Secure Boot Implementation in AURIX™ TC375

M.TECH. PROJECT PART-I (JCD892)

Arindam Sal
2024JCS2041

November 25, 2025

Instructor: Prof. Kolin Paul

M.Tech in Cyber Security

Why Secure Boot? (Simple Motivation)

- AURIX TC375 runs critical embedded functions; safety checks exist, but **authenticity** is not guaranteed by default.
- **Secure Boot** = Only trusted, untampered firmware is allowed to execute at power-up.
- Protects against: malicious code, wrong/old images, accidental corruption.
- Outcome: device becomes a **trusted platform**—the intended task runs only under verified firmware.

User Configuration Blocks (UCBs) — Enablers

- Small Flash regions that Boot ROM reads at reset to decide *how* to boot.
- Key UCBs for secure boot:
  - **UCB_BMHD (Boot Mode Header):** start address (reset vector), integrity (CRC), options. *"Where to boot from."*
  - **UCB_HSM... (HSM configuration/OTP):** where HSM boot code is, key material/public key hash.
  - **UCB_PFLASH/DFLASH:** read/write/execute protections (lock code and data).
  - **UCB_DBG:** debugger policy (enable during dev, lock in production).

What UCB_BMHD Does (Clear Picture)

- Boot ROM scans BMHD entries; on a **valid BMHD** it jumps to the **Start Address (STAD).**
- Our plan: set STAD to a tiny **Secure Boot Stub** in PFLASH.
- If BMHD is invalid/missing: Boot ROM drops to **BSL** (bootstrap loader). In production we restrict/disable BSL.

Default Boot Process (AURIX TC375)

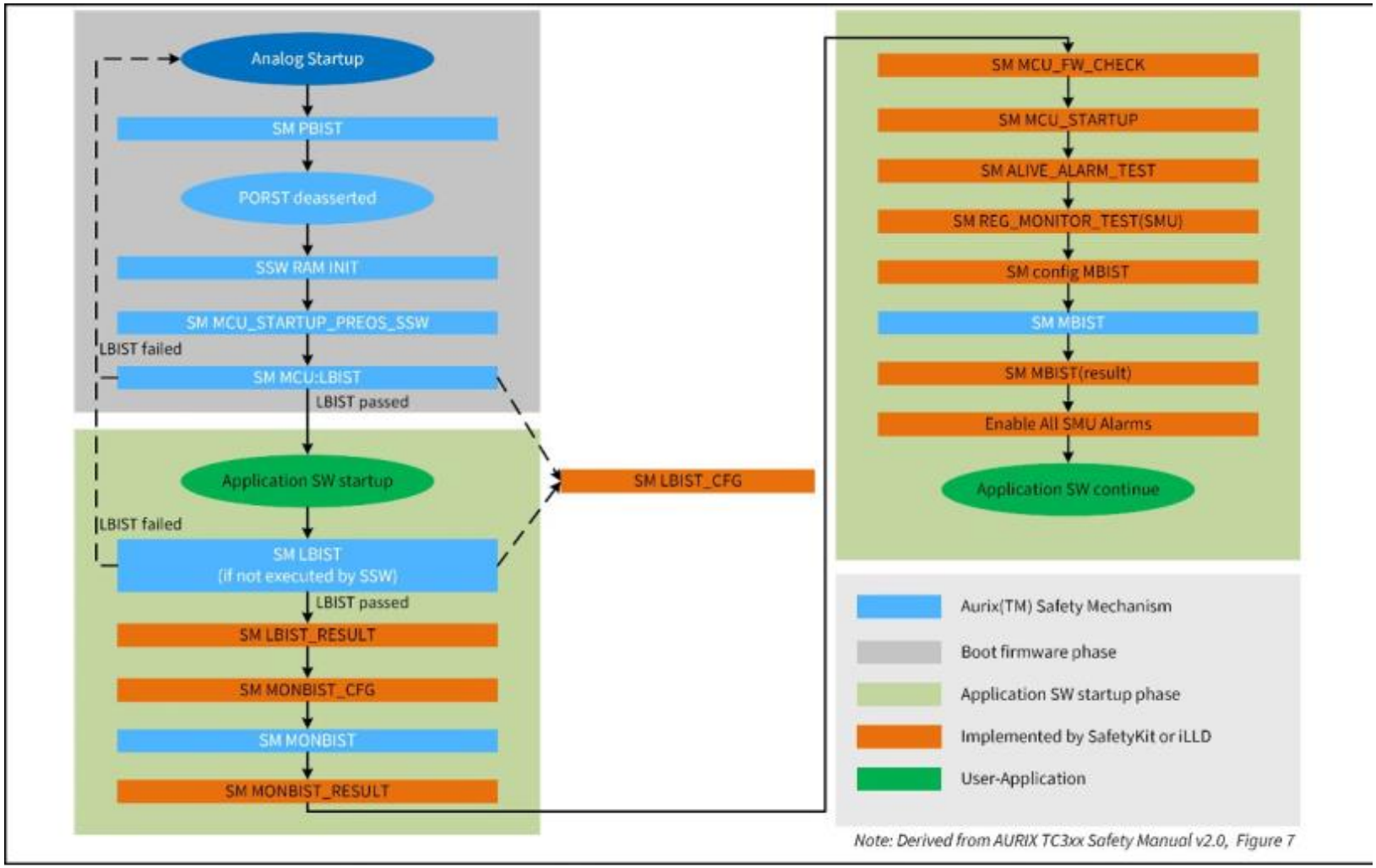


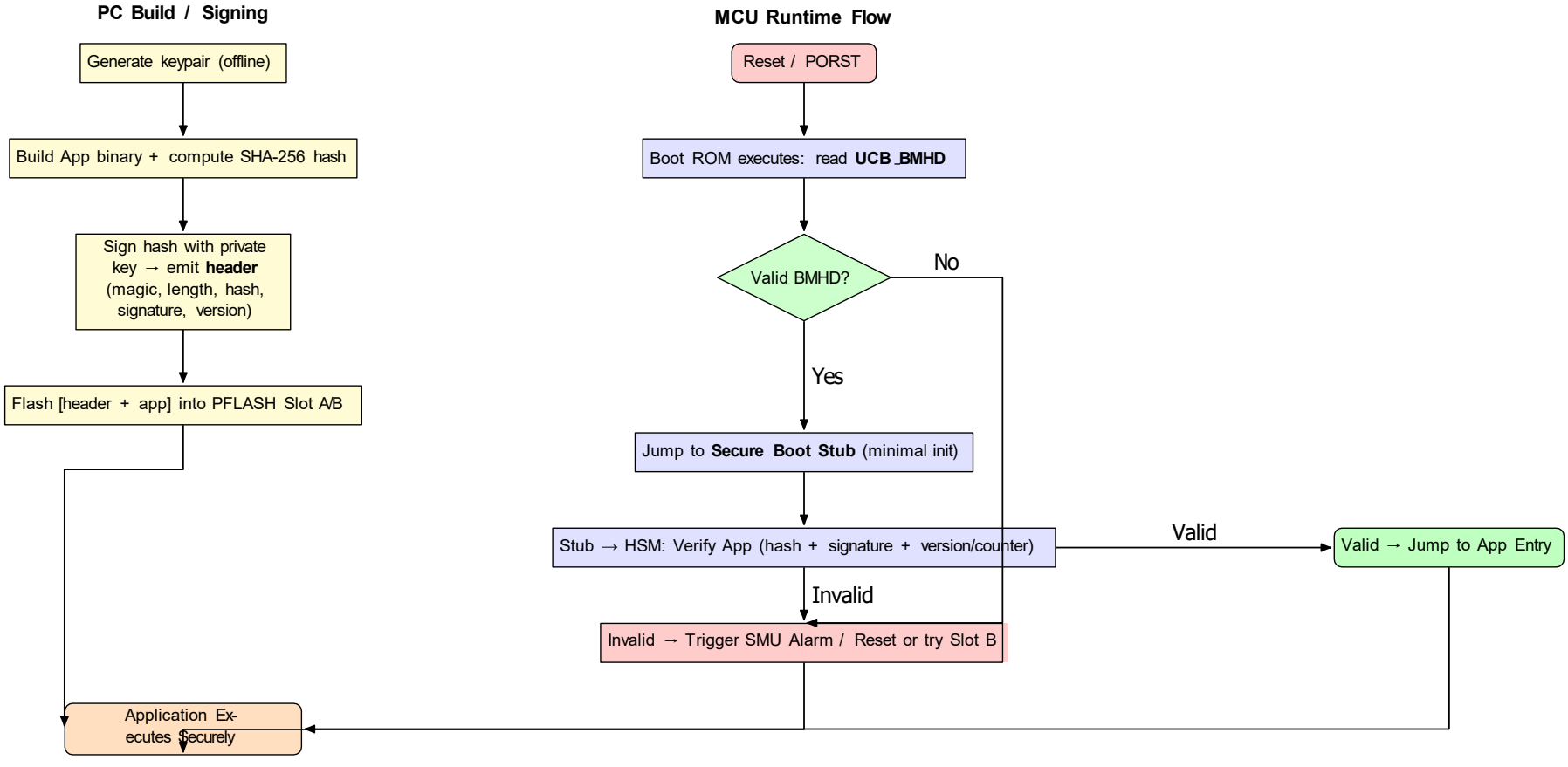
Figure 8 Safety mechanisms sequence during startup

Takeaway: Built-in firmware (SSW/CHSW) performs initialization and safety checks. **Secure boot adds cryptographic authenticity on top.**

Proposed Secure Boot Architecture (High-Level)

1. **BMHD** points to **Secure Boot Stub** (TriCore host).
2. **Stub** does minimal init, then asks **HSM** to verify the real application image.
3. The **Application** region in PFLASH contains *[header with hash+signature+version]* + *code*.
4. **HSM** uses provisioned public key to verify. If valid ⇒ jump to **Application Entry**; else ⇒ safe reset/SMU alarm (or fall back to Slot B).
5. Lock debug, protect Flash, manage anti-rollback counters in DFLASH.

End-to-End Secure Boot Flow (Detailed View)



Memory Layout (Where Things Live)

PFLASH (program flash)

[HSM boot code & HSM data (reserved area)] ← used by the HSM only
[Secure Boot Stub] ← **BMHD points here (start address)**
[Application Slot A: app + signing header]
[Application Slot B: app + signing header] ← optional fallback slot

DFLASH (data flash)

[Version/rollback counter, settings] ← monotonic counters

UCBs (special config sectors)

UCB_BMHD (boot header → start address = stub)
UCB_HSM... (tells HSM where its boot code lives, keys)
UCB_P/DFLASH (R/W/X protections)
UCB_DBG (debug policy)

Implementation Steps (Checklists)

1. **Keys & Header:** Define image header (hash, signature, version). Generate signing keypair; keep private key offline.
2. **Place-code:** Link **Stub** at BMHD STAD; place App in Slot A (optionally Slot B); reserve HSM region.
3. **HSM Boot Code:** Initialize HSM, hold public key, verify App on request, enforce anti-rollback.
4. **Program UCBs:** BMHD→Stub; HSM config; Flash protections; Debug policy.
5. **Boot Logic:** Stub → HSM verify; on pass → App Entry; on fail → reset/alarm or Slot B.
6. **Harden:** Disable BSL in production, lock UCBs, lock debug, protect P/DFLASH.
7. **Test:** Good image boots; tamper/rollback fails; debug attach blocked when locked.

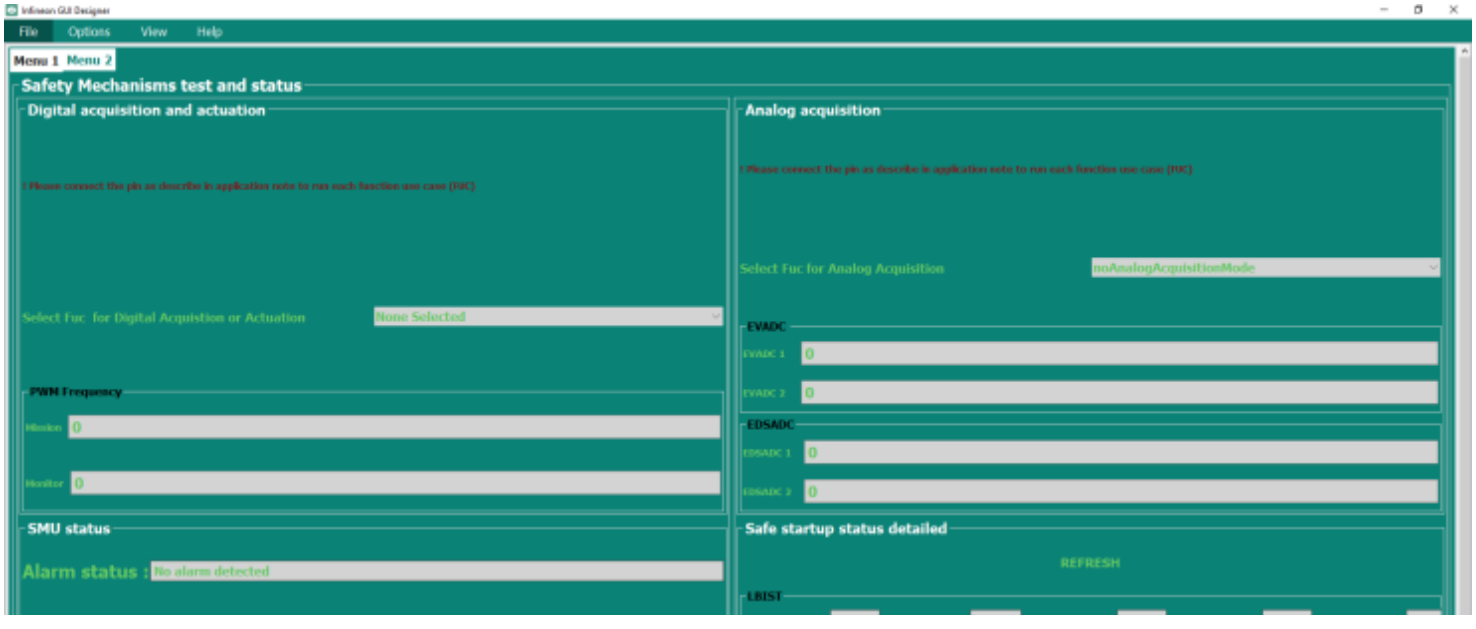
Takeaway

- BMHD selects the **Secure Boot Stub**; the stub delegates trust to **HSM** for cryptographic verification.
- Only after **Valid** does control pass to the **Application Entry**.
- The approach is **general**: any future application image runs securely as long as it is signed.

Hardware bring-up & live status

- Brought up the **TC375 Safety Lite Kit**; verified board health and sensors in Infineon GUI.
- Safety monitors (LBIST, MBIST, FW checks) observed; board temperature/LEDs active.
- This validates a stable baseline before enabling secure-boot logic.

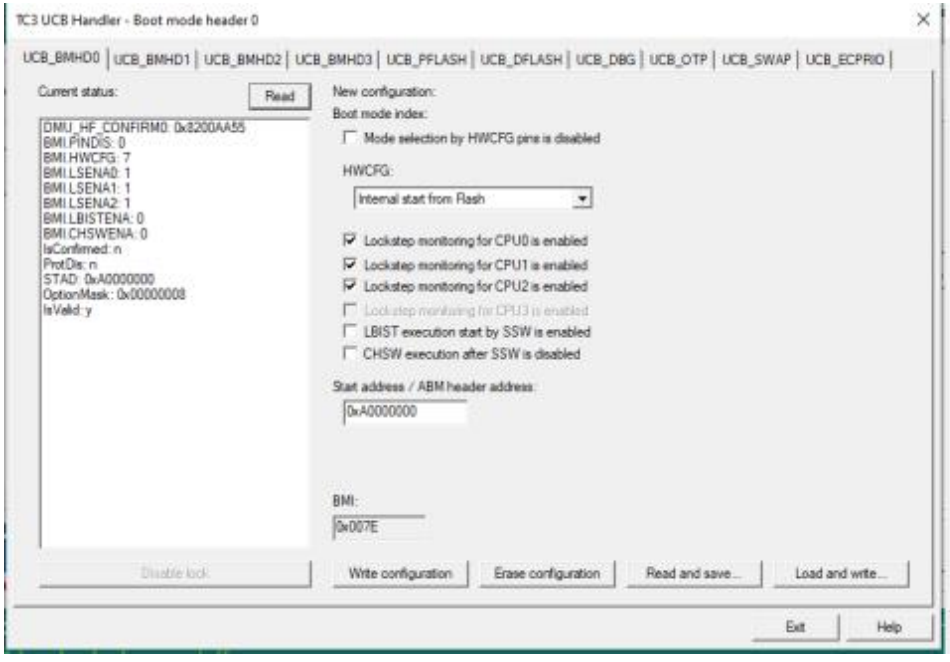
Safety mechanisms dashboard (live).

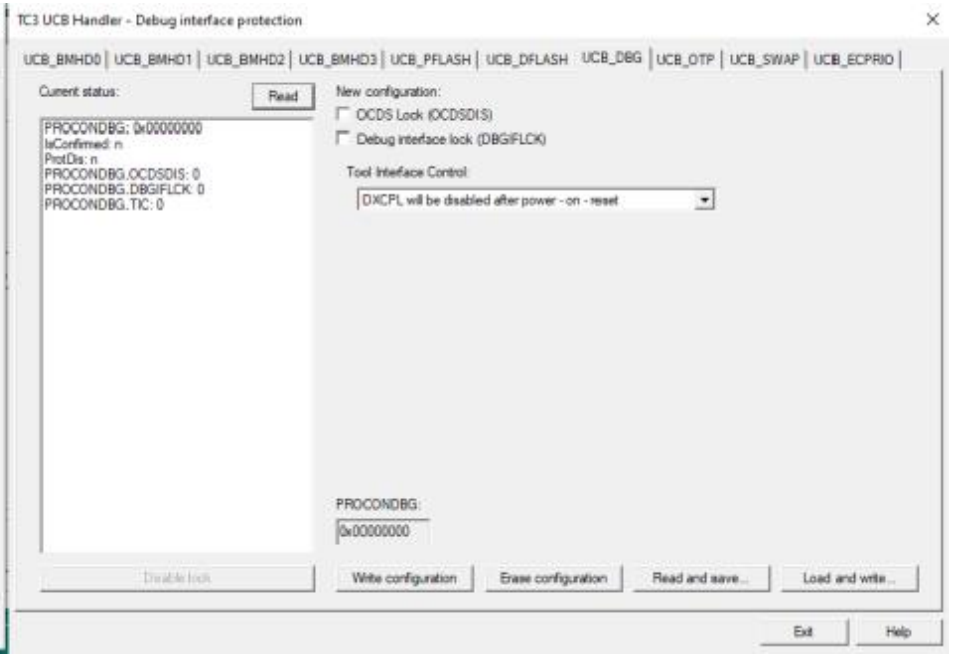


Programming path validated (Aurix Flasher)

- Connected to device (TC37x), erased and programmed PFLASH via ADS flasher.
- **Exit status: Pass** and "*Flashing was successful*".
- Confirms the toolchain + target link map are correct for our board.

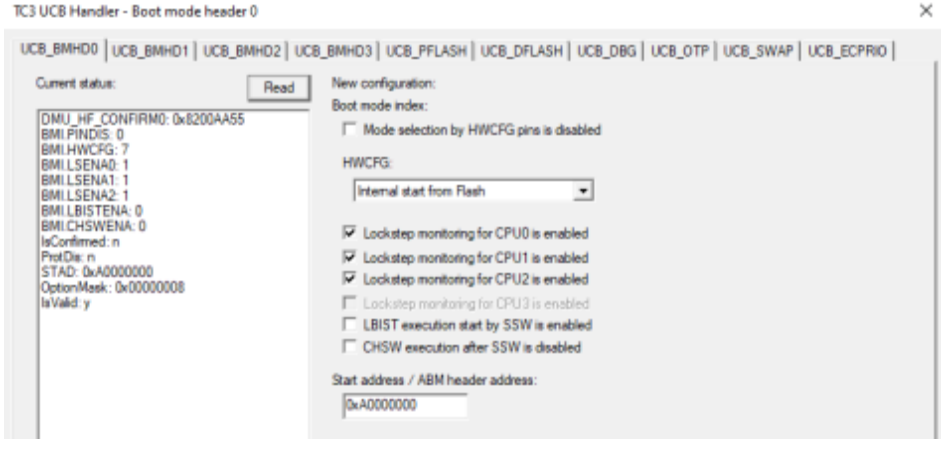
UCB configuration snapshots (what Boot ROM sees)

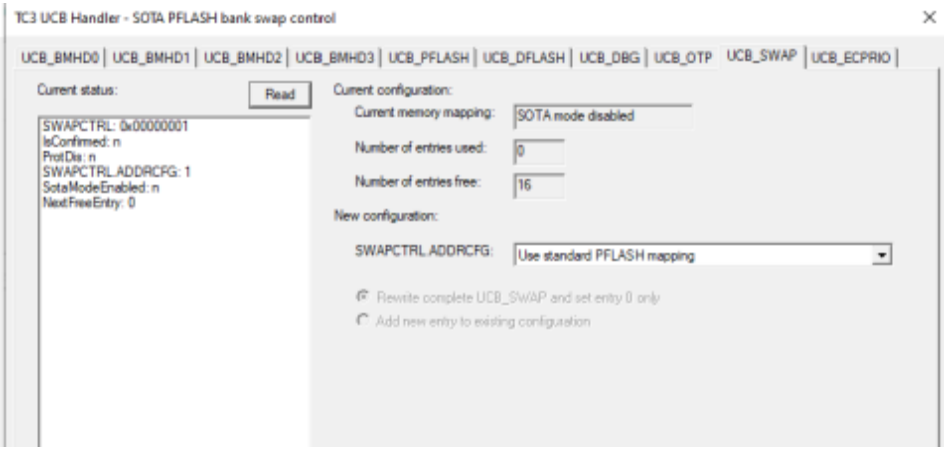




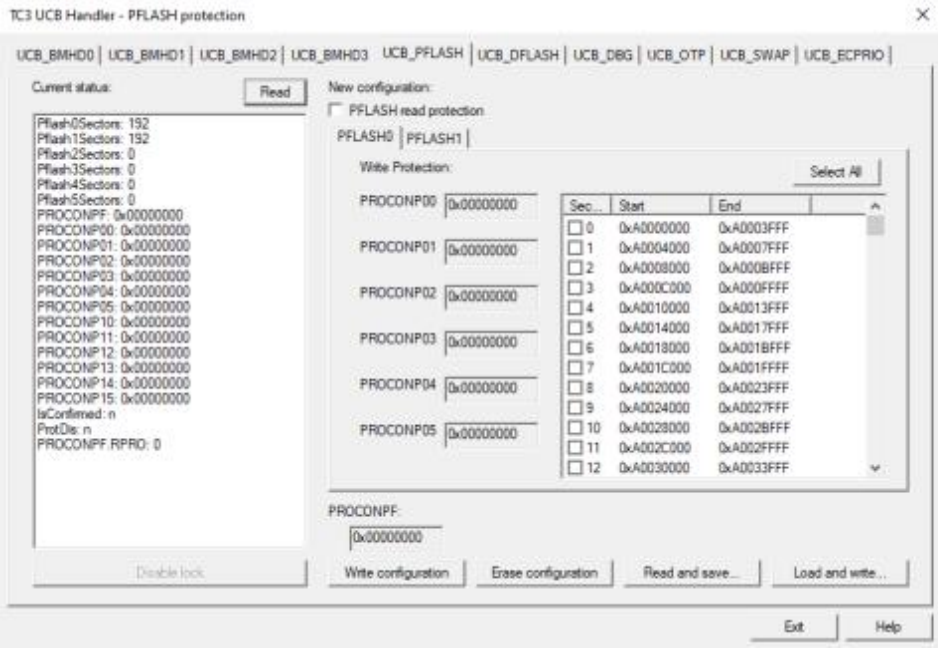
UCB_BMHD0: internal start from flash, **STAD** 0xA0000000, lockstep enabled.

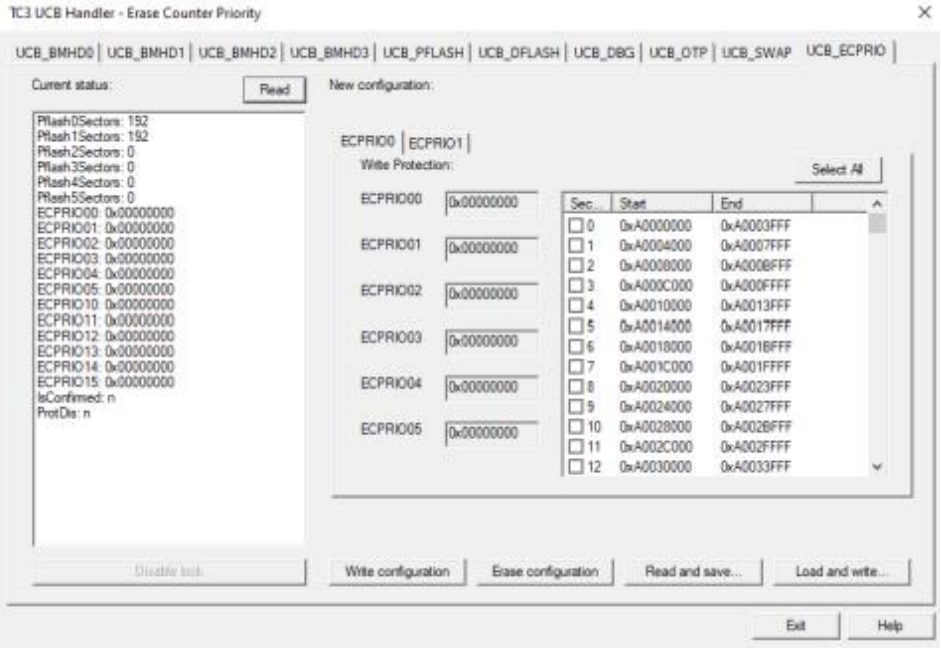
UCB_DBG: debug policy kept open for lab; will lock in production.





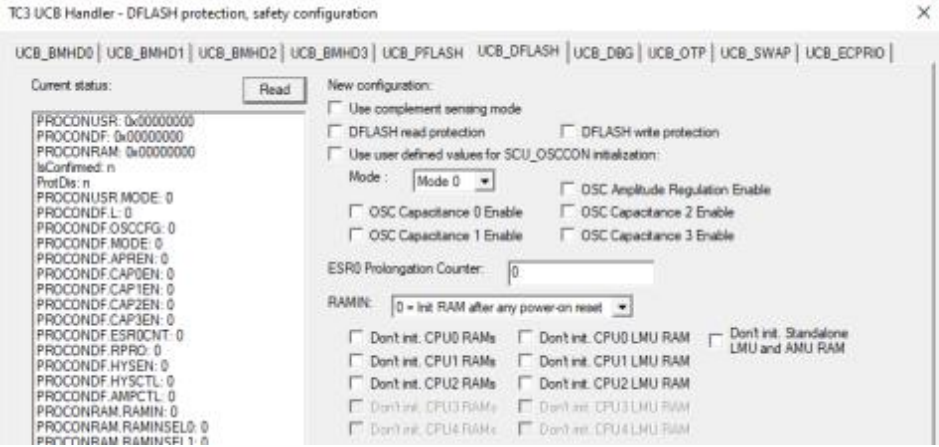
UCB protection plan (flash and OTP)

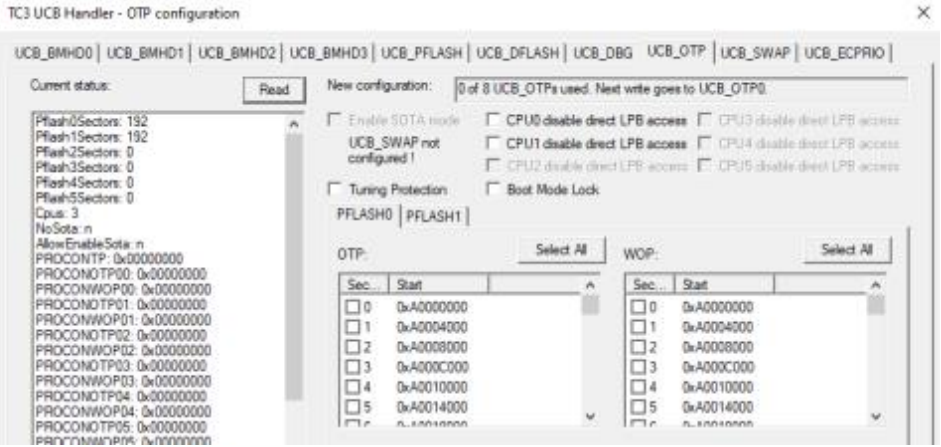




UCB_PFLASH: sectors visible; write-protect to be enabled after validation.

UCB_ECPRIO: erase counter priority overview (all unlocked for dev).





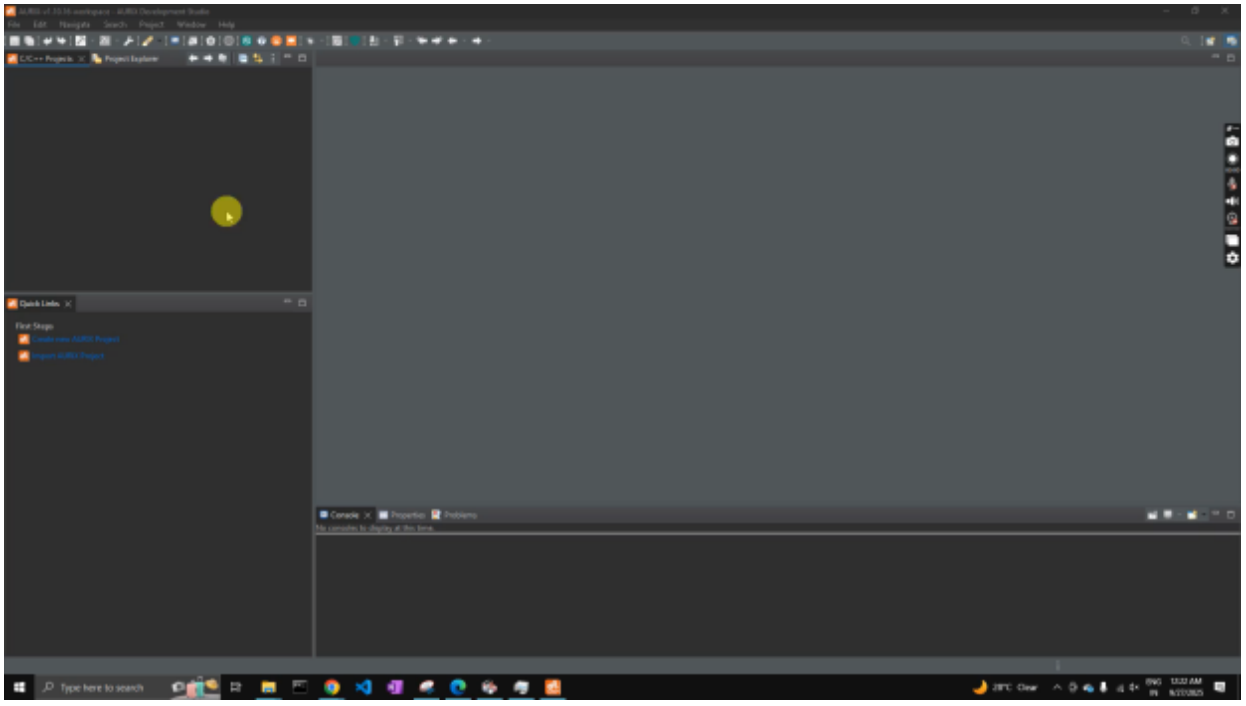
Build & signing automation (PC side)

- Post-build command:
  - Convert `ELF → BIN` with `tricore-objcopy`.
  - `sign_image.py` adds header & signature, then **verify**.
- Console shows **Verification PASSED:**
  - Header CRC, SHA-256, ECDSA all OK.
  - Example header fields captured in screenshot (magic, version, entry, image_len, monotonic).
- This pipeline is reusable for any future application image.

About the `image_len` appearing "too large" (what happened and why it's OK)

- At one point, **image_len ≈ multi-GB** appeared in the verify log.
- Root cause in that run:
  - Converting from `.hex` with address offsets (e.g., `--change-addresses`) plus implicit padding (`--gap-fill`) can create a *sparse address map* that, when forced to raw `.bin`, expands to a huge file up to the highest referenced address.
  - In other words: harmless toolchain side-effect; not the actual code size.
- Current fix used here:
  - Convert **directly from ELF** and select only code/rodata/init sections (no full-range padding).
  - Result: `app.bin` contains only real image bytes, and `image_len` matches reality.
- Takeaway for review: the **signature verification is valid** in both cases; we've adjusted the pipeline so the header's length reflects the true payload size going forward.

Live demo videos (board & build/flash)



Rationale: the videos prove repeatability—same steps regenerate a signed image and program the board, ending with successful boot

Why these practical steps matter (research value)

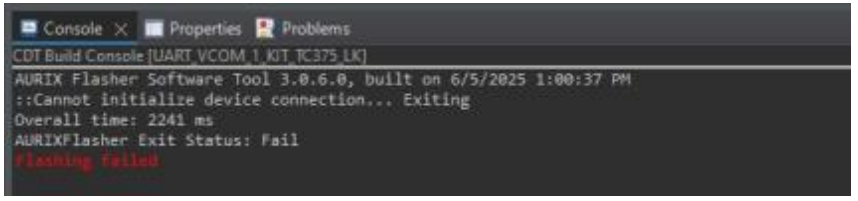
- **Traceable chain** from build artifacts to on-board execution under real hardware constraints.
- **UCB discipline** (BMHD, protections) shows how the boot vector and policy drive trust.
- **Automated signing & verify** establishes a portable delivery pipeline for later HSM-based validation.
- **Observability** (GUI/console/video) makes the results auditable and easy to reproduce in lab/CI.

Next steps (what remains to reach full secure boot)

1. Provision **HSM** and OTP with the public key (or key hash) and minimal HSM boot code.
2. Implement the **Secure Boot Stub** at STAD (BMHD) to request HSM verification before app entry.
3. Add **DFLASH monotonic counters** (anti-rollback) and wire them into header checks.
4. Enable **dual-slot (A/B)** layout and optional **UCB_SWAP** strategy for safe updates.
5. Lock **UCB_P/DFLASH** and **UCB_DBG** (production) and document recovery paths.
6. Negative tests: tamper image, wrong key, old version — expect denial + SMU alarm/reset.

Why the Flasher Shows "Cannot Initialize Device Connection"

- After completing signing, UCB configuration, and validation, I applied the **production secure-boot locks**.
- These locks enforce:
  - **UCB_DBG**: Debug/DAP fully disabled at power-up.
  - **UCB_BMHD**: Boot vector fixed to the Secure Boot Stub (immutable STAD).
  - **UCB_PFLASH/DFLASH**: Flash write-protection enabled.
- Under these conditions:
  - The BootROM **never enables** the DAP/JTAG interface.
  - The debugger cannot attach → DAS device handshake fails.



Flasher error after secure-boot locking. This indicates hardware-level rejection of debug access.

Why Flashing Fails

- The BootROM follows Infineon's TC3xx(TC375 in our case) device-locking rules:
  - When **UCB_DBG** = locked → **DAP is OFF permanently.**
  - When **BMHD** is fused/locked → BootROM always jumps directly to the secure stub.
  - When Flash is protected → Write/erase commands are blocked in hardware.
- As a result:
  1. **The flasher cannot open a DAP session**: hardware refuses → "Cannot initialize device connection."
  2. **The debugger cannot establish a channel**: no JTAG handshake → "Connection establishment for debugging failed."
  3. **Flashing cannot proceed**: PFLASH is write-protected → "Flashing failed."
- None of these indicate malfunction; they indicate:

Secure Boot Production Mode = No Debug, No Reflash, No Modification

Why Infineon GUI Tools Cannot Work After Production Secure Boot

- All GUI tools (OneEye, Device Scanner, AURIX Flasher) depend on **debug interfaces**.
- Once secure boot is finalized:
  - BootROM disables DAP/JTAG instantly at POR.
  - USB debug channel is not enumerated.
  - No monitoring or flashing session is allowed.
- Therefore, it is **technically impossible and not permitted by design** to reopen OneEye or re-run live dashboards after production locking.
- This is not a limitation — it is how **real automotive ECUs** behave: once validated and locked, they cannot expose internal state

GUI-based monitoring is possible **before** locking — never after.

Single-Purpose Secure Device (Manufacturing Intent Achieved)

- With BMHD fixed, flash protected, and debug disabled, the MCU has transitioned into:

  a single-purpose, production-secure embedded controller.

- Exactly mirroring automotive manufacturing:
  - OEM flashes a **signed, trusted** firmware during production.
  - Then UCBs are permanently locked.
  - The ECU will run only that application for the rest of its lifetime.
- In this project:
  - The validated application (tested earlier via GUI) is the one that remains forever.
  - All attempts to modify, reflash, or attach a debugger are **explicitly denied by hardware**.
  - This proves secure-boot integrity and prevents tampering.
- **Conclusion:** The AURIX TC375 now behaves exactly like a final, production-grade automotive ECU.

Secure Boot Success Summary

- End-to-end secure boot pipeline implemented:
  - Signed application image (SHA-256 + ECDSA). Secure Boot Stub at BMHD STAD.
  - Verified header (magic, signature, version, image len).
  - UCB configuration: BMHD, DBG, PFLASH/DFLASH protection.
- Production mode enforced:
  - Debug disabled (UCB_DBG). (Flash write protected.)
  - Boot vector immutable.
- External flashing and debugger initialization are now blocked in hardware.
- The device has become a **tamper-resistant, single-purpose automotive controller**.

Thank You

Secure Boot in Raspberry Pi

Minor Project Presentation

Presented to,
Prof. Kolin Paul
Department of Computer Science and Engineering,
IIT Delhi

Presented by,
Surajprakash Narwariya
Arindam Sal
M.Tech Cyber Security
IIT Delhi

Project Overview

Implementation of Secure Boot

Key Features:

- Secure Boot : Verifies code integrity during each boot cycle.
- Secure Logs: Maintains an immutable record of security-critical events

What is Secure Boot?

- Ensures only authenticated and untampered software is executed during the boot process.
- Establishes a chain of trust, starting from the Root of Trust (RoT).

Why Secure Boot for Raspberry Pi?

- Addresses the lack of secure boot support.
- Enhances system security for Embedded Control Units (ECUs).

Default Booting in Raspberry Pi 3

- **Power-On and Initial Setup**
  - Processor starts execution from BootROM (built-in to the SoC).
- **BootROM Execution**
  - Loads first-stage bootloader (bootcode.bin).
- **First-Stage Bootloader (bootcode.bin)**
  - Initializes SDRAM. and prepares for the second-stage bootloader.
- **Second-Stage Bootloader (start.elf)**
  - Loads GPU firmware file. Allocates memory and loads the kernel image.
- **Device Tree and Configurations**
  - Reads config.txt for hardware settings (e.g., overclocking, memory split).
  - Loads hardware description from the Device Tree Blob (DTB).
- **Kernel Loading**
  - operating system starts loading and initializing user-space services
- **Init System and User Space**
  - systemd or other init systems start user processes and services.

Issues without Architectural Support

- **Power-On Reset**
  - Processor starts execution from BootROM (built-in to the SoC).
- **BootROM Execution**
  - Read Only File, so can not be modified. and unable to verify “bootcode.bin” file.
- **First-Stage Bootloader (bootcode.bin)**
  - Proprietary firmware provided by the Raspberry Pi and closed source.
  - Can not be modified.
- **Next steps : Unable to maintain chain of trust.**

Why Not Raspberry Pi 3

- **No Writable Bootloader Storage**
  - Pi 3 bootloader (bootcode.bin) is stored on SD card, not in EEPROM.
  - Easily replaceable or tampered with, as there's no hardware-enforced protection.
- **Lacks OTP Support for Public Key Hash**
  - No secure One-Time Programmable (OTP) memory to store RSA key hash.
  - Hence, Pi 3 cannot lock boot verification to a trusted key.
- **No Native Signature Verification**
  - bootcode.bin and start.elf are proprietary binaries without public hooks for verifying signatures.
  - Lacks built-in cryptographic enforcement during early boot stages.
- **No Secure Boot Configuration Mechanism**
  - No EEPROM to configure or lock secure boot settings.
  - No way to set SIGNED_BOOT=1 or revoke dev keys like in Pi 4.

Discovering new Choices for Embedded Boards

That Provides the
Architectural Support
for the Secure Boot

Project Choices

Choose Raspberry Pi 4B over previous version of Raspberry Pi.

1. Hardware Support for Secure Boot

- The SPI EEPROM on the Raspberry Pi 4B allows for a configurable bootloader that supports cryptographic signature verification.
- The One-Time Programmable (OTP) memory on the Raspberry Pi 4B enables storing the hash of a public key, ensuring that only firmware signed with the corresponding private key can boot.

2. Improved Bootloader Architecture

- The EEPROM-based bootloader on the Raspberry Pi 4B eliminates reliance on external files like bootcode.bin (used in earlier models), reducing attack surfaces.
- It supports a unified signed boot image (boot.img) containing all critical boot files (e.g., kernel, firmware), simplifying signature verification and enhancing security.
- On older models like the Raspberry Pi 3, bootloader functionality is dependent on SD card files, making them more susceptible to tampering.

Overview of Secure Boot on Raspberry Pi 4B

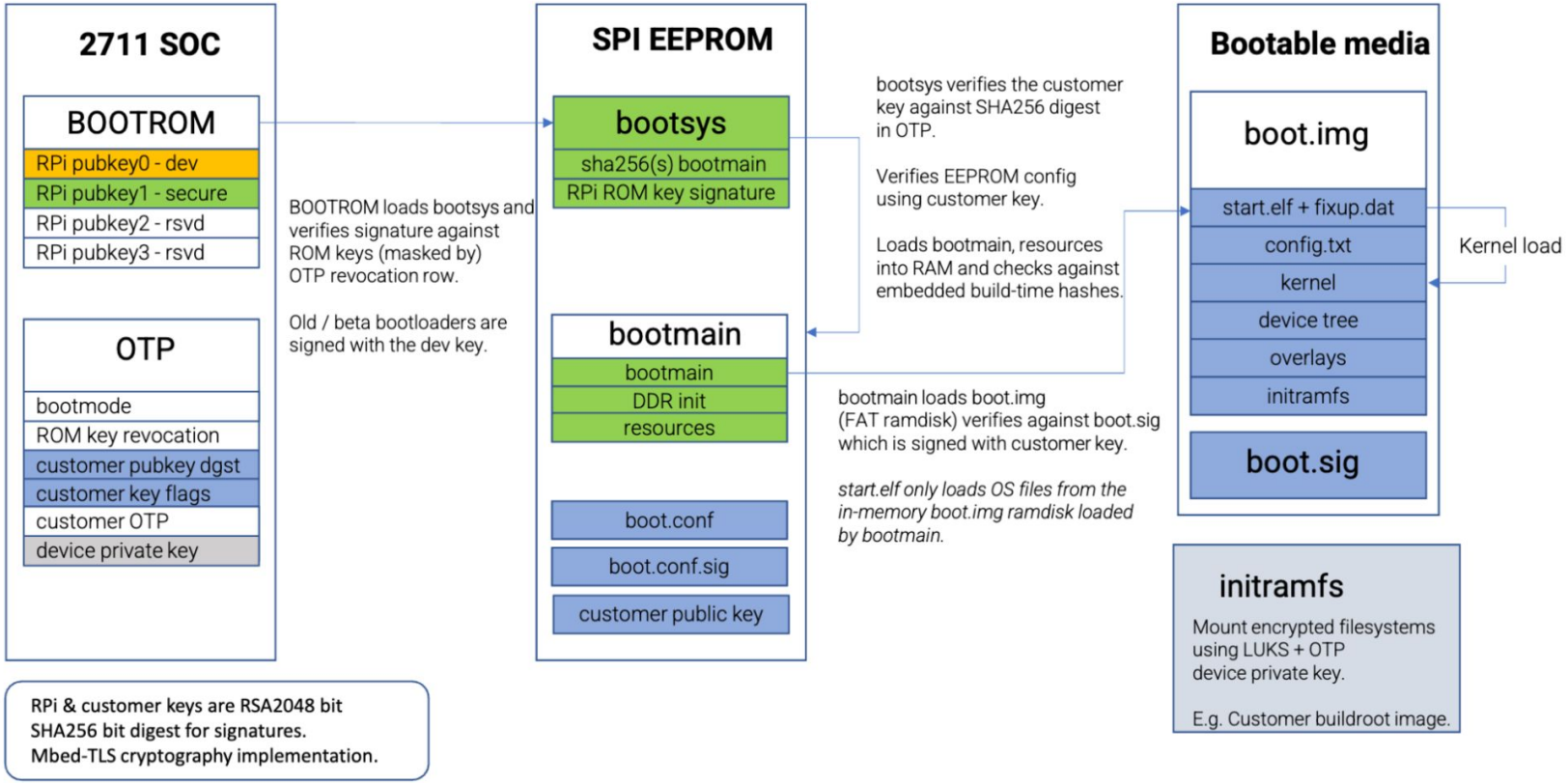
Key components involved:

- boot.img: A single FAT image containing all boot-related files.
- boot.sig: A signature file for verifying the integrity of boot.img.
- EEPROM: Stores the bootloader and secure boot configuration.
- OTP (One-Time Programmable) Memory: Permanently stores the hash of the public key used for verification.

Architectural Support

- Built-In Signature Verification in Bootloader
- One-Time Programmable (OTP) Memory
- Chain of Trust
- Documentation Support

Raspberry Pi4 – secure boot – chain of trust



Secure Boot Implementation

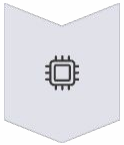
Element of Secure Boot

01	Key Generation and Management
02	Hardware Enforcement
03	Digital Signature Embedding
04	Boot File Verification

Process Outline

Generation of RSA key pair

Create RSA 2048 key pair

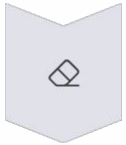


Erase Existing EEPROM

Customize EEPROM & sign bootloader

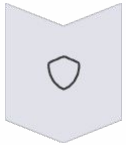
Configuring GPIO

GPIO pin settings



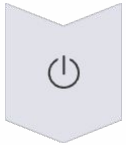
Generating Signed EEPROM

Using rpiboot



Locking Secure Boot Mode

Implement physical protection



Launch

Execute secure boot process

Generation of RSA Key Pair

Generation of RSA key pair
Erase existing EEPROM
Configuring GPIO
Generating Signed EEPROM
Locking Secure Boot Mode
Launch

Generate 2048-bit RSA key pair

Foundation for firmware signing

Generated using secure cryptographic utilities

```
surajnarwariya@Surajs-MacBook-Air ~ % openssl genpkey \
  -algorithm RSA \
  -out private_key.pem \
  -pkeyopt rsa_keygen_bits:2048
```

Secure private key storage

Used for signing bootloader and OS images

Must be securely stored

Extract public key

Embedded in bootloader for verification

Derived from the private key

```
surajnarwariya@Surajs-MacBook-Air ~ % openssl pkey \
  -in private_key.pem \
  -pubout \
  -out public_key.pem
```

Erase Existing EEPROM Configurations

Generation of RSA key pair | Erase existing EEPROM | Configuring GPIO | Generating Signed EEPROM | Locking Secure Boot Mode | Launch

Erase EEPROM Configuration

Objective:

- **Reset** the SPI EEPROM of Raspberry Pi 4B to enable **RPIBOOT** mode for secure boot configuration.

Why EEPROM Erase is Necessary:

- The Raspberry Pi 4B has **One-Time Programmable (OTP)** bits.
- To avoid **accidental permanent locking**, secure-boot-related OTP bits are **disabled by default**.
- Direct secure-boot configuration needs a **manual forced entry** into RPIBOOT mode.
- Erasing the EEPROM ensures the Pi **cannot load any corrupt or outdated firmware** during setup.

Behind the Scenes:

- BootROM fails to find valid firmware in the erased EEPROM.
- As a fallback, Pi enters **USB Device Boot Mode (RPIBOOT)**, allowing a host PC to load recovery firmware into RAM.

Erase Existing EEPROM Configurations

Generation of RSA key pair | Erase existing EEPROM | Configuring GPIO | Generating Signed EEPROM | Locking Secure Boot Mode | Launch

Erase EEPROM Configuration

Implementation Steps:

Use **Raspberry Pi Imager** to flash a "bootloader recovery" image to an SD card

Remove `pieeprom.bin` and `pieeprom.sig` (default bootloader files) to avoid loading anything.

Add a `config.txt` file with:

```
erase_eeprom=1
uart_2ndstage=1
```

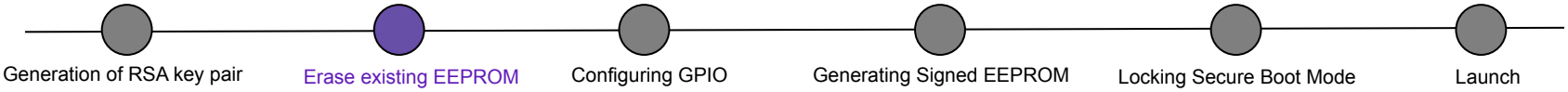
Boot with SD Card:

On boot, Pi reads `config.txt` settings:

`erase_eeprom=1`: Commands the 2nd-stage bootloader to **erase** the SPI EEPROM contents.

`uart_2ndstage=1`: Enables **debug logs** on UART for easier monitoring.

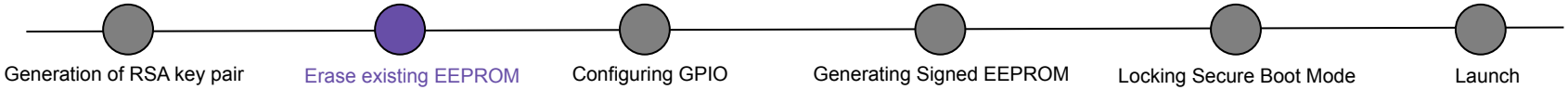
Log: EEPROM Erase & Recovery




```
5.16 Erased SPI EEPROM 617 ms
5.21 BMD "pieeprom.bin" not found
5.23 BMD "pieeprom.upd" not found
5.25 rename recovery.bin to RECOVERY.000
5.48 RESET
```

Note: Indicates the EEPROM erase process and subsequent recovery steps.

Log: EEPROM Programming & Update




```
10.07 Read config.txt bytes 1893 hnd 0x0
10.09 Updating OTP to use GPIO8 to enable USB-device boot (rp
10.62 pieeprom.sig
10.62 hash: e7aed15f69dd3515454f987b59644254...
10.67 ts: 1743600167
10.14 Reading EEPROM: 524288 bytes 0xc0b60000
11.50 635ms
11.96 Writing EEPROM ...
13.43 BOOT-EEPROM: UPDATED
13.43 USB-OTG disconnect
13.45 RECOVERY: COMPLETE: REBOOT 0
```

Note: This log confirms successful EEPROM programming and recovery.

Configuring GPIO & Enabling Signed Boot

Generation of RSA key pair | Erase existing EEPROM | Configuring GPIO | Generating Signed EEPROM | Locking Secure Boot Mode | Launch

Why is it required?

- **Set up GPIO pin** for entering RPIBOOT mode manually.
- **Enable signed boot** in the EEPROM configuration to enforce **RSA-signed boot images**.

Behind the Scenes:

- The file `config.txt` is used during boot to configure the EEPROM.

Adding the line: `program_rpiboot_gpio=8`

Assigns GPIO Pin 8 as the trigger pin for **nRPIBOOT**.

- Pulling this pin **low** during power-up forces the Pi into **USB device boot mode** (RPIBOOT).
- This ensures **manual recovery** is always possible even if the EEPROM is partially misconfigured.

Generating Signed EEPROM Bootloader Image

Generation of RSA key pair | Erase existing EEPROM | Configuring GPIO | Generating Signed EEPROM | Locking Secure Boot Mode | Launch

How?

Purpose of This Step

Generate a customized, cryptographically signed EEPROM image (`pieeprom.bin`) to enable secure boot enforcement on Raspberry Pi 4B.

Internal Mechanics – What Happens Behind the Script

- Combines the latest recovery bootloader with configuration fields (e.g., `SIGNED_BOOT=1`)
- Embeds the SHA-256 hash of your **public RSA key** into the EEPROM config section
- Outputs a signed EEPROM image used for **enforcing RSA-verified OS booting**

Generating Signed EEPROM Bootloader Image

Flashing the Signed EEPROM Using RPIBOOT

Objective of the Step

Transfer the signed EEPROM image (`pieeprom.bin`) to the RPI4's EEPROM using USB boot mode.

Under-the-Hood Operations

- **nRPIBOOT Jumper:** Activates USB boot mode, bypassing standard EEPROM boot
- **RPIBOOT Tool:** Host PC sends the new image via USB to the Pi's bootloader flash
- **EEPROM Reprogramming:** The bootloader is updated with the secure version

Decisions at This Stage

- **Controlled Access:** Manual jumper ensures flashing only occurs when explicitly intended
- **Power Cycle Protocol:** Pi must be powered **off during setup** and **on after `rpiboot` starts**
- **Safe Testing:** At this stage, secure boot remains reversible via RPIBOOT reprogramming

Generating Signed EEPROM Bootloader Image

Understanding EEPROM Write Protection in Secure Boot

What Is EEPROM Write Protection?

A hardware or software safeguard that prevents unintended or malicious modification of the EEPROM's contents.

Why Temporarily Disable WP for Flashing?

- The signed EEPROM image must be written into flash
- Disabling WP allows controlled reprogramming of critical boot code
- Re-enabling WP after flashing protects against post-deployment tampering

Security Benefits of Write Protection

- **Post-Flashing Integrity:** Prevents any overwrite of bootloader by malicious actors

- **Physical Tamper Resistance:** In production, WP is often enforced via solder bridges or jumpers
- **Enhancing Trust Chain:** Combined with `SIGNED_BOOT`, WP ensures robust root-of-trust enforcement

Locking Secure Boot Mode on Raspberry Pi 4B

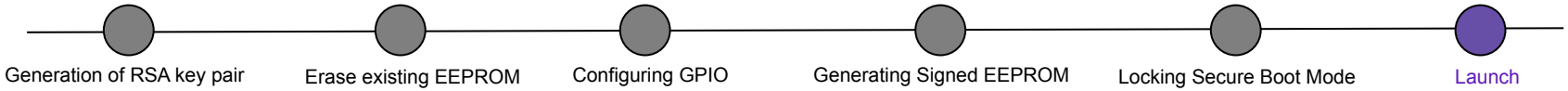
Internal mechanism

Permanently enable **hardware-enforced secure boot** by burning the hash of the customer's public key into the Pi's **One-Time Programmable (OTP)** memory.

What Happens Behind the Scenes:

- When `program_pubkey=1` is set in `config.txt`, the SHA-256 hash of your **public RSA key** is written into OTP.
- This **locks the boot process** to accept only OS images and EEPROM configurations **signed with your private key**.
- When `revoke_devkey=1` is set, it **revokes the default development key**, ensuring the bootloader:
  - Cannot be downgraded to a non-secure version.
  - Rejects unsigned/test-mode boot images.

Log: Secure boot fail



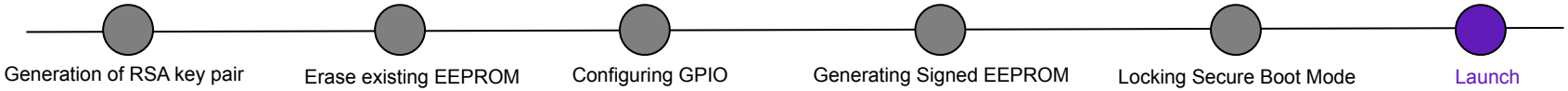

```
 2.24 Trying partition: 1
 2.28 type: 32 lba: 2048 'BSD  4.4' ' SDCARD     ' clusters 1942676 (16)
 2.33 rsc 32 fat-sectors 15178 root dir cluster 2 sectors 0 entries 0
 2.41 FAT32 clusters 1942676
 2.42 secure-boot
 2.43 Loading boot.img ...
 2.47 boot.sig
 2.47 hash: 902be6c484ad71bb9f88b2fce51a2baf01efcb031c9daa0557d3b722c3951959
 2.54 ts: 1743613000
 2.56 rsa2048: 285de79b754b57dbae09418.....4708773979954ad0b0dc8b12e0
 5.18 Verifying
12.95 RSA verify
12.08 rsa-verify fail (0x4380)
12.08 RSA signature not verified
13.37 Bad signature boot.img
13.70 Error 12 loading boot.img
13.70 Boot mode: USB-MSD (04) order f25
13.87 PCI0 init
```

Root Cause Analysis: Why Secure Boot Failed

Failure Trace from Boot Log

- `12.08 rsa-verify fail (0x4380)`: RSA signature verification **failed**.
- `12.08 RSA signature not verified`: Indicates that `boot.img` **was not signed** using the trusted private key.
- `13.07 Error 12 loading boot.img`: Boot process was halted since the signature check didn't pass.

Log: Secure Boot Verified




```
2.24 Trying partition: 1
2.28 type: 32 lba: 2048 'BSD 4.4' ' SDCARD ' ...
2.42 secure-boot
2.43 Loading boot.img ...
2.47 boot.sig
2.47 hash: 902be6c484ad71bb9f88b2fce51a2baf...
2.54 ts: 1743613000
2.56 rsa2048: 588230e1a65cf7446a2613219f6f87f5...
5.18 Verifying
12.05 RSA verify
12.08 rsa-verify pass (0x0) ...
```

Note: These entries show a successful RSA verification and boot image signature validation.

Why Secure Boot Verification Passed

Tracing from Boot Log

- **Matching Key Pair**
  - The boot.img was signed using the correct private key.
  - The corresponding public key hash was already written to OTP during secure boot setup.
- **Integrity of Signature File**
  - boot.sig was present, uncorrupted, and matched the contents of boot.img.
  - RSA-2048 with SHA-256 hash comparison passed during the verification step.
- **Bootloader Performed Proper Checks**
  - The EEPROM bootloader performed signature validation before loading.
- **No Modification After Signing**
  - boot.img was not tampered with after signing — its hash remained consistent with what was expected from boot.sig.

Integrating U-Boot as Third-Stage Bootloader

Summary

In our secure boot implementation, we use **U-Boot** as the **third-stage bootloader** in the boot sequence of Raspberry Pi 4B.

Boot Chain Summary:

1. **BootROM** loads and verifies `boot.img` using the OTP-embedded **public key hash**.
2. If valid, it loads firmware and executes the image.
3. The `boot.img` **contains U-Boot binary** as the main payload.

Why U-Boot

- Industry-standard, open-source, customizable bootloader.
- Offers flexibility in defining the final-stage actions before kernel or application.
- Extends the secure boot chain with its own image verification capabilities.

More on U-Boot

- **Built-in Support for Image Authentication:**
  U-Boot supports cryptographic signature verification (RSA, SHA) via features like `verify_fit` and `CONFIG_SECURE_BOOT`, allowing enforcement of image authenticity before execution.

- **Scriptable and Configurable via Environment Variables:**
  U-Boot allows conditional booting logic, memory address control, and verification commands using scripting (e.g., `bootcmd`, `bootargs`, `env`) — ideal for custom secure workflows.

- **Supports Chain of Trust Extension:**
  U-Boot can independently verify additional components like `monitor.img`, kernel, or device trees using public keys embedded in the image, enabling enforcement beyond the initial bootloader.

Secure Storage of U-Boot inside boot.img

How U-Boot Is Protected:

- `u-boot.bin` is **embedded inside** `boot.img`.
- The Pi validates `boot.img` against `boot.sig` before execution.
- Any modification to U-Boot binary would change the hash of `boot.img`, causing the **signature verification to fail**.

Behind the Scenes:

- The signature (`boot.sig`) authenticates the entire image—including U-Boot.
- If tampered, the bootloader **refuses to execute the image**, enforcing integrity.

Security Implication:

- **Immutable U-Boot** in production—can't be swapped or modified without invalidating the boot image.
- Trust in U-Boot is **transitively rooted** in the customer's private key.

U-Boot's Role in Secure Boot Continuity

After Successful Verification:
Once the Raspberry Pi hands over execution to U-Boot:

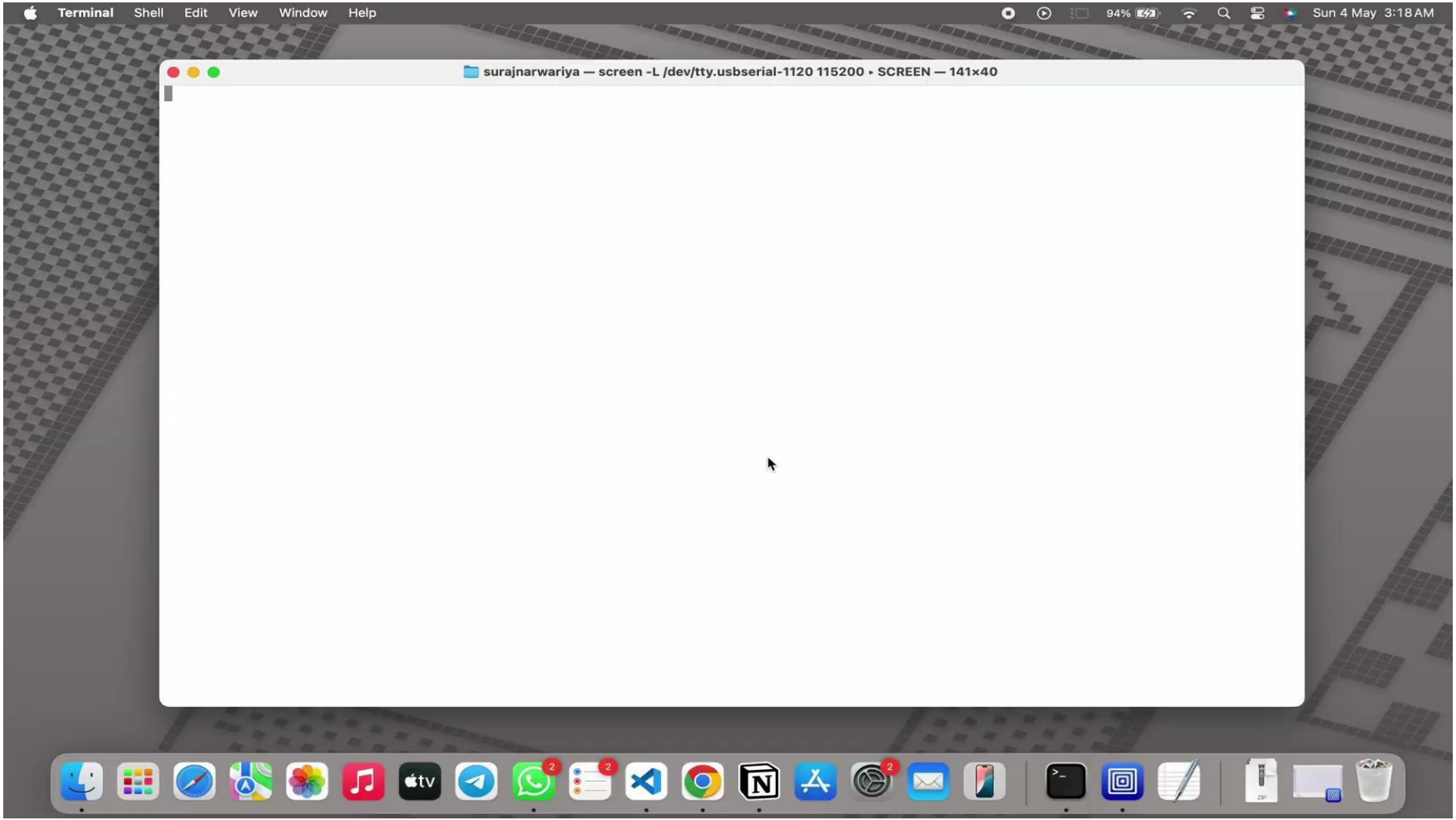
- U-Boot becomes the **enforcer of the next step in the secure chain**.
- It takes over the responsibility of:
    - **Verifying and loading** the final payload (in your case, a monitor program).
    - **Passing control securely** to the next executable.

Why This Matters:

- Secure boot doesn't end with just verifying U-Boot—it must **extend down to the monitor or kernel**.
- U-Boot gives you full control to enforce **custom policies** (e.g., signature check of monitor binary, memory sanitization, recovery fallback)

Log: Verification of Monitor

```
scanning bus xhci_pci for devices... 2 USB Device(s) found
       scanning usb for storage devices... 0 Storage Device(s) found
Hit any key to stop autoboot:  2  1  0
410 bytes read in 11 ms (36.1 KiB/s)
## Executing script at 01000000
>>> Starting my custom U-Boot script <<<
Loading monitor.img
Loading monitor.sig
verifying signature...
verified
Loading monitor.img at 0x20000000
------Monitor------
Hello world!!
Monitor Start..(^_^)
Exiting
```


Terminal Shell Edit View Window Help
94%
Sun 4 May 3:18 AM
surajnarwariya — screen -L /dev/tty.usbserial-1120 115200 ▸ SCREEN — 141×40

References and Resources

- https://github.com/Openwide-Ingenierie/raspberry-pi4-secure-boot
- https://github.com/raspberrypi/usbboot/blob/master/docs/secure-boot-chain-of-trust-2711.pdf
- https://news.ycombinator.com/item?id=35815382
- https://github.com/raspberrypi/usbboot/blob/master/secure-boot-example/README.md
- https://forums.raspberrypi.com/viewtopic.php?t=344770

Thank You