Aggiornamenti OTA con Secure-Boot e Flash encryption sul dispositivo embedded ESP32

UniVR - Dipartimento di Informatica

Fabio Irimie

Tesi di laurea 2025/2026

Indice

1 Introduzione

Questo progetto consiste nell'implementazione di un sistema che permetta di aggiornare il firmware dell'ESP32 da remoto (Over The Air) tramite Wi-Fi. L'obiettivo principale è quello di attivare le funzionalità di sicurezza del microcontrollore in modo da proteggere il dispositivo da accessi non autorizzati. Le funzionalità di sicurezza includono:

- **Secure OTA**: Garantisce che il nuovo firmware sia autentico e non compromesso
- **Secure Boot**: Impedisce l'esecuzione di firmware non autorizzato
- **Flash Encryption**: Protegge i dati memorizzati nella memoria flash del dispositivo

2 Cenni teorici

2.1 Aggiornamenti OTA (Over The Air)

Gli aggiornamenti OTA permettono di aggiornare il firmware del dispositivo durante la sua normale esecuzione, senza la necessità di collegarlo fisicamente a un computer. Le modalità di aggiornamento si distinguono in base alla vulnerabilità del sistema:

- **Modalità sicura**: L'aggiornamento di alcune partizioni è resiliente, cioè garantisce l'operabilità del dispositivo anche in caso di perdita di alimentazione o di errore durante l'aggiornamento. Solo il seguente tipo di partizione supporta la modalità sicura:
  - **Application**: OTA configura la partition table in modo da avere due partizioni per l'aggiornamento (`ota_0` e `ota_1`) e una partizione per lo stato di boot (`ota_data`). Durante l'aggiornamento il nuovo firmware viene scritto nella partizione OTA attualmente non selezionata per il boot. Una volta completato l'aggiornamento, la partizione `ota_data` viene aggiornata per indicare che la partizione OTA appena scritta deve essere utilizzata al boot successivo. Se la partizione `ota_data` non contiene alcun dato il dispositivo esegue il boot dalla partizione `factory`.

    La partition table con due partizioni OTA è la seguente:

    ```
    # ESP-IDF Partition Table
    # Name,   Type, SubType, Offset,  Size, Flags
    nvs,      data, nvs,     0x9000,  0x4000,
    otadata,  data, ota,     0xd000,  0x2000,
    phy_init, data, phy,     0xf000,  0x1000,
    factory,  app,  factory, 0x10000,  1M,
    ota_0,    app,  ota_0,   0x110000, 1M,
    ota_1,    app,  ota_1,   0x210000, 1M,

    ```

- **Modalità non sicura**: L'aggiornamento di alcune partizioni è vulnerabile, cioè in caso di perdita di alimentazione o di errore durante l'aggiornamento il dispositivo potrebbe non essere più operabile. Una partizione

temporanea riceve i dati della nuova immagine e, una volta completato il trasferimento, l'immagine viene copiata nella partizione di destinazione. Se l'operazione di copia viene interrotta potrebbero verificarsi problemi di boot. Le partizioni che supportano la modalità non sicura sono:

- **Bootloader**
- **Partition Table**
- **Partizioni data** (ad esempio NVS, FAT, ecc...)

2.1.1 Partizione OTA Data

Al primo avvio del dispositivo la partizione `ota_data` deve essere vuota (tutti i byte a 0xFF) in modo da far eseguire il boot dal'applicazione nella partizione `factory`. Se l'applicazione in `factory` non è presente viene eseguito il boot della prima partizione OTA disponibile (di solito `ota_0`).

Dopo il primo aggiornamento OTA, la partizione `ota_data` viene aggiornata per indicare quale partizione OTA deve essere utilizzata al successivo boot. La dimensione di `ota_data` è di due settori (0x2000 bytes = 8192 bytes) in modo da evitare errori mentre si scrive la partizione. I due settori sono cancellati indipendentemente e scritti con gli stessi dati. In questo modo se i dati dei due settori non coincidono viene usato un counter per determinare quale settore è stato scritto più recentemente.

2.1.2 App rollback

L'obiettivo dell'app rollback è quello di tenere il funzionante il dispositivo dopo un aggiornamento e permette di tornare alla versione precedente del firmware se la nuova versione non funziona correttamente (solo le partizioni OTA possono effettuare il rollback). Dopo un aggiornamento OTA con rollback attivo si hanno le seguenti possibilità:

- Se l'app funziona bene `esp_ota_mark_app_valid_cancel_rollback()` imposta lo stato dell'applicazione a `ESP_OTA_IMG_VALID`.
- Se l'app non funziona correttamente il dispositivo esegue il rollback alla versione precedente e `esp_ota_mark_app_invalid_rollback()` imposta lo stato dell'applicazione a `ESP_OTA_IMG_INVALID`.
- Se l'impostazione `CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE` è abilitata e viene effettuato un reset, allora viene effettuato un rollback senza chiamare nessuna funzione nell'applicazione. Questa opzione permette di intercettare la prima esecuzione di una nuova applicazione per confermare che funzioni correttamente.

Gli stati che controllano il processo di selezione dell'applicazione sono:

Stato	Restrizioni sulla nuova app
ESP_OTA_IMG_VALID	Nessuna restrizione. Verrà selezionata
ESP_OTA_IMG_UNDEFINED	Nessuna restrizione. Verrà selezionata
ESP_OTA_IMG_INVALID	Non verrà selezionata
ESP_OTA_IMG_ABORTED	Non verrà selezionata
ESP_OTA_IMG_NEW	Se l'opzione CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE è abilitata, l'app verrà selezionata solo una volta. Nel bootloader lo stato viene subito impostato a ESP_OTA_IMG_PENDING_VERIFY.
ESP_OTA_IMG_PENDING_VERIFY	Se l'opzione CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE è abilitata, l'app non verrà selezionata. Nel bootloader lo stato viene impostato a ESP_OTA_IMG_ABORTED.

Tabella 1: Stati dell'applicazione OTA

L'impostazione di questi stati avviene nei seguenti casi:

- ESP_OTA_IMG_VALID: impostato dalla funzione esp_ota_mark_app_valid_cancel_rollback().
- ESP_OTA_IMG_UNDEFINED: impostato dalla funzione esp_ota_set_boot_partition() se l'impostazione CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE è disabilitata.
- ESP_OTA_IMG_NEW: impostato dalla funzione esp_ota_set_boot_partition() se l'impostazione CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE è abilitata.
- ESP_OTA_IMG_INVALID: impostato dalla funzione esp_ota_mark_app_invalid_rollback() o esp_ota_mark_app_invalid_rollback_and_reboot().
- ESP_OTA_IMG_ABORTED: impostato se l'operabilità dell'applicazione non è stata confermata e avviene un reboot quando l'impostazione CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE è abilitata.
- ESP_OTA_IMG_PENDING_VERIFY: impostato nel bootloader se l'impostazione CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE è abilitata e l'applicazione selezionata è nello stato ESP_OTA_IMG_NEW.

2.2 eFuse

Gli eFuse (electronic fuses) sono memorie non volatili che possono essere programmate (burned) una sola volta. Gli eFuse sono campi da **un bit** che può essere impostato a 1, ma non può essere riportatoo a 0 e servono per memorizzare valori di sistema o dell'utente. I bit sono raggruppati in 4 blocchi da 256 bit e ogni blocco è suddiviso in 8 registri da 32 bit. Alcuni blocchi sono riservati per i valori di sistema e altri sono liberi per l'uso dell'utente. I blocchi degli eFuse sono divisi in:

- `EFUSE_BLK0`: è interamente riservato ai valori di sistema
- `EFUSE_BLK1`: è utilizzato per le chiavi della Flash Encryption se attivata, altrimenti può essere usato dall'utente
- `EFUSE_BLK2`: è utilizzato per la chiave del Secure Boot se attivo, altrimenti può essere usato dall'utente
- `EFUSE_BLK3`: può essere riservato per memorizzare un indirizzo MAC personalizzato oppure può essere usato dall'utente. Alcuni bit sono già usati in ESP-IDF.

Non tutti i bit possono essere utilizzati.

2.3 Secure Boot

Il Secure Boot protegge il dispositivo dall'esecuzione di firmware non autorizzato verificando che ogni software (second stage bootloader e ogni applicazione) che viene eseguito sia firmato. Il first stage bootloader non deve essere firmato in quanto è memorizzato in una memoria ROM (di sola lettura). La firma è una coppia di chiavi (privata e pubblica) generata tramite l'algoritmo RSA.

- La chiave privata viene utilizzata per firmare il software prima che venga caricato nel dispositivo.
- La chiave pubblica viene memorizzata nel dispositivo e viene utilizzata per verificare la firma del software prima che venga eseguito.

In breve, il processo di Secure Boot funziona come segue:

1. Il first stage bootloader carica il second stage bootloader e ne verifica la firma. Solo se la verifica va a buon fine il second stage bootloader viene eseguito.
2. Il second stage bootloader carica l'applicazione e ne verifica la firma. Solo se la verifica va a buon fine l'applicazione viene eseguita.

2.3.1 Vantaggi

I vantaggi del Secure Boot sono:

- La chiave pubblica è memorizzata sul dispositivo, mentre quella privata è tenuta in un posto sicuro e non viene mai usata dal dispositivo.
- Solo una chiave pubblica può essere generata e memorizzata nel chip durante la fase di produzione.
- Viene usato lo stesso formato di immagine del firmware e della firma sia per il second stage bootloader che per le applicazioni.
- Nessun dato segreto viene memorizzato nel dispositivo.

2.3.2 Formato del signature block

Il signature block è una struttura che contiene la firma del software. Il blocco inizia su un confine allineato a 4 KB e ha un proprio settore di flash, cioè 4096 byte. La firma viene calcolata su **tutti i byte** dell'immagine, inclusi i byte di "secure padding" (vedi capitolo 2.3.3). Ogni signature block contiene la firma dell'immagine a cui appartiene insieme alla chiave pubblica RSA usata per la verifica della firma. Il formato del signature block è il seguente:

Offset	**Dimensione (byte)**	**Descrizione**
0	1	Byte magico (deve essere `0xE7`)
1	1	Byte della versione, per Secure Boot v2 è `0x02`
2	2	Byte di padding. Riservati e devono essere a 0
4	32	Hash SHA-256 del contenuto dell'immagine, senza considerare il signature block
36	384	Modulo RSA usato per la verifica della firma. (valore `"n"` nella specifica RFC8017)
420	4	Esponente pubblico RSA usato per la verifica della firma. (valore `"e"` nella specifica RFC8017)
424	384	Valore precalcolato di `"R"`, derivato da `"n"`, usato per l'algoritmo di moltiplicazione di Montgomery
808	4	Valore precalcolato di `"M'"`, derivato da `"n"`, usato per l'algoritmo di moltiplicazione di Montgomery
812	384	Firma RSA-PSS (sezione 8.1.1 della RFC8017) del contenuto dell'immagine, calcolata usando i seguenti parametri PSS: hash SHA-256, funzione MGF1, lunghezza del salt 32 bytes, campo trailer di default 0xBC.
1196	4	CRC32 (checksum) dei precedenti 1196 bytes
1200	16	Byte di padding a 0 per arrivare a lunghezza 1216 bytes

Tabella 2: Formato del signature block

Lo spazio rimanente dopo il signature block (4096 - 1216 = 2880 bytes) è memoria flash cancellata, cioè tutti i byte a 0xFF, che può essere usata per scrivere altri signature block dopo il precedente.

Un signature block è valido se soddisfa entrambe le condizioni:

- Il byte magico è corretto (`0xE7`)

- La checksum CRC32 è corretta

altrimenti viene considerato non valido.

2.3.3 Secure padding

Il secure padding è un'area di memoria che viene aggiunta alla fine di ogni immagine del firmware per allineare l'immagine al confine della dimensione di pagina della flash MMU (Memory Management Unit con dimensione default di 64KB). Questo viene fatto per assicurare che soltanto contenuti verificati vengano mappati nella memoria indirizzabile del dispositivo. La firma dell'immagine viene calcolata dopo aver aggiunto il secure padding e solo dopo viene aggiunto alla fine il signature block (4KB).

2.3.4 Verifica di un'immagine

Un esempio di applicazione firmata è il seguente:

Offset	**Dimensione (KB)**	**Descrizione**
0	580	Contenuto di esempio di un'applicazione non firmata
580	60	Secure padding per allineare l'immagine al prossimo confine di 64KB
640	4	Signature block

Tabella 3: Esempio di applicazione firmata

L'immagine dell'applicazione inizia sempre al prossimo confine di pagina della flash MMU, di default 64KB, e quindi lo spazio rimanente dopo il signature block può essere utilizzato per memorizzare altre partizioni di dati, ad esempio `nvs`.

Un'immagine è verificata se la chiave pubblica memorizzata in qualsiasi signature block è valida per quel dispositivo, e se la firma RSA-PSS nel signature block coincide con la firma calcolata per i dati dell'immagine letti dalla flash. La verifica dell'immagine non viene effettuata soltanto ad ogni boot, ma anche dopo ogni aggiornamento OTA. Se la verifica della nuova immagine ottenuta tramite OTA fallisce, il bootloader cercherà un'altra immagine valida da eseguire. I passaggi per verificare un'immagine sono i seguenti:

1. Confrontare l'hash SHA-256 della chiave pubblica memorizzata nel signature block del bootloader con quello memorizzato negli `eFuse` del dispositivo. Se non coincidono, l'immagine non è valida.

2. Generare l'hash dell'immagine dell'applicazione e confrontarlo con l'hash memorizzato nel signature block dell'immagine. Se non coincidono, l'immagine non è valida.

3. Usare la chiave pubblica per verificare la firma dell'immagine del bootloader usando RSA-PSS (sezione 8.1.2 della RFC8017) per confrontarla con l'hash dell'immagine calcolato nel passo 2.

2.3.5 Processo di Secure Boot

I passi eseguiti durante il processo di Secure Boot sono i seguenti:

1. All'avvio, il codice salvato in ROM (first stage bootloader) controlla il bit di Secure Boot v2 negli `eFuse` del dispositivo. Se il bit è disabilitato, il dispositivo esegue il boot normalmente, altrimenti procede al passo successivo.

2. Il codice in ROM verifica il signature block del bootloader (vedi 2.3.2). Se fallisce il processo di boot viene interrotto.

3. Il codice in ROM verifica l'immagine del bootloader usando i dati dell'immagine, i signature block corrispondenti e gli `eFuse` (vedi 2.3.4). Se la verifica fallisce il processo di boot viene interrotto.

4. Il codice in ROM esegue il bootloader.

5. Il bootloader verifica il signature block dell'immagine dell'applicazione. Se fallisce il processo di boot viene interrotto.

6. Il bootloader verifica l'immagine dell'applicazione usando i dati dell'immagine, i signature block corrispondenti e gli `eFuse`. Se la verifica fallisce il processo di boot viene interrotto, ma se viene trovata un'altra immagine, allora il bootloader proverà a verificare quell'immagine tornando ad eseguire i punti dal 5 al 7. Questo viene ripetuto finchè non viene trovata un'immagine valida o non ci sono più immagini da verificare.

7. Il bootloader esegue l'immagine dell'applicazione verificata.

2.4 Flash Encryption

La Flash Encryption permette di crittografare la memoria flash del dispositivo. Una volta attivata, il firmware viene flashato sottoforma di testo non cifrato che verrà crittografato al primo boot. Una volta attivata, le seguenti partizioni vengono crittografate di default:

- Second stage bootloader
- Partition table
- Partizione OTA data
- Partizioni di tipo `app`

Altre partizioni si possono crittografare condizionalmente:

- Partizioni con la flag `encrypted` impostata nella partition table
- Il digest (hash) del bootloader di Secure Boot se il Secure Boot è abilitato

2.4.1 eFuse rilevanti

I seguenti `eFuse` sono rilevanti per la Flash Encryption:

- `CODING_SCHEME` (2 bit): definisce il numero di bit del `block1` usati per derivare la chiave di crittografia finale. I possibili valori sono:
  - `0`: per 256 bits
  - `1`: per 192 bits
  - `2`: per 128 bits

  Il metodo di derivazione è basato sul valore dell'eFuse `FLASH_CRYPT_CONFIG`.
- `flash_encryption` (`block1` 256 bit): contiene la chiave di crittografia
- `FLASH_CRYPT_CONFIG` (4 bit): controlla il processo di crittografia
- `DISABLE_DL_ENCRYPT` (1 bit): se abilitato disattiva la flash encryption mentre il dispositivo viene eseguito in modalità Firmware Download
- `DISABLE_DL_DECRYPT` (1 bit): se abilitato disattiva la flash decryption mentre il dispositivo viene eseguito in modalità UART Firmware Download
- `FLASH_CRYPT_CNT` (7 bit): è un numero del tipo 2^n indica se i contenuti della memoria flash sono stati crittografati.
  - Se un numero dispari di bit sono impostati a 1, indica che i contenuti della memoria flash sono crittografati e devono essere decrittografati durante la lettura.
  - Se un numero pari di bit sono impostati a 1, indica che i contenuti della memoria flash non sono crittografati.

  Ogni aggiornamento della memoria flash con dati non crittografati e con l'esecuzione della crittografia il bit più significativo di `FLASH_CRYPT_CNT` viene impostato a 1.

2.4.2 Processo della Flash Encryption

Supponendo che i valori degli eFuse siano nello stato iniziale e che il second stage bootloader sia stato compilato per supportare la flash encryption, il processo di crittografia è il seguente:

1. Al primo avvio, tutti i dati nella memoria flash non sono crittografati (testo in chiaro) e il first stage bootloader carica il second stage bootloader.
2. Il second stage bootloader controlla il valore dell'eFuse `FLASH_CRYPT_CNT`. Poichè tutti i bit sono a 0 (stato di default), cioè un numero pari di bit impostati, il bootloader configura e attiva il *flash encryption block*. Viene anche impostato il valore dell'eFuse `FLASH_CRYPT_CONFIG` a `0xF`
3. Il second stage bootloader controlla se una chiave è già presente nell'eFuse:
   - Se è presente, il processo di generazione della chiave viene saltato e viene usata la chiave esistente.

- Se non è presente, viene generata casualmente una nuova chiave di crittografia AES-256 bit e viene scritta nell'eFuse `flash_encryption`.

L'intero processo di crittografia avviene via hardware e la chiave non è accessibile dal software.

4. Il *flash encryption block* crittografa tutti i dati nella memoria flash, cioè il second stage bootloader, le applicazioni e tutte le partizioni etichettate con la flag `encrypted` nella partition table.

5. Il second stage bootloader imposta il primo bit disponibile di `FLASH_CRYPT_CNT` a 1 per indicare che i dati nella memoria flash sono crittografati.

6. In modalità `development`, il second stage bootloader abilita soltanto gli eFuse `DISABLE_DL_ENCRYPT` e `DISABLE_DL_CACHE` per permettere al bootloader in modalità Firmware Download di flashare binari crittografati. Inoltre i bit dell'eFuse `FLASH_CRYPT_CNT` non sono protetti in scrittura.

7. In modalità `release`, il second stage abilita i bit degli eFuse `DISABLE_DL_ENCRYPT`, `DISABLE_DL_ENCRYPT` e `DISABLE_DL_CACHE` per impedire che il bootloader decrittografi i dati durante la modalità Firmware Download. Inoltre i bit dell'eFuse `FLASH_CRYPT_CNT` sono protetti in scrittura.

8. Il dispositivo viene riavviato per eseguire l'immagine crittografata. Il second stage bootloader chiama il *flash decryption block* per decrittografare i dati della memoria flash e caricare i contenuti decrittografati nella memoria IRAM (Instruction RAM).

2.4.3 Modalità di sviluppo e di rilascio

Durante la fase di sviluppo è necessario poter flashare più volte il dispositivo, con diverse immagini non crittografate per testare il processo di crittografia, di conseguenza la modalità Firmware Download deve essere attivata. Questa modalità però dovrebbe essere disabilitata nella versione finale del prodotto per impedire ulteriore accesso alla memoria flash. Per questo motivo esistono due modalità di funzionamento della Flash Encryption:

- **Development mode**: In questa modalità è possibile flashare nuove immagini non crittografate sul dispositivo e il bootloader le crittograferà utilizzando la chiave memorizzata nell'hardware. Questo però permette, indirettamente, di leggere i contenuti in chiaro dell'immagine in memoria flash.

- **Release mode**: In questa modalità non è più possibile flashare nuove immagini sul dispositivo senza conoscere la chiave di crittografia.

3 Implementazione

3.1 Aggiornamenti OTA

3.1.1 Configurazione del progetto

Per abilitare gli aggiornamenti OTA è stato necessario configurare il progetto ESP-IDF con le seguenti componenti:

- **Partition table e memoria flash**: è stato creato il file `sdkconfig.defaults` che contiene la configurazione di default del progetto. All'interno è stato definito l'utilizzo della partition table con due partizioni OTA:

```
CONFIG_PARTITION_TABLE_TWO_OTA=y
```

  e sono state impostate le dimensioni della memoria flash a 4MB:

```
CONFIG_ESPTOOLPY_FLASHSIZE_4MB=y
```

- **Server HTTP**: per permettere il download del nuovo firmware è stato creato un server HTTP in `server/pytest_simple_ota.py` che rende disponibile il file binario del firmware aggiornato.

  Il server richiede, nella sua cartella, la presenza di un certificato SSL per permettere connessioni sicure attraverso HTTPS. Il certificato è stato generato eseguendo il seguente comando nella cartella `server/`:

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

  Nota: durante la creazione del certificato nel campo `Common Name (CN)` deve essere inserito il nome host del server. Se il server viene eseguito in locale il campo deve essere impostato con l'indirizzo IP del server.

  Una volta generato il certificato bisogna flashare il file `ca_cert.pem` sul dispositivo in modo che possa verificare l'autenticità del server. Per fare ciò bisogna copiare il file nella cartella `server_certs/` del progetto:

```
cp ca_cert.pem ../server_certs
```

  Inoltre è necessario modificare il file `main/CMakeLists.txt` per includere il certificato nel firmware:

```
idf_build_get_property(project_dir PROJECT_DIR)
idf_component_register(SRCS "main.c" INCLUDE_DIRS "."
    EMBED_TXTFILES ${project_dir}/server_certs/ca_cert.pem)
```

  Una volta configurato il tutto, il server può essere avviato eseguendo il comando:

```
python pytest_simple_ota.py <BIN_DIR> <PORT> [CERT_DIR]
```

  Dove:

  - `<BIN_DIR>` è la cartella che contiene il file binario del firmware.
  - `<PORT>` è la porta su cui eseguire il server HTTP.

– `[CERT_DIR]` (opzionale) è cartella che contiene il certificato SSL del server. Se non viene specificata viene usata la cartella corrente.

Se tutto è andato a buon fine l'output del server sarà simile al seguente:

```
$ python pytest_simple_ota.py build 8070
Starting HTTPS server at "https://:8070"
192.168.10.106 - - [01/Jan/2026 12:00:00] "GET /
    esp32_secure_ota.bin HTTP/1.1" 200 -
```

- **Supporto del versionamento**: per tenere traccia delle versioni del firmware è stato aggiunto il file `version.txt` nella cartella principale del progetto che contiene il numero di versione corrente. Questo numero viene inserito nel file binario del firmware durante la compilazione e può essere letto dall'applicazione per confrontare la versione corrente con quella disponibile sul server ed evitare aggiornamenti non necessari.
- **Accesso Wi-Fi**: per connettere il dispositivo alla rete Wi-Fi è stata implementata la logica di connessione nel file `main/wifi.c`.
- **Configurazione Wi-Fi e server HTTP**: per fornire le credenziali di accesso alla rete Wi-Fi e l'indirizzo del server HTTP è stata creata una sezione nel menu di configurazione del progetto (menuconfig), nel file `main/Kconfig.projbuild`. Per accedere al menuconfig bisogna eseguire il comando:

```
idf.py menuconfig
```

e navigare fino alla sezione `Over The Air Updates configuration` dove è possibile impostare:

  – **Wifi configuration**
    * `SSID`: nome della rete Wi-Fi
    * `Password`: password della rete Wi-Fi
    * `Maximum retries`: numero massimo di tentativi di riconnessione
    * `WPA3 SAE mode selection`: modalità di autenticazione WPA3
    * `Password identifier`: identificatore della password WPA3
    * `WiFi Scan auth mode threshold`: modalità di autenticazione minima accettata durante la scansione delle reti
  – **Upgrade server**
    * `Firmware upgrade URL`: URL del server HTTP che ospita il file binario del firmware. Deve essere nel formato:

      ```
      https://<host-ip-address>:<host-port>/<firmware-image-
          filename>
      ```

      dove:
      · `<host-ip-address>`: hostname o indirizzo IP del server HTTP.
      · `<host-port>`: porta su cui è in esecuzione il server HTTP.
      · `<firmware-image-filename>`: nome del file binario del firmware. **Deve coincidere con il nome del file messo a disposizione dal server**.

* `Skip server certificate CN field check`: se abilitato, il dispositivo non verificherà il campo `Common Name (CN)` del certificato SSL. Utile per testare il server in locale senza dover generare un certificato con l'indirizzo IP come CN.
* `Skip firmware version check`: se abilitato, il dispositivo non confronterà la versione del firmware corrente con quella disponibile sul server prima di effettuare l'aggiornamento OTA.
* `OTA recieve timeout`: tempo massimo (in millisecondi) per ricevere una risposta dal server HTTP durante l'aggiornamento OTA.

3.1.2 Connessione Wi-Fi

La connessione alla rete Wi-Fi viene gestita nel file `main/wifi.c` che fornisce la funzione `connect_wifi()`. Questa funzione gestisce la connessione alla rete Wi-Fi utilizzando le credenziali fornite nel menuconfig e la riconnessione in caso di disconnessione. La funzione esegue i seguenti passaggi:

1. Inizializza il driver Wi-Fi
2. Chiama un handler per gestire gli eventi di connessione e disconnessione dalla rete Wi-Fi
3. Chiama un handler per gestire l'acquisizione dell'indirizzo IP
4. Imposta il device in modalità stazione Wi-Fi con la configurazione fornita dal menuconfig
5. Avvia la connessione alla rete Wi-Fi
6. Attende fino a quando il dispositivo non si connette alla rete o raggiunge il numero massimo di tentativi di riconnessione
7. Restituisce lo stato della connessione (successo o fallimento)

3.1.3 Aggiornamento OTA

L'aggiornamento OTA viene gestito nel file `main/ota.c` che fornisce le funzione `download_new_firmware()` e `diagnose_new_firmware()`. La funzione `download_new_firmware()` connette il dispositivo al server HTTP e scarica il nuovo firmware, se disponibile, e lo imposta come nuova applicazione di boot. I passi eseguiti sono i seguenti:

1. Recupera le partizioni di boot e OTA correnti
2. Si connette al server HTTP utilizzando l'URL fornito nel menuconfig
3. Recupera la partizione OTA non attualmente in uso per il boot
4. Legge il file binario del firmware dal server HTTP
5. Confronta la versione del firmware corrente con quella scaricata
    - Se l'aggiornamento non è necessario viene eseguito un loop infinito che attende un reset del dispositivo per ritentare l'aggiornamento

6. Inizializza l'aggiornamento OTA
7. Scrive i dati del firmware scaricato nella partizione OTA non in uso
8. Controlla che tutti i dati siano stati scritti correttamente
9. Disattiva l'aggiornamento OTA
10. Imposta la partizione OTA appena scritta come partizione di boot
11. Riavvia il dispositivo per eseguire il nuovo firmware

La funzione `diagnose_new_firmware()` permette di verificare se il nuovo firmware funziona correttamente e, in caso negativo, effettua il rollback alla versione precedente. I passi eseguiti sono i seguenti:

1. Controlla se l'applicazione corrente è al primo avvio dopo un aggiornamento OTA
2. Esegue la funzione di diagnosi del firmware (in questo caso un semplice delay di 5 secondi che simula un controllo)
   - Se la diagnosi ha esito positivo, conferma che il nuovo firmware è valido e cancella il rollback
   - Se la diagnosi ha esito negativo, marca il firmware come non valido ed esegue il rollback alla versione precedente

3.1.4 Applicazione principale

L'applicazione principale si trova nel file `main/main.c` e utilizza le funzioni definite nei file `wifi.c` e `ota.c` per connettere il dispositivo alla rete Wi-Fi e scaricare il nuovo firmware. I passi eseguiti sono i seguenti:

1. Controlla il nuovo firmware tramite la funzione `diagnose_new_firmware()`
2. Si connette alla rete Wi-Fi tramite la funzione `connect_wifi()`
3. Crea una task per gestire l'aggiornamento OTA chiamando la funzione `download_new_firmware()`
4. Crea una task per eseguire l'applicazione principale (in questo caso un semplice loop che stampa un messaggio ogni 5 secondi)

Bibliografia

[1] Espressif Systems. *Documentation of QEMU Fork with Espressif patches.* URL: `https://github.com/espressif/esp-toolchain-docs/blob/main/qemu/esp32/README.md`. (accessed: 05.11.2025).

[2] Espressif Systems. *eFuse.* URL: `https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/efuse.html`. (accessed: 30.10.2025).

[3] Espressif Systems. *Enable Flash Encryption and Secure Boot Externally*. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/security/security-features-enablement-workflows.html#enable-flash-encryption-and-secure-boot-v2-externally. (accessed: 05.11.2025).

[4] Espressif Systems. *Flash Encryption*. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/security/flash-encryption.html. (accessed: 04.11.2025).

[5] Espressif Systems. *Flash Encryption Example*. URL: https://github.com/espressif/esp-idf/tree/v5.5.1/examples/security/flash_encryption. (accessed: 05.11.2025).

[6] Espressif Systems. *Fork of QEMU with Espressif patches*. URL: https://github.com/espressif/qemu. (accessed: 05.11.2025).

[7] Espressif Systems. *Over The Air Examples*. URL: https://github.com/espressif/esp-idf/tree/master/examples/system/ota. (accessed: 20.10.2025).

[8] Espressif Systems. *Over The Air Updates (OTA)*. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/ota.html. (accessed: 20.10.2025).

[9] Espressif Systems. *Secure Boot v2*. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/security/secure-boot-v2.html#. (accessed: 28.10.2025).

[10] Espressif Systems. *Security Features Example*. URL: https://github.com/espressif/esp-idf/tree/v5.5.1/examples/security/security_features_app#enable-security-features-with-help-of-qemu. (accessed: 05.11.2025).