Trusted firmware for A-profile systems

Firmware security

Trainer: Sumit Garg
Linaro Support and Solutions Engineering

Linaro
Developer Services

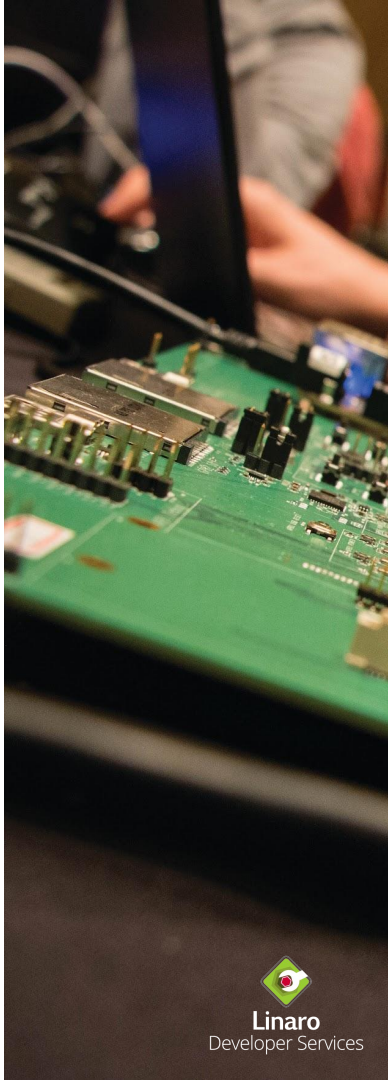
Logistics

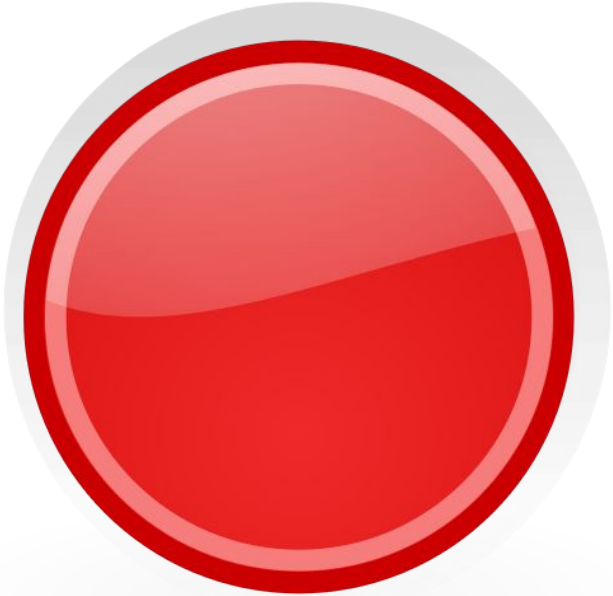
These sessions will be recorded. Turn off your camera if you do not want to appear in the recording.

Questions welcome! There is time allocated for Q&A at the end of today's session but you can ask relevant questions verbally or in the chat as we go.

Please keep your microphone muted when not speaking!

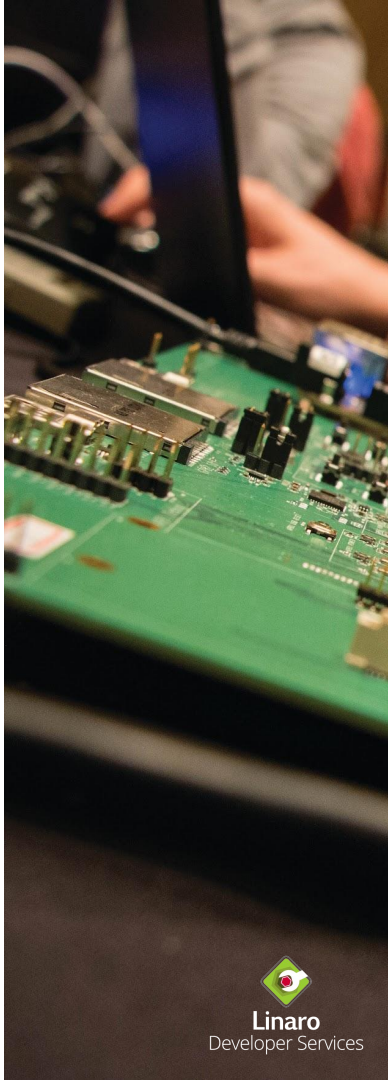
Slides and lab resources can be downloaded from:
https://fileserver.linaro.org/s/fE6iBYca8bYqrrk





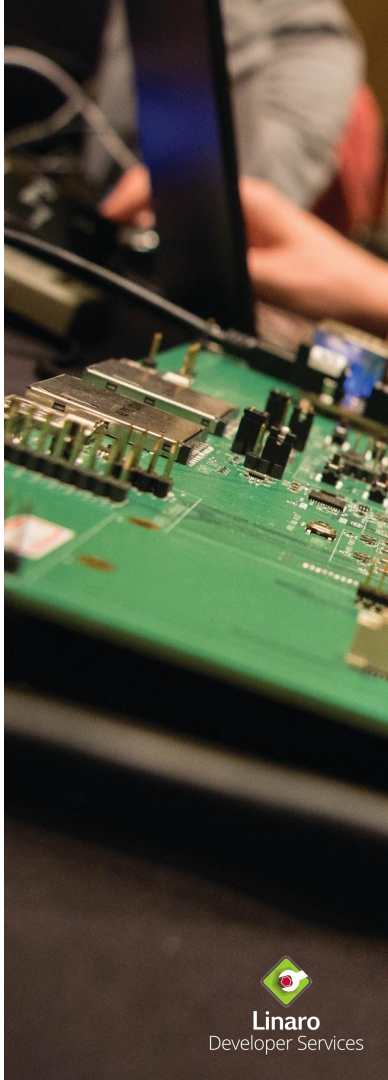
Training modules overview

1. Introduction to TF-A
2. Generic boot
3. **Firmware security**
4. Secure/Realm world interfaces



Firmware security

- **Generic threat model**
- Trusted Board Boot
  - Chain of Trust
  - Authentication Framework
- Measured boot
- Firmware update
- Firmware encryption



Why threat modeling?

- Security is **not** a turn key solution but rather made of many different components
- There is no such thing as “**a secure system**”, only **secure enough**
- Threat modelling is essential and an important part of **Secure Development Lifecycle (SDL)**
  - Knowledge of system **assets**
  - **Threats** to them
  - **Mitigated** via security features
- Firmware security **features** such as authentication, measurement, encryption, updates etc. are **tools** in your toolbox.

TF-A threat model

TF-A codebase is **highly configurable** to allow tailoring it best for each platform's needs. So providing a holistic threat model covering all of its features **is not necessarily** the best approach.

Instead, TF-A provides a **collection of documents** which, together, form the project's threat model. These are articulated around a core document, called the **Generic Threat Model**, which focuses on the **most common** configuration we expect to see. The other documents typically focus on **specific features** not covered in the core document.

TF-A follows STRIDE threat modeling technique to identify a list of threats and corresponding mitigations.

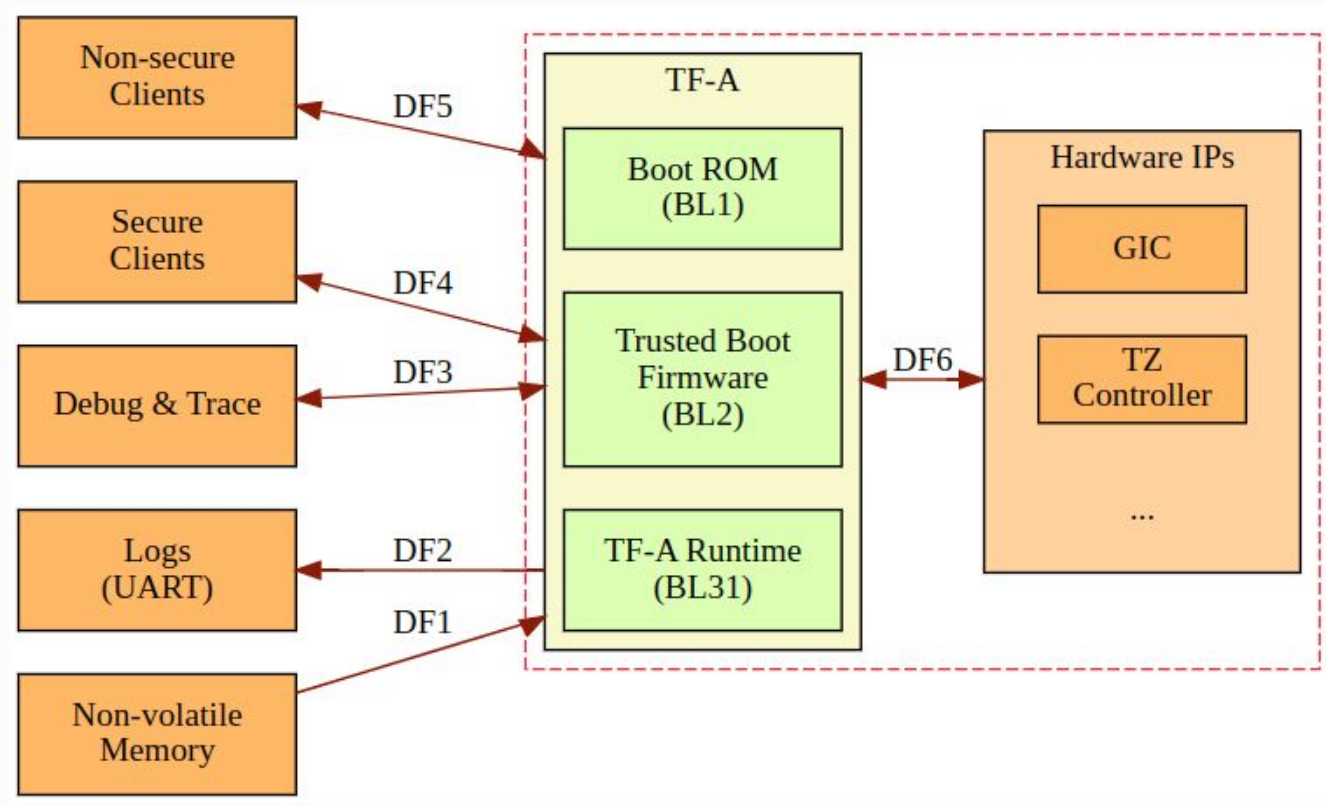
Generic threat model: Target of evaluation

In this threat model, the target of evaluation is the Trusted Firmware for A-class Processors **(TF-A)**. This includes the **boot ROM (BL1)**, the **trusted boot firmware (BL2)** and the **runtime EL3 firmware (BL31)**.

Generic threat model targets TF-A most basic configuration:

- All TF-A images are run from **either ROM or on-chip trusted SRAM**.
  - TF-A is not vulnerable to an attacker that can probe or tamper with off-chip memory.
- **Trusted boot** is enabled.
  - An attacker can't boot arbitrary images that are not approved by platform providers.
- There is **no Secure-EL2**.
- **No experimental** features are enabled.

Generic threat model: Data Flow Diagram



Generic threat model: Data Flow Diagram - II

Diagram Element	Description
DF1	At boot time, images are loaded from **non-volatile memory** and verified by TF-A boot firmware. These images include TF-A BL2 and BL31 images, as well as other secure and non-secure images.
DF2	TF-A log system framework outputs debug messages over a **UART interface**.
DF3	**Debug and trace IP** on a platform can allow access to registers and memory of TF-A.
DF4	**Secure world** software (e.g. trusted OS) interact with TF-A through **SMC call interface** and/or shared memory.
DF5	**Non-secure world** software (e.g. rich OS) interact with TF-A through **SMC call interface** and/or shared memory.
DF6	This path represents the interaction between TF-A and various hardware IPs such as **TrustZone controller and GIC**. At boot time TF-A configures/initializes the IPs and interacts with them at runtime through interrupts and registers.

Generic threat model: Assets

Asset	Description
Sensitive Data	These include sensitive data that an attacker must not be able to tamper with (e.g. the **Root of Trust Public Key**) or see (e.g. secure logs, debugging information such as crash reports).
Code Execution	This represents the requirement that the platform should **run only TF-A code approved by the platform provider**.
Availability	This represents the requirement that TF-A services should **always be available** for use.

Generic threat model: Threat agents

Threat agent	Description
NSCode	Malicious or faulty code running in the **non-secure world**, including NS-EL0 NS-EL1 and NS-EL2 levels
SecCode	Malicious or faulty code running in the **secure world**, including S-EL0 and S-EL1 levels
AppDebug	Physical attacker using **debug signals** to access TF-A resources
PhysicalAccess	Physical attacker having access to external **device communication** bus and to external **flash communication** bus using common hardware

Generic threat model: Threat assessment

ID	Threat	Mitigations	Implemented?
1	An attacker can **mangle** firmware images to execute arbitrary code	1) Implement the **Trusted Board Boot (TBB)** feature which prevents malicious firmware from running on the platform by authenticating all firmware images. 2) Perform **extra checks on unauthenticated data**, such as FIP metadata, prior to use.	1) **Yes** with **TRUSTED_BOARD_BOOT** build option is set to 1. 2) **Yes**.
2	An attacker may attempt to boot **outdated, potentially vulnerable** firmware image	Implement **anti-rollback protection** using non-volatile counters (NV counters) as required by TBBR-Client specification.	Yes / **Platform specific**.
3	An attacker can use Time-of-Check-Time-of-Use (TOCTOU) attack to bypass image authentication during the boot process	Copy image to **on-chip memory** before authenticating it.	**Platform specific.**

Generic threat model: Threat assessment - II

4	An attacker with physical access can execute arbitrary image by **bypassing** the signature verification stage **using glitching techniques**.	Mechanisms to **detect** clock glitch and power variations.	**No**.
5	Information **leak** via UART logs	Remove **sensitive information** logging in production releases. **Do not conditionally** log information depending on potentially sensitive data. Do not log **high precision timing** information.	Yes / Platform Specific. Requires the **right build options** to be used.
6	An attacker can read sensitive data and execute arbitrary code through the **external debug and trace** interface	**Disable** the debug and trace capability for production releases or enable proper **debug authentication**.	Platform specific.
7	An attacker can perform a **denial-of-service attack** by using a broken SMC call that causes the system to reboot or enter into unknown state.	**Validate SMC function** ids and arguments before using them.	Yes / Platform specific.

Generic threat model: Threat assessment - III

8	Memory corruption due to **memory overflows and lack of boundary checking** when accessing resources could allow an attacker to execute arbitrary code, modify some state variable to change the normal flow of the program, or leak sensitive information	1) Use proper input validation. 2) Code reviews, testing.	1) Yes. Data received from normal world are **sanitized** before being used. 2) Yes. TF-A uses a combination of **manual code reviews** and automated program analysis and testing.
9	**Improperly handled SMC calls** can leak register contents	**Save and restore registers** when switching contexts.	Yes.
10	SMC calls can leak sensitive information from TF-A memory via **microarchitectural side channels** such as **Spectre**	Enable appropriate **side-channel protections**.	Yes / Platform specific.
11	**Misconfiguration** of the Memory Management Unit (**MMU**) may allow a normal world software to access sensitive data, execute arbitrary code or access otherwise restricted HW interface	When configuring access permissions, the **principle of least privilege** ought to be enforced.	Platform specific.

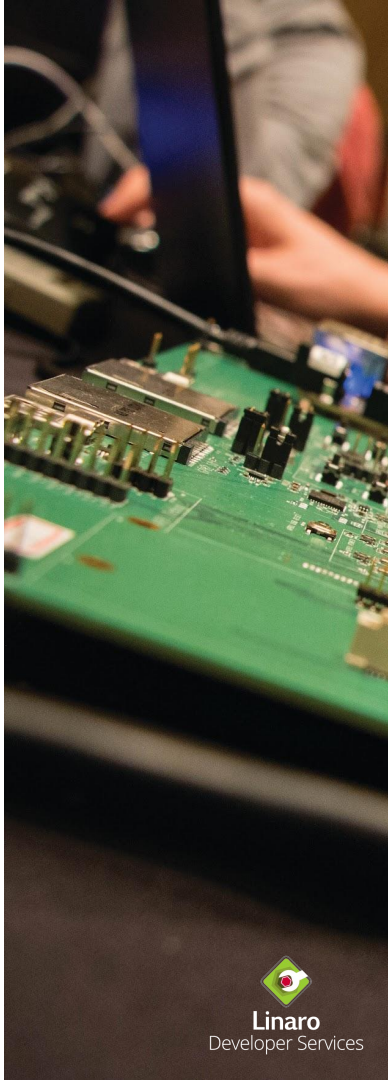
Generic threat model: Threat assessment - IIII

12	Incorrect configuration of **Performance Monitor Unit (PMU)** counters can allow an attacker to mount side-channel attacks using information exposed by the counters	Follow mitigation strategies as described in Secure Development Guidelines.	Yes / platform specific.
13	Leaving **sensitive information** in the memory, can allow an attacker to retrieve them.	**Clear the sensitive data** from internal buffers as soon as they are not needed anymore.	Yes / Platform specific
14	Attacker wants to execute an arbitrary or untrusted binary as the secure OS when the option **OPTEE_ALLOW_SMC_LOAD** is enabled.	When enabling the option OPTEE_ALLOW_SMC_LOAD, the **non-secure OS must be considered a closed platform** up until the point the SMC can be invoked to load OP-TEE.	**None** in TF-A itself. It has to be implemented in non-secured OS eg. ChromeOS.

For details, refer to: https://trustedfirmware-a.readthedocs.io/en/latest/threat_model/threat_model.html

Firmware security

- Generic threat model
- **Trusted Board Boot**
  - **Chain of Trust**
  - **Authentication Framework**
- Measured boot
- Firmware update
- Firmware encryption



Trusted Board Boot (TBB)

The Trusted Board Boot (TBB) feature **prevents malicious firmware** from running on the platform by authenticating all firmware images up to and including the normal world bootloader. It does this by **establishing a Chain of Trust** using Public-Key-Cryptography Standards (**PKCS**).

TF-A TBB implementation meets Trusted Board Boot Requirements (TBBR) specification.

The Chain of Trust established by TBB is **only upto normal world bootloader** like u-boot, edk2 etc. It has to be extended upto non-secure OS like Linux via bootloader features like **UEFI Secure boot, U-boot FIT signatures, Android verified boot** etc.

Chain of Trust (CoT)

A Chain of Trust (CoT) starts with a set of implicitly trusted components known as **Root of Trust (RoT)** anchor. From TF-A perspective these are:

- A SHA-256 hash of the **Root of Trust Public Key (ROTPK)**. It is stored in the trusted root-key storage registers.
- The **BL1 or BootROM** image, on the assumption that it resides in ROM so cannot be tampered with.

The remaining components in the CoT are **either certificates or boot loader images**. The certificates follow the X.509 v3 standard. In the TBB CoT all certificates are **self-signed**. There is **no need** for a Certificate Authority (CA) because the CoT is **not established** by verifying the validity of a certificate's issuer but **by the content of the certificate extensions**.

Keys and certificates

The certificates are categorised as **“Key”** and **“Content”** certificates.

- **Key certificates** are used to verify public keys which have been used to sign content certificates.
- **Content certificates** are used to store the hash of a boot loader image. An image can be authenticated by calculating its hash and matching it with the hash extracted from the content certificate.

The **public keys and hashes** are included as **non-standard** extension fields in the X.509 v3 certificates.

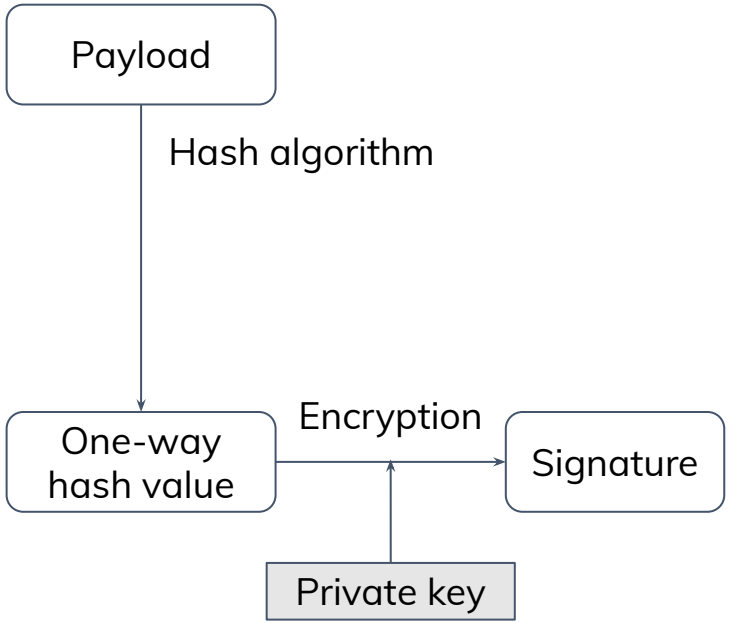
Keys management

The keys used to establish the CoT are:

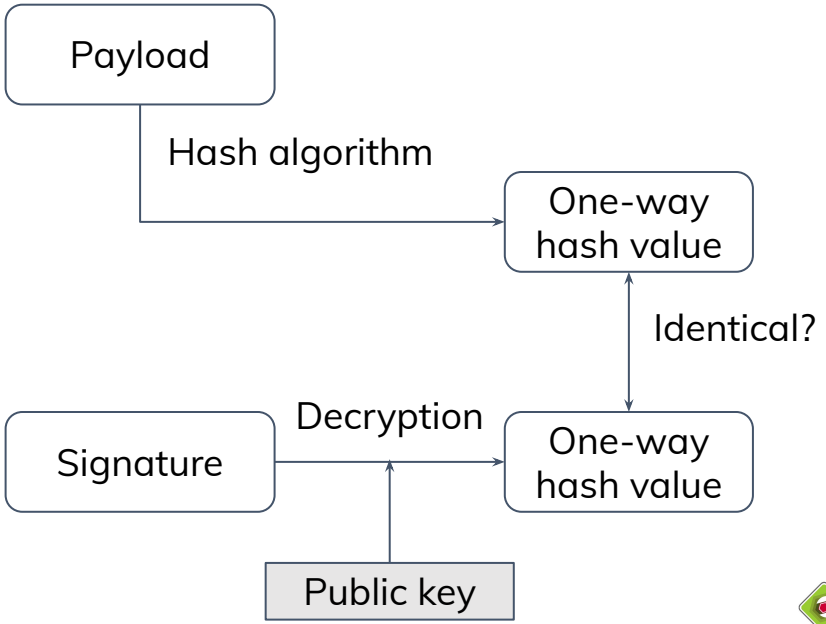
- **Root of trust key:** The private part of this key is used to sign the **BL2 content** certificate and the **trusted key** certificate. The public part is the **ROTPK**.
- **Trusted world key:** The private part is used to sign the **key certificates** corresponding to the **secure world** images (SCP_BL2, BL31 and BL32). The public part is stored in one of the extension fields in the trusted world certificate.
- **Non-trusted world key:** The private part is used to sign the **key certificate** corresponding to the **non secure world** image (BL33). The public part is stored in one of the extension fields in the trusted world certificate.
- **BL3X keys:** For each of SCP_BL2, BL31, BL32 and BL33, the private part is used to sign the **content certificate** for the BL3X image. The public part is stored in one of the extension fields in the corresponding key certificate.

Certificates and authentication

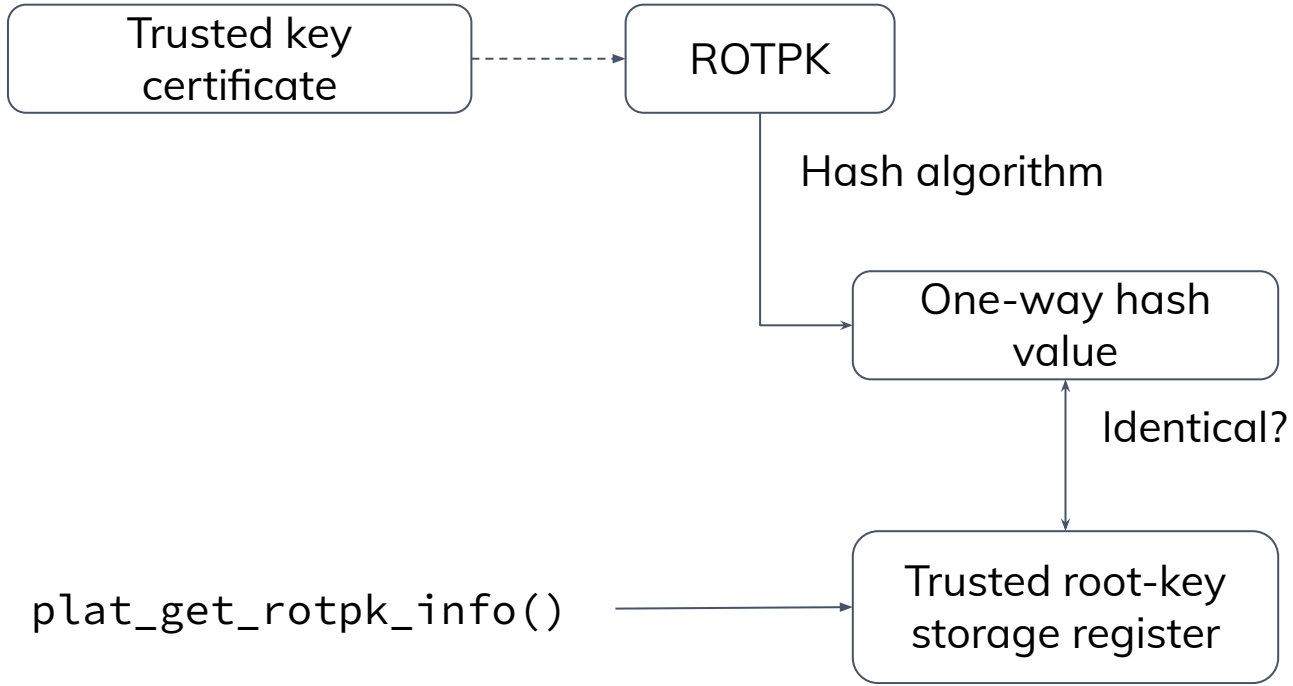
Certificate (build phase)



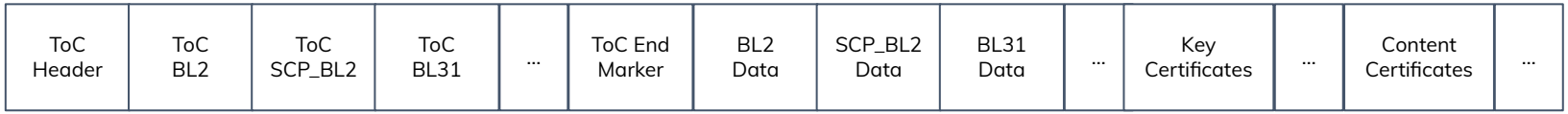
Authentication (boot phase)



ROTPK authentication

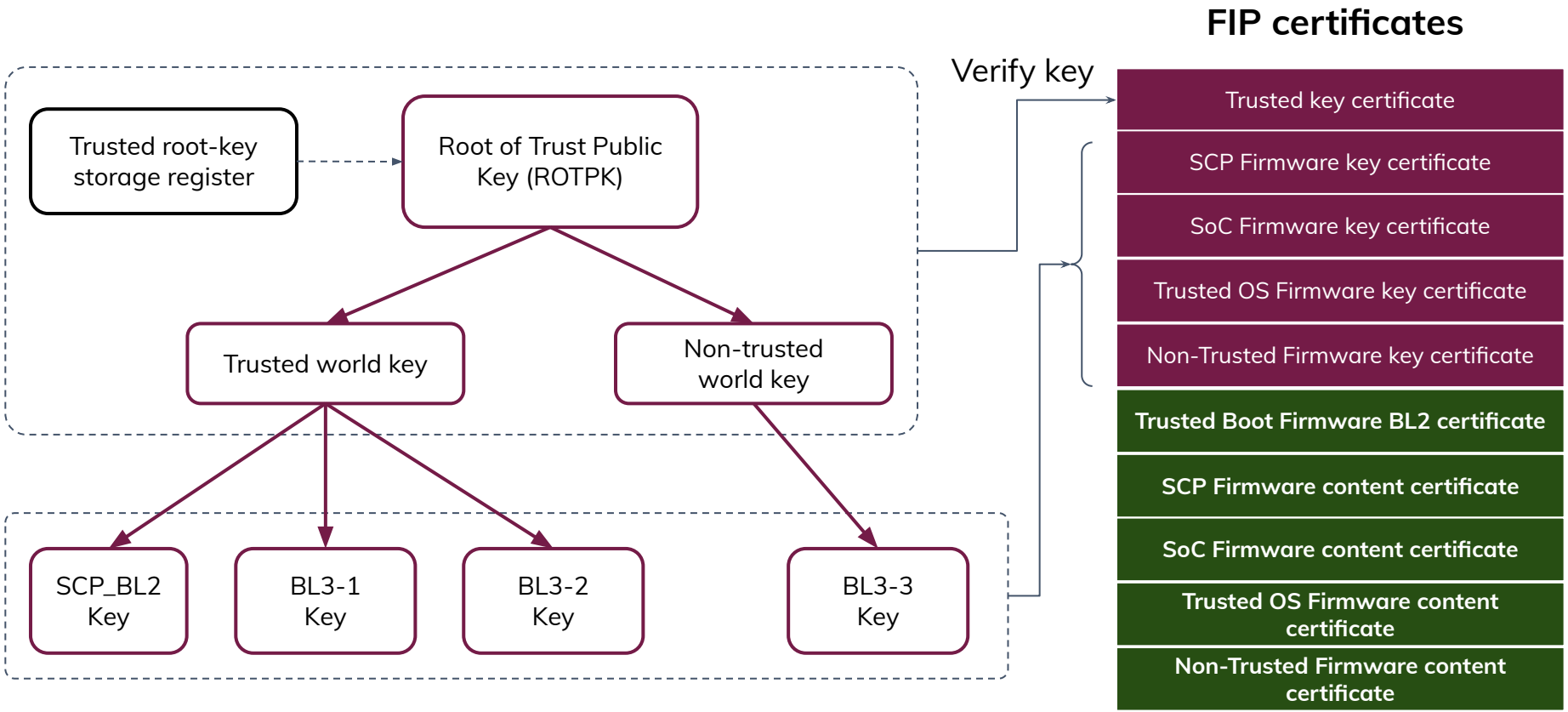


FIP with certificates bundled

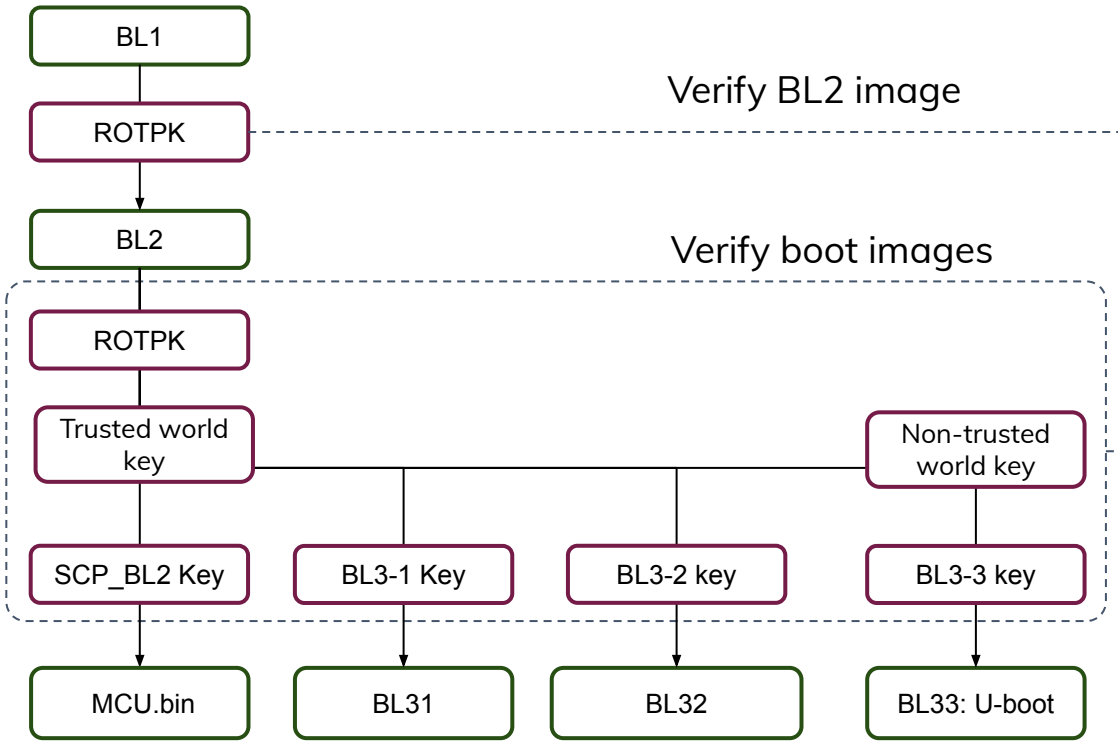



```
$ tools/fiptool/fiptool info .../fip.bin
Trusted Boot Firmware BL2: offset=0x400, size=0x13611, cmdline="--tb-fw"
SCP Firmware SCP_BL2: offset=0x13C00, size=0x35088, cmdline="--scp-fw"
EL3 Runtime Firmware BL31: offset=0x48E00, size=0xC021, cmdline="--soc-fw"
Secure Payload BL32 (Trusted OS): offset=0x55000, size=0x1C, cmdline="--tos-fw"
Secure Payload BL32 Extra1 (Trusted OS Extra1): offset=0x55200, size=0x97758, cmdline="--tos-fw-extra1"
Secure Payload BL32 Extra2 (Trusted OS Extra2): offset=0xECA00, size=0x0, cmdline="--tos-fw-extra2"
Non-Trusted Firmware BL33: offset=0xECA00, size=0xF0000, cmdline="--nt-fw"
Trusted key certificate: offset=0x1DCA00, size=0x60E, cmdline="--trusted-key-cert"
SCP Firmware key certificate: offset=0x1DD200, size=0x4DA, cmdline="--scp-fw-key-cert"
SoC Firmware key certificate: offset=0x1DD800, size=0x4DA, cmdline="--soc-fw-key-cert"
Trusted OS Firmware key certificate: offset=0x1DDE00, size=0x4E8, cmdline="--tos-fw-key-cert"
Non-Trusted Firmware key certificate: offset=0x1DE400, size=0x4EA, cmdline="--nt-fw-key-cert"
Trusted Boot Firmware BL2 certificate: offset=0x1DEA00, size=0x4B6, cmdline="--tb-fw-cert"
SCP Firmware content certificate: offset=0x1DF000, size=0x3E9, cmdline="--scp-fw-cert"
SoC Firmware content certificate: offset=0x1DF400, size=0x430, cmdline="--soc-fw-cert"
Trusted OS Firmware content certificate: offset=0x1DFA00, size=0x4CE, cmdline="--tos-fw-cert"
Non-Trusted Firmware content certificate: offset=0x1E0000, size=0x440, cmdline="--nt-fw-cert"
```

Key management chain



Chain of Trust



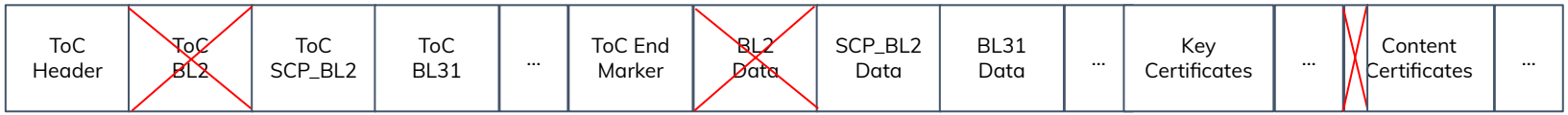
FIP certificates

- Trusted key certificate
- SCP Firmware key certificate
- SoC Firmware key certificate
- Trusted OS Firmware key certificate
- Non-Trusted Firmware key certificate
- **Trusted Boot Firmware BL2 certificate**
- **SCP Firmware content certificate**
- **SoC Firmware content certificate**
- **Trusted OS Firmware content certificate**
- **Non-Trusted Firmware content certificate**

Enabling trusted board boot

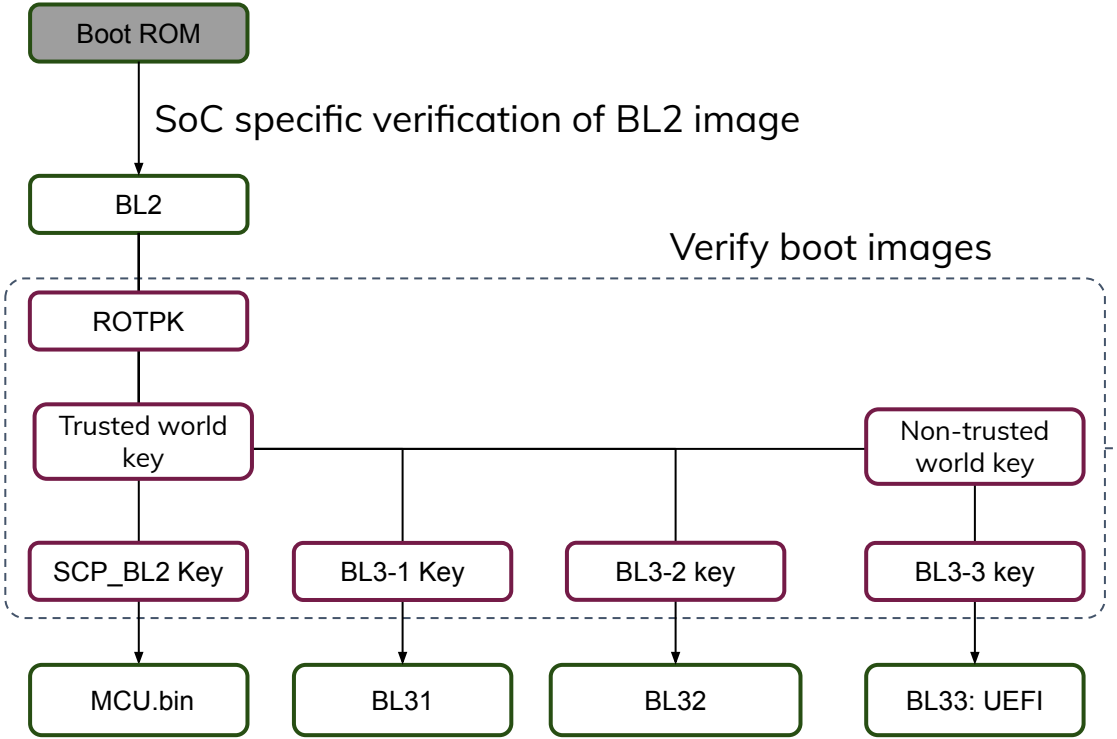
```
$ git clone https://github.com/ARMmbed/mbedtls.git
$ cd mbedtls
$ git checkout mbedtls-3.4.0

$ cd trusted-firmware-a
$ make PLAT=hikey960 SPD=opteed \
    SCP_BL2=.../lpm3.img \
    BL32=.../tee-header_v2.bin \
    BL32_EXTRA1=.../tee-pager_v2.bin \
    BL32_EXTRA2=.../tee-pageable_v2.bin \
    BL33=.../BL33_AP_UEFI.fd \
    TRUSTED_BOARD_BOOT=1 GENERATE_COT=1 SAVE_KEYS=1 \
    MBEDTLS_DIR=/path/to/mbedtls V=1 all fip
```

FIP with certificates bundled: RESET_TO_BL2




```
$ tools/fiptool/fiptool info .../fip.bin
Trusted Boot Firmware BL2: offset=0x400, size=0x13C11, cmdline="--tb-fw"
SCP Firmware SCP_BL2: offset=0x13C00, size=0x35088, cmdline="--scp-fw"
EL3 Runtime Firmware BL31: offset=0x48E00, size=0xC021, cmdline="--soc-fw"
Secure Payload BL32 (Trusted OS): offset=0x55000, size=0x1C, cmdline="--tos-fw"
Secure Payload BL32 Extra1 (Trusted OS Extra1): offset=0x55200, size=0x97758, cmdline="--tos-fw-extra1"
Secure Payload BL32 Extra2 (Trusted OS Extra2): offset=0xECA00, size=0x0, cmdline="--tos-fw-extra2"
Non-Trusted Firmware BL33: offset=0xECA00, size=0xF0000, cmdline="--nt-fw"
Trusted key certificate: offset=0x1DCA00, size=0x60E, cmdline="--trusted-key-cert"
SCP Firmware key certificate: offset=0x1DD200, size=0x4DA, cmdline="--scp-fw-key-cert"
SoC Firmware key certificate: offset=0x1DD800, size=0x4DA, cmdline="--soc-fw-key-cert"
Trusted OS Firmware key certificate: offset=0x1DDE00, size=0x4E8, cmdline="--tos-fw-key-cert"
Non-Trusted Firmware key certificate: offset=0x1DE400, size=0x4EA, cmdline="--nt-fw-key-cert"
Trusted Boot Firmware BL2 certificate: offset=0x1DEA00, size=0x4B6, cmdline="--tb-fw-cert"
SCP Firmware content certificate: offset=0x1DF000, size=0x3E9, cmdline="--scp-fw-cert"
SoC Firmware content certificate: offset=0x1DF400, size=0x430, cmdline="--soc-fw-cert"
Trusted OS Firmware content certificate: offset=0x1DFA00, size=0x4CE, cmdline="--tos-fw-cert"
Non-Trusted Firmware content certificate: offset=0x1E0000, size=0x440, cmdline="--nt-fw-cert"
```

Chain of trust with RESET_TO_BL2

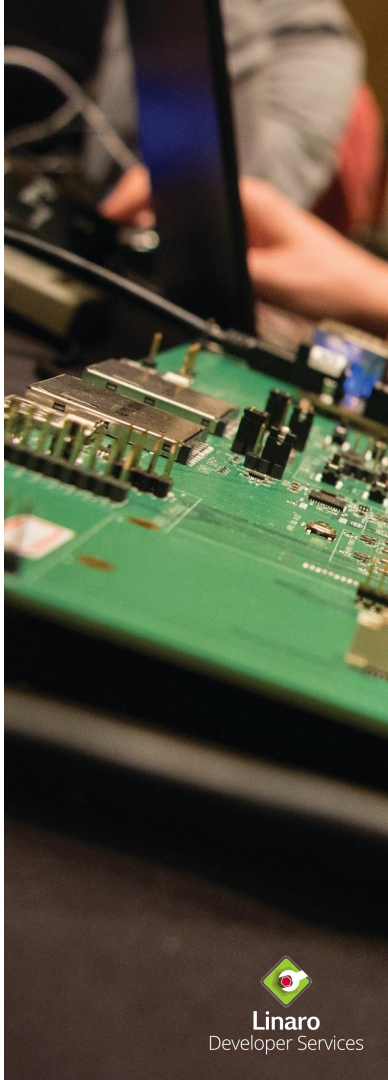


FIP certificates

- Trusted key certificate
- SCP Firmware key certificate
- SoC Firmware key certificate
- Trusted OS Firmware key certificate
- Non-Trusted Firmware key certificate
- **SCP Firmware content certificate**
- **SoC Firmware content certificate**
- **Trusted OS Firmware content certificate**
- **Non-Trusted Firmware content certificate**

Firmware security

- Generic threat model
- Trusted Board Boot
  - Chain of Trust
  - Authentication Framework
- **Measured boot**
- Firmware update
- Firmware encryption



Measured boot

Measured Boot is a boot flow that **computes and securely records** hashes of code and other critical data at each stage in the boot chain. A **Trusted Platform Module (TPM)** is (typically) used to hold the measurements.

Usage:

- Allows a remote entity to use these securely stored measurements (records) for **remote device attestation**.
- Allows a device to **enforce security policies** on the basis of these measurements.

Measured boot: TPM

A TPM is a module that can **securely store artifacts** used to authenticate a computing platform. Typically a TPM is implemented as a **discrete silicon**, however, it can also be implemented in **firmware** (e.g. in TrustZone) aka fTPM.

Some of the **secure services** offered by TPM:

- Provides **attestation, crypto services and Key Management**.
- Securely store measurements (e.g. hashes) of all code and data loaded. The information is recorded (extended) into the **Platform Configuration Registers (PCRs)**.

TF-A measured boot

Ideally **every boot stage** should have a TPM driver in order to extend measurements into TPM PCRs. However, in case of TF-A with:

- **fTPM in OP-TEE**: It won't be up and running at BL1 or BL2 stage.
- **Discrete TPM**: Currently TF-A doesn't have corresponding TPM driver.

TF-A rather does the measurements and **store them in secure memory**. It then passes on measurements and corresponding event log to **later boot stages** in order to extend them into TPM PCRs.

- **BL1** measures BL2.
- **BL2** measures the rest of the images and data.
- All the measurements are recorded on the event log in **Secure Memory**.
- Measurements are passed via devicetree node, compatible property: "**arm,tpm_event_log**".
- Enabled via build flag: **MEASURED_BOOT=1**.

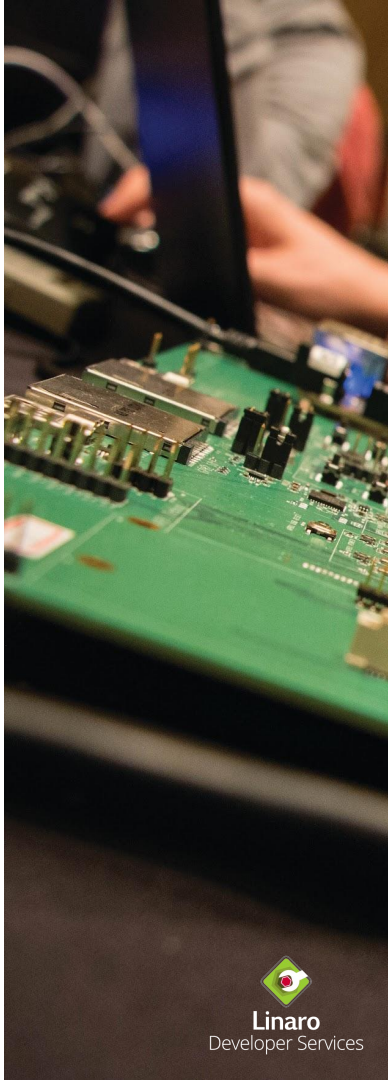
Short break

Based on feedback from previous courses...

... we added this to the middle...

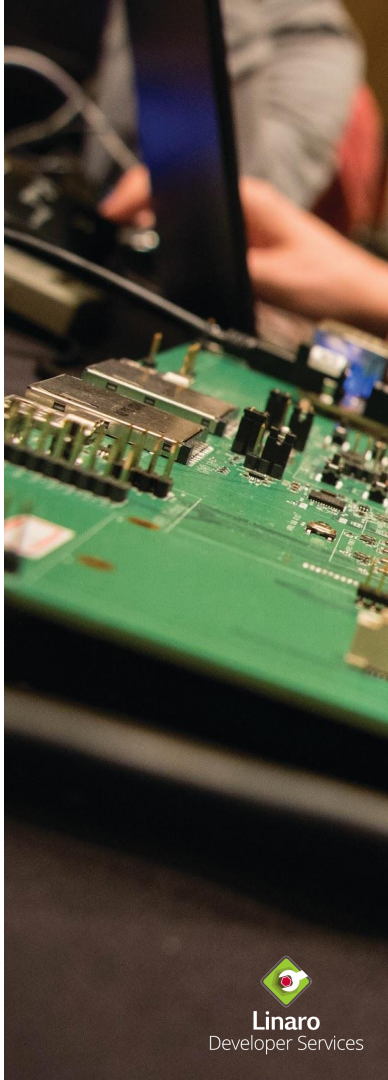
... because it is important ...

... and after two hours it is hard to recollect bits

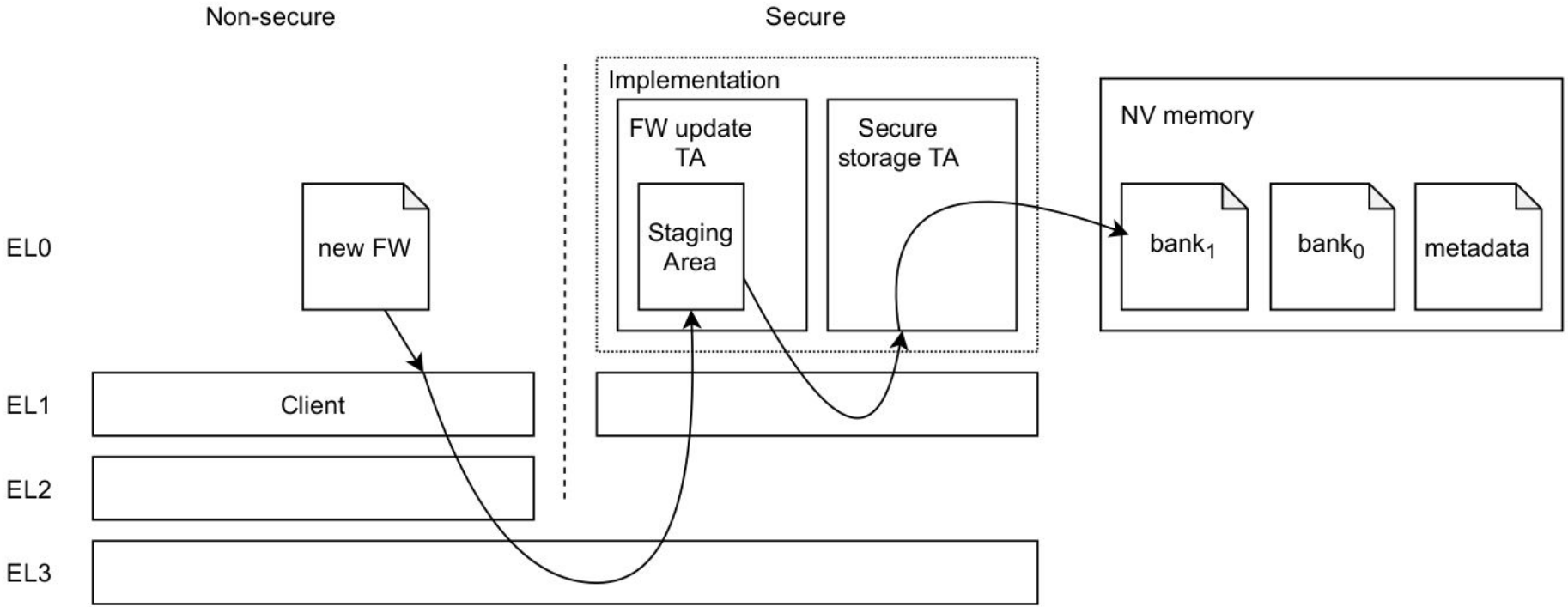


Firmware security

- Generic threat model
- Trusted Board Boot
  - Chain of Trust
  - Authentication Framework
- Measured boot
- **PSA Firmware Update**
- Firmware encryption



PSA Firmware Update (FWU)



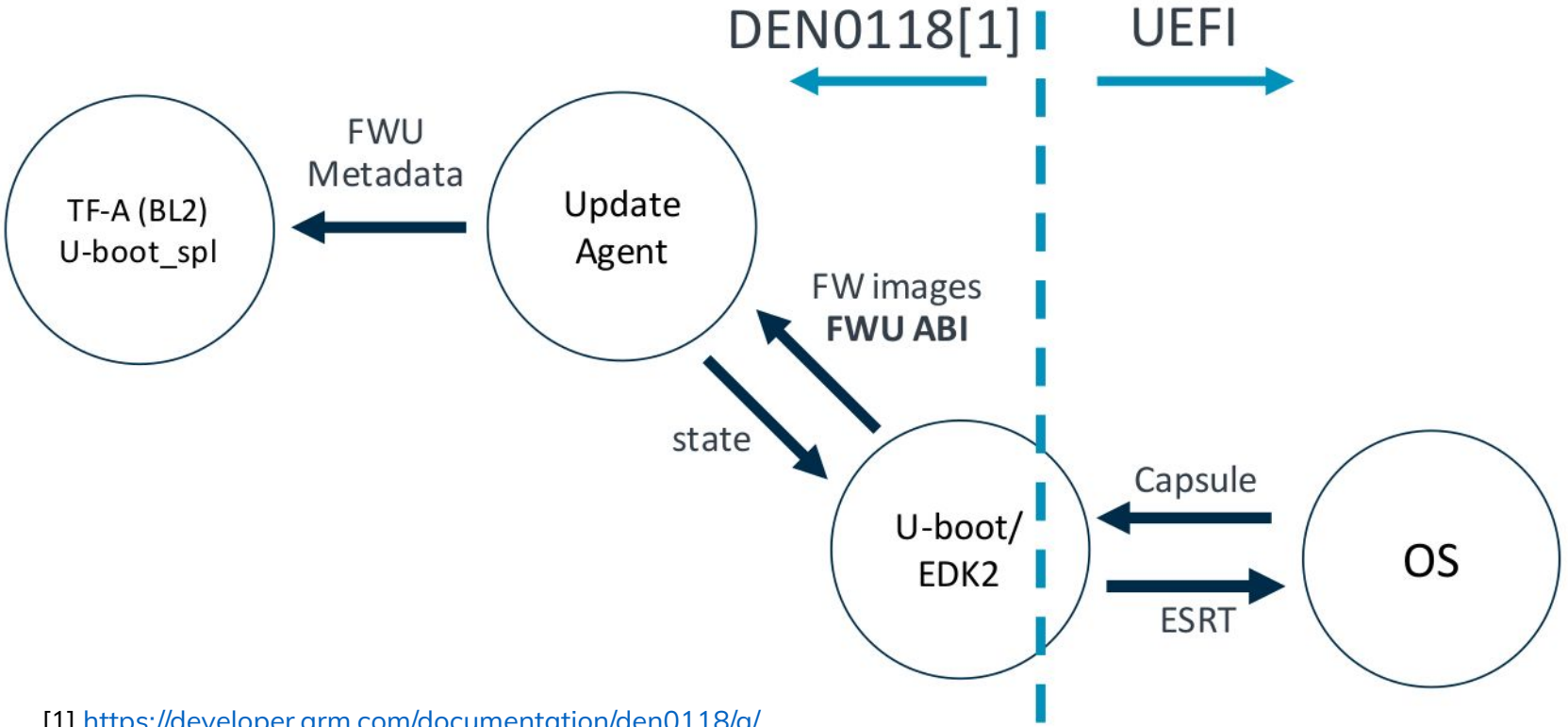
PSA FWU terminologies

A system implementing PSA FWU must contain the following entities:

- **FW update client (Client)**
  - **Originator** of the FW images to be updated.
- **FW update agent (Update Agent)**
  - Execution context **isolated** from the Client. It receives the FW images transmitted from the Client and is responsible for **serializing those to a NV storage**.
  - **Optionally**, the Update Agent can perform **FW image authentication** before updating the NV storage.

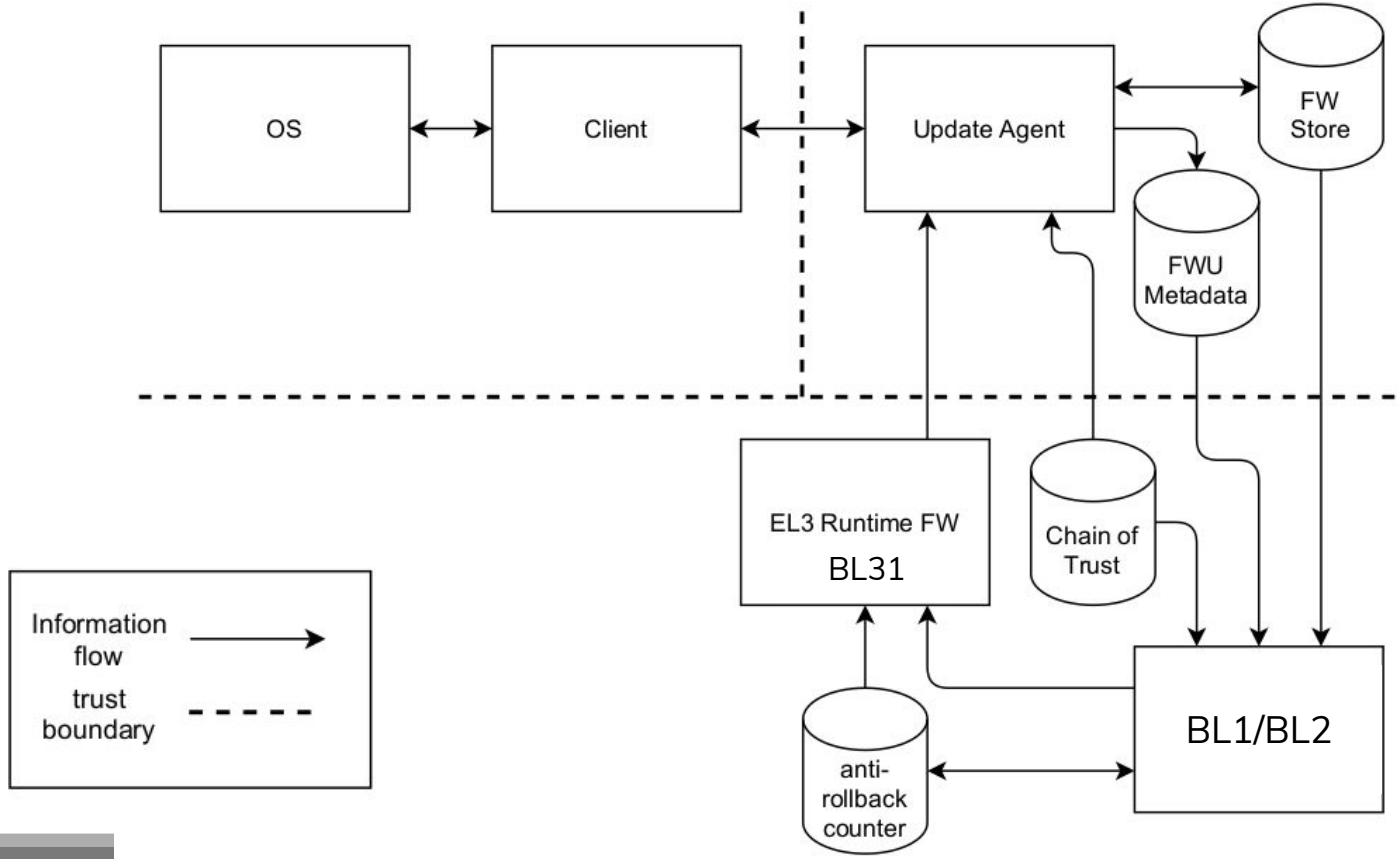
PSA FWU allows support for **multiple** firmware banks. The common approach is to have **two FW image banks** (bank0 and bank1). At any point in time, there exists an **active** and an **update** bank.

PSA FWU: UEFI OS interface

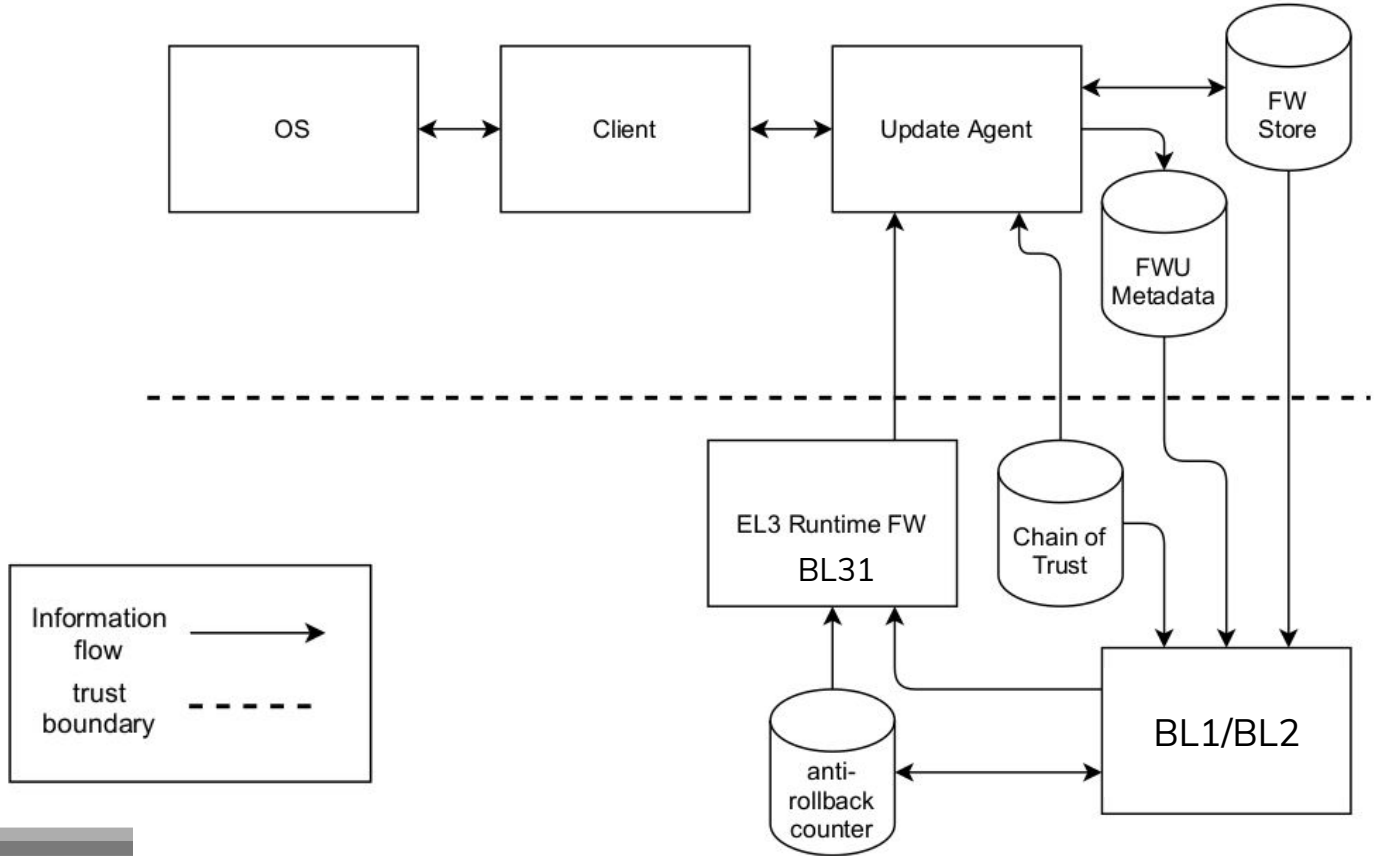


[1] https://developer.arm.com/documentation/den0118/a/

PSA FWU: Update agent in Secure world



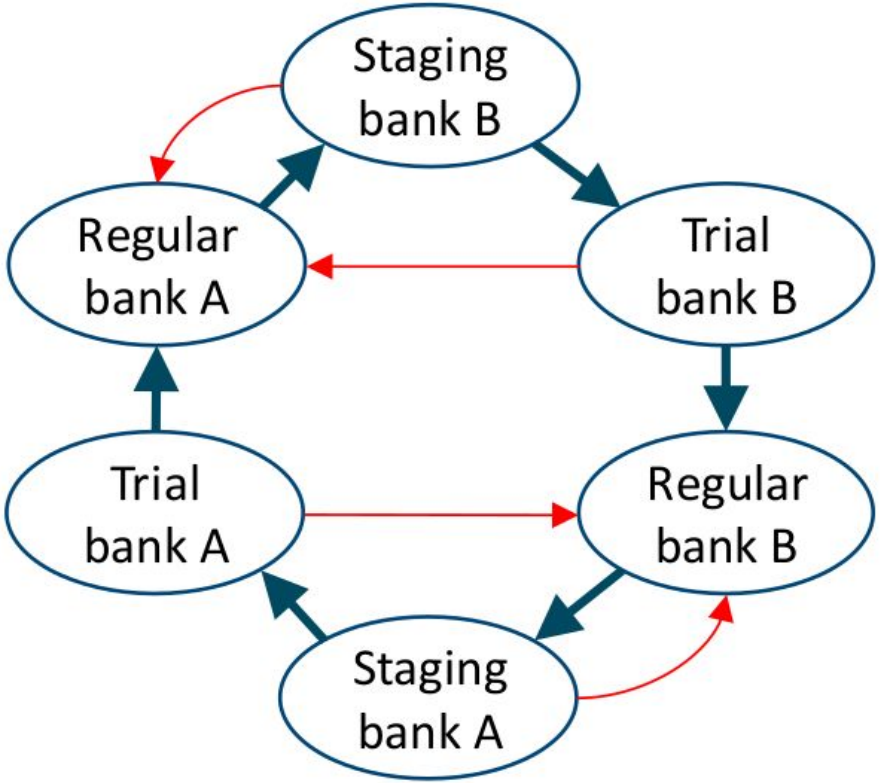
PSA FWU: Update agent in Non-secure world



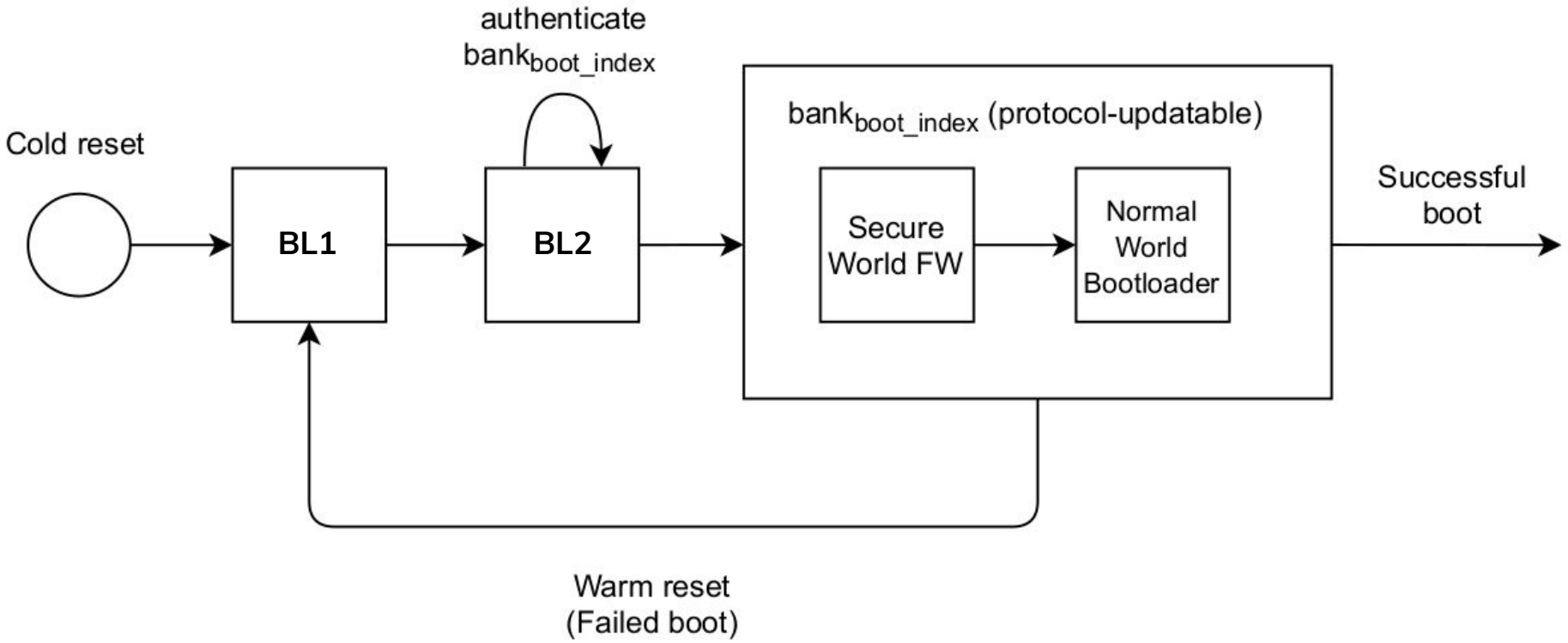
PSA FWU: ABI and FSM

PSA FWU ABI:

- fwu_discover
- fwu_begin_staging
- fwu_end_staging
- fwu_cancel_staging
- fwu_open
- fwu_write_stream
- fwu_read_stream
- fwu_close
- fwu_accept_image
- fwu_set_active



PSA FWU: Platform boot



PSA FWU: Role of BL2

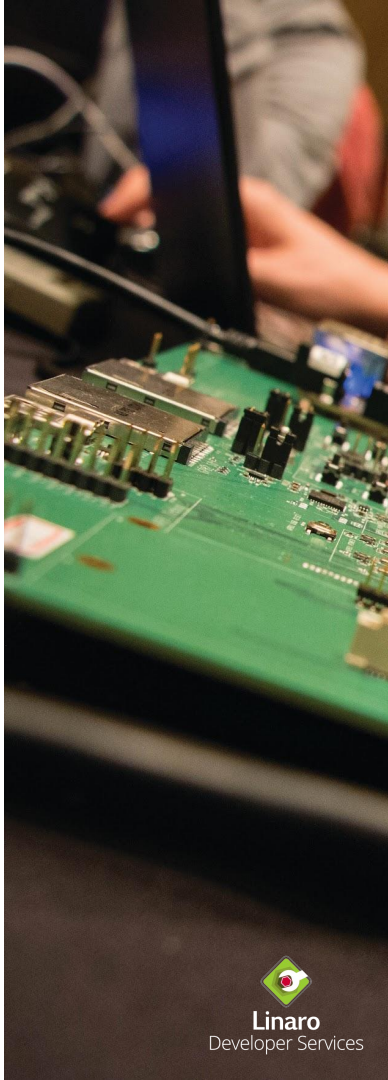
Once firmwares are updated in the **update bank** of the non-volatile storage, then **update agent** marks the update bank as the active bank, and write updated FWU metadata in non-volatile storage. On subsequent reboot, **the second stage bootloader (BL2)** performs the following actions:

- **Read FWU metadata** in memory.
- Retrieve the **image specification** (offset and length) of updated images present in non-volatile storage with the help of FWU metadata.
- Uses the **active bank** of non-volatile storage to boot the images in **trial state**.

Once images pass the **chain of trust** and if the system **booted successfully** then update agent marks the update as **accepted**. The second stage Bootloader (BL2) avoids upgrading the **platform NV-counter** until it's been confirmed that given update is accepted.

Firmware security

- Generic threat model
- Trusted Board Boot
  - Chain of Trust
  - Authentication Framework
- Measured boot
- Firmware update
- **Firmware encryption**



Firmware encryption

FIP: Firmware Image Package

Firmware encryption allows us to achieve **confidentiality** and in turn **integrity** for a firmware image bundled as part of FIP, using:

- **Symmetric** encryption
  - Reason to not use **asymmetric** encryption: boot time limitation.
- **Authenticated** encryption (eg. AES-GCM)
  - Ensures integrity of **encrypted** firmware blob.

Firmware encryption: Assets?

Possible firmware assets to protect:

- **Software IP**
  - Allow confidentiality protection for software IP.
- **Device secrets**
  - Allow firmware image to act as secret store (though unlikely to be suitable for high value secrets).
- **Implementation details**
  - Make it harder to develop exploits for any vulnerabilities in the firmware.

Firmware encryption: Use-cases?

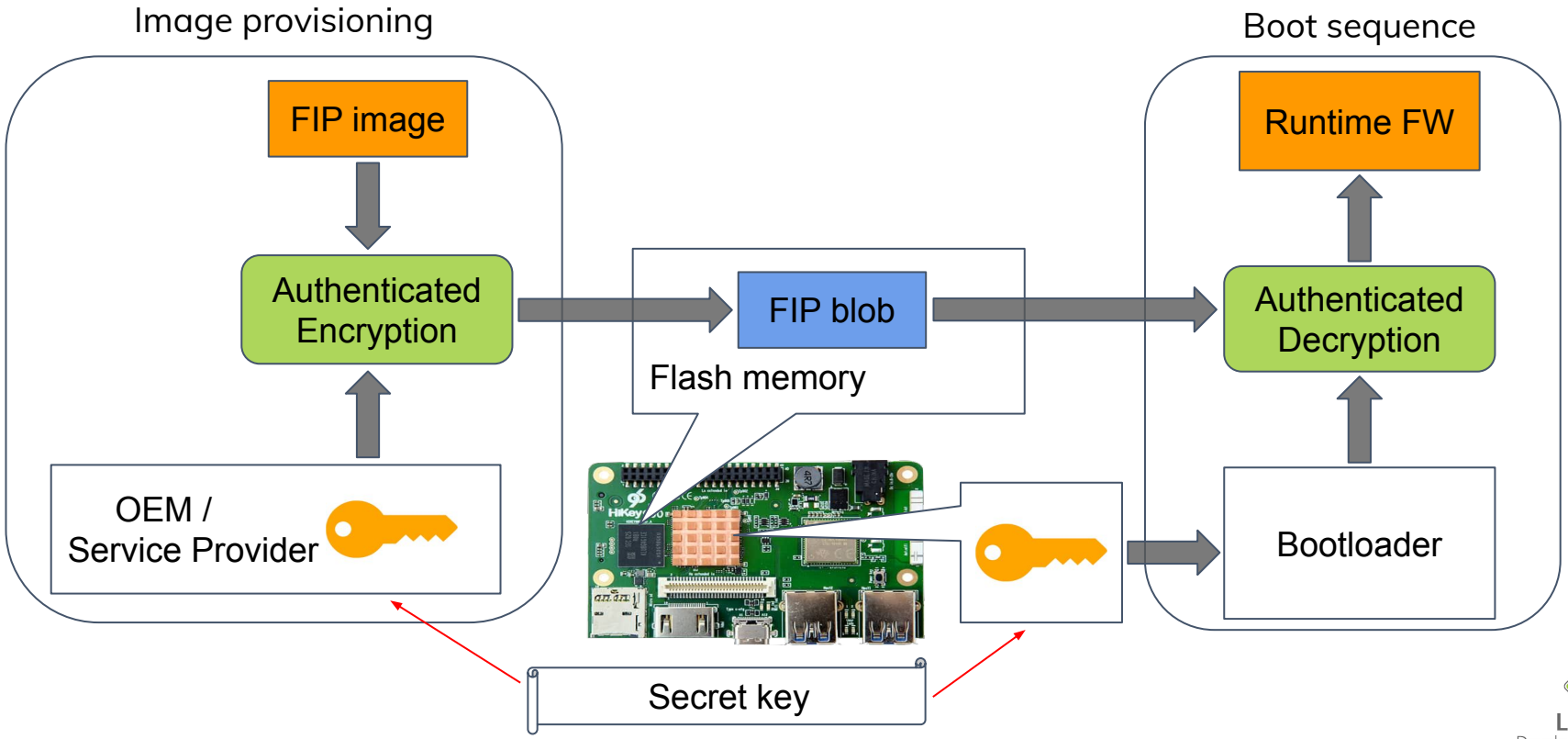
The major drivers for this feature are the emerging robustness requirements for software **Digital Rights Management** (DRM) implementations.

Make it even harder to reverse engineer Trusted Execution Environment (TEE) and therefore would like to see that **Trusted OS** is not just signed, but also **encrypted**.

TEE assets:

- DRM software IP.
- DRM implementation details.

Firmware encryption



Challenge: Secret key protection?

Secret key protection may vary from one platform to another depending on use-case and hardware capabilities like:

- Key is derived from **device secrets** like OTP or such.
- Key is provisioned into **secure fuses** on the device.
- Key is provisioned into **hardware crypto accelerator**.
- Key is provisioned into platform **secure storage** like non-volatile SRAM etc.

Solution: Secret key protection

In order to address this varying requirement, we need to provide an **abstraction layer** to retrieve **secret key / secret key handle** and platform can provide underlying implementation.

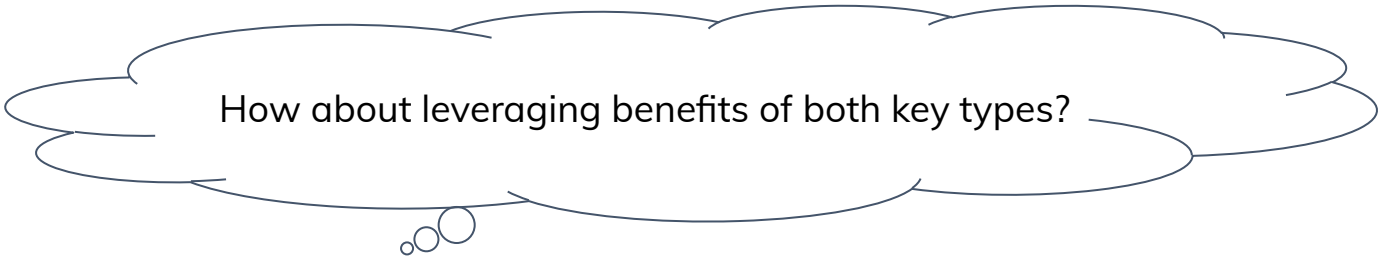
TF-A provides:

```
int plat_get_enc_key_info(enum fw_enc_status_t fw_enc_status,
                          uint8_t *key, size_t *key_len,
                          unsigned int *flags, const uint8_t *img_id,
                          size_t img_id_len);
```

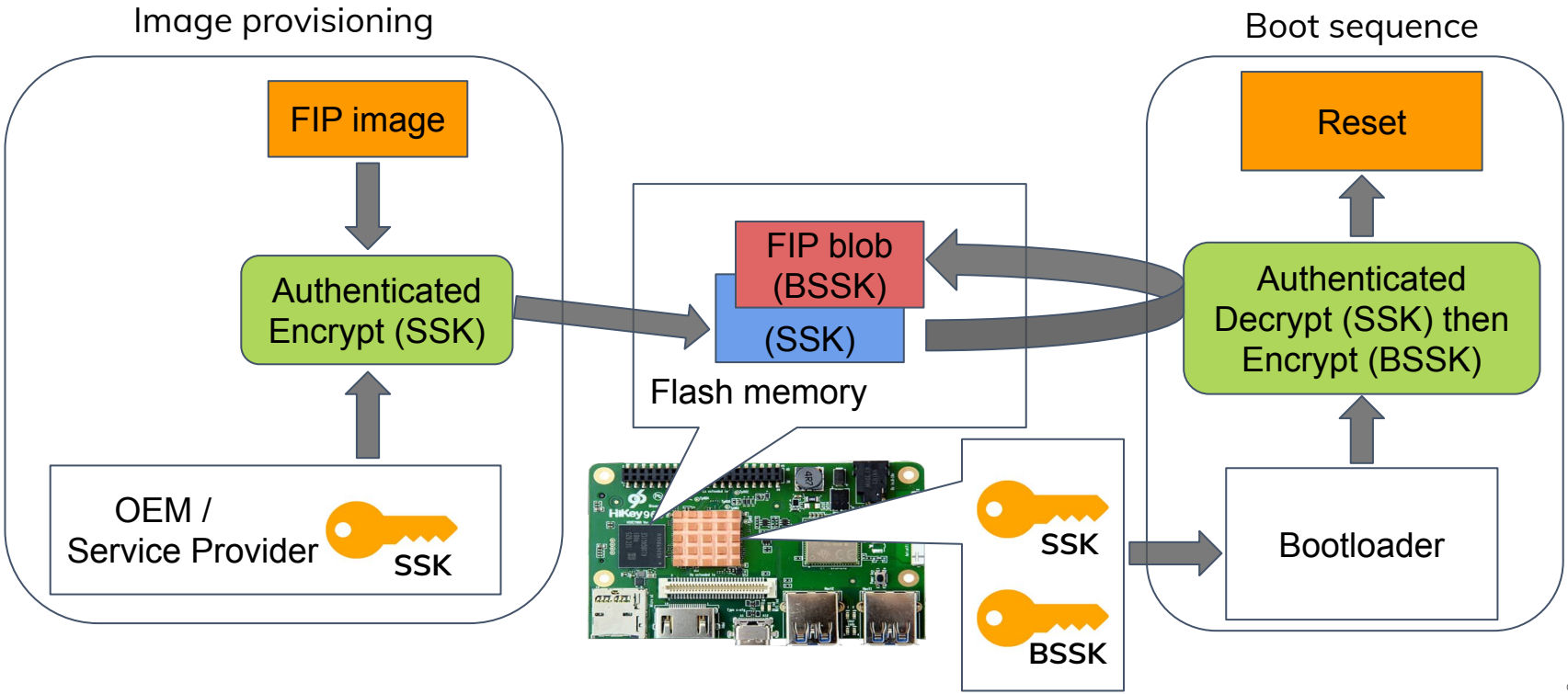
Challenge: Device unique or class wide key?

Secret key type?

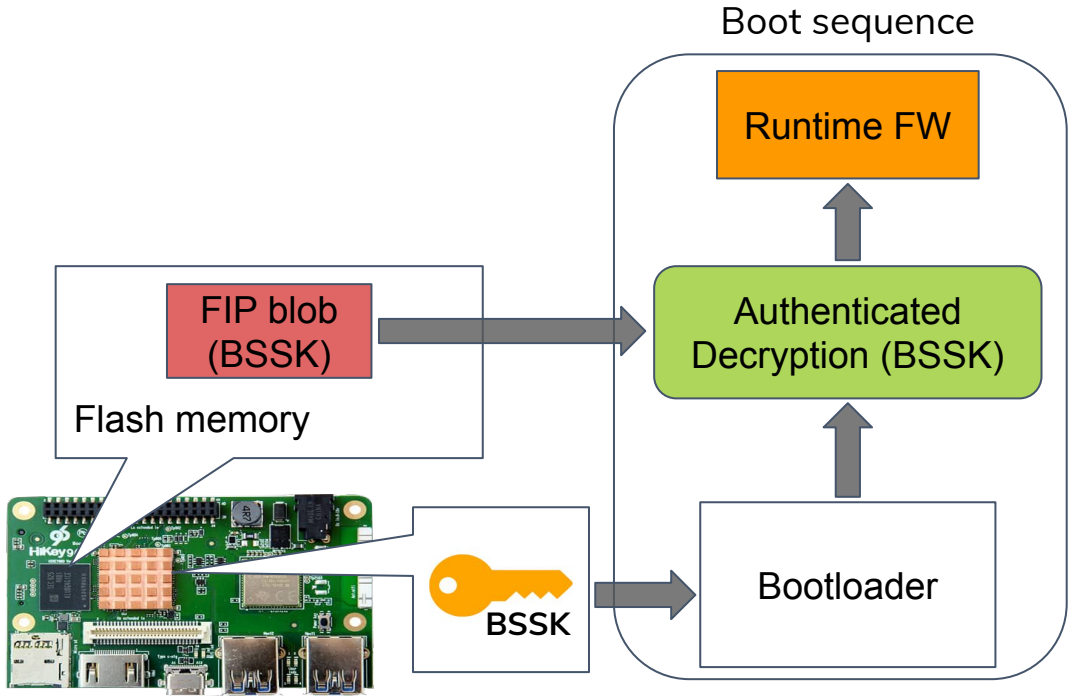
- **Device unique key:** Unique per device, aka Binding Secret Symmetric Key (**BSSK**)
  - **Pros**: limits attacks surface to per device, provides protection against software cloning.
  - **Cons**: scalability issue to manage per device unique firmware images.
- **Class wide key:** Common shared key for a class of devices, aka Shared Secret Key (**SSK**)
  - **Pros**: single firmware image, easy to deploy and update.
  - **Cons**: comparatively larger attack surface, class wide attacks.



Firmware encryption: first boot (firmware binding)



Firmware encryption: subsequent boot



Challenge: encryption + signature?

Encryption and **signature** schemes are well known cryptographic constructs but when their combination is to be used:

- Proper attention is required towards **achievable** security properties

Possible combinations:

- **Encrypt-then-sign**
- **Sign-then-encrypt**
- **Sign-then-encrypt-then-MAC**

Challenge: encryption + signature?

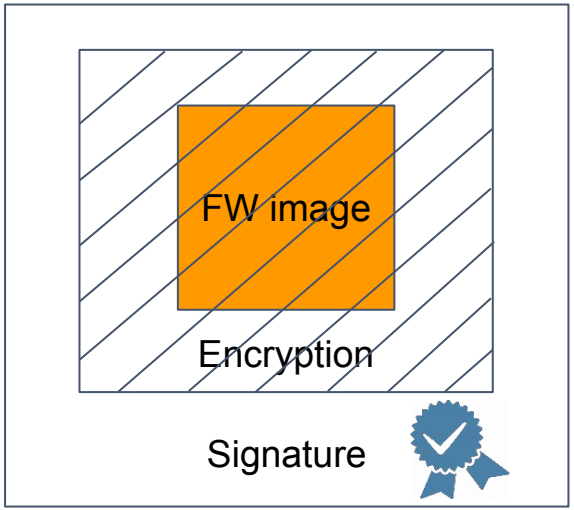
Encrypt-then-sign

Security properties:

- Confidentiality
- Integrity
- Authentication
- Authorization

Shortcomings:

- Only encrypted firmware blob is **non-repudiable** to OEM / SP.
- Signing encrypted blob makes it **immutable**
  - Doesn't allow **re-encryption** on device, aka firmware binding.



Challenge: encryption + signature?

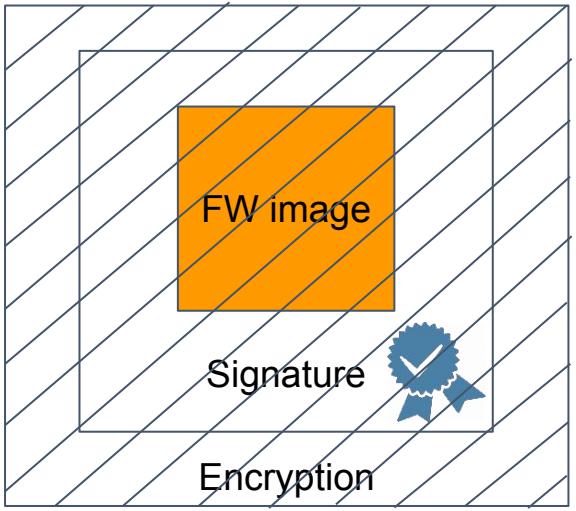
Sign-then-encrypt

Security properties:

- Confidentiality
- Authentication
- Authorization
- Non-repudiation

Shortcomings:

- **Plain** encryption doesn't assure integrity of encrypted blob.
  - Vulnerable to **Chosen Ciphertext Attacks** (CCAs).



Solution: encryption + signature

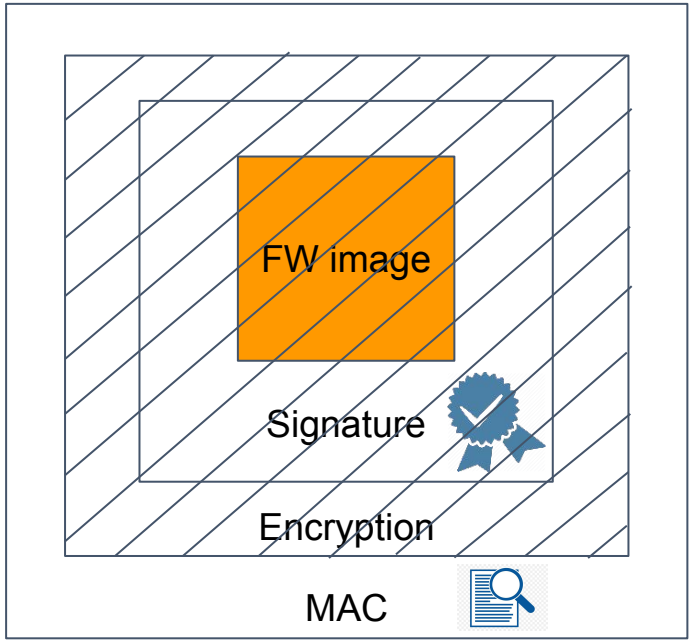
Sign-then-encrypt-then-MAC

Security properties:

- Confidentiality
- Integrity
- Authentication
- Authorization
- Non-repudiation

Concerns addressed:

- MAC tag assures **integrity** of encrypted blob.
- Allows firmware **re-encryption**.



Challenge: Firmware updates?

Generally, following approaches are used to apply firmware updates:

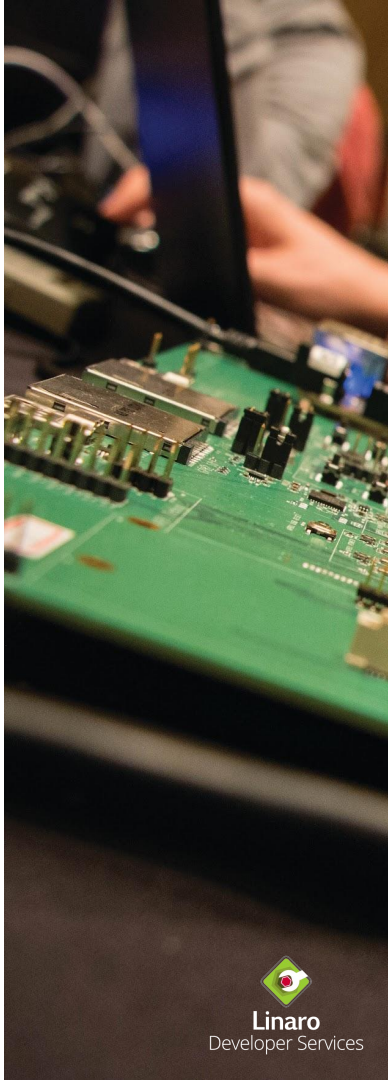
- Update complete FIP **partition**
  - Encryption **doesn't** adds any complexity
    - Updater could verify overall FIP partition signature.
- Update **individual** images in FIP
  - Encryption **adds** complexity:
    - Updater needs to verify each individual image, requires access to encryption key.
    - Either updater needs to be a secure world entity or leverages secure world decrypt and verify service.

Implementation

Trusted Firmware-A (TF-A) supports an **I/O encryption layer** (drivers/io/io_encrypted.c):

- Layered on top of any base I/O layer (eg. drivers/io/io_fip.c)
  - To allow loading of corresponding encrypted firmware payload.
- Approach used: **sign-then-encrypt-then-MAC**
  - Leveraging existing **TBBR** Chain of Trust.
- Uses **encrypt_fw** tool (tools/encrypt_fw/) to encrypt firmwares during build.
- Build options:
  - **DECRYPTION_SUPPORT**: enables firmware decryption layer (values: **aes_gcm** or **none**)
  - **FW_ENC_STATUS**: firmware encryption key flag (values: 0 -> **SSK**, 1 -> **BSSK**)
  - **ENC_KEY**: 32-byte (256-bit) symmetric key
  - **ENC_NONCE**: 12-byte (96-bit) encryption nonce or Initialization Vector (IV)
  - **ENCRYPT_{BL31/BL32}**: flag to enable BL31/BL32 encryption

Lab sessions



Reminder: Preparation and boot

Preparatory steps remains similar as for TFA-01 lab sessions.

Boot cmdline:

```
qemu-system-aarch64 \
    -machine virt,secure=on -cpu cortex-a57 \
    -smp 4 -nographic -m 1G -bios flash.bin \
    -drive \
     file=./core-image-tfa-qemuarm64-secureboot.wic.qcow2,if=virtio,format=qcow2 \
    -netdev user,id=eth0,hostfwd=tcp::2222-:22 \
    -device virtio-net-device,netdev=eth0
```

Don’t forget to source the SDK environment setup script!

LAB1 - Enable Trusted Board Boot on Qemu

Enabling trusted board boot

Build TF-A using following commands to enable trusted board boot, try to boot it.

```
$ git clone https://github.com/ARMmbed/mbedtls.git
$ cd mbedtls
$ git checkout mbedtls-3.4.0

$ cd trusted-firmware-a
$ LDFLAGS= make CFLAGS= PLAT=qemu DEBUG=1 \
    BL33=$SDKTARGETSYSROOT/boot/u-boot.bin BL32_RAM_LOCATION=tdram \
    TRUSTED_BOARD_BOOT=1 GENERATE_COT=1 SAVE_KEYS=1 \
    MBEDTLS_DIR=/path/to/mbedtls all fip
```

Things to explore?

Your trainer would like you to explore trusted board boot and find answers to following questions:

- Find out the **key and content certificates** used? Use fiptool to analyze the generated "`fip.bin`".
- Find out where the **ROTPK hash** is stored for Qemu platform?
- Find out the **signature and hashing algorithm** used for Qemu platform?
- Find out **custom X509.v3 extension fields** utilized by trusted board boot?
- Try to **maliciously modify** any of the FIP images (fiptool can be handy here) and see if trusted board boot is truly enforced.

LAB2 - Enable firmware encryption on Qemu

Enabling firmware encryption

Build TF-A using following commands to enable firmware encryption, try to boot it.

```
$ git clone https://github.com/ARMmbed/mbedtls.git
$ cd mbedtls
$ git checkout mbedtls-3.4.0

$ cd trusted-firmware-a
$ LDFLAGS= make CFLAGS= PLAT=qemu DEBUG=1 \
    BL33=$SDKTARGETSYSROOT/boot/u-boot.bin BL32_RAM_LOCATION=tdram \
    TRUSTED_BOARD_BOOT=1 GENERATE_COT=1 SAVE_KEYS=1 \
    MBEDTLS_DIR=/path/to/mbedtls all fip \
    DECRYPTION_SUPPORT=aes_gcm FW_ENC_STATUS=0 \
    ENCRYPT_BL31=1
```

Things to explore?

Your trainer would like you to explore firmware encryption and find answers to following questions:

- Find out **where encryption key is stored** on Qemu platform?
- Find out the **encryption algorithm** used?
- Does Qemu uses a **class wide or device unique** key?
- Explore the order in which **decryption and signature verification** are done at runtime.

Thank you

support@linaro.org

Linaro
Developer Services