Politecnico di Torino

Cybersecurity for Embedded Systems
01UDNOV

Master's Degree in Computer Engineering

Secure OTA Update

Project Report

Candidates:
Daniele Berretta (305426)
Andrea Cencio (288687)
Mohamed Shehab (289509)

Referents:
Prof. Paolo Prinetto
Dr. Matteo Fornero
Dr. Vahid Eftekhari

Contents

List of Figures

Abstract

This document reports the development of the secure OTA update project, for the course Cybersecurity for Embedded Systems. It describes how this OTA protocol was develop and then designed with a look at possible and future developments, starting from a simpler working base. By Reading this report, we will review some concepts for wireless update (using an internet connection) and safety concepts that we want to guarantee and apply to our protocol. The main actors in this protocol are a common PC, used to upload the new binary file to be downloaded to the device, the internet connection to the exchange server (in this case we choose Google Drive as a prototype server to check that our mechanism works correctly), and the boards used for IoT communications (ESP32s).

At the end of the report you will be able to implement step by step a working update protocol for devices that use ESP32, being aware of the secure mechanism used. At the links below you can find the presentation slides with video demos and the source code of the project:

- Code: `https://drive.google.com/drive/folders/12X8qwaMzzthq3UGNoWMIBG-sxKIKymrK`
- Presentation: `https://1drv.ms/p/s!AvgcmezmQyo_k142UmFzocKHarcV?e=8MCfQo`

CHAPTER 1

Introduction

An over-the-air (OTA) update is the wireless delivery of new software, firmware, or other data to mobile devices. Wireless carriers and original equipment manufacturers (OEMs) typically use OTA updates to deploy firmware and configure phones to use on their networks over Wi-Fi or mobile broadband. The initialization of a newly purchased phone, for example, requires an Over-The-Air update. With the rise of smartphones, tablets and internet of things (IoT) devices, carriers and manufacturers have turned to different Over-The-Air update architecture methods to deploy new operating systems (OSes) to these devices.

1.1 Goal

The goal of this project is to define and implement an inter-cluster protocol where nodes verify the binary of other nodes by means of co-operative hash checking. Defining/Implementation of mechanism to receive/execute the updates.

1.2 Constraints

The target design/implementation is for IoT devices that are communication within cluster of nodes,a real time operating system,such as freertos shall run in our boards.

1.3 Contextualization

Imagine a situation in which a company has several IoT applications scattered around the factory, or a home automation where we have some board for different applications. We need to update them, but we want to update all without having to manually reprogram them one by one. To do this we need a OTA protocol that also ensure that this update is completed correctly and in a secure way, reducing the possibility of attacks and file corruptions.

CHAPTER 2

Background

In this chapter we introduce some concepts and tools used for the project development.

2.1 Security aspect

In order to be sure that the OTA procedure is completed correctly and no one or nothing can change the update file, we want to maintain security concepts:

- **Protecting the update Integrity**: We want to ensure that the source file sent is identical to the one received
- **Protecting the update Authentication**: We want to make sure that the update arrives from the original OEM
- **Protecting the update Authenticity**: We want to make sure that the update is sent to the required device
- **Protecting the update Confidentiality**: We want that no one is able to read the data during transmission

2.2 Hash function

To ensure the **integrity** of this procedure and update file we use an Hash function.
Hash is a function that can be used to map data of arbitrary size to fixed-size values. The values returned by an hash function are called hash digests.
Digest is a unique result for a given input, there cannot be (within a certain precision) two identical results starting from two different inputs.
Using these considerations, if two digests are identical, the two inputs files are necessarily the same, and we use this principle to ensure that the file we send for the update is the same as the one that is received and subsequently installed.

Figure 2.1: Hash Function

2.3 HTTP/HTTPS

For the communication between server and client we used HTTP as communication protocol. HTTP is a request/response protocol that match our needs for the download of the new update file. In particular we want to access to the SSL layer (Secure Socket Layer), and this is possible using HTTPS protocol. SSL is a standard technology that ensures the security of an Internet connection session and protects sensitive data exchanged between the two systems by preventing cybercriminals from reading and modifying the information transferred (known as MITM attack). Below there is a summary of the exchange of asymmetric and symmetric keys and certified.

- Client (IoT) Request secure connection
- Server receives request and generates two asymmetric keys: one public and one private
- Server sends the public key to the IoT device with the Certificate Authority signature
- IoT device keeps public key and check the CA signature
- If IoT device trust the CA, generates two identical encryption keys (session keys)
- IoT device encrypts session key using public key and send the encrypted key to the server
- Server decrypts the message with his private key
- The two parties have an identical key and can communicate securely

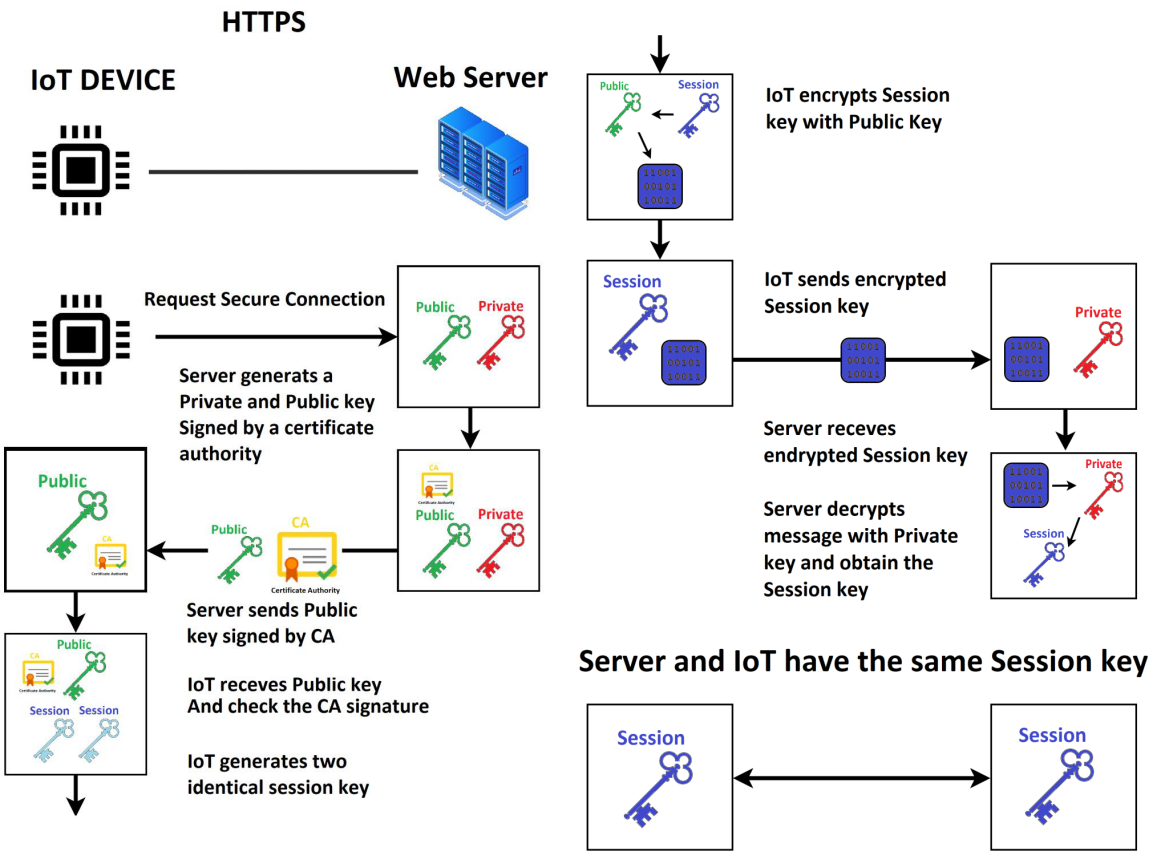


Figure 2.2: HTTPS Protocol Schematic

2.3.1 Certificate Authority

A certificate authority (CA) is a third party actor or organization that acts to validate the identities (such as websites, server our case) and bind them to cryptographic keys through the digital certificates.

2.4 FreeRTOS

FreeRTOS is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of IoT libraries suitable for use across all industry sectors. FreeRTOS is built with an emphasis on reliability and ease of use.
`https://www.freertos.org/`

2.5 Preliminary

In **appendice B** is possible to see preliminary conceptual diagram and first simulation, using OpenSSL.
Appendice B

CHAPTER 3

Implementation Overview

For the development of the OTA protocol we decided to use a **Google Cloud server**, and **ESP32** board for the IoT connectivity.
The IoT update travel through the server and is downloaded from the device under a specific command. For the upload of the binary file on the server we can easy go on the correct drive page, linked to the related IoT and copy there.
Once the binary file has been updated, the OTA procedure can start. Upon request, the IoT downloads the binary file from the server using the SSL channel and trust it, thanks to the hard-coded Certificate which is embedded in each device.

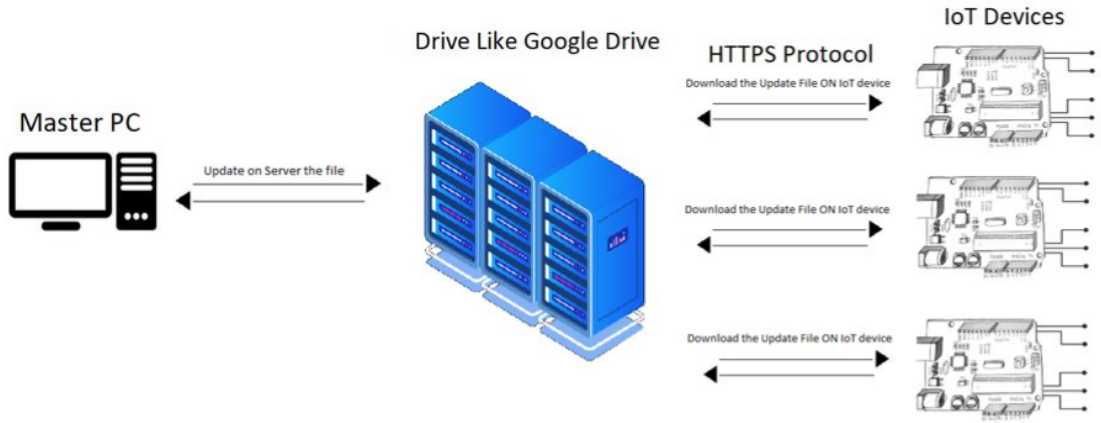


Figure 3.1: Summary Scheme

If all the procedure and control goes without problem, the downloaded binary file is flashed on one of the two free partitions of the device.
The memory of ESP32 is partitioned in 3 part: one, the P0, is the back-up partition where there is the first flashed version. The other two P1 and P2 are for future OTA updates. First we boot on P1 then P2 and then return to P1, and so on, we toggle the area at every update.

CHAPTER 4

Implementation Details

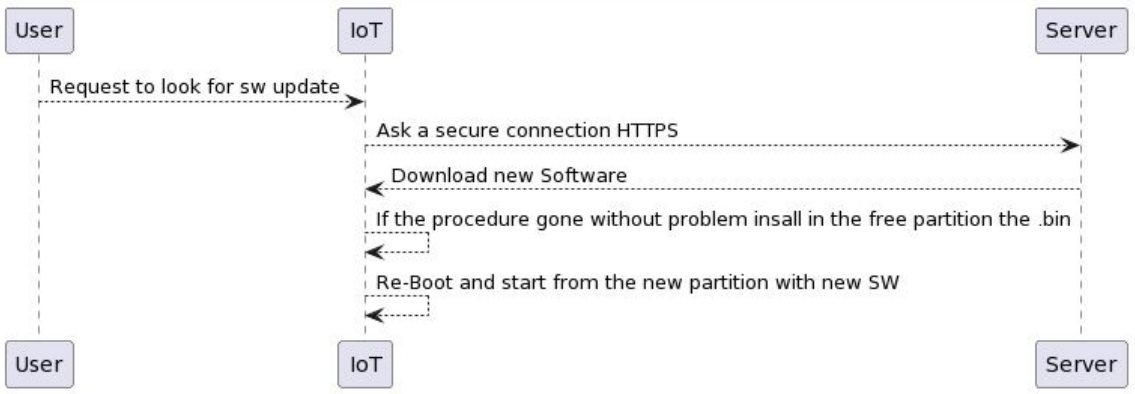


Figure 4.1: Process Diagram

In this section we can go on the implementation detail of this protocol, and the operational steps to set up the machine and work with ESP32 board. As a remind, this is the folder project:
`https://drive.google.com/drive/folders/12X8qwaMzzthq3UGNoWMIBG-sxKIKymrK`

4.1 Include

4.1.1 Wireless Connectivity

Wireless connectivity is required for the OTA update procedure. One of the key point for the procedure, is the usage of internet connectivity.
This could gives us potential benefits: we can run the update without connection to a local network, the devices can communicate even if they are not within the same intranet, but on the same time we could be more sensitive to cyber attacks.
As mentioned, we use an ESP32 board, that is equipped with Wi-Fi module and good library well explained to use it. It also has good libraries for wireless communication and secure boot.

- `esp_wifi.h` - wifi connectivity
- `nvs_flash.h` - wifi configuration
- `esp_http_client.h` - using client server

4.1.2 FreeRTOS

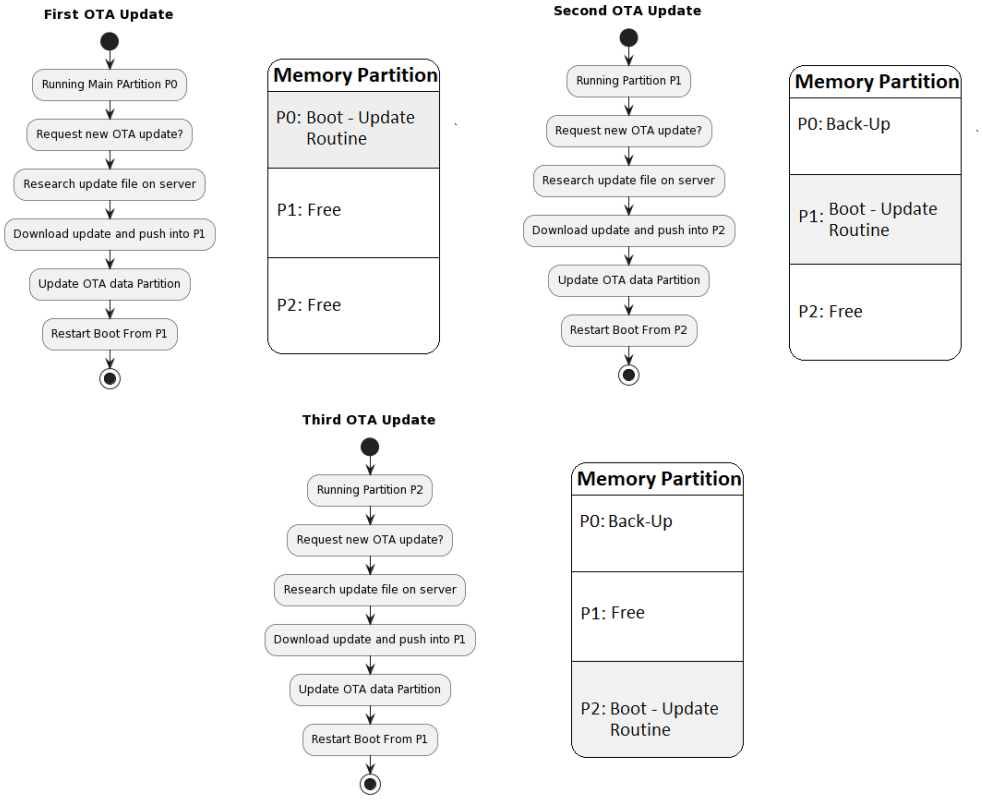
FreeRTOS is used as an operating system for tasks and for interrupt handling. In particular, an interrupt is used on one of the buttons on the board, to start the OTA download and control routine. We could implement any other method to activate the procedure, eg. a message sent from another ECU, timer or any other signal, we had many GPIOs available, however the button seemed to us for the demonstration demo to be a quick and convenient solution to use this feature. For future developments it is therefore possible to think about different solutions more suitable for the application of use. Libraries used:

- `freertos/FreeRTOS.h` - freertos library
- `freertos/task.h` - freertos library
- `freertos/semphr.h` - freertos library
- `driver/gpio.h` - resource library

4.2 Memory Partition

The memory of the board is divided in 3 partition of 1MB.

- P0 : Partition flashed with the first Software, used also has back-up (Flashed with wired connection)
- P1-P2 : Partitions free for future update (Flash with OTA protocol)



4.3 Start Communication

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/protocols/esp_http_client.ht

Communication happens using http client server. Configuration:

```
// as we use http protocol:
esp_http_client_config_t clientConfig = {
.url = "Update Link", // here where we have the binary file
.event_handler = client_event_handler,
.cert_pem = (char *)server_cert_pem_start // Certificate Cfg
```

4.4 Download and install in partition

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/esp_https_ota.html

OTA APIs used: `esp_https_ota`

```
// esp_https_ota returns ok and the binary file is putted in the ota partition that we setted
if(esp_https_ota(&clientConfig) == ESP_OK)
{
  ESP_LOGI(TAG,"firmware update is successful now we are in version %d.",firmware_version);
  vTaskDelay(pdMS_TO_TICKS(5000)); //some delay before restarting
  esp_restart(); //restaring our microcontroller and start from the new memory partition
}
```

CHAPTER 5

Results

In conclusion we can say that we have developed a working OTA protocol. You will have seen or will see in the demo that the OTA update, on the board, takes place without problems. It is possible to repeat the operation several times and multiplexed on multiple devices.

5.1 Known Issues

The major problem that we see on this solution is the server used. From one point of view, Google drive provides us advantages in terms of simplicity of use and versatility; Drive already gives us a well-developed server that is easy to use and within everyone's reach and in addition, the Certified CA that is very important for the security side.
But using Google Drive, that is a third party server, accessible from anyone, does not give us absolute security during the loading and saving phase of the .bin. For sure we can guarantee for the download and upload a HTTPS protocol, but while the file is on the server someone can stole my credential and modify my new software.
One solution could be create a own server with own CA, preferably on a local network, but this takes away the possibility of uploading the board remotely away from the device. Another possibility could be adding a subsequent hash check that we try to explain in the Future Work Chapter.

5.2 Future Work

5.2.1 Cooperative hash check

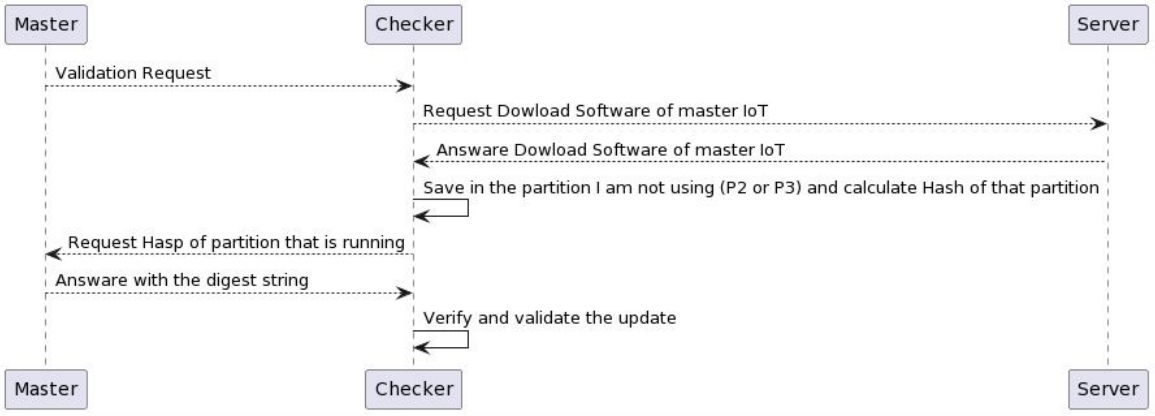
At the end of the OTA update process the new software is safely downloaded from Google Drive. We want to add the possibility, perhaps under a specific command, to verify the hash of the .bin file downloaded by another IoT connected on the same network. Below is represented one operations sequence that allows you to do this check.

- Master send to Checker Validation Request
- Checker send to Server the Request download Software of master IoT
- Server Answer to checker with the download Software of master IoT
- Checker Saves in the partition that is not using (or P2 or P3) and calculate Hash of that partition

- Checker Request to Master the Hash of partition that is running
- Master Answer to Checker with the digest string
- Checker verify and validate the update

Precondition

- The IoT devices to perform this check need to be in the same network
- All the IoT devices are configurable as Master or Checker depending on the check request
- All the IoT devices have a static address. All the IoT devices know all the address of the cluster
- The links to download the bin of other IoT device is hard-coded on all the board. This means that an IoT device can download all the software related to its cluster



5.2.2 Generation hash for inter cluster check

Using FreeRTOS library

`https://www.freertos.org/pkcs11/pkcs11-mechanisms-digest-demo.html`

Core PKCS 11 is a library belonging to FreeRTOS that allows us to hash a given array. At the link above you will find an example of how the digest of the SHA256 function is calculated.

```
/*******************************************************************************/
/* Hash with SHA256 mechanism. */
xDigestMechanism.mechanism = CKM_SHA256;

/* Initializes the digest operation and sets what mechanism will be used
* for the digest. */
xResult = pxFunctionList->C_DigestInit( hSession,
                                        xDigestMechanism );
configASSERT( CKR_OK == xResult );

/* Pass a pointer to the buffer of bytes to be hashed, and its size. */
xResult = pxFunctionList->C_DigestUpdate( hSession,
                                          pxKownMessage,
                                          sizeof( pxKownMessage ) - 1 );
```

```
configASSERT( CKR_OK == xResult );

/* Retrieve the digest buffer. Since the mechanism is an SHA-256 algorithm,
* the size will always be 32 bytes. If the size cannot be known ahead of time,
* a NULL value to the second parameter xDigestResult, will set the third parameter,
* pulDigestLen to the number of required bytes. */
xResult = pxFunctionList->C_DigestFinal( hSession,
                                         xDigestResult,
                                         ulDigestLength );
configASSERT( CKR_OK == xResult );
/*-----------------------------------------------------------------------------*/
```

Using ESP library

In our case is not possible to consider the load for the digest as a file or a string but better consider it as a portion of memory, in particular in this case a partition.

`https://www.esp32.com/viewtopic.php?t=16327`

```
/*-----------------------------------------------------------------------------*/
 #include <sys/stat.h>
 #include "esp_err.h"
 #include "esp_log.h"
+#include "esp_partition.h"
 #include "esp_spiffs.h"

 static const char *TAG = "example";
@@ -96,4 +97,13 @@ void app_main(void)
     // All done, unmount partition and disable SPIFFS
     esp_vfs_spiffs_unregister(NULL);
     ESP_LOGI(TAG, "SPIFFS unmounted");
+
+    uint8_t hash[32] = {0};
+    const esp_partition_t *p = esp_partition_find_first(ESP_PARTITION_TYPE_DATA,
+                                                        ESP_PARTITION_SUBTYPE_DATA_SPIFFS, NULL);
+    ESP_ERROR_CHECK( esp_partition_get_sha256(p, hash) );
+    printf("SPIFFS partition hash ");
+    for (int i = 0; i < sizeof(hash); i++) {
+        printf("%02x ", hash[i]);
+    }
+    printf("\n");
 }
/*-----------------------------------------------------------------------------*/
```

5.2.3 Communication between IoT (ESP32)

To allow the ESP32 to start communication among them, we can use for our project the Protocol Communication (Protocomm).

`https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/provisioning/protocomm.html`

At this link is possible to see the documentation and related example how to use protocomm API for a secure communication session.

CHAPTER 6

Conclusions

Over-The-Air Update Service is and will be more and more fundamental for the Software Update. The protocol described in this report can be the starting point for other development. Using this protocol you have an working OTA update usable with ESP32 Board, that is versatile for different application.
There would be others possible ideas to improve and specialize the protocol, making it perhaps more resilient to be used on different board, and not only on ESP32, but also perhaps introducing the possibility of using a proprietary server and certificate.

Bibliography

[1] Developed in partnership with the world's leading chip companies over an 18-year period, and now downloaded every 170 seconds, FreeRTOS is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of IoT libraries suitable for use across all industry sectors. FreeRTOS is built with an emphasis on reliability and ease of use. `https://www.freertos.org/index.html`

[2] RIOT powers the Internet of Things like Linux powers the Internet. RIOT is a free, open source operating system developed by a grassroots community gathering companies, academia, and hobbyists, distributed all around the world. RIOT supports most low-power IoT devices, microcontroller architectures (32-bit, 16-bit, 8-bit), and external devices. RIOT aims to implement all relevant open standards supporting an Internet of Things that is connected, secure, durable and privacy-friendly. `https://www.riot-os.org/`

[3] v4.2 of ESP-IDF.
`https://docs.espressif.com/projects/esp-idf/en/v4.2/esp32/get-started/index.html`

APPENDIX A

User Manual

In this chapter is described how to use the OTA protocol and retry to perform itself the demo seen on the project presentation Demo.

A.1 Step 1: ESP32 ESP-IDF

To use the ESP32 board we have to install the ESP-IDF, the development framework used to build and flash our project. There you can find the link with more information and how to download it:
`https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html`
As IDE we used VS Code, but it is possible use other editor tool, as Eclipse etc. Once you have the all tools installed, you can open the ESP-IDF tool, (is like a prompt command), and go in the project folder "cd C:project path". than using the command:

- `idf.py build` - compile and build the project
- `idf.py flash` - research ESP32 COM port and flash it
- `idf.py monitor` - show the serial comunication in orther to follow the update step

Also at this link you can find other useful command:
`https://docs.espressif.com/projects/esp-idf/en/v4.2/esp32/api-guides/build-system.html`

A.2 Step 2: Configure the Wi-Fi

In order to connect the board on the Wi-Fi network you need to change and modify the Wi-Fi credential with your actual connection. Open the folder at this link:
`https://drive.google.com/drive/folders/12X8qwaMzzthq3UGNoWMIBG-sxKIKymrK`
Download it and open with a text editor. Replace now `"myssid"` with actual SSID connection and change `"mypassword"` with the connection password.

A.3 Step 3: Compile and Flash ESP32

we have everything set to build the code. Open ESP-IDF and go on the project path. digit `idf.py build` and the build start. At the end press `idf.py flash` and the SW will Flash on the board. This is the version 0 that is flashed on partition zero, used also as back up.

A.4 Step 4: Start procedure

At this point everything is configured and ready, you can start to use the OTA protocol. To see the protocol working you can modify the software adding or changing the release number and compile it. Once compiled, copy the bin and paste on Google Drive folder (in our case the link is that one, but is possible to use your own drive page), if changed the related link on the code. To flash the upload you can press the button on the ESP32 Board and see the OTA update working. If all gone fine you have displayed the new release SW.

APPENDIX B

Preliminary

When we approached the project, we looked at which solutions had already been implemented. Below are some links to slides, docs and video courses that introduce security and OTA protocol.

- Slide from ST:
  `https://drive.google.com/drive/folders/18AetVN1BpxSz1GdH3sN6-Kgl6JH-_lUo`
- Video lesson from ST:
  `https://www.youtube.com/playlist?list=PLnMKNibPkDnGd7J7fV7tr-4xIBwkNfD--`
- Documentation from TI:
  `https://www.ti.com/lit/wp/sway021/sway021.pdf?ts=1650211817745`
- SSL certificate:

  `https://www.youtube.com/watch?v=33VYnE7Bzpk`

B.1 OpenSSL and first demo

At this point, before starting with a targeted development on a specific OS and on a specific board, we have implemented a simulation using the OpenSSL library.
For further detail: `https://www.openssl.org/` The first diagram developed for this application was this:

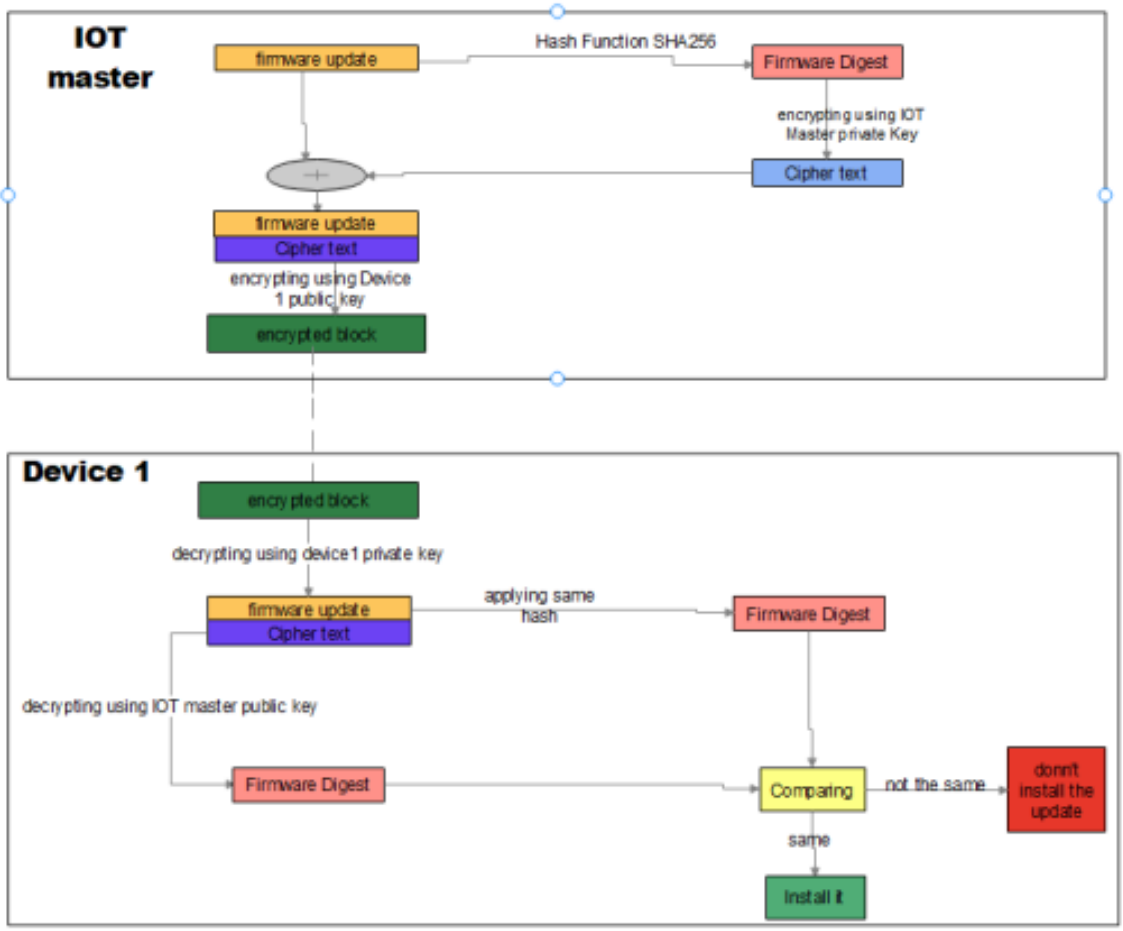


Figure B.1: Procedure Diagram

In this diagram: IoT master will send a firmware update to the device 1. First, it apply hash function SHA256 and encrypt it with the master private key, then combine together the firmware and cipher text, and then encrypt them by the public key and send them to device 1. Device 1 will decrypt them with its private key and now we have the firmware update and the cipher text, applying same hash used by the master and decrypt by master public key and then comparing the two hashes together, if are the same it's ok and authenticity is guaranteed.

To simulate this procedure we can install on PC a board simulator or install OpenSSL usable by prompt command. Below links:

- FreeRTOS simulator: `https://www.freertos.org/install-and-start-qemu-emulator/`
- Raspberry simulator: `https://www.raspberrypi.com/software/raspberry-pi-desktop/`

B.1.1 Command OpenSSL

Master:

```
openssl genpkey-algorithm rsa-out master_private_key.pem
openssl pkeyin master private_key.pem pubout out master_public_key.pem
openssl dgst-sha256 -sign master private_key.per out cipher_text firmwareupdate.c
```

Device A:

```
openssl genpkey -algorithm rsa -out deviceA_private_key.pen
openssl okey to deviceA_private_key.pen -pubout -out deviceA_public_key.pem
openssl dgst-sha256 -verify master public key.pen signature cipher_text
firmwareupdate.c
```

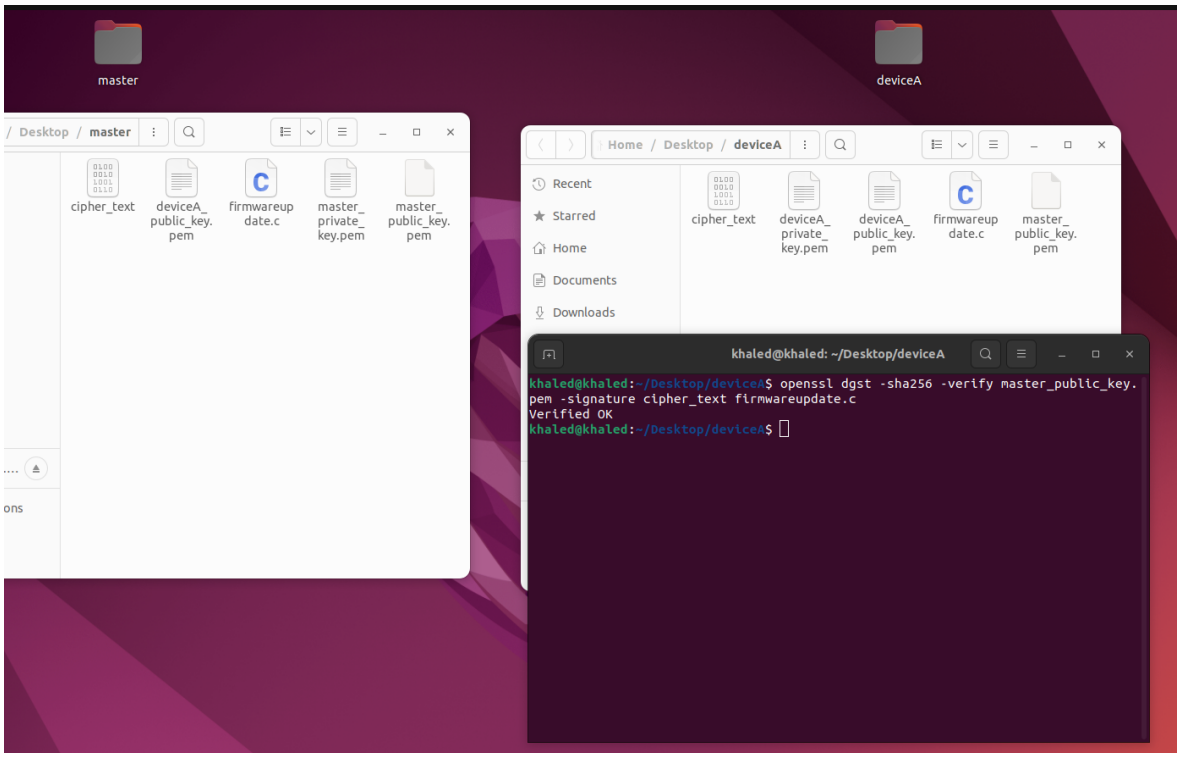


Figure B.2: First Simulation Result

Resource File:

`https://drive.google.com/drive/folders/1Cbt19ni2mlKOWGnxKtJ_QcgR-52DhUYt?usp=sharing`

B.2 RIOT - FreeRTOS

After having made a base project and have understood the OTA protocol and necessary security, we looked at which OS could best fit our application.

- RIOT: `https://www.riot-os.org/`
- FreeRTOS: `www.freertos.org`

Both OSes were compatible, but we found more documentation and examples on FreeRTOS.