Secure Boot Framework For STM32 Microcontrollers - Final Report

Oğuz Mert COŞKUN-S034085 Mehmet Arda GÜLER-S027709 Mert KIRGIN-S034556

CS350 Operating Systems Project, Özyeğin University

mert.kirgin@ozu.edu.tr, arda.guler@ozu.edu.tr, mert.coskun@ozu.edu.tr

Abstract – We have successfully designed and implemented a software-based Secure Boot Framework [1] tailored for STM32 microcontrollers which are tested working in STM32F746G and STM32F411. This framework addressed the lack of hardware-based security in embedded systems, and establishes a Root of Trust (RoT) entirely in software. It guarantees firmware authenticity with ECDSA-P256 signing and integrity using SHA-256. The bootloader provides software transmission confidentiality option during updates via AES-128 encryption, and enforces memory isolation using the Memory Protection Unit (MPU). It also implements a dual-bank update mechanism was implemented to support fail-safe updates.

1 Introduction

Embedded systems in IoT and automotive sectors are increasingly vulnerable to reverse engineering and malicious firmware modifications. While modern MCUs include hardware security (e.g., TrustZone mentioned in the proposal), widely used mainstream MCUs like the STM32F7 series lack these features. The goal of this project was to bridge this gap by developing a portable secure bootloader that:

1. Verifies the integrity and authenticity of the application before every execution using ECDSA[2] and SHA-256[3].
2. Ensures firmware confidentiality during transmission by decrypting updates securely with AES-128.
3. Enforces runtime memory isolation using the Memory Protection Unit (MPU) to prevent the application from compromising the bootloader.
4. Provides a fail-safe update mechanism with automatic rollback capabilities using a dual-slot architecture.

2 System Architecture

2.1 Memory Architecture

We partitioned the STM32 Flash memory into logical regions to isolate between the trusted bootloader and the user application. These partitions are aligned with the sector organization of the STM32F746G shown in Figure 1. Aligning logical partitions with physical flash sectors is important because flash memory erase operations occur at the sector level. If a boundary splits a sector, erasing one region would unintentionally corrupt the data in the adjacent region. For example, deleting the data in Active Application can corrupt the Download Slot too. In order to maintain portability across different STM32 families, this layout is designed to be configurable from `mem_layout.h` and end users must be adapt to the sector boundaries of the their target STM32 device. As defined in `mem_layout.h`, the memory partition for our test device STM32F746G is as follows:

Table 3. STM32F756xx and STM32F74xxx Flash memory organization

Block	Name	Bloc base address on AXIM interface	Block base address on ICTM interface	Sector size
Main memory block	Sector 0	0x0800 0000 - 0x0800 7FFF	0x0020 0000 - 0x0020 7FFF	32 Kbytes
	Sector 1	0x0800 8000 - 0x0800 FFFF	0x0020 8000 - 0x0020 FFFF	32 Kbytes
	Sector 2	0x0801 0000 - 0x0801 7FFF	0x0021 0000 - 0x0021 7FFF	32 Kbytes
	Sector 3	0x0801 8000 - 0x0801 FFFF	0x0021 8000 - 0x0021 FFFF	32 Kbytes
	Sector 4	0x0802 0000 - 0x0803 FFFF	0x0022 0000 - 0x0023 FFFF	128 Kbytes
	Sector 5	0x0804 0000 - 0x0807 FFFF	0x0024 0000 - 0x0027 FFFF	256 Kbytes
	Sector 6	0x0808 0000 - 0x080B FFFF	0x0028 0000 - 0x002B FFFF	256 Kbytes
	Sector 7	0x080C 0000 - 0x080F FFFF	0x002C 0000 - 0x02F FFFF	256 Kbytes
Information block	System memory	0x1FF0 0000 - 0x1FF0 EDBF	0x0010 0000 - 0x0010 EDBF	60 Kbytes
	OTP	0x1FF0 F000 - 0x1FF0 F41F	0x0010 F000 - 0x0010 F41F	1024 bytes
	Option bytes	0x1FFF 0000 - 0x1FFF 001F	-	32 bytes

Figure 1: STM32F746G Memory Sector Table [4] (AXIM interface is used)

- **Secure Bootloader (Sectors 0-1):** Stores the RoT code.
- **Configuration Sector (Sector 2):** Stores persistent boot state (`magic_number`, `system_status`, `current_version`).

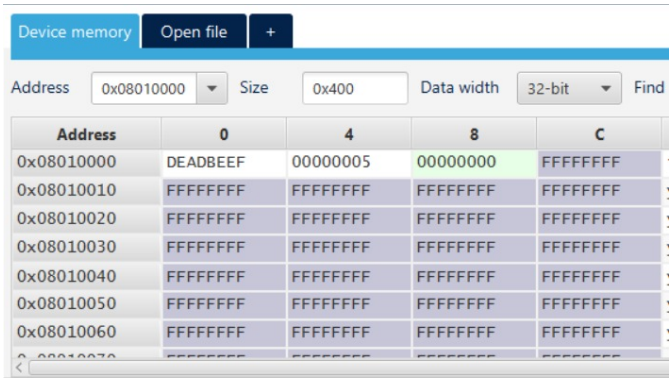

Figure 2: Sector 2 Magic Number and System Status Shown in the Memory. Status is Ready to Update

- **User Data / Reserved (Sectors 3-4):** These sectors are intentionally left unused by the bootloader. This creates a ”safe zone” for the user application to store persistent data—such as calibration values, logs, or file systems—without risk of

being overwritten during bootloader operations.

- **Active Application (Slot A - Sector 5):**Where the verified application runs. We specifically assigned one of the largest physical sectors (256 KB) to this slot to accommodate substantial user applications.
- **Download Slot (Slot B - Sector 6):** Stores encrypted updates. Like the active slot, this is mapped to a 256 KB sector to ensure it can buffer full-size update images.
- **Scratchpad (Sector 7):** A temporary buffer used during the decryption and swapping process.

2.2 Security Mechanism

- **Cryptographic Engine:** We integrated Intel's TinyCrypt library to perform cryptographic operations . Although TinyCrypt codebase is platform-free, its default build and library configuration targets UNIX-like systems. We ported the its codebase compatible to the ARM Cortex-M GCC toolchain used in the STM32CubeIDE environment. Therefore, we created a software-based cryptographic backend.
  Applying crpytography in software guarantees portability across different STM32 families that may lack dedicated crypytographic hardware peripherals. However, the modular design of the code enables devices equipped with hardware cryptographic accelerators to seamlessly replace the software backend easily.This design guarantees bootloader to be seamlessly deployed on a wide spectrum of devices, from low budget MCUs to high-end performance lines.

- **Verification and Confidentiality:** We developed a Python-based "binary packaging tool" that is integrable into the application software build process to ensure firmware integrity and confidentiality. This tool performs a two-stage security procedure on the compiled application binary. Firstly, it calculates the SHA-256 hash of the binary to enable bootloader to ensure authenticity and then signs it using the developer's ECDSA private key. A metadata footer containing this signature, version information and SHA-256 hash is appended to the binary.
  Secondly, to achieve confidentiality of the binary during the transmission of the binary file, the tool encrypts the entire signed package using AES-128 in CBC mode. It generates a random Initialization Vector (IV) for each update, which is stored in a update header appended to the encrypted payload. The Python script is designed to work on every build chain, allowing developers to easily integrate it into systems like CMake or Makefiles to automatically generate secure, ready-to-flash update packages.

- **Memory Protection:** The Memory Protection Unit (MPU) of the Arm Cortex-M7 is configured to establish a hardware-enforced trust boundary for the system's most critical component. The Bootloader region (Sectors 0 & 1) is explicitly marked as Privileged Read-Only (RO). This configuration prevents the application or any runaway code from modifying the bootloader, effectively neutralizing threats where malware might attempt to inject itself into the boot sequence to gain permanent control. In contrast, the Configuration Sector is intentionally configured as Read-Write (RW).
  This design choice is architecturally necessary because this sector serves as the non-volatile state storage for the bootloader's finite state machine. The system must be able to write to this sector dynamically to persist critical flags—such as `STATE_UPDATE_REQ` or `STATE_ROLLBACK`—across system resets. Making this sector Read-Only would render the bootloader unable to track its own state or perform updates. The remaining memory regions, including the Active Application slot, currently follow the default privileged access rules.

2.3 Software Update Logic

One of the design decisions we made in this framework is the decoupling of the firmware transport layer from the secure boot logic. The Active Application is responsible for receiving the signed binary using whichever peripheral is appropriate for the deployment environment, writing the data to the Download Partition (Slot B), and notifying the bootloader by updating the `system_status` flag in the shared Configuration Partition.

This architectural decision makes the bootloader lightweight and secure by minimizing the attack surface, as it does not directly handle external data transmission. It also ensures compatibility with any application; while we implemented a UART-based transmission for testing, developers can freely utilize Ethernet, Wi-Fi or other communication protocols for their specific use cases.

Update Trigger Mechanism: To initiate the update process after downloading the binary, the application should modify the persistent state stored in the Configuration Sector (Sector 2). Since this region resides in the internal Flash memory, the application is required to execute the following sequence to communicate with bootloader:

1. **Unlock Flash Access:** The application must first unlock the Flash control register using `HAL_FLASH_Unlock` to permit write and erase operations on the non-volatile memory.

2. **Sector Erasure:** The Configuration Sector (Sector 2) must be erased completely with `HAL_FLASHEx_Erase`.

3. **State Programming:** The application populates a `BootConfig_t` structure (defined in `bootloader_interface.h`) with the magic number `0xDEADBEEF` and sets the system status flag to `STATE_UPDATE_REQ` (integer value: 5). It then programs this structure to the base address of Sector 2.

4. **Lock Flash:** The Flash control register must be locked immediately (using `HAL_FLASH_Lock`).

5. **System Reset:** Finally, the application must trigger a soft reset (e.g., calling `NVIC_SystemReset()`). This

forces the CPU to restart execution from the Bootloader's address (Sector 0) and allows the bootloader to detect the newly set request flag and begin the update process.

However, the condition that triggers this sequence is flexible. The framework does not impose a strict update trigger policy; developers can integrate the update trigger logic based on their specific requirements, such as a scheduled timer, a network command, or a specific diagnostic event. To validate this flexibility during our testing, we implemented a manual update trigger using the on-board User Button. It is important to note that this physical button is primarily intended for debugging and demonstration purposes. In a production environment, this logic can easily be replaced by a software-based trigger, effectively giving the developer the choice of control.

3 Implementation Details

3.1 Static Code Size Optimization for Lightweight Design

A critical requirement for the bootloader was to remain lightweight enough to be deployed on constrained devices with limited Flash memory. Also, our primary objective in developing this bootloader is to maximize the Flash memory available for user applications. Although our development platform, the STM32F746G, features a generous 1MB of Flash memory, we must apply code size optimization because widely adopted entry-level devices like the STM32F1 series are typically limited to just 128KB.

We avoided the standard C input/output library `stdio.h`. Standard printf implementations typically introduce significant code bloat.Instead, we implemented a custom console driver, `tiny_printf`. This module provides a streamlined implementation of printf that supports only the essential format specifiers required for debugging and bypasses the standard library to write directly to the UART peripheral in debugging.

To ensure firmware portability across different STM32 device families, we utilized the STM32 Hardware Abstraction Layer (HAL). While effective for abstraction, the HAL introduces significant code size overhead due to its comprehensive error handling and state management features. To mitigate this, we employed a two-tiered optimization strategy.

First, we configured the compiler with the -Os (Optimize for Size) flag, instructing it to prioritize binary size reduction over execution speed for individual compilation units. Second, and most critically, we enabled Link Time Optimization (-flto). This setting allows the linker to perform cross-module analysis of the entire program structure. By viewing the application as a single unit, the linker can identify and rigorously discard unused functions, data structures, and peripheral drivers that would otherwise be retained in the final binary. These combined strategies successfully reduced the static code size from 88 KB to approximately 32 KB.

3.2 Boot State Machine

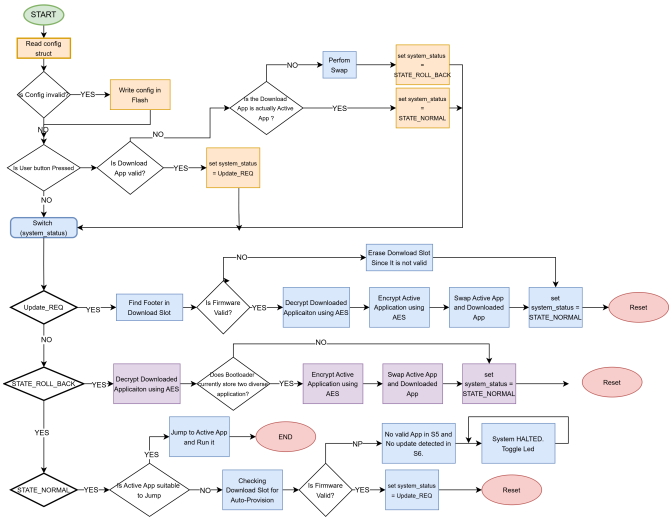

Figure 3: Flow Chart

The core boot logic is in `main.c`, implementing a finite state machine that manages the device's startup behavior. Upon reset, the bootloader performs the following sequence:

1. **Initialization:** Configures the system clock, initializes the HAL library, and re-enables interrupts (`__enable_irq`) to allow peripheral operation.

2. **State Retrieval:** Reads the `BootConfig_t` structure from the dedicated Configuration Partition (Sector 2) to determine the last known system state.

3. **Update Handling:** If the retrieved state is `STATE_UPDATE_REQ`, the system initiates the update process by calling the Swap Manager.

4. **Application Verification & Jump:** If the state is `STATE_NORMAL`, the bootloader proceeds to verify the Active Application (Sector 5). This involves two distinct checks:

   - **Cryptographic Check:** The `Firmware_Is_Valid()` function calculates the SHA-256 hash of the application and verifies it against the ECDSA signature stored in the footer.
   - **Sanity Check:** The Reset Vector (first word of the binary) is checked to ensure it points to a valid address within the application's memory range.

   If both checks pass, the bootloader transfers control to the application using the `Bootloader_JumpToApp` method.

5. **Automatic Recovery:** If the Active Application is invalid (signature mismatch) or empty, the bootloader scans the Inactive Slot (Sector 6). If a valid firmware image is detected there, the system automatically transitions to `STATE_UPDATE_REQ`, saves the new state, and resets.

This triggers the installation of the valid image, self-healing the device.

6. **Critical Error:** If no valid application exists in the Active Slot and no valid replacement is found in the Inactive Slot, the system enters a critical error state. Program execution halts, and the error indicator LED (GPIO PI1) is toggled to signal a fatal failure.

7. **Rollback Mode:** The state machine also supports a `STATE_ROLLBACK` mode. If the user holds the User Button during reset but the firmware in the Download Slot is an older or equal version (indicating a backup), the system interprets this as a rollback request. It invokes `BL_Rollback()` to restore the previous stable firmware and transitions back to `STATE_NORMAL`.

3.3 Cryptographic Verification

Our code abstracts the complexity of the TinyCrypt library by implementing a high-level wrapper in `Cryptology_Control.c`. The verification function, `Firmware_Is_Valid`, first locates the firmware footer by scanning backwards from the end of the firmware slot for the distinct `FOOTER_MAGIC` marker code. Once the footer is parsed, the function reads the firmware payload directly from flash memory and calculates its SHA-256 digest. Finally, instead of comparing the hash directly, it uses this calculated digest to cryptographically verify the ECDSA signature stored in the footer using the `uECC_verify` function, ensuring the binary was signed by the authorized private key holder.

3.4 Secure Update Manager

Once the downloaded binary is verified by the cryptographic mechanism, the update process is orchestrated by the `BL_Swap_NoBuffer` function in `BL_Functions.c`. To ensure atomicity and prevent data loss during power failures, the system utilizes a **Scratch Partition (Sector 7)** as an intermediate buffer. The update workflow executes in three synchronized stages:

1. **Decryption:** The function first decrypts the AES-128-CBC encrypted update package from the Download Slot (Sector 6) and writes the plaintext firmware to the Scratch Partition.

2. **Backup:** Before overwriting the active firmware, the system reads the current application from the Active Slot (Sector 5), encrypts it using AES-128-ECB, and saves it to the Download Slot. This ensures a valid backup is available for rollback.

3. **Installation:** Finally, the validated plaintext firmware is copied from the Scratch Partition to the Active Slot (Sector 5), completing the update.

3.5 Deployment of the Framework to the Devices

To deploy the secure bootloader framework, the following configuration steps must be performed to establish the Root of Trust and ensure correct memory mapping for the user application:

1. **Key Generation:** First, the cryptographic keys required for the system are generated using the `keygen.py` script. This script produces the AES-128 secret key for encryption and the ECDSA key pair (Private/Public) for signing. To facilitate insertion into the embedded code, the `extract_pubkey.py` utility is executed, which formats and prints the `AES_SECRET_KEY` and the `ECDSA_public_key_xy` (Public Key) as C-style arrays, as shown in Figure 4.

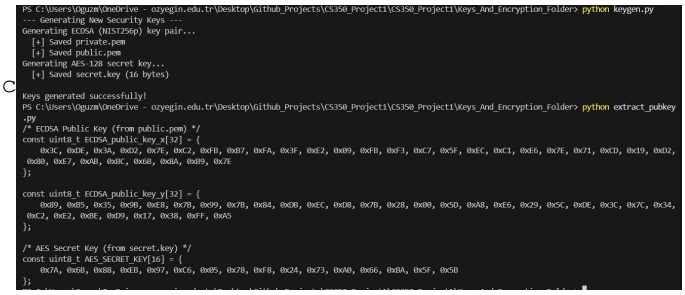

Figure 4: Keygen.py and extract_pubkey.py outputs

1. **Shared Configuration Integration:** Since the Bootloader and Application are separate projects, they must share a common understanding of the memory map and data structures. Two critical header files must be included in the Application's source tree:

   - `mem_layout.h`: This file serves as the "Single Source of Truth" for the physical memory partitioning (e.g., defining that the Active Slot starts at Sector 5). It ensures that both the Bootloader and Application calculate addresses (like `APP_ACTIVE_START_ADDR`) identically.
   - `bootloader_interface.h`: This file defines the logical interface, specifically the `BootConfig_t` structure and the status flags (e.g., `STATE_UPDATE_REQ`). This allows the application to correctly format the data written to the Configuration Sector (Sector 2) when requesting an update.

2. **Key Insertion:** The cryptographic keys generated in the previous step must be embedded directly into the Bootloader's source code to establish the chain of trust. The formatted arrays are copied into `Core/Src/keys.c`. This hardcodes the AES key for decryption and the ECDSA Public Key for signature verification into the bootloader's immutable read-only memory.(*See Figure 5*)

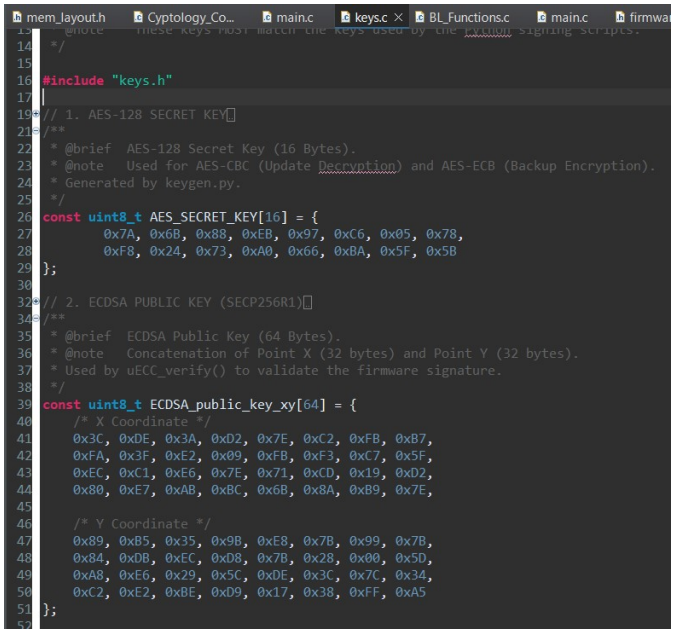

Figure 5: Assinging keys into the keys.c file

3. **Application Linker Configuration:** To ensure the user application executes from the designated Active Slot (Sector 5) rather than overwriting the bootloader (Sector 0), the application's linker script (`.ld`) must be modified. As defined in the memory layout, the `FLASH` origin is offset to `0x08040000`, and the length is restricted to 256KB to fit within the slot boundaries.(*See Figure 6*)

```
/* Entry Point */
ENTRY(Reset_Handler)

/* Highest address of the user mode stack */
_estack = ORIGIN(RAM) + LENGTH(RAM); /* end of "RAM" Ram type memory */

_Min_Heap_Size = 0x200; /* required amount of heap */
_Min_Stack_Size = 0x400; /* required amount of stack */

/* Memories definition */
MEMORY
{
  RAM    (xrw)    : ORIGIN = 0x20000000,   LENGTH = 320K
  FLASH    (rx)    : ORIGIN = 0x08040000,   LENGTH = 256K
}
```

Figure 6: Application's Linker File Configuration

4. **Vector Table Relocation:** Since the application is shifted to a new address, the Interrupt Vector Table must be relocated to match the new flash origin. In `system_stm32f7xx.c`, the `USER_VECT_TAB_ADDRESS` macro must be uncommented, and the `VECT_TAB_OFFSET` must be set to `0x00040000` (the offset for Sector 5). This ensures that the Cortex-M7 core (SCB → VTOR) correctly jumps to the application's interrupt handlers instead of the bootloader's.(*See Figure 7*)

```
#define USER_VECT_TAB_ADDRESS

#if defined(USER_VECT_TAB_ADDRESS)
/*!< Uncomment the following line if you need
     in Sram else user remap will be done in
/* #define VECT_TAB_SRAM */
#if defined(VECT_TAB_SRAM)
#define VECT_TAB_BASE_ADDRESS   RAMDTCM_BASE

#else
#define VECT_TAB_BASE_ADDRESS   FLASH_BASE

#endif /* VECT_TAB_SRAM */
#if !defined(VECT_TAB_OFFSET)
#define VECT_TAB_OFFSET         0x00040000U

#endif /* VECT_TAB_OFFSET */
#endif /* USER_VECT_TAB_ADDRESS */
```

Figure 7: Application's system_stm32f7xx.c Configuration

5. **Interrupt Re-enabling:** To ensure a safe transition, the bootloader disables all interrupts globally before jumping to the application. Consequently, the application must explicitly re-enable interrupts at the very beginning of its execution. This is achieved by calling the `__enable_irq()` intrinsic function within the `main()` initialization sequence, as shown in Figure 8. Without this step, the application's peripheral drivers (e.g., HAL Timebase, UART) would fail to function.

```
HAL_Init();

/* USER CODE BEGIN Init */

__enable_irq();
/* USER CODE END Init */

/* Configure the system clock */
SystemClock_Config();

/* USER CODE BEGIN SysInit */
/* USER CODE END SysInit */

/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_USART1_UART_Init();
/* USER CODE BEGIN 2 */
//uint32_t current_sp = __get_MSP();
  printf("Application1'den selam\r\n");
  //printf("Current Stack Pointer (MSP): 0x%08lX\r\n", current_sp);

  HAL_GPIO_WritePin(GPIOI, GPIO_PIN_1, GPIO_PIN_SET);
```

Figure 8: Basic Structure of Application's main.c file

6. **Firmware Packaging:** Once the application is compiled into a raw binary (`.bin`), it must be packaged with cryptographic metadata to be accepted by the bootloader. The `generate_update.py` script is used for this purpose. It accepts the application binary, version number, and cryptographic keys as input. The script calculates the SHA-256 hash, signs it with the ECDSA private key, encrypts the binary with AES-128-CBC, and appends the custom footer containing the signature and version information. The output is a secure firmware blob ready for transmission.(*See Figure 9*)

Figure 9: Generate_update.py encrypts the compiled application .bin file

7. **Readout Protection (RDP):** For the final production deployment, the STM32 Readout Protection (RDP) Level 1 or Level 2 mechanism must be enabled. This hardware feature locks the Flash memory from external access via the debug interface (SWD/JTAG), effectively preventing physical tampering, reverse engineering, and firmware cloning.

   Crucially, enabling RDP renders standard external tools such as STM32CubeProgrammer or ST-Link unusable for flashing or debugging the device. Consequently, to ensure field updatability without reliance on these locked hardware interfaces, we implemented a dedicated UART terminal solution on the STM32F4 system. This allows the device to receive secure firmware updates directly via serial communication. In this project, however, RDP was intentionally left disabled to maintain debug access for development and testing purposes.

4 Runtime Operation and Update Logic

The core operational logic of the bootloader is governed by a finite state machine (FSM) implemented in `main.c`. This FSM orchestrates the decision-making process between normal boot, firmware updates, and system recovery.

4.1 Startup & User Interaction

Upon system reset, the bootloader initializes the HAL and system clock, then immediately checks the state of the User Button (GPIO PI11). This physical interaction serves as a hardware trigger for manual interventions:

- **Normal Boot (Button Released):** The system defaults to checking the configuration sector. If no update flag is set, it proceeds to verify the Active Application.
- **Update Request (Button Pressed):** If the button is held during reset, the bootloader scans the Download Slot (Sector 6). If a valid, new firmware image is detected (verified by a newer version number in the footer), the state is forced to `STATE_UPDATE_REQ`.
- **Manual Rollback (Button Pressed + No Update):** If the button is held but the Download Slot contains an older or equal version (indicating a backup exists), the system infers a rollback request. The state transitions to `STATE_ROLLBACK`, allowing the user to restore the previous stable firmware.

4.2 Cryptographic Verification Chain

Before any code is executed or installed, it must pass a rigorous "Authenticate-then-Decrypt" verification process in `Cryptology_Control.c`:

1. **Footer Parsing:** The system scans the end of the Download Slot for the `FOOTER_MAGIC` marker to locate metadata.
2. **Integrity Check:** The SHA-256 digest of the *encrypted* binary blob is calculated.
3. **Authenticity Check:** This digest is verified against the ECDSA signature stored in the footer using the public key embedded in `keys.c`. This ensures the code originates from a trusted source.

4.3 Atomic Swap Mechanism

To ensure system resilience against power failures, the update process uses a Scratch Partition (Sector 7) as an intermediate buffer. The `BL_Swap_NoBuffer` function executes the following atomic sequence:

1. **Decryption:** The AES-128-CBC encrypted image is read from the Download Slot, decrypted using the secret key, and written to the Scratch Partition.
2. **Backup (Anti-Brick):** The current valid Active Application (Sector 5) is read, encrypted with AES-128-ECB, and written to the Download Slot. This overwrites the update package with a backup of the old system.
3. **Installation:** The decrypted new firmware is copied from the Scratch Partition to the Active Slot.
4. **State Cleansing:** The configuration sector is updated to `STATE_NORMAL` to prevent boot loops.

4.4 Bootloader Handover

If the verification succeeds (or after an update completes), the bootloader performs the final handover sequence:

1. It disables all interrupts to ensure a clean state.
2. It reads the application's Reset Vector from the beginning of Sector 5.
3. It sets the Main Stack Pointer (MSP) to the application's stack address.
4. It jumps to the application's entry point.

The application then immediately calls `_enable_irq()` to restore interrupt functionality and begins execution.

5 Testing and Validation

We validated the reliability and security of the framework through a combination of unit tests for individual components and comprehensive integration tests.

5.1 Unit Testing

To ensure the correctness of critical security and memory mechanisms before full system integration, we implemented the following runtime self-tests:

- **MPU Violation Test:** To verify memory isolation, we implemented a test function `Test_MPU_Violation()` that attempts to write data to the Bootloader's protected flash region (`0x08000000`). As expected, this operation triggered a `MemManage_Handler` hard fault. This shows that the MPU correctly prevents the active application from modifying the Root of Trust.
- **Cryptographic Testing:** We included a `BL_Test_Crypto()` function to validate the TinyCrypt library port. This test compared the output of on-device SHA-256 hashing and AES-CBC encryption/decryption operations against known standard test vectors. The completion of these tests confirmed that the software cryptographic backend was functioning correctly.

5.2 Integration Testing

We performed a complete end-to-end secure update scenario to validate the interaction between the Application, Bootloader, and Cryptographic Engine. The test procedure was as follows: To validate the system's decision-making logic, we tested the bootloader under four distinct runtime conditions. The following console logs document the system's behavior in each case.

1. **Critical Failure & Halt (Figure 10):** When both the Active Slot (Sector 5) and the Download Slot (Sector 6) are empty, the bootloader correctly identifies that the system is unbootable. It outputs "System Halted" and toggles the error LED, preventing undefined behavior.

```
=====================================
Starting Bootloader Version-(1,7)
=====================================
[BL] State: NORMAL. Checking Active Application (S5)...
[BL] S5 Empty or Invalid! Checking S6 for Auto-Provisioning...
[ERROR] No valid app in S5, and no update in S6.
[ERROR] System Halted.
```

Figure 10: System Halted when both slots are empty.

2. **Auto-Provisioning / Self-Healing (Figure 11):** In this test (Figure 11), the Active Slot was intentionally erased, but a valid firmware image was uploaded to the Download Slot. Upon reset, the bootloader detected the "S5 Empty" condition, scanned the Download Slot, found a valid image, and automatically triggered an update. This proves the system can self-heal without manual intervention.

```
=====================================
Starting Bootloader Version-(1,7)
=====================================
[BL] State: NORMAL. Checking Active Application (S5)...
[BL] S5 Empty or Invalid! Checking S6 for Auto-Provisioning...
[BL] Valid Image found in S6! Triggering Update...

=====================================
Starting Bootloader Version-(1,7)
=====================================
[BL] State: UPDATE REQUESTED.
[BL] Verifying Signature... OK!
[BL] Valid Update! Ver: 256, Payload: 17424
[1/3] Decrypting S6 -> S7...
[2/3] Backing up S5 -> S6...
  [DEBUG] Erasing Backup Sector (S6)... OK
  [DEBUG] Encrypting & Backing up 262144 bytes...
  Backup Complete.
[3/3] Installing S7 -> S5...
  [DEBUG] Erasing Sector 5... OK
  [DEBUG] Writing 262144 bytes to 0x8040000... OK
[BL] Update Successful! Setting State to NORMAL.
Swap Complete. Resetting...

=====================================
Starting Bootloader Version-(1,7)
=====================================
[BL] State: NORMAL. Checking Active Application (S5)...
[BL] Valid App found at 0x8040000. Jumping...
Application1'den selam
```

Figure 11: Auto-Provisioning triggered when Active App is missing but valid firmware exists in Download Slot.

3. **Manual Rollback (Figure 12):** In this test, the `STATE_UPDATE_REQ` triggered by pressing User Button, despite, second firmware is not uploaded. Since there were no valid firmware in Download Slot, it rolled back to the Active firmware (App1).

```
=====================================
Starting Bootloader Version-(1,7)
=====================================
[BL] Button Pressed! Determining Mode...
 -> Download Slot has data (Backup). Requesting SWAP/ROLLBACK.

[BL] Starting Rollback/Toggle...
[1/3] Decrypting Backup (S6 -> S7)...
[ERROR] The Backup in S6 is Empty or Invalid!
[ERROR] Reset Vector: 0x08X. Aborting Swap to protect Active App.
[BL] Rollback Failed. Reverting state to NORMAL.

=====================================
Starting Bootloader Version-(1,7)
=====================================
[BL] State: NORMAL. Checking Active Application (S5)...
[BL] Valid App found at 0x8040000. Jumping...
Application1'den selam
```

Figure 12: Manual Rollback initiated by User Button, restoring the previous firmware (App 1).

4. **Switch to Uploaded Firmware (Figure 13):** In this test, the second firmware (App 2) is uploaded into Download Slot. Then `STATE_UPDATE_REQ` triggered by pressing User Button. This time, the Bootloader scanned the Download Slot and found new Application. Finally, it switched to newly uploaded Fimrware(App 2) and run it.

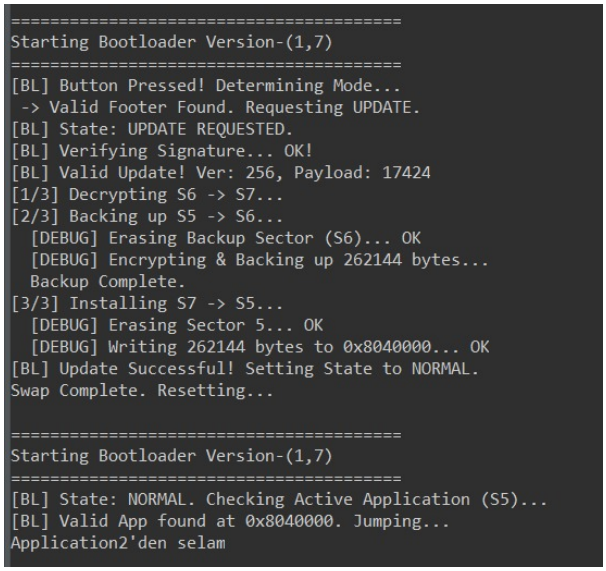

Figure 13: Manual Rollback. The user button triggered an update and runs the newly updated firmware.

5. **Swap between Applications (Figure 14):** In this test, we have two diverse firmware in both slots(App1 and App2). In order to swap from App1 to App2, we pressed the User Button. Bootloader scanned the Download Slot and found that this stored fimrware is not newly updated. So it just swapped the firmwares and run the App2.

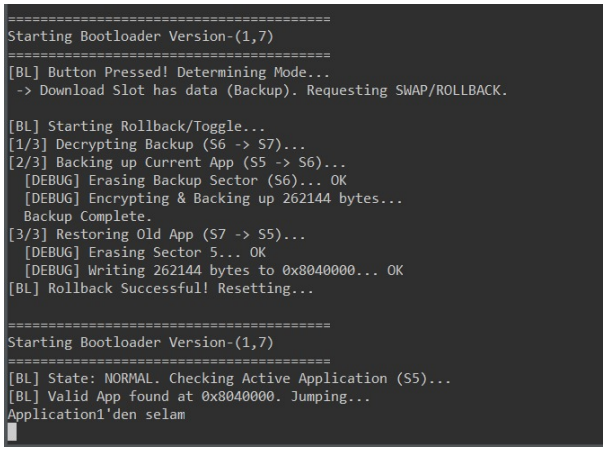

Figure 14: Manual Swap. The user button triggered a swap, restoring the previous firmware from the backup slot.

5.3 Portability Verification

To validate the cross-family portability of the framework, we deployed the bootloader on an STM32F411 microcontroller in addition to the primary STM32F746G. The STM32F411 differs significantly from the F7 series with featuring a Cortex-M4 core and a different Flash memory sector layout and less flash memory capacity. By simply reconfiguring the sector definitions in `mem_layout.h` to match the F411's memory map, we successfully compiled and ran the bootloader without modifying the core of the framework. The device successfully performed a secure boot and accepted an encrypted update. It proved the portable design of the framework.

6 Conclusion

This project delivered a fully functional software-based secure boot framework for the STM32F7. By implementing a software-based Root of Trust with industry-standard cryptography, we achieved the security goals outlined in our proposal. The addition of the MPU protection and the scratch-sector swap mechanism ensures that the system is resilient against both software attacks and power failures during updates.

7 Future Work

While the current framework provides a robust and secure foundation for firmware updates, several enhancements can further optimize performance and versatility.

- **Firmware Compression:** To reduce transmission time for low-bitrate transmission channels we can integrate a decompression stage into the update pipeline. By compressing the signed binary before encryption, we can significantly decrease the size of the update package, resulting in faster OTA updates and lower power consumption during reception.
- **Diverse Transmission Protocols:** Currently, we only provided firmware transport examples for UART protocol. Future implementations can expand the reference example applications to support Ethernet, Wi-Fi, or CAN bus for automotive use cases.

References

[1] Oğuz Mert Coşkun, Mehmet Arda Güler, and Mert Kırgın. Secure Boot Framework For STM32 Microcontrollers - Project Source Code. https://github.com/Omert2004/CS350_Project, 2026. GitHub repository.

[2] Don Johnson, Alfred J. Menezes, and Scott A. Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International Journal of Information Security*, 1(1):36–63, 2001.

[3] National Institute of Standards and Technology. Fips pub 180-4: Secure hash standard (shs). Technical Report FIPS PUB 180-4, U.S. Department of Commerce, 2015.

[4] STMicroelectronics. *RM0385 Reference manual: STM32F745/746 and STM32F755/756 advanced ARM-based 32-bit MCUs*. STMicroelectronics, rev 10 edition, 2021. Available at https://www.st.com/resource/en/reference$_manual/rm0385-stm32f745746-stm32f755756-advanced-armbased-32bit-mcus-stmicroelectronics.pdf$.