. Its benefit is that it performs data integrity check and in case that the public key used for the firmware signing is only used within the secure build environment and not available publicly it can be considered as server authenticity verification.

Based on the NIST, the acceptable key lengths for the signing are 2048 bit for RSA and 224 bit for ECDSA, and acceptable hash functions are the SHA-2 family or newer. Eight other, less computationally intensive, algorithms exist (e.g. hashing functions like MD5, SHA-1, etc.) but have been found to be susceptible to various forms of attacks. [22]

The firmware integrity can be verified in a multiple steps. The first verification can be done when the firmware image is being downloaded from the remote server to the device. The firmware

[4] https://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf

update process usually requires restart of the device and in case that the firmware image integrity check fails, the device will not reboot and will continue running the current version of the firmware. Secondly the firmware integrity can be verified during the boot sequence of the processor by the bootloader. In case that the verification fails, the device will launch previous firmware (the previously downloaded firmware via the OTA or even the factory firmware which was uploaded during the manufacturing process. [23]

2.5 Server Authentication

This section describes how the firmware update server authentication can be done and why it is crucial for the FOTA.

The authentication of the update server and verification of its identity is important step of the FOTA to ensure that the server is trusted and known by the device. If this step is not performed as part of the update process then the device could connect to a non-trusted server and download an image created by the attacker. Just like the other potential risks this could result into the attacker gaining control of our device and use it for further attacks.

There are multiple tools to verify the server authenticity:

X.509 certificates are probably the most common way for the authentication (more details in 2.5.1) and is part of the TLS negotiation,

Trusted Platform Module (TPM) is a hardware platform which can be used to store X.509 certificates (more details in 2.6.3.2),

Symmetric Key is known to both server and device, it is easy to establish, but it is less secure than X.509 and is not considered best practice,

Shared Symmetric Key between multiple devices should never be used. In case of loosing control of one device, attacker has access to all the others. [24]

2.5.1 Server Authentication using PKI

The CA root certificate or intermediate certificate must be present in the device to verify the authenticity of the server. It can be uploaded as a part of the firmware or independently. It is typically uploaded during the manufacturing process or during the the device commissioning. [18]

The certificate always has a limited validity (in case of IoT devices it's typically 10 years). After this time (or earlier in case that the certificate is invalidated) the device needs to obtain a new certificate. It is a good practice to have multiple CA certificates present in the device in parallel to ensure the smooth transition from the old certificate to the new one. In case that the certificate is not updated on time and it expired, the device would lose the possibility to connect to the server and other devices and will require more complex solution (for example a physical connection to the device). [24]

An alternative solution to the complete certificate being stored in the device is to store just a thumbprint (hash) of the certificate to verify it. [24]

2.6 Device Authentication

There are multiple IoT authentication methods, including the digital certificates or the two-factor authentication and also hardware options such as hardware Root of Trust or Trusted Platform Module.

2.6.1 Two-factor Authentication

The two-factor authentication (2FA) method used to access websites works also for IoT devices. For the machine to machine (M2M) communication the 2FA requires a specific Bluetooth beacon or a near-field communication dongle in the requesting device that the remote server can use to ensure the authenticity of the device. [25]

2.6.2 Other Software Authentication Methods

"Depending on the IoT device and its network role, IT admins can use other software authentication methods such as digital certificates, organization-based access control and distributed authentication through the Message Queuing Telemetry Transport (MQTT) protocol. The MQTT connects the IoT device to a broker – a server that stores digital identities or certificates – to verify its identity and grant the access. Many manufacturers and vendors adopt the protocol because it's scalable to monitor thousands of IoT devices." [25]

2.6.3 Hardware Authentication

Hardware-based authentication methods, such as the hardware Root of Trust (RoT) and the Trusted Execution Environment (TEE), have become industry standards to secure IoT devices.

2.6.3.1 Hardware Root of Trust

RoT security model is a separate computing engine that manages devices' trusted cryptographic processors. This model finds application mainly in the IoT domain. This helps to protect the device from being hacked and keeps it locked onto the relevant network. The hardware RoT protects the devices from the hardware tampering and automates the reporting of unauthorized activity. [25]

Ideal implementation is used for devices, that transmit a high-value data such as financial or health, devices in remote locations and devices in public areas. [25]

2.6.3.2 Trusted Platform Module

"Another hardware authentication method is Trusted Platform Module (TPM), a specialized IoT device chip that stores host-specific encryption keys for hardware authentication. Within the chip, software can't access authentication keys, which makes them safe from digital hacks. When the device tries to connect to the network, the chip sends the appropriate keys and the network attempts to match them to known keys. If they match and have not been modified, the network grants access. If they don't match, the device locks and the network sends notifications to the appropriate monitoring software." [25]

Ideal implementation is used for devices connecting to a single server or other devices within the network. [25]

2.6.3.3 Trusted Execution Environment

"The TEE authentication method isolates authentication data from the rest of the IoT device's main processor through higher level encryption. The method runs parallel to the device's OS and any other hardware or software on it. IT admins find the TEE authentication method ideal for IoT devices because it puts no additional strain on the device's speed, computing power or memory." [25]

Ideal implementation is used for devices, that transmit a high-value data, devices which have less powerful CPUs and smaller memory caches and for the IoT gateways and other sensitive devices requiring higher security protection. [25]

2.7 Hardware Supporting Cryptography

This section describes hardware solutions allowing a simple implementation of the cryptographic mechanism and algorithms. The previous section 2.6.3 already described the hardware solutions that can be used for the authentication of the IoT devices.

2.7.1 Hardware Acceleration

The cryptographic functions discussed in this thesis (including the decryption, hashes such as SHA1 / SHA256 / HMAC, and also the Random Number Generation (RNG), Physically Unique Function (PUF) and the Secure Boot) are slow and consume significant amounts of the CPU power when implemented solely in a firmware. The hardware acceleration offers orders of magnitude differences in performance, as well as very large savings in the RAM and program space utilization. Most of the modern CPUs offer these cryptographic capabilities without a significant cost increases. These hardware accelerated cryptographic functions are compatible with the results of the software based cryptographic functions. They can be used interchangeably in the device firmware. Whenever possible the hardware accelerated functions should be used in preference to the software-based operations. [22]

2.7.2 Key Management

Most of the cryptographic principles rely on a “shared secrets” such as cryptographic encryption keys. These keys should be ephemeral[5] and unique to the target device to mitigate the risk that any additional devices could be attacked if one device’s keys are exposed. [22]

Ideally these keys are created / exchanged and stored in the device in “secured nonvolatile memory” in a manner that should protect these keys from exposure. To mitigate the risk of key exposure a proper key management techniques should be followed. [22]

The attacker may try to read the keys from the device’s memory during the decryption phase. Against this level of attacker the device would be better protected through usage of a hardware component designed specifically to store, use, and process the cryptographic keys. [22]

[5]Cryptographic key is called ephemeral if it’s generated for each execution of a key establishment process. [26]

Chapter 3

Security Features of ESP32

This chapter describes the ESP32 platform in general, it introduces the hardware variants and their features and it provides a list of ESP32 security related features.

3.1 ESP32 platform

ESP32 is *"a feature-rich MCU with integrated Wi-Fi and Bluetooth connectivity for a wide-range of applications"* [27]. It combines many different features normally provided by a different hardware components in a single chip.

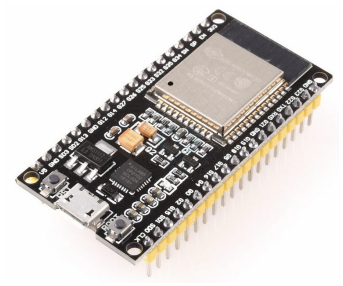


Figure 3.1 ESP-WROOM-32 development board [28]

The ESP32 provides robust design capable of functioning reliably in industrial environments, with an operating temperature ranging from –40°C to +125°C. It is powered by an advanced calibration circuits. [27]

The platform has ultra-low power consumption and therefore is an ideal solution for mobile, wearable and IoT applications and devices. The ESP32 also includes state-of-the-art features, such as fine-grained clock gating, various power modes and dynamic power scaling. [27]

The ESP32 is highly-integrated with in-built antenna switches, RF balun, power amplifier, low-noise receive amplifier, filters, and power management modules. The ESP32 provides extremely large functionality in one chip with a minimal Printed Circuit Board (PCB) requirements. [27]

The ESP32 can perform as a standalone system or as a slave device to a host MCU. It can interface with other systems to provide Wi-Fi and Bluetooth functionality through its SPI / SDIO or I2C / UART interfaces. [27]

3.1.1 ESP32 Hardware Versions

The ESP32 modules are split into the following series (each having an integrated flash memory):

ESP32-WROOM are modules with the Wi-Fi and Bluetooth/Bluetooth LE-based support a a dual-core processor,

ESP32-WROVER have the SPIRAM[1] and they contain a dual-core processor and are suited for applications requiring more memory (e.g. AIoT or gateway applications),

ESP32-MINI provide a cost-effective solution for simple Wi-Fi and Bluetooth/Bluetooth LE-based connectivity applications. [29]

3.1.2 ESP32 Family

All the module series and families are available in versions with either an integrated PCB antenna or with a connection to an external IPEX[2] antenna.

There are 4 families (also called series) of the ESP32 modules which are described in the next sections.

3.1.2.1 ESP32-S3

The newest S3 family has hav32-bit MCU & 2.4 GHz Wi-Fi & Bluetooth 5 (LE) and the following key parameters:

- Xtensa® 32-bit LX7 dual-core microprocessor, up to 240 MHz,
- 512 KB of SRAM and 384 KB of ROM on the chip,
- up to 32 MB flash, up to 8 MB PSRAM (depending on the module),
- Vector instructions to accelerate neural network computing and signal processing,
- 45 programmable GPIOs,
- security features: RSA-based secure boot, AES-XTS-based flash encryption, digital signature and the HMAC. [30]

3.1.2.2 ESP32-S2

The S2 family contains 32-bit MCU & 2.4 GHz Wi-Fi, but compared to other families it doesn't support the Bluetooth communication. These are its key parameters:

- Xtensa® single-core 32-bit LX7 microprocessor, up to 240 MHz,
- 320 KB SRAM and 128 KB ROM,
- up to 16 MB of Flash and up to 2 MB of PSRAM (depending on the module),
- ultra-low-power performance: fine-grained clock gating, dynamic voltage and frequency scaling,
- 37 programmable GPIOs,
- Security features: eFuse, flash encryption, secure boot, signature verification, integrated AES, SHA and RSA algorithms. [30][31]

[1]RAM memory connected via SPI bus.
[2]Standard connector used for antennas.

3.1.2.3 ESP32-C / ESP32-C3

The C family has single-core 32-bit RISC-V[3] MCU & 2.4 GHz Wi-Fi & Bluetooth 5 (LE) and the following key parameters:

- 32-bit RISC-V single-core processor, up to 160 MHz,
- 400KB of SRAM and 384 KB of ROM on the chip,
- up to 4MB of flash memory on the chip (depending on the model),
- state-of-the-art power and RF performance,
- 15 programmable GPIOs,
- Security features: RSA-3072-based secure boot, AES-128-XTS-based flash encryption, digital signature and the HMAC. [30][32]

3.1.2.4 ESP32

The original ESP32 family is still available and supported, but most of the modules from this family are not recommended by the manufacturer for the development of a new products.

This version of ESP32 has one or two 32-bit MCUs & 2.4 GHz Wi-Fi & Bluetooth/Bluetooth LE and following parameters:

- One or two Xtensa® 32-bit LX6 microprocessor(s), 80 MHz to 240 MHz,
- 520 KB of SRAM and 448 KB of ROM on the chip,
- up to 16 MB of Flash, up to 2 MB PSRAM in the chip and up to 8 MB external (depending on the module),
- sleep current below 5 μA,
- Bluetooth & Bluetooth Low Energy (Bluetooth LE),
- support for capacitive touch sensors, Hall sensor, SD card interface and Ethernet.
- 26 programmable GPIOs. [30][33]

The figure 3.2 shows architecture of an ESP32-WROOM module.

3.2 ESP32 Security

This section describes both the hardware and software features that are offered by the ESP32 modules and by the development framework provided by the manufacturer.

[3] An international non-profit organization supporting the free and open RISC instruction set architecture and extensions.

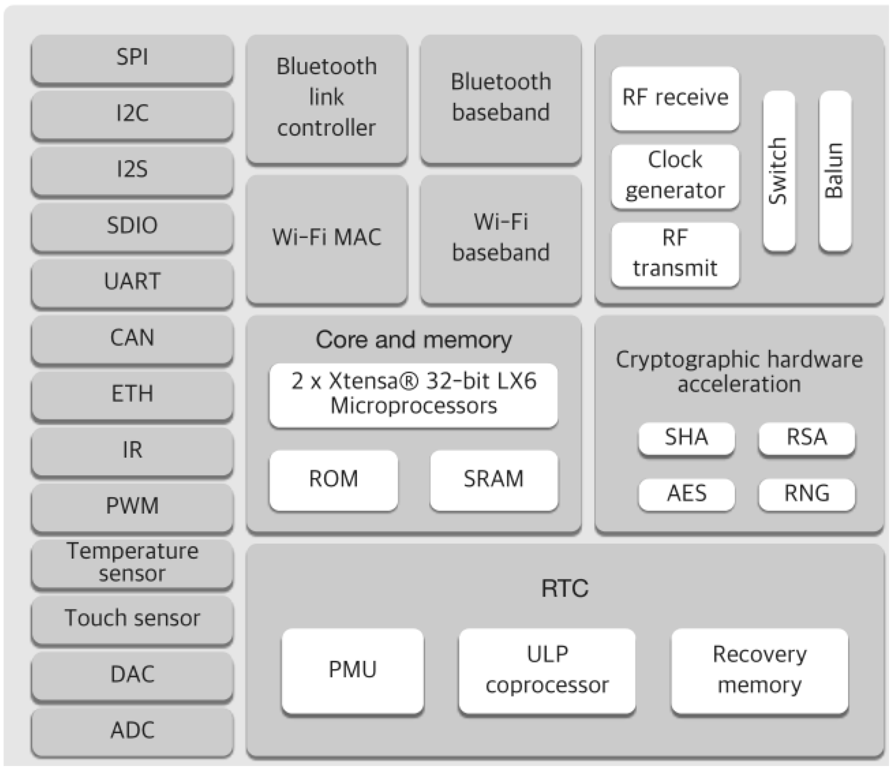


Figure 3.2 Architecture of the ESP32-WROOM series [34]

3.2.1 eFuse

"The ESP32 has a number of eFuses which store system parameters. Fundamentally, an eFuse is a single bit of non-volatile memory with the restriction that once an eFuse bit is programmed to 1, it can never be reverted to 0. Software can instruct the eFuse Controller to program each bit for each system parameter as needed. Some of these system parameters can be read by software using the eFuse Controller. Some of the system parameters are also directly used by hardware modules." [35]

The ESP32 has 4 eFuse blocks each of the size of 256 bits:

EFUSE_BLK0 is used entirely for the system purposes,

EFUSE_BLK1 is used for the flash encrypt key. If not using the Flash Encryption feature, they can be used for another purpose,

EFUSE_BLK2 is used for the security boot key. If not using the Secure Boot feature, they can be used for another purpose,

EFUSE_BLK3 can be partially reserved for the custom MAC address, or used entirely for the user application. Note that some bits are already used in IDF. [35]

3.2.2 Hardware Cryptography Feature

"The ESP32 platform provides hardware acceleration for the AES, SHA, RSA and RNG cryptographic features. It allows the ESP32 based devices to use strong encryption algorithms for

example for the flash encryption, image signature verification and TLS protocol support. Without the hardware support the ESP32 platform would not have enough power to use all of these features." [23]

3.2.2.1 Advanced Encryption Standard

"The Advanced Encryption Standard (AES) Accelerator speeds up AES operations significantly, compared to AES algorithms implemented solely in software. The AES Accelerator supports six algorithms of FIPS PUB 197, specifically AES-128, AES-192 and AES-256 encryption and decryption." [23]

3.2.2.2 Secure Hash Algorithm

"The Secure Hash Algorithm (SHA) Accelerator is included to speed up SHA hashing operations significantly, compared to SHA hashing algorithms implemented solely in software. The SHA Accelerator supports four algorithms of FIPS PUB 180-4, specifically SHA-1, SHA-256, SHA-384 and SHA-512." [23]

3.2.2.3 RSA Cipher

"The RSA Accelerator provides hardware support for multiple precision arithmetic operations used in RSA asym- metric cipher algorithms. Sometimes, multiple precision arithmetic is also called "bignum arithmetic", "bigint arithmetic" or "arbitrary precision arithmetic"." The RSA Accelerator supports keys of length up 4096 bits long. [23]

3.2.3 Random Number Generator

The ESP32 contains the hardware Random Number Generator (RNG), which produces truly random numbers during the following conditions:

- RF subsystem is enabled (Wi-Fi or Bluetooth),
- the internal entropy source has been enabled by calling *bootloader_random_enable()*. [36]

When one of the conditions is true, the samples of a physical noise are mixed into the internal hardware RNG state to provide the entropy. If none of the above conditions are true, the output of the RNG should be considered pseudo-random only. [36]

3.2.4 Secure Boot V2

This part describes the Secure Boot V2, which is supported by the chips with version ECO3 onwards. The previous versions only support the the Secure Boot V1. [23]

The Secure Boot protects a device from running any software, which is not signed. This software includes the second stage bootloader and each application binary. The first stage bootloader does not require to be signed as it cannot be changed during the firmware updates. [23]

If the Secure Boot V2 is enabled in the corresponding eFuse the boot sequence will proceed in the following steps:

1. the second stage bootloader's RSA-PSS signature block is verified,
2. the bootloader image is verified,
3. the bootloader image is executed,
4. the bootloader verifies the application image's RSA-PSS signature block,

5. the bootloader verifies the application image
6. if the previous check fails, the bootloader will try to verify next image using from the step 5,
7. the bootloader executes application image (if any found). [23]

The RSA-3072 public key is stored on the device and it's digest is stored in an eFuse. The corresponding RSA private key is kept at a secret place and is never accessed by the device. Only one public key can be generated and stored in the chip during manufacturing. No secrets are stored on the device and it is immune to a passive side-channel attacks (for example timing or power analysis). The same image format and signature verification method is used to sign applications and bootloader images. [23]

3.2.5 Flash Encryption

"Flash encryption is intended for encrypting the contents of the ESP32's off-chip flash memory. Once this feature is enabled, firmware is flashed as plaintext, and then the data is encrypted in place on the first boot. As a result, physical readout of flash will not be sufficient to recover most flash contents." Flash encryption protects firmware against unauthorised readout and modification. [37]

All the production firmware images should be published in the "Release" mode with the encryption enabled. When the encryption is enabled then the bootloader, the partition table and all "app" partitions are encrypted. Other partitions can also be encrypted based on the configuration. As of now the Non-volatile Storage (NVS) used for data storing cannot be encrypted due to incompatibility between the flash encryption and the NVS library. [37]

The flash encryption operation is controlled by the various eFuses available on the ESP32. It can for example disable the flash encryption when performing the firmware update via the UART (serial connection). One of the eFuses stores the encryption key used for the AES-256 encryption and it is by default protected from the software access. [37]

Flash encryption does not prevent an attacker from understanding the high-level layout of the flash, because the same AES key is used for the different blocks and the blocks with the identical content (such as empty or padding areas) result into a matching pairs of the encrypted blocks. [37]

It is recommended to always use the Secure Boot (V1 or V2) as the flash encryption alone may not prevent an attacker from modifying the firmware of the device. [37]

3.2.6 Firmware Signature Verification

When using the Secure Boot V2 3.2.4 the image is being signed and the signature is not only verified during the boot sequence, but also during the FOTA update. [23]

The ESP32 also allows a verification of the firmware's signature without the use of the Secure Boot. The signing scheme is exactly the same as when the Secure Boot is being used. The firmware signing adds the information described in the table 3.1 to the firmware image file. This option can be used in case that the device needs to boot up very quickly and to avoid the Secure Boot process which would prolong the start up sequence. In this case the attacker should not be allowed to gain physical access to the device, which would allow him tampering of the device and eventually changing the bootloader or the actual firmware in the device. [23]

With this option the currently running firmware is being considered as a trusted one. It must contain the public key in its signature block which is used to verify the signature of the newly updated firmware. The check can be performed during the firmware start and also during the FOTA process. It's essential that the initial app flashed to the device is also signed. [23]

Older revisions of the ESP32 hardware (below ESP32-ECO3) only support the ECDSA based signing scheme. All the newer revisions (ESP32-ECO3 and newer, ESP32-S2, ESP32-C3 and

Table 3.1 Content of the Signature Block [23]

Offset	Size (bytes)	Description
0	1	Magic byte
1	1	Version number byte (currently 0x02), 0x01 is for Secure Boot V1.
2	2	Padding bytes, Reserved. Should be zero.
4	32	SHA-256 hash of only the image content, not including the signature block.
36	384	RSA Public Modulus used for signature verification. (value 'n' in RFC8017).
420	4	RSA Public Exponent used for signature verification (value 'e' in RFC8017).
424	384	Pre-calculated R, derived from 'n'.
808	4	Pre-calculated M', derived from 'n'
812	384	RSA-PSS Signature result (section 8.1.1 of RFC8017) of image content, computed using following PSS parameters: SHA256 hash, MFG1 function, salt length 32 bytes, default trailer field (0xBC).
1196	4	CRC32 of the preceding 1095 bytes.
1200	16	Zero padding to length 1216 bytes.

ESP32-S3) support the RSA-3072 based Signature. To ensure the support of the RSA, the configuration flag *CONFIG_ESP32_REV_MIN* has to be set to Rev 3. This and other firmware signing settings can be modified in the *Security features* within the Project Configuration Menu. [23]

If possible the device should always use the Secure Boot signature verification, which also verifies the signature of the bootloader instead of this simple signature verification. [23]

By default the image signing is performed as part of the build process. An alternative solution is to perform a remote signing described in the next section.

3.2.6.1 Remote Signing of Images

In case that the *Sign binaries during build* option is disabled in the Project Configuration Menu, then the image needs to be signed remotely (typically as part of the Continuous Integration process). In the production, it is good practice to store the signing key on a remote secured server, which is not the same machine as the one used for the creation of the image.[23]

The *espsecure.py* command line program can be used to sign the firmware image on a remote server:

Code listing 3.1 Remote signing of a firmware image [23]

```
espsecure.py sign_data --version 2 --keyfile PRIVATE_SIGNING_KEY \
    --output SIGNED_BINARY_FILE BINARY_FILE
```

3.2.7 TLS Support

The ESP32 has support for the TLS communication (both hardware and software).

The latest stable version of ESP-IDF (version 4.4) supports various version of SSL/TLS up to TLS 1.2. There is already support for the TLS 1.3 possible using WolfSSL, but it has much stronger license and is not free for use in commercial projects. [38]

The currently developed version of ESP-IDF (5.0) will have support for the TLS 1.3 using an open source library MbedTLS (moving from version 2.X to 3.1.0). As of writing this thesis the

MbedTLS support is only experimental and only implemented for the client part of the HTTPS connections (the server side support is not yet added). The new version of the ESP-IDF will also restrict all the older and insecure protocols and ciphers. Therefore it will not be possible to use TLS 1.1 and older. The TLS 1.2 will be fully supported and the TLS 1.3 should also be fully supported. [38][39]

3.2.8 Automatic Rollback

The main purpose of the application rollback is to keep the device working after the update.” In case that a new firmware has critical errors or when the device restarts during the start up sequence of the new firmware then the device will roll-back and start the previous version of the firmware. The implementation has multiple parts. After the new firmware is downloaded, verified and stored in the flash, the device will restart. The new firmware will start and it should perform a self-check. In case that the self check is successful, the firmware will call *esp_ota_mark_app_valid_cancel_rollback()* and the new firmware will be considered as a valid from that moment. [40]

The feature can be enabled by the *CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE* flag in the project configuration. [40]

3.2.9 Anti-rollback

”Anti-rollback prevents rollback to application with security version lower than one programmed in eFuse of chip.” [40]

This function works if the *CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK* option is set. The version in the new firmware must be greater than or equal to the version stored in the eFuse on the device. The number of bits in the eFuse is limited to 32 bits (4 bytes). And as each of the bits can only be changed from 0 to 1 and not back, there can only be up to 32 different security versions preventing the rollback. [40]

3.3 Software Solution for ESP32

”ESP-IDF is Espressif’s official IoT Development Framework for the ESP32, ESP32-S and ESP32-C series of SoCs. It provides a self-sufficient SDK for any generic application development on these platforms, using programming languages such as C and C++. ESP-IDF currently powers millions of devices in the field, and enables building a variety of network-connected products, ranging from simple light bulbs and toys to big appliances and industrial devices.” [41]

3.3.1 ESP-IDF Features

The ESP-IDF is an open source project available on the GitHub licensed mostly under the Apache 2.0 license. It has defined release process offering stable versions of the source code as well as a work in progress repository allowing the developers to test the latest features in the development. [41]

The ESP-IDF offers large support and the components for peripherals, networking, communication protocols and many other features and it is officially supported by the Microsoft Viusal Studio Code and Eclipse. [41]

It comes with a comprehensive documentation, API description and manuals and the project contains over 100 examples of using various peripherals and features. [41]

The figure 3.3 shows the key software components and the key features of the ESP-IDF are:

- RTOS Kernel,
- Standard Programming Interface,
- Peripheral Drivers,
- Wi-Fi,
- Bluetooth & Bluetooth LE,
- Networking Protocols,
- Power Management,
- Storage,
- Security,
- Network Provisioning,
- Build Systems,
- Developer Tools,
- IDE Support. [41]

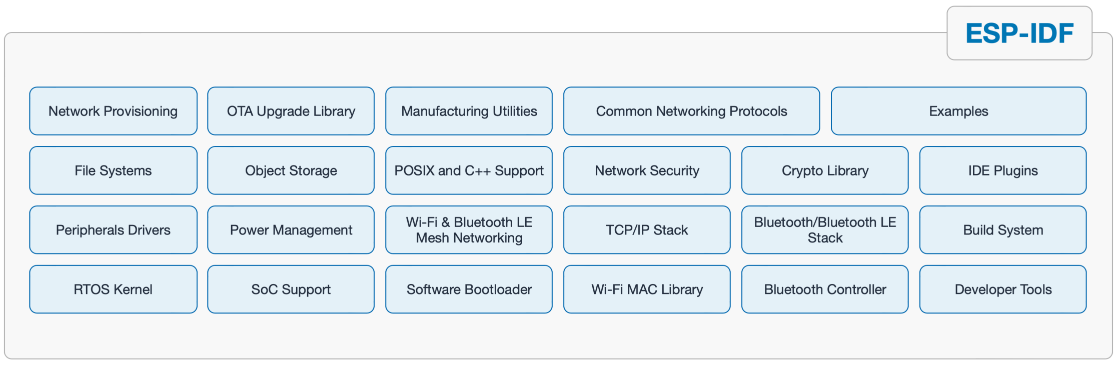


Figure 3.3 ESP-IDF Software Components [41]

3.3.2 ESP-TLS Component

"The ESP-TLS component provides a simplified API interface for accessing the commonly used TLS functionality. It supports common scenarios like CA certification validation, SNI, ALPN negotiation, non-blocking connection among others. [42]

The component has an option to use the MbedTLS or WolfSSL (an embedded SSL/TLS libraries providing secure communication for IoT) as their underlying library. By default MbedTLS is used. The ESP-IDF provides the source codes and examples which are useful for understanding the APIs. [42]

Chapter 4

Practical Part - Secure FOTA on ESP32

This chapter speaks about the difference between the insecure and secure version of the ESP32 Firmware Over The Air update and describes how both of these versions were implemented as part of this thesis.

4.1 Environment and Toolchain

This section describes the model of ESP32 platform used for the proof of concept design and verification. It also contains a list of the software tools used in this work. Lastly it contains instructions to set up the environment and to compile the firmware on the ESP32 platform.

4.1.1 Software Toolchain

To develop the proof-of-concept firmware for the purpose of this tesis and for the validation of the firmware behavior the author has used the following tools including their versions listed:

- macOS Monterey (12.1)
- Visual Studio Code (1.65)
- Visual Studio Espressif Integrated Development Framework (ESP-IDF) Extension (v1.4.0)
- Visual Studio ESP-IDF esp_encrypted_img Component (2.0.2)[1]
- Python (3.8.5)
- Screen (4.00.03)
- OpenSSL (1.1.1m)
- Espressif ESP-IDF[2](different versions described further in this chapter)

[1]https://github.com/espressif/idf-extra-components/tree/master/esp_encrypted_img
[2]https://github.com/espressif/esp-idf

4.1.2 Hardware

The author has decided to use ESP32-WROOM (D0WDQ6) ver. 1 without any peripherals connected to it.

4.2 Common Part Of The Implementations

This sections describes implementation, setup up and verification parts that are common to all the examples and implementations mentioned in the following sections.

4.2.1 High-level Implementation Architecture

The implementations contain 2 main threads. The first thread is used to handle the HTTP requests and to provide a web page displaying a version of the currently running firmware (shown as a screenshot in the attachment A.1). The second thread handles the checking of the availability of the new firmware version and also performs the Over The Air firmware update itself.

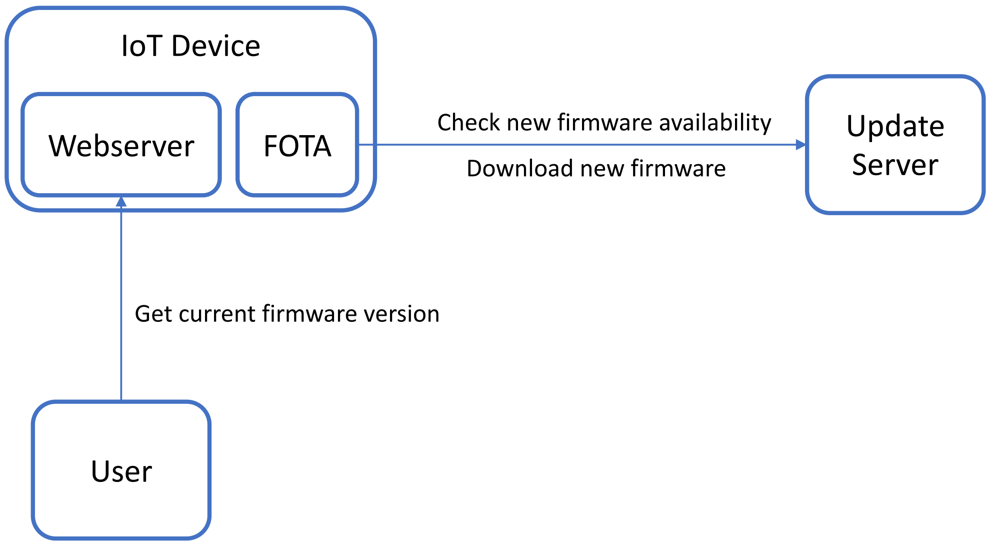


Figure 4.1 High-level implementation architecture, created by the author

4.2.2 Periodical Firmware Check

The implementations described in the following sections periodically check the availability of the new firmware on the update server. In case that the version of the firmware on the server is newer than the version currently running on the tested device, the application will start the firmware update process.

Compared to the real life environment, the examples in this thesis check the presence of a new version of the firmware on the update server periodically every 60 seconds. In the real-life use case the period might be set to a several hours interval to preserve the battery capacity and prolong the life-time of the IoT device.

4.2.3 Project Configuration

The prerequisite for the further steps is installation of all the tools listed in the section 4.1.1.

The ESP32 based device connects to the local WIFI network using the configuration described in this section.

To define the macros via the project configuration and to enable some of the macros that are defined within the source code provided by Espressif it is necessary to create and update the following configuration files.

The lines from the code listing 4.1 must be added into the *sdkconfig.defaults* file and the ”hostname” replaced with the actual hostname or IP address of the update server, ”nnn” with the firmware version in digits, ”ssid” with the actual SSID and ”password” with the actual WIFI password.

Code listing 4.1 Common content of the sdkconfig.defaults

```
CONFIG_FIRMWARE_UPGRADE_URL="hostname"
CONFIG_APP_PROJECT_VER_FROM_CONFIG=y
CONFIG_APP_PROJECT_VER="nnn"
CONFIG_EXAMPLE_CONNECT_WIFI=y
CONFIG_EXAMPLE_WIFI_SSID="ssid"
CONFIG_EXAMPLE_WIFI_PASSWORD="password"
CONFIG_EXAMPLE_CONNECT_IPV6=y
```

Kconfig.projbuild file must be created and filled with the content shown in the code listing 4.2.

Code listing 4.2 Common content of the Kconfig.projbuild

```
menu "FOTA Configuration"

    config FIRMWARE_UPGRADE_URL
        string "firmware upgrade url endpoint"
        default "192.168.1.200"
        help
            URL of server which hosts the firmware
            image.

    config APP_PROJECT_VER_FROM_CONFIG
        bool "Get the project version from Kconfig"
        default y
        help
            If this is enabled, then config item APP_PROJECT_VER
            will be used for the variable PROJECT_VER.
            Other ways to set PROJECT_VER will be ignored.

    config APP_PROJECT_VER
        string "Project version"
        default "100"
        depends on APP_PROJECT_VER_FROM_CONFIG
        help
            Project version

endmenu
```

After the update of the files it is necessary to run *idf.py menuconfig* script to apply the changes.

4.3 Basic Insecure Firmware Over the Air

This section describes implementation of the most basic Firmware Over The Air update on the ESP32 platform, which doesn't implement any of the common cyber security practices. It only supports the HTTP connection between the ESP32 and the update server containing the firmware package.

The basic implementation is based on the stable ESP-IDF release branch version 4.4[3].

4.3.1 Implementation

To perform the FOTA update without any security features, the HTTP (non-secure) has to be explicitly enabled in the project configuration using the following macro.

Code listing 4.3 Additional content of sdkconfig.defaults

```
CONFIG_ESP_HTTP_CLIENT_ENABLE_HTTPS=y
```

To use this basic HTTP version, there is no certificate needed (neither for version checking, nor for the OTA itself).

The application will download a file called *version* from the IP address defined in the configuration, which contains 3 digit number representing the firmware version of the newest firmware file available on the server. It will compare the number in that file with the currently running version of the firmware. If the new version is larger than the current one, then the simple FOTA update via HTTP will be performed. The prototype code uses *esp_https_ota()* function call which handles both the download of the new image as well as the firmware upgrade itself.

The firmware file has the following name: *imageXXX.bin*, where XXX is replaced with the number read from version file (for example *image101.bin*).

4.3.2 Verification Of The Results

To test this basic version the author used a basic HTTP server provided by python. The author created two firmware files (*image100.bin* and *image101.bin*) and the *version* file containing the number *100*. The HTTP server was started using the command shown in the code listing 4.4.

Code listing 4.4 Run (insecure) HTTP server

```
python -m http.server 80
```

The ESP32 board was flashed with the firmware version 100. During the first minute the device verified presence of a new firmware. The update was not performed as the firmware currently running was matching the version of the firmware available on the server.

In the next step the author changed the content of the *version* file from *100* to *101*. After this change the device compared the current version with the one obtained from the *version* file. The new version was greater than the previous one and the device performed successful FOTA update.

[3]https://github.com/espressif/esp-idf/tree/release/v4.4

4.4 Secure Firmware Over the Air

The secure implementation of the FOTA in this thesis uses three following security mechanisms (described in more detail in the next sections):

1. secure file transfer using TLS 1.2,
2. firmware signing,
3. firmware image encryption.

These mechanisms are used in a different stages of the firmware creation process and its transfer to the device. Figure 4.2 illustrates that the first step after the firmware image is compiled is signing of the image followed by the its encryption. During the FOTA process the file is transferred to the device using TLS 1.2 and the image decryption and verification is done in the reverse order (firstly the firmware needs to be decrpyted and in the next step its signature is verified). If all these steps are performed successfully, the firmware is upgrade is performed and the device is restarted.

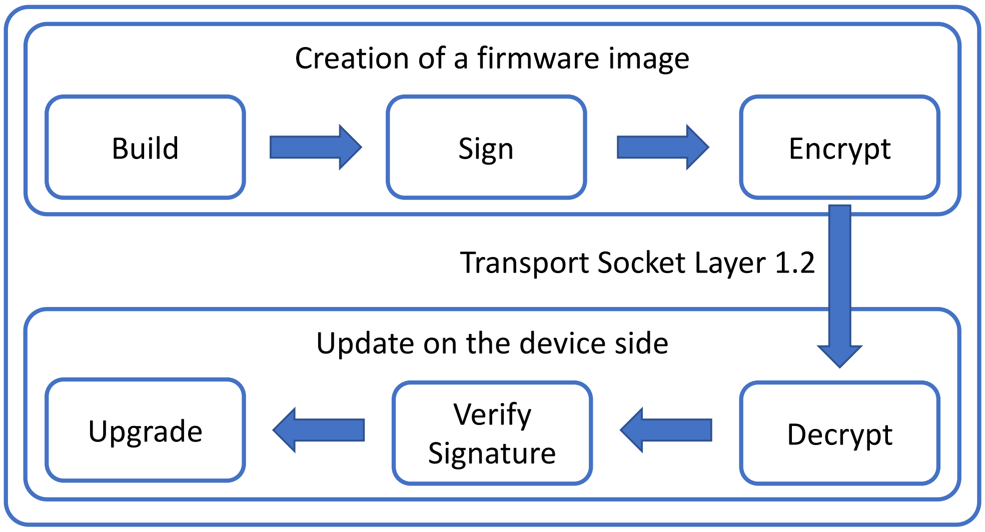

Creation of a firmware image
Build
Sign
Encrypt
Transport Socket Layer 1.2
Update on the device side
Upgrade
Verify Signature
Decrypt

Figure 4.2 Life-cycle of the firmware image, created by the author

Each of the security mechanisms requires an encryption key, certificate or key-pair. They need to be integrated into the firmware build process. The keys / certificates are listed in the table 4.1 and the way to obtain the keys is described in the related sections.

Table 4.1 Cryptographic keys and certificates included in the source code

File	Key type	Purpose / Section
server_certs/ca_cert.pem	RSA4096	Secure file transfer 4.4.1
secure_boot_signing_key.pem	RSA3072	Firmware image signing 4.4.2
rsa_key/private.pem	ECDSA256	Firmware image encryption 4.4.3

The secure firmware update implementation is based on the most recent version of the ESP-IDF[4], which contains many new security implementations and features including the firmware signing, image encryption and draft of the TLS 1.3 protocol.

[4]https://github.com/espressif/esp-idf/commit/45c1d1cba212b2012d53a55fcf9d338a959c2ece.

4.4.1 Secure File Transfer

The section 3.2.7 already describes, that the latest version of MbedTLS supporting the TLS 1.3 is being integrated into the ESP-IDF platform at the time of creation of this thesis. The available version supports TLS 1.2 and it forbids using any older TLS/SSL versions and related ciphers. For the purpose of the proof of concept application only the TLS 1.2 protocol is enabled.

The author investigated whether the implementation of TLS 1.3 version is sufficient for the purpose of this thesis. Unfortunately the implementation still contained errors and missing implementation. Example of the missing implementation is a missing macro in the error check in the *check_config.h*[5] file (code listing 4.5). The implementation is very basic and is more of a starting point to implement all the features that the TLS 1.3 offers.

Code listing 4.5 Example of the incomplete TLS 1.3 implementation in the mbedTLS, the TLS 1.3 is missing in the list and enabling only TLS 1.3 support causes a compilation error

```
#if defined(MBEDTLS_SSL_TLS_C)
          && (!defined(MBEDTLS_SSL_PROTO_SSL3) && \
    !defined(MBEDTLS_SSL_PROTO_TLS1)
          && !defined(MBEDTLS_SSL_PROTO_TLS1_1) && \
    !defined(MBEDTLS_SSL_PROTO_TLS1_2))
#error "MBEDTLS_SSL_TLS_C defined, but no protocols are active"
#endif
```

The TLS protocol firmware requires presence of the trusted Certification Authority certificate in the firmware running on the device. The same authority must sign the certificate of the FOTA server. The process is illustrated in the figure 4.3 and the shell commands which were used to create the keys and certificates are in the code listing 4.6.

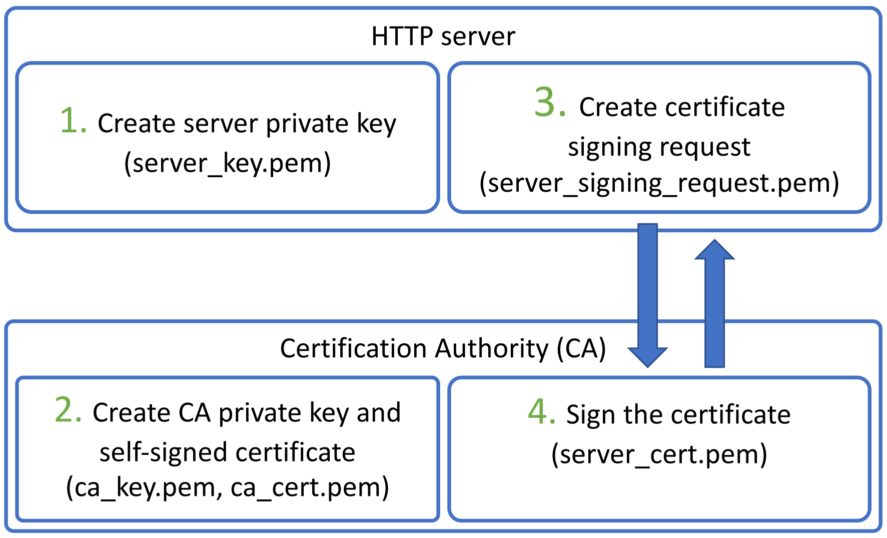

HTTP server
1. Create server private key (server_key.pem)
3. Create certificate signing request (server_signing_request.pem)
Certification Authority (CA)
2. Create CA private key and self-signed certificate (ca_key.pem, ca_cert.pem)
4. Sign the certificate (server_cert.pem)

Figure 4.3 Generating Certification Authority and server keys and certificates

The *ca_cert.pem* file must be copied to the *server_certs* folder in the project before the compilation. All the certificates signed by this certification authority are trusted by the device.

[5]https://github.com/wolfeidau/mbedtls/blob/master/mbedtls/check_config.h

Code listing 4.6 Generating CA and server keys and certificates

```
#Generate the private key of the web server
openssl genrsa -out server_key.pem 4096

#Create CA private key and certificate
openssl req -x509 -newkey rsa:4096 -keyout ca_key.pem \
    -out ca_cert.pem -sha256 -days 3650 \
    -subj "/C=CZ/O=Marek's CA/CN=localhost" -nodes

#Create Certificate Signing Request (CSR),
#Common Name (CN) has to be set to the FQDN or IP address of the server
openssl req -new -sha256 -key server_key.pem \
    -out server_signing_request.pem \
    -subj "/C=CZ/O=FOTA HTTP server/CN=192.168.50.203"

#Sign the server certificate
openssl x509 -req -sha256 -in server_signing_request.pem \
    -CA ca_cert.pem -CAkey ca_key.pem \
    -CAcreateserial -out server_cert.pem -days 3650
```

Verification of the TLS functionality is described in the chapter 4.4.4.1.

4.4.2 Firmware Signing

Firmware signing is a feature present in the ESP32 platform. The firmware signing is used together with the firmware encryption and therefore the signing has to be the first step after the compilation. The verification of the signature in the device during the update can only be done after the firmware is decrypted (encryption and decryption is described in the next section).

The firmware signing uses the same headers and structures as the Secure Boot feature. The proof of concept application uses the firmware signing without the Secure Boot feature being enabled.

Revision of the hardware used for this thesis is below rev. 3, which means that the ECDSA algorithm had to be used for the image signing (as explained in section 3.2.6).

The signing has been included in the build process, which produces both the firmware files with and without signature.

The certificate used for the firmware signing has been generated using the command shown in the code listing 4.7.

Code listing 4.7 Generate RSA signing key

```
~/esp-idf/components/esptool_py/esptool/espsecure.py \
    generate_signing_key --version 1 \
    --scheme ecdsa256 secure_boot_signing_key.pem
```

The signing key has to be saved in the project source code root folder and the the line shown in the code listing 4.8 must be added into the *main/CMakeList.txt* file.

Code listing 4.8 Include the signing key into project configuration

```
EMBED_TXTFILES ${project_dir}/secure_boot_signing_key.pem
```

To enable the signing verification and generation the configuration has to be updated using *idf.py menuconfig* command or by adding it to the *sdkconfig* file. The required configuration is done in the *sdkconfig.defaults* file which is part of the source code. The lines added are shown in the code listing 4.9. When building the project for the first time the *sdkconfig* file inherits the content from the *sdkconfig.defaults* file.

Code listing 4.9 Enable firmware signing in the project configuration

```
#
# Security features
#
CONFIG_SECURE_SIGNED_ON_UPDATE=y
CONFIG_SECURE_SIGNED_APPS=y
CONFIG_SECURE_BOOT_V1_SUPPORTED=y
CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT=y
CONFIG_SECURE_SIGNED_APPS_ECDSA_SCHEME=y
CONFIG_SECURE_SIGNED_ON_UPDATE_NO_SECURE_BOOT=y
CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES=y
CONFIG_SECURE_BOOT_SIGNING_KEY="secure_boot_signing_key.pem"
# end of Security features
```

Verification of the firmware signing functionality is described in the chapter 4.4.4.2.

4.4.3 Encryption Of The Firmware

The previous section mentions that the firmware encryption is the last step of the image building process and at the same time it is the first step of verification during the FOTA update (firmware needs to be decrypted prior verification of the image signature).

To add support for the firmware image encryption, the author has installed a Visual Studio Code (VS Code) component called esp_encrypted_img. The code of the component and link for the Component Manager can be found in a Github repository containing idf-extra-components[6]. Installation using VS Code can be done using *ESP-IDF: Component Registry* from the Command Palette.

The ESP-IDF provides an example application of the firmware encryption, which was inspiration for the proof of concept application developed by the author. This example called pre_encrypted_ota[7] can be found in the Espressif Github repository.

The project doesn't require any further configuration as the encryption is enabled by adding the related callbacks into the application.

The encryption of the image needs to be done after the successful build process is completed. The firmware encryption requires RSA key to be generated. The code listing 4.10 shows the script which the author used to generate the encryption key. To encrypt the built image using the generated key, the author has executed command shown in the code listing 4.11. The command uses the *esp_enc_img_gen.py*[8] script.

Code listing 4.10 Generating key for the firmware encryption

```
openssl genrsa -out private.pem 3072.
```

Code listing 4.11 Encryption of the firmware

```
./esp_enc_img_gen.py encrypt secure_ota.bin private.pem image201.bin
```

Verification of the firmware encryption functionality is described in the chapter 4.4.4.3.

[6]https://github.com/espressif/idf-extra-components
[7]https://github.com/espressif/esp-idf/tree/master/examples/system/ota/pre_encrypted_ota
[8]https://github.com/espressif/idf-extra-components/blob/master/esp_encrypted_img/tools/esp_enc_img_gen.py

4.4.4 Verification of the results

Multiple versions of the firmware are used for the verification of the FOTA feature. The starting version which is in the source code is 200. Version 201 is used for the positive test scenario with a successful firmware update result and for the test with the invalid TLS certificate. The versions 202 to 205 are used to simulate an unsuccessful FOTA update. All of these scenarios are described in the following sections.

Same as the basic version, the HTTP server provides *version* file containing version of the newest firmware (or better version of the firmware which should be downloaded). At the beginning the version is set to *200*. The version is changed to a value between *201* and *205* depending on the test being performed. After this change the device compares the current firmware version with the one obtained from the *version* file from the server. If the new version is greater than the previous one then the device performs the FOTA update.

The implemented security features must be tested all together for the positive test result. For the negative test results of the individual features the author performed test of only one feature at a time (for example by providing wrong image signature or wrong encryption key), but never a combination of the two features at a time.

All the binaries used for the FOTA verification are stored on the attached media inside the *tests/advanced* folder.

All the performed tests include secure file transfer. For the initial positive result tests there is no limitation of the TLS protocol version and the server key and certificate created earlier in section 4.4.1 are used. The next section describes how to start the HTTP server using TLS.

4.4.4.1 Secure File Transfer

To start the server using the TLS the author has used the command shown in the code listing 4.12. The server must be started from the folder which contains all the test files (*image201.bin*, . . . , and *version*).

Code listing 4.12 Run HTTPS server

```
openssl s_server -WWW -key server_key.pem -cert server_cert.pem -port 443
```

To verify the functionality of the secure data transfer using TLS the author has performed the following tests:

Enforcing TLS version 1.1 results into the error shown in the 4.14, because the firmware update mechanism only supports TLS 1.2 protocol. Any older versions are not supported. Running the server with enforced TLS 1.1 protocol is done using the command shown in the code listing 4.13. The error returned in the message means *"Handshake protocol not within min/max boundaries."* [43].

Enforcing TLS version 1.2 works as expected and the firmware update is performed successfully. The command used to run server with restriction to only the TLS 1.2 protocol is shown in the code listing 4.15. The serial console log is shown in the attachment A.2.

Malicious Certification Authority was tested by generating a new server certificate using exactly the same workflow as described in the 4.6 section with exception of the server private key. The same private key was used both for the original server certificate for the previous tests and for this test. The new server certificate is signed by the new certification authority while the device firmware contains the certificate of the original authority. Even though the information listed in the certificate seems to be correct and the certificates contain the right attributes, the CA is different than the original one and therefore the server certificate is not trusted and the server authenticity verification fails with the message shown in 4.16. The alternate certificates are stored on the attached media in the *alternate_ca* folder.

Code listing 4.13 Run HTTPS server with only TLS 1.1 supported

```
openssl s_server -WWW -key server_key.pem -cert server_cert.pem \
    -port 443 -tls1_1
```

Code listing 4.14 Test result with TLS 1.1 enforced by the server (serial console log from the device)

```
E (126259) esp-tls-mbedtls: mbedtls_ssl_handshake returned -0x6E80
I (126269) esp-tls-mbedtls: Certificate verified.
E (126269) esp-tls: Failed to open new connection
E (126269) TRANSPORT_BASE: Failed to open a new connection
E (126279) HTTP_CLIENT: Connection failed, sock < 0
E (126279) ota.c: Error perform http request ESP_ERR_HTTP_CONNECT
```

Code listing 4.15 Run HTTPS server with only TLS 1.2 supported

```
openssl s_server -WWW -key server_key.pem -cert server_cert.pem \
    -port 443 -tls1_2
```

Code listing 4.16 Test result when the server certificate is signed by another certification authority

```
I (16239) esp-tls-mbedtls: Failed to verify peer certificate!
```

4.4.4.2 Firmware Signature

As described in the section 4.4.2, the firmware is signed automatically as part of the build process (in this work the configuration flag to enable this behavior has been set).

Apart from the positive test scenario it is important to verify also the negative scenarios which should fail. For the firmware signing feature verification the author has created two firmware files to test with. The test files are *image202.bin* which is not signed at all and *image203.bin* which is signed by another key than the one used for verification.

Signed image using the correct key results into the expected behavior.

Unsigned image caused an error as expected. The error message is shown in the code listing 4.17.

Signed image by another key results into an error. The error message is shown in the code listing 4.18. To create the firmware image signed by another certificate, the author has followed the same steps as described in the section 4.4.2. Before compiling with the new signing key the author recommends to delete the content of the build folder to avoid any issues caused by caching and potential reuse of the previously provided signing key.

Based on the serial console logs provided by the device, it is not possible to distinguish between the firmware image which doesn't have the signature at all and the image, which is signed by another key.

Author has also tested the newer version of the firmware signing using RSA, but unfortunately that version is only available on the newer hardware revisions, which the author didn't have available to perform such a test.

■ **Code listing 4.17** Test result with the unsigned firmware image

```
E (48669) secure_boot_v1: image has invalid signature version field
0xffffffff (image without a signature?)
E (48679) esp_image: Secure boot signature verification failed
I (48679) esp_image: Calculating simple hash to check for corruption...
W (48939) esp_image: image valid, signature bad
E (48949) ota.c: Image validation failed, image is corrupted
E (48949) ota.c: ESP_HTTPS_OTA upgrade failed 0x1503
```

■ **Code listing 4.18** Test result with a firmware image signed by another key

```
E (47690) secure_boot_v1: image has invalid signature version field
0xffffffff (image without a signature?)
E (47700) esp_image: Secure boot signature verification failed
I (47700) esp_image: Calculating simple hash to check for corruption...
W (47970) esp_image: image valid, signature bad
E (47970) ota.c: Image validation failed, image is corrupted
E (47970) ota.c: ESP_HTTPS_OTA upgrade failed 0x1503
```

4.4.4.3 Encryption Of The Firmware

Three different tests were performed to verify the firmware encryption feature. Firmware:

Encrypted with the correct RSA key which resulted into the correct behavior.

Not encrypted at all resulting into an error and the device logged the message shown in the code listing 4.19. The firmware file *image204.bin* was used for this test.

Encrypted with a different RSA key also resulting into an error and the device logged the message shown in the code listing 4.20. The firmware file *image205.bin* was used for this test. The procedure to encrypt the firmware and to generate the alternate RSA key is the same as described in the section 4.4.3.

■ **Code listing 4.19** Test result with a unnencrypted firmware (device console log)

```
E (9269) esp_encrypted_img: Magic Verification failed
E (9279) esp_https_ota: Decrypt callback failed -1
E (9279) esp_https_ota: Decryption of image header failed
E (9289) ota.c: Complete data was not received.
E (9299) ota.c: ESP_HTTPS_OTA upgrade failed
```

■ **Code listing 4.20** Test result with a firmware encrypted by a wrong RSA key (device console log)

```
E (9659) esp_encrypted_img: failed
  ! mbedtls_pk_decrypt returned -0x4100

E (9659) esp_encrypted_img: Unable to decipher GCM key
E (9669) esp_https_ota: Decrypt callback failed -1
E (9669) esp_https_ota: Decryption of image header failed
E (9679) ota.c: Complete data was not received.
E (9689) ota.c: ESP_HTTPS_OTA upgrade failed
```

Chapter 5

Summary and Discussion

5.1 Results of the Implementation and Tests

Table 5.1 contains the summary of all the performed tests. By default the Transport Layer Security (TLS) 1.2, firmware signing and firmware encryption have been used. Each line in the table represents different test conditions. The specific conditions which deviated from the default ones are described in the first column. The second column shows the firmware file name with the version, which was used for the test and which is stored on the attached media. The third column shows the expected result of the FOTA update attempt and the last column shows whether the test resulted into the expected behavior. Each of the tests was performed from a version 200 running on the device.

Table 5.1 Summary of the performed tests

Changed conditions	Firmware file	Expected result	Worked as expected
None	image201.bin	PASS	Yes
Server restricted to TLS 1.1	image201.bin	Fail	Yes
Server restricted to TLS 1.2	image201.bin	PASS	Yes
Server certificate signed by another CA	image201.bin	Fail	Yes
Binary not signed	image202.bin	Fail	Yes
Binary signed by another key	image203.bin	Fail	Yes
Binary not encrypted	image204.bin	Fail	Yes
Binary encrypted using another key	image205.bin	Fail	Yes

5.2 Security Recommendations

This section provides security recommendations for the secure firmware over the air update based on the analysis done in this thesis and also based on the authors experience and the challenges experienced during the work on this bachelor thesis:

Use new TLS protocol versions which means disabling the Transport Layer Security (TLS) 1.1 and older versions and enabling the TLS 1.2 and also the TLS 1.3 in case that it is supported.

Enable all security features provided by the platform (both hardware and software) such as the Secure Boot (v1 or better v2), flash encryption, firmware signing, or anti-rollback.

Use the latest revision of the hardware to avoid any limitations due to an insufficient hardware acceleration of the cybersecurity features.

Never provide development version to customers or anyone else. If some of the security features are not enabled in the beta version it is easier for the potential attacker to gain the access to the device, analyze the firmware and discover weaknesses of the system.

Use secure environment for firmware signing purpose and storing the private keys.

Generate private keys on a system with a high quality source of the entropy - outside of the build server.

5.3 Secure File Transfer

The new versions the ESP-IDF (later than 4.4) do not support the Transport Layer Security (TLS) version 1.1 or older by default and it is strongly recommended not use any of the disabled protocols and related ciphers.

The author attempted to enable the TLS 1.3 protocol support, but unfortunately the implementation done using the MbedTLS library was not yet fully done nor integrated into the ESP-IDF release 5.0, which was under development at the time of writing this thesis.

For the purpose of the proof of concept application the author has created certificates with ten years of validity. For the real application it would be sufficient to use shorter validity and provide a new Certification Authority (CA) certificate as part of the future firmware images to make a smooth transition between the old and new CA certificates.

5.4 IoT Device Authentication

The device authentication is out of the scope of this thesis and was only introduced in its theoretical part.

At the same time the firmware encryption was performed using asymmetric cryptography (public and private key pair). The public key was used to encrypt the firmware while the private key stored on the device was used for the firmware decryption.

In case that each device has its own private key (obtained from a hardware security module, created by the device, or uploaded by a manufacturer during the manufacturing process within the secure environment) and the matching public key is maintained by the firmware update server and it is used to sign the firmware update files individually for each device by the corresponding public key, then the firmware encryption mechanism ensures also the device identity verification. Although that any malicious device would be able to download the firmware binary, the binary would be encrypted and therefore the full firmware update process would not be completed.

The ideal device identity verification would confirm the device's authenticity before allowing the device to download the actual firmware package.

Chapter 6
Conclusion

The main goal of this thesis was to develop a proof of concept application which will perform the over the air firmware update using the ESP32 platform. This goal has been successfully achieved.

To achieve the goal the author had to analyze the general cybersecurity concepts and the features used for the IoT (Internet of Things) applications and specifically for the firmware over the air update feature. The next step was to understand the ESP32 platform, its hardware and software features and the related development tools. The result of the analysis helped the author to understand the differences between the ESP32 hardware variants and versions and their compatibility with the cybersecurity features.

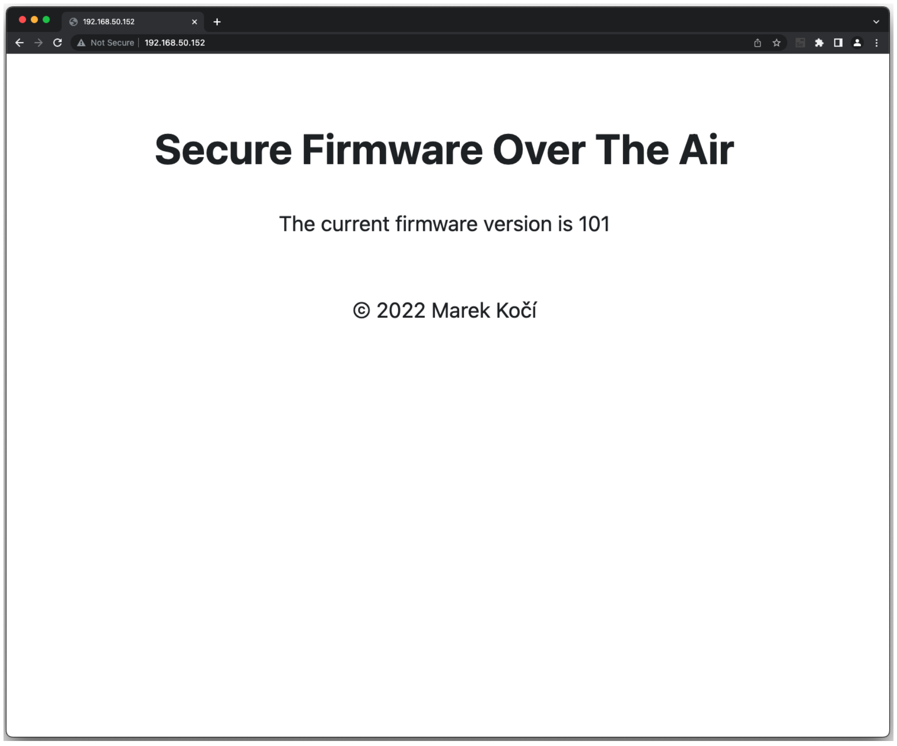
The provided solution implements the secure file transfer using the TLS protocol, the firmware image encryption and the firmware signing features. One of the main challenges was the implementation of the secure file transfer using the TLS protocol. The author attempted to use the TLS version 1.3, which at the time of writing this thesis had not yet been fully supported by the MbedTLS library providing the ESP32 platform with the TLS capabilities.

The author believes that the result of the thesis will be used as a source of inspiration and the starting point for developers implementing IoT solutions and mainly solutions based on the ESP32 platform.

The future works following up on this thesis could focus on the TLS 1.3 implementation, deeper analysis of the authentication of the devices or the most recent models of the ESP32 microcontrollers and their cybersecurity features.

Attachment A

Web Interface Screenshot and Serial Console Logs


192.168.50.152

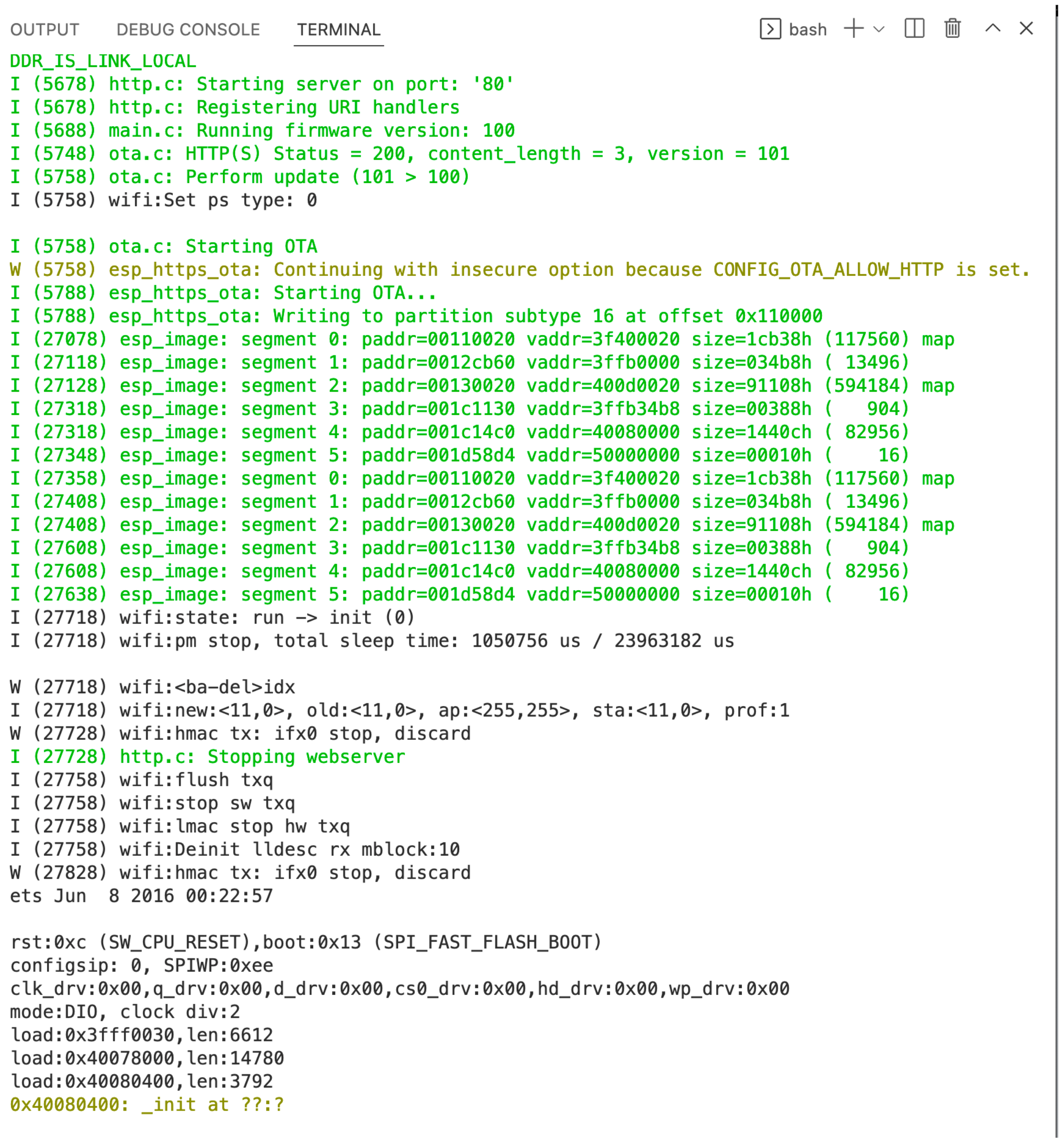
Not Secure | 192.168.50.152

Secure Firmware Over The Air

The current firmware version is 101

© 2022 Marek Kočí

Figure A.1 Screenshot of the device web interface
Screenshot of the device web interface. The webpage is accessible from a web serve running on the tested device. It displays the currently running version of the firmware. The FOTA starts at timestamp 5758.



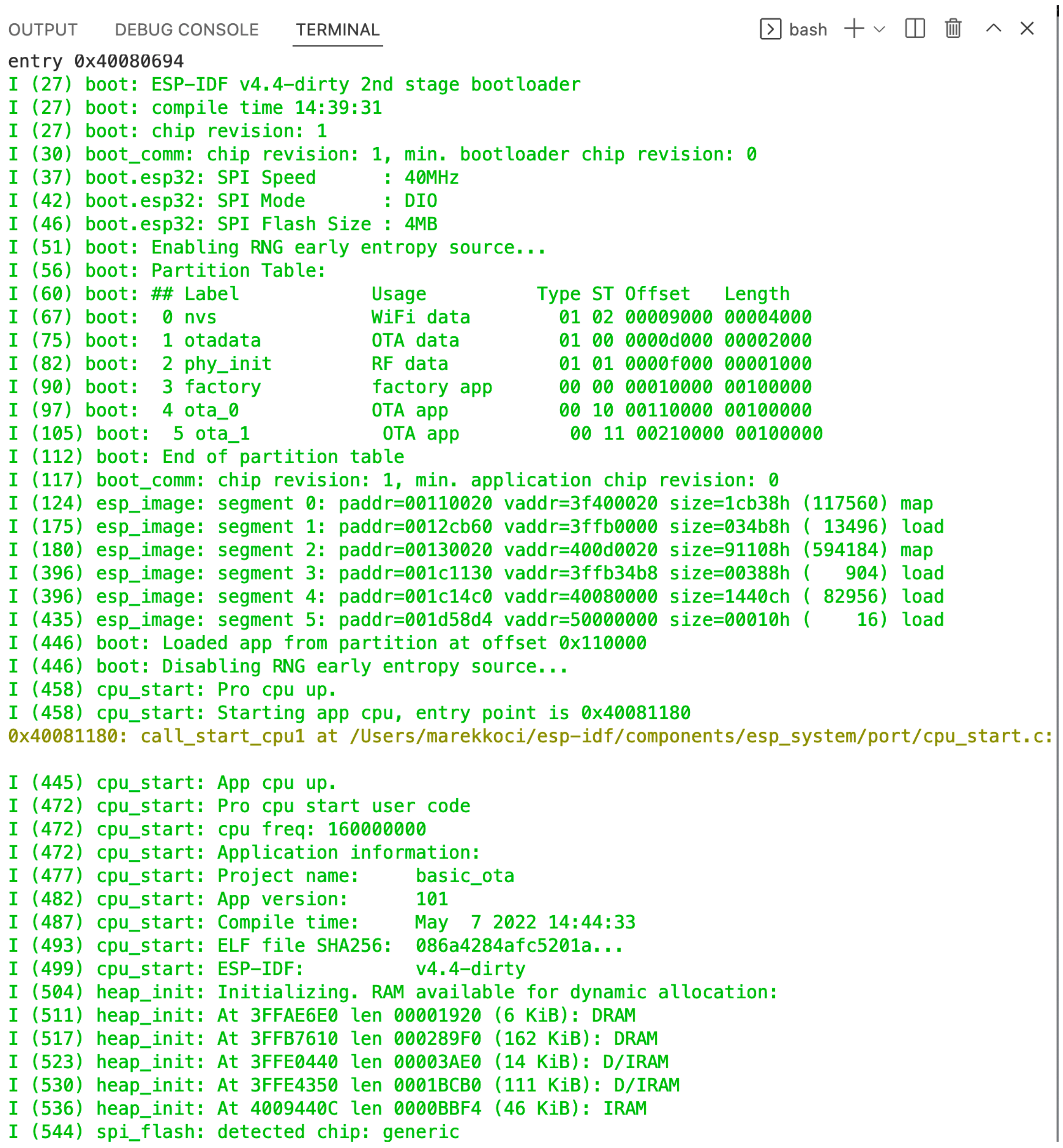
```
DDR_IS_LINK_LOCAL
I (5678) http.c: Starting server on port: '80'
I (5678) http.c: Registering URI handlers
I (5688) main.c: Running firmware version: 100
I (5748) ota.c: HTTP(S) Status = 200, content_length = 3, version = 101
I (5758) ota.c: Perform update (101 > 100)
I (5758) wifi:Set ps type: 0

I (5758) ota.c: Starting OTA
W (5758) esp_https_ota: Continuing with insecure option because CONFIG_OTA_ALLOW_HTTP is set.
I (5788) esp_https_ota: Starting OTA...
I (5788) esp_https_ota: Writing to partition subtype 16 at offset 0x110000
I (27078) esp_image: segment 0: paddr=00110020 vaddr=3f400020 size=1cb38h (117560) map
I (27118) esp_image: segment 1: paddr=0012cb60 vaddr=3ffb0000 size=034b8h ( 13496)
I (27128) esp_image: segment 2: paddr=00130020 vaddr=400d0020 size=91108h (594184) map
I (27318) esp_image: segment 3: paddr=001c1130 vaddr=3ffb34b8 size=00388h (   904)
I (27318) esp_image: segment 4: paddr=001c14c0 vaddr=40080000 size=1440ch ( 82956)
I (27348) esp_image: segment 5: paddr=001d58d4 vaddr=50000000 size=00010h (    16)
I (27358) esp_image: segment 0: paddr=00110020 vaddr=3f400020 size=1cb38h (117560) map
I (27408) esp_image: segment 1: paddr=0012cb60 vaddr=3ffb0000 size=034b8h ( 13496)
I (27408) esp_image: segment 2: paddr=00130020 vaddr=400d0020 size=91108h (594184) map
I (27608) esp_image: segment 3: paddr=001c1130 vaddr=3ffb34b8 size=00388h (   904)
I (27608) esp_image: segment 4: paddr=001c14c0 vaddr=40080000 size=1440ch ( 82956)
I (27638) esp_image: segment 5: paddr=001d58d4 vaddr=50000000 size=00010h (    16)
I (27718) wifi:state: run -> init (0)
I (27718) wifi:pm stop, total sleep time: 1050756 us / 23963182 us

W (27718) wifi:<ba-del>idx
I (27718) wifi:new:<11,0>, old:<11,0>, ap:<255,255>, sta:<11,0>, prof:1
W (27728) wifi:hmac tx: ifx0 stop, discard
I (27728) http.c: Stopping webserver
I (27758) wifi:flush txq
I (27758) wifi:stop sw txq
I (27758) wifi:lmac stop hw txq
I (27758) wifi:Deinit lldesc rx mblock:10
W (27828) wifi:hmac tx: ifx0 stop, discard
ets Jun  8 2016 00:22:57

rst:0xc (SW_CPU_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:6612
load:0x40078000,len:14780
load:0x40080400,len:3792
0x40080400: _init at ??:?
```

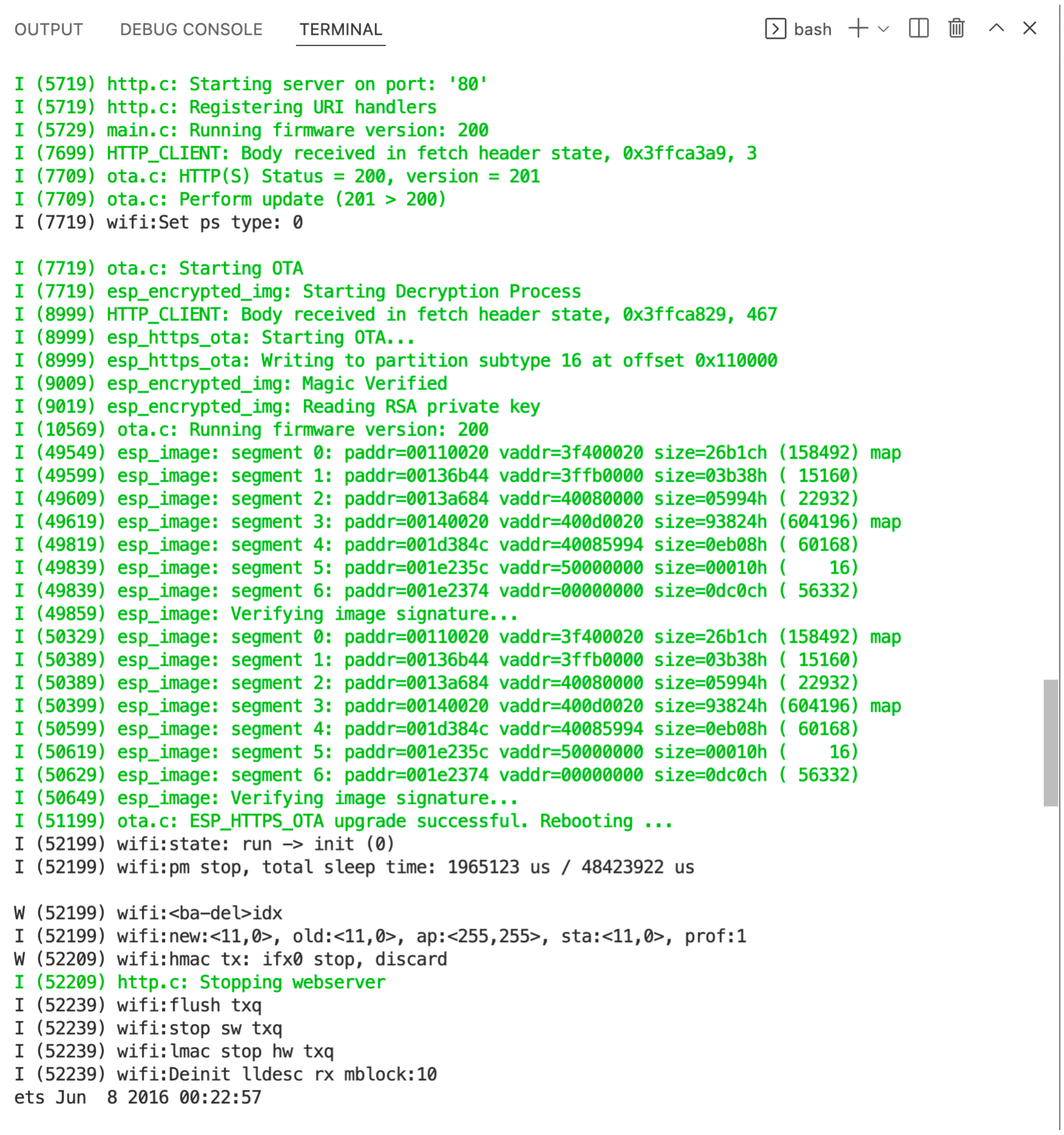
Figure A.2 Serial console log from the device (basic implementation - first part)
Serial console log from the device (basic implementation - first part). The log is showing a successful update of the firmware from the version 100 to 101. The device started using a new firmware (timestamp 482)



```
entry 0x40080694
I (27) boot: ESP-IDF v4.4-dirty 2nd stage bootloader
I (27) boot: compile time 14:39:31
I (27) boot: chip revision: 1
I (30) boot_comm: chip revision: 1, min. bootloader chip revision: 0
I (37) boot.esp32: SPI Speed      : 40MHz
I (42) boot.esp32: SPI Mode       : DIO
I (46) boot.esp32: SPI Flash Size : 4MB
I (51) boot: Enabling RNG early entropy source...
I (56) boot: Partition Table:
I (60) boot: ## Label            Usage          Type ST Offset   Length
I (67) boot:  0 nvs              WiFi data        01 02 00009000 00004000
I (75) boot:  1 otadata          OTA data         01 00 0000d000 00002000
I (82) boot:  2 phy_init         RF data          01 01 0000f000 00001000
I (90) boot:  3 factory          factory app      00 00 00010000 00100000
I (97) boot:  4 ota_0            OTA app          00 10 00110000 00100000
I (105) boot:  5 ota_1            OTA app          00 11 00210000 00100000
I (112) boot: End of partition table
I (117) boot_comm: chip revision: 1, min. application chip revision: 0
I (124) esp_image: segment 0: paddr=00110020 vaddr=3f400020 size=1cb38h (117560) map
I (175) esp_image: segment 1: paddr=0012cb60 vaddr=3ffb0000 size=034b8h ( 13496) load
I (180) esp_image: segment 2: paddr=00130020 vaddr=400d0020 size=91108h (594184) map
I (396) esp_image: segment 3: paddr=001c1130 vaddr=3ffb34b8 size=00388h (   904) load
I (396) esp_image: segment 4: paddr=001c14c0 vaddr=40080000 size=1440ch ( 82956) load
I (435) esp_image: segment 5: paddr=001d58d4 vaddr=50000000 size=00010h (    16) load
I (446) boot: Loaded app from partition at offset 0x110000
I (446) boot: Disabling RNG early entropy source...
I (458) cpu_start: Pro cpu up.
I (458) cpu_start: Starting app cpu, entry point is 0x40081180
0x40081180: call_start_cpu1 at /Users/marekkoci/esp-idf/components/esp_system/port/cpu_start.c:

I (445) cpu_start: App cpu up.
I (472) cpu_start: Pro cpu start user code
I (472) cpu_start: cpu freq: 160000000
I (472) cpu_start: Application information:
I (477) cpu_start: Project name:     basic_ota
I (482) cpu_start: App version:      101
I (487) cpu_start: Compile time:     May  7 2022 14:44:33
I (493) cpu_start: ELF file SHA256:  086a4284afc5201a...
I (499) cpu_start: ESP-IDF:          v4.4-dirty
I (504) heap_init: Initializing. RAM available for dynamic allocation:
I (511) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
I (517) heap_init: At 3FFB7610 len 000289F0 (162 KiB): DRAM
I (523) heap_init: At 3FFE0440 len 00003AE0 (14 KiB): D/IRAM
I (530) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (536) heap_init: At 4009440C len 0000BBF4 (46 KiB): IRAM
I (544) spi_flash: detected chip: generic
```

Figure A.3 Serial console log from the device (basic implementation - second part)
Serial console log from the device (basic implementation - second part). The log is showing a successful update of the firmware from the version 100 to 101.

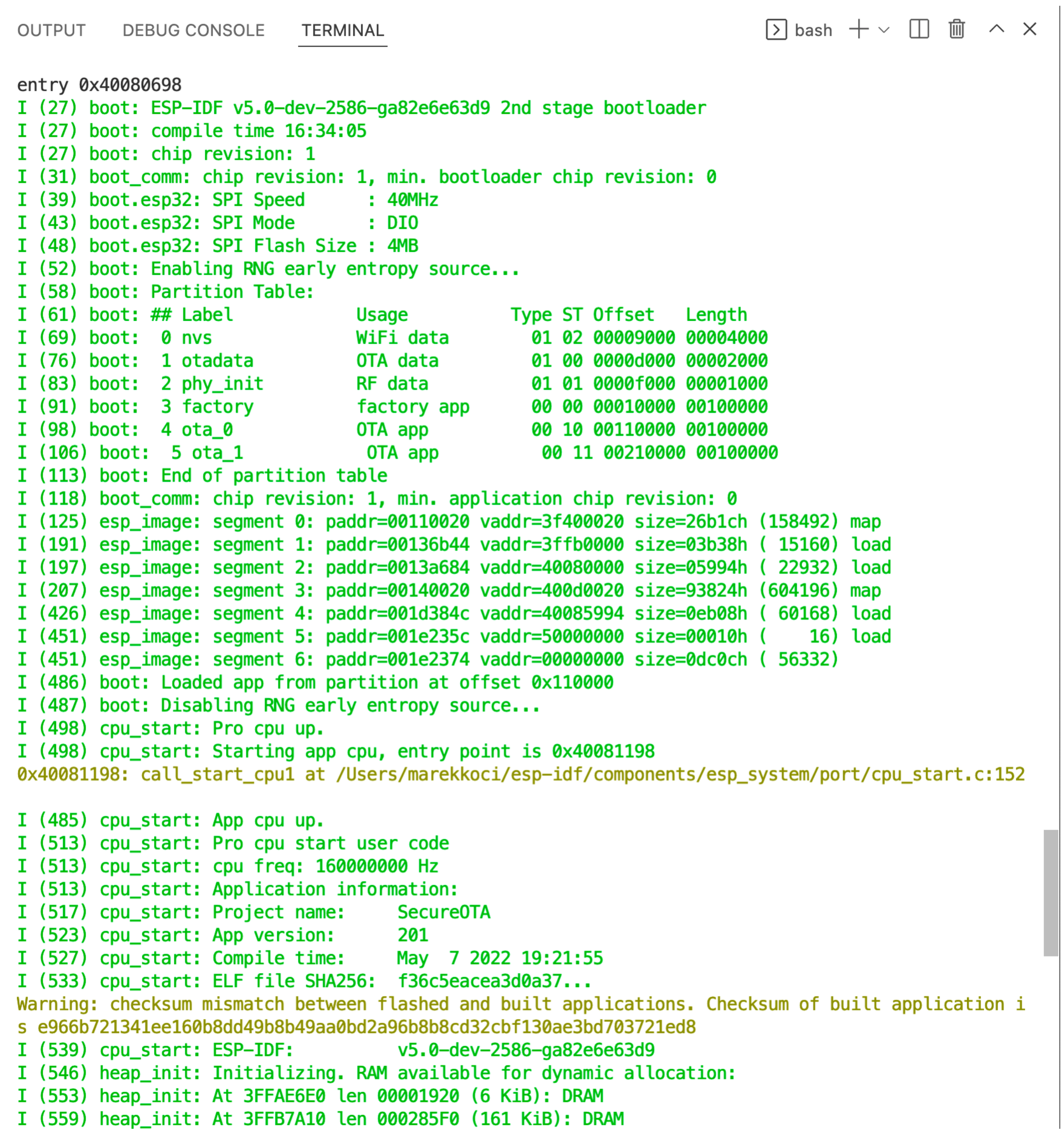


```
I (5719) http.c: Starting server on port: '80'
I (5719) http.c: Registering URI handlers
I (5729) main.c: Running firmware version: 200
I (7699) HTTP_CLIENT: Body received in fetch header state, 0x3ffca3a9, 3
I (7709) ota.c: HTTP(S) Status = 200, version = 201
I (7709) ota.c: Perform update (201 > 200)
I (7719) wifi:Set ps type: 0

I (7719) ota.c: Starting OTA
I (7719) esp_encrypted_img: Starting Decryption Process
I (8999) HTTP_CLIENT: Body received in fetch header state, 0x3ffca829, 467
I (8999) esp_https_ota: Starting OTA...
I (8999) esp_https_ota: Writing to partition subtype 16 at offset 0x110000
I (9009) esp_encrypted_img: Magic Verified
I (9019) esp_encrypted_img: Reading RSA private key
I (10569) ota.c: Running firmware version: 200
I (49549) esp_image: segment 0: paddr=00110020 vaddr=3f400020 size=26b1ch (158492) map
I (49599) esp_image: segment 1: paddr=00136b44 vaddr=3ffb0000 size=03b38h ( 15160)
I (49609) esp_image: segment 2: paddr=0013a684 vaddr=40080000 size=05994h ( 22932)
I (49619) esp_image: segment 3: paddr=00140020 vaddr=400d0020 size=93824h (604196) map
I (49819) esp_image: segment 4: paddr=001d384c vaddr=40085994 size=0eb08h ( 60168)
I (49839) esp_image: segment 5: paddr=001e235c vaddr=50000000 size=00010h (    16)
I (49839) esp_image: segment 6: paddr=001e2374 vaddr=00000000 size=0dc0ch ( 56332)
I (49859) esp_image: Verifying image signature...
I (50329) esp_image: segment 0: paddr=00110020 vaddr=3f400020 size=26b1ch (158492) map
I (50389) esp_image: segment 1: paddr=00136b44 vaddr=3ffb0000 size=03b38h ( 15160)
I (50389) esp_image: segment 2: paddr=0013a684 vaddr=40080000 size=05994h ( 22932)
I (50399) esp_image: segment 3: paddr=00140020 vaddr=400d0020 size=93824h (604196) map
I (50599) esp_image: segment 4: paddr=001d384c vaddr=40085994 size=0eb08h ( 60168)
I (50619) esp_image: segment 5: paddr=001e235c vaddr=50000000 size=00010h (    16)
I (50629) esp_image: segment 6: paddr=001e2374 vaddr=00000000 size=0dc0ch ( 56332)
I (50649) esp_image: Verifying image signature...
I (51199) ota.c: ESP_HTTPS_OTA upgrade successful. Rebooting ...
I (52199) wifi:state: run -> init (0)
I (52199) wifi:pm stop, total sleep time: 1965123 us / 48423922 us

W (52199) wifi:<ba-del>idx
I (52199) wifi:new:<11,0>, old:<11,0>, ap:<255,255>, sta:<11,0>, prof:1
W (52209) wifi:hmac tx: ifx0 stop, discard
I (52209) http.c: Stopping webserver
I (52239) wifi:flush txq
I (52239) wifi:stop sw txq
I (52239) wifi:lmac stop hw txq
I (52239) wifi:Deinit lldesc rx mblock:10
ets Jun  8 2016 00:22:57
```

Figure A.4 Serial console log from the device (secure implementation - first part)
Serial console log from the device (secure implementation - first part). The log is showing a successful update of the firmware from the version 200 to 201. The FOTA starts at timestamp 7709.



```
entry 0x40080698
I (27) boot: ESP-IDF v5.0-dev-2586-ga82e6e63d9 2nd stage bootloader
I (27) boot: compile time 16:34:05
I (27) boot: chip revision: 1
I (31) boot_comm: chip revision: 1, min. bootloader chip revision: 0
I (39) boot.esp32: SPI Speed      : 40MHz
I (43) boot.esp32: SPI Mode       : DIO
I (48) boot.esp32: SPI Flash Size : 4MB
I (52) boot: Enabling RNG early entropy source...
I (58) boot: Partition Table:
I (61) boot: ## Label            Usage          Type ST Offset   Length
I (69) boot:  0 nvs              WiFi data        01 02 00009000 00004000
I (76) boot:  1 otadata          OTA data         01 00 0000d000 00002000
I (83) boot:  2 phy_init         RF data          01 01 0000f000 00001000
I (91) boot:  3 factory          factory app      00 00 00010000 00100000
I (98) boot:  4 ota_0            OTA app          00 10 00110000 00100000
I (106) boot:  5 ota_1            OTA app          00 11 00210000 00100000
I (113) boot: End of partition table
I (118) boot_comm: chip revision: 1, min. application chip revision: 0
I (125) esp_image: segment 0: paddr=00110020 vaddr=3f400020 size=26b1ch (158492) map
I (191) esp_image: segment 1: paddr=00136b44 vaddr=3ffb0000 size=03b38h ( 15160) load
I (197) esp_image: segment 2: paddr=0013a684 vaddr=40080000 size=05994h ( 22932) load
I (207) esp_image: segment 3: paddr=00140020 vaddr=400d0020 size=93824h (604196) map
I (426) esp_image: segment 4: paddr=001d384c vaddr=40085994 size=0eb08h ( 60168) load
I (451) esp_image: segment 5: paddr=001e235c vaddr=50000000 size=00010h (    16) load
I (451) esp_image: segment 6: paddr=001e2374 vaddr=00000000 size=0dc0ch ( 56332)
I (486) boot: Loaded app from partition at offset 0x110000
I (487) boot: Disabling RNG early entropy source...
I (498) cpu_start: Pro cpu up.
I (498) cpu_start: Starting app cpu, entry point is 0x40081198
0x40081198: call_start_cpu1 at /Users/marekkoci/esp-idf/components/esp_system/port/cpu_start.c:152

I (485) cpu_start: App cpu up.
I (513) cpu_start: Pro cpu start user code
I (513) cpu_start: cpu freq: 160000000 Hz
I (513) cpu_start: Application information:
I (517) cpu_start: Project name:     SecureOTA
I (523) cpu_start: App version:      201
I (527) cpu_start: Compile time:     May  7 2022 19:21:55
I (533) cpu_start: ELF file SHA256:  f36c5eacea3d0a37...
Warning: checksum mismatch between flashed and built applications. Checksum of built application i
s e966b721341ee160b8dd49b8b49aa0bd2a96b8b8cd32cbf130ae3bd703721ed8
I (539) cpu_start: ESP-IDF:          v5.0-dev-2586-ga82e6e63d9
I (546) heap_init: Initializing. RAM available for dynamic allocation:
I (553) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
I (559) heap_init: At 3FFB7A10 len 000285F0 (161 KiB): DRAM
```

Figure A.5 Serial console log from the device (secure implementation - second part)
Serial console log from the device (secure implementation - second part). The log is showing a successful update of the firmware from the version 200 to 201. The device started using a new firmware (timestamp 523).

Bibliography

1. GILLIS, Alexander. *What is internet of things (IoT)?* [Online]. 2021. Available also from: `https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT`. Last accessed 2nd April 2022.
2. RAY, Partha Pratim. An introduction to dew computing: Definition, concept and implications. *IEEE Access*. 2018, vol. 6, pp. 723–737. Available from DOI: `10.1109/ACCESS.2017.2775042`.
3. DUGGAL, Nikita. *What Are IoT Devices : Definition, Types, and 5 Most Popular Ones for 2022* [online]. 2022. Available also from: `https://www.simplilearn.com/iot-devices-article`. Last accessed 3rd April 2022.
4. GREGERSEN, Carsten. *Complete Guide to IoT Protocols & Standards In 2021* [online]. 2020. Available also from: `https://www.nabto.com/guide-iot-protocols-standards`. Last accessed 3rd April 2022.
5. CHAKRABORTY, Kuntal. *Firmware* [online]. 2021. Available also from: `https://www.techopedia.com/definition/2137/firmware`. Last accessed 7th April 2022.
6. LITINGTUN, Shawn. *Introduction to Embedded Firmware Development* [online]. 2021. Available also from: `https://predictabledesigns.com/introduction-to-embedded-firmware-development/`. Last accessed 7th April 2022.
7. ECLYPSIUM. *Enterprise Best Practices for Firmware Updates* [online]. 2020. Available also from: `https://securityboulevard.com/2020/04/enterprise-best-practices-for-firmware-updates/`. Last accessed 7th April 2022.
8. RYABKOV, Deomid. *Updating firmware reliably* [online]. 2016. Available also from: `https://www.embedded.com/updating-firmware-reliably/`. Last accessed 10th April 2022.
9. XJTAG. *What is JTAG?* [Online]. 2022. Available also from: `https://www.xjtag.com/about-jtag/what-is-jtag/`. Last accessed 8th April 2022.
10. EL JAOUHARI, Saad; BOUVET, Eric. Secure firmware Over-The-Air updates for IoT: Survey, challenges, and discussions. *Internet of Things*. 2022, vol. 18, p. 100508. ISSN 2542-6605. Available from DOI: `https://doi.org/10.1016/j.iot.2022.100508`.
11. CLOUDFLARE. *What is encryption? — Types of encryption* [online]. [N.d.]. Available also from: `https://www.cloudflare.com/en-gb/learning/ssl/what-is-encryption/`. Last accessed 15th April 2022.
12. CRANE, Casey. *What Is a Hash Function in Cryptography? A Beginner's Guide* [online]. 2021. Available also from: `https://www.thesslstore.com/blog/what-is-a-hash-function-in-cryptography-a-beginners-guide/`. Last accessed 15th April 2022.

13. FRUHLINGER, Josh. *What is PKI? And how it secures just about everything online* [online]. 2020. Available also from: `https://www.csoonline.com/article/3400836/what-is-pki-and-how-it-secures-just-about-everything-online.html`. Last accessed 15th April 2022.
14. TEAM, SSL.com Support. *What Is a Certificate Authority (CA)?* [Online]. 2021. Available also from: `https://www.ssl.com/faqs/what-is-a-certificate-authority/`. Last accessed 15th April 2022.
15. RAVNEET KAUR, Amandeep Kaur. Digital Signature. In: *2012 International Conference on Computing Sciences*. Phagwara, India: IEEE, 2012.
16. ACDX. *Digital Signature diagram* [online]. 2012. Available also from: `https://commons.wikimedia.org/w/index.php?curid=5251510`. CC BY-SA 3.0, Last accessed 15th April 2022.
17. LAKE, Josh. *What is TLS and how does it work?* [Online]. 2021. Available also from: `https://www.comparitech.com/blog/information-security/tls-encryption/`. Last accessed 16th April 2022.
18. SSL2BUY. *Expired Root Certificates: The Main Reason to Weaken IoT Devices* [online]. 2022. Available also from: `https://www.ssl2buy.com/wiki/expired-root-certificates-main-reason-to-weaken-iot-devices`. Last accessed 23rd April 2022.
19. REDHAT. *What is a CVE?* [Online]. 2020. Available also from: `https://www.redhat.com/en/topics/security/what-is-cve`. Last accessed 17th April 2022.
20. LEE, Vincent. *CVE-2021-22909 – DIGGING INTO A UBIQUITI FIRMWARE UPDATE BUG* [online]. 2021. Available also from: `https://www.zerodayinitiative.com/blog/2021/5/24/cve-2021-22909-digging-into-a-ubiquiti-firmware-update-bug`. Last accessed 17th April 2022.
21. OWASP. *OWASP Internet of Things Project* [online]. 2018. Available also from: `https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project`. Last accessed 17th April 2022.
22. CAPABILITIES, Technical; PATCHING EXPECTATIONS WORKING GROUP, NTIA. *Voluntary Framework for Enhancing Update Process Security* [online]. 2017. Available also from: `https://www.ntia.doc.gov/files/ntia/publications/ntia_iot_capabilities_oct31.pdf`. Last accessed 17th April 2022.
23. ESPRESSIF. *Secure Boot V2* [online]. 2022. Available also from: `https://docs.espressif.com/projects/esp-idf/en/v4.3.2/esp32/security/secure-boot-v2.html`. Last accessed 17th April 2022.
24. BERDY, Nicole. *IoT device authentication options* [online]. 2018. Available also from: `https://azure.microsoft.com/es-es/blog/iot-device-authentication-options/`. Last accessed 23rd April 2022.
25. BORGINI, Julia. *IoT device authentication methods that increase security* [online]. 2021. Available also from: `https://www.techtarget.com/iotagenda/tip/IoT-device-authentication-methods-that-increase-security`. Last accessed 18th April 2022.
26. NIST. *NIST Gloassary - Ephemeral Key* [online]. 2022. Available also from: `https://csrc.nist.gov/glossary/term/ephemeral_key`. Last accessed 21st April 2022.
27. SYSTEMS, Espressif. *ESP32* [online]. 2022. Available also from: `https://www.espressif.com/en/products/socs/esp32`. Last accessed 24th April 2022.
28. *IoT ESP-WROOM-32 2.4GHz Dual-Mode WiFi+Bluetooth rev.1, CP2102* [online]. 2022. Available also from: `https://www.laskakit.cz/iot-esp-32s-2-4ghz-dual-mode-wifi-bluetooth-rev-1--cp2102`. Last accessed 9th April 2022.

29. SYSTEMS, Espressif. *ESP32 Series of Modules* [online]. 2022. Available also from: `https://www.espressif.com/en/products/modules/esp32`. Last accessed 24th April 2022.

30. SYSTEMS, Espressif. *Modules* [online]. 2022. Available also from: `https://www.espressif.com/en/products/modules`. Last accessed 24th April 2022.

31. SYSTEMS, Espressif. *ESP32-S2-WROVER ESP32-S2-WROVER-I Datasheet* [online]. 2022. Available also from: `https://www.espressif.com/sites/default/files/documentation/esp32-s2-wrover_esp32-s2-wrover-i_datasheet_en.pdf`. Last accessed 24th April 2022.

32. SYSTEMS, Espressif. *ESP32-C3-WROOM-02 ESP32-C3-WROOM-02U Datasheet* [online]. 2022. Available also from: `https://www.espressif.com/sites/default/files/documentation/esp32-c3-wroom-02_datasheet_en.pdf`. Last accessed 24th April 2022.

33. SYSTEMS, Espressif. *ESP32-WROOM-32E ESP32-WROOM-32UE Datasheet* [online]. 2022. Available also from: `https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf`. Last accessed 24th April 2022.

34. WILLIAMS, Elliot. *ESP32 HANDS-ON: AWESOME PROMISE* [online]. 2016. Available also from: `https://hackaday.com/2016/09/15/esp32-hands-on-awesome-promise/`. Last accessed 9th April 2022.

35. SYSTEMS, Espressif. *ESP32 Technical Reference Manual* [online]. 2021. Available also from: `https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf`. Last accessed 29th April 2022.

36. SYSTEMS, Espressif. *Random Number Generation* [online]. 2022. Available also from: `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/random.html`. Last accessed 29th April 2022.

37. SYSTEMS, Espressif. *Flash Encryption* [online]. 2022. Available also from: `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/flash-encryption.html`. Last accessed 29th April 2022.

38. SYSTEMS, Espressif. *Migration of Protocol Components to ESP-IDF 5.0* [online]. 2022. Available also from: `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/migration-guides/protocols.html`. Last accessed 6th May 2022.

39. SYSTEMS, Espressif. *Mbed TLS* [online]. 2022. Available also from: `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/protocols/mbedtls.html`. Last accessed 6th May 2022.

40. SYSTEMS, Espressif. *Over The Air Updates (OTA)* [online]. 2022. Available also from: `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/ota.html`. Last accessed 1st May 2022.

41. SYSTEMS, Espressif. *ESP-IDF Official IoT Development Framework* [online]. 2022. Available also from: `https://www.espressif.com/en/products/sdks/esp-idf`. Last accessed 29th April 2022.

42. SYSTEMS, Espressif. *ESP-TLS* [online]. 2022. Available also from: `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/protocols/esp_tls.html`. Last accessed 29th April 2022.

43. MBED, arm. *ssl.h File Reference* [online]. 2015. Available also from: `https://tls.mbed.org/api/ssl_8h.html#ae8285bd18c5cbf25d9a9b6780f335081`. Last accessed 8th May 2022.

Contents of enclosed media

/
- readme.txt — basic information about the thesis
- source
  - advanced — source code of the secure implementation
    - main — folder with the main source code files
    - managed_components — source code of the firmware encryption component
    - rsa_key — RSA key for the firmware encryption
    - server_certs — folder with Certification Authority certificates
    - seure_boot_signing_key.pem — ECDSA encryption key for the firmware signing
  - basic — source code of the non-secure implementation
  - tex — thesis source files in the LaTeXformat
- tests — test files used to verify the firmware
  - 201 — working binary, keys and certificates
  - 202_no_signing — unsigned binary
  - 203_different_signing_key — binary and alternate signing key
  - 204_no_encryption — unencrypted binary
  - 205_different_encryption_key — binary and alternate encryption key
  - alternate_ca — CA certificate used for the TLS test
- pdf — textual part
  - kocimar8_2022.pdf — thesis text in the pdf format