ARM TrustZone

Miguel Quaresma

November 11, 2019

Contents

1 Prerequisites **1**
1.1 Development Setup 2

2 ARM TrustZone **3**
2.1 Context Switching Worlds 4
2.1.1 Exception handling in ARMv7 4
2.1.2 Exception handling in ARMv8 4

3 OP-TEE **5**
3.1 OP-TEE Components 5
3.2 Architecture 5
3.2.1 Secure Storage 7
3.3 Trusted Applications 12
3.3.1 Instances, Sessions and Commands 12

4 Trusted Firmware **13**
4.1 Trusted Board Boot - Secure Boot 13

5 Terms **16**

1 Prerequisites

ARM processors, although common, are not present in most personal computers where x86 is the dominant architecture. As such, and since TrustZone is a low level feature, it's useful to setup a development environment using either simulated or emulated hardware to run firmware and software written for this types of processors. The following section will compare two alternatives to exploring ARM's TrustZone and associated firmware and software:

- QEMU emulating an ARMv8 CPU
- ARM's FVP(Fixed Virtual Platform) which simulate a generic ARMv8 CPU

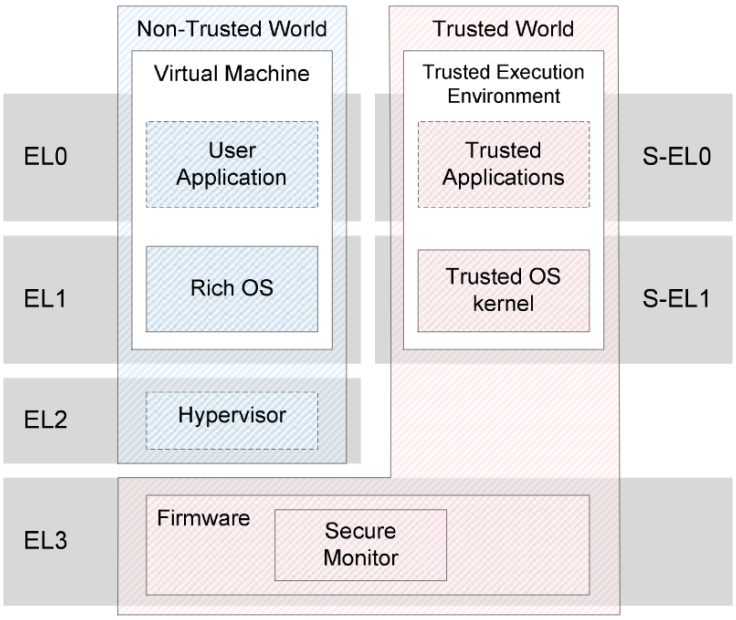
It's important to note that no approach is better than the other one, both have their use cases where they excel.

1.1 Development Setup

2 ARM TrustZone

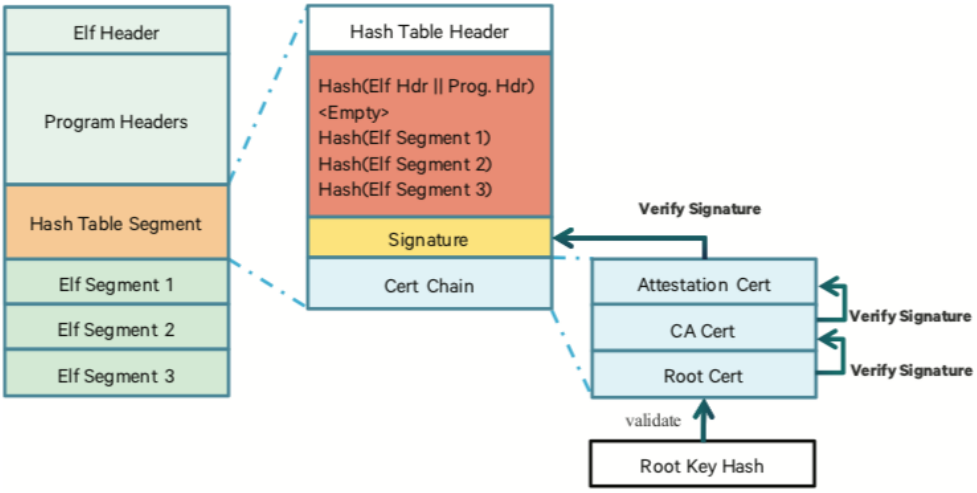
ARM TrustZone is a hardware technology present in ARM processors that allows for the implementation of Trusted Execution Environments (TEE). This technology is implemented in ARM processors via Security Extensions whereby each physical core turns into two virtual cores, one executing in a Secure World and the other in a Non-Secure World. To switch between these two worlds, the core enters a dedicated mode called the monitor mode. A TEE implemented using TrustZone requires a Monitor and might also be comprised of a TrustZone OS as well as some Trusted Applications (TAs) to provide additional functionality. Some examples of TEEs implemented using TrustZone are:

- **OP-TEE**
- **Trusty**
- **QSEE** (Qualcomm Secure Execution Environment)
- **Kinibi**
- **TrustedCore**



The Secure Monitor runs at the highest privilege level, EL3, alongside the SoC firmware which is responsible for context switching between the Secure and Normal world. The trusted world kernel runs at EL1 while the TAs run at EL0. In the normal world there can be a Hypervisor running at EL2, mediating hardware access to the kernel, running at EL1.

TAs, usually called trustlets, are integrity checked before executing. This check is achieved via a certificate chain in which the leaf certificate is used to sign a hash table where each element is the hash of an ELF segment of the TA binary. This hash table, as well as the certificate chain, are embedded in the trustlet:



2.1 Context Switching Worlds

The world in which the processor is currently running in is determined by the Non-Secure bit in the Secure Configuration Register which also enables main memory isolation. The mechanism used to context switch between these two worlds, from the Non-Secure world, is the execution of a Secure Monitor Call (SMC), which raises an "excecption" to be handled in monitor mode.

2.1.1 Exception handling in ARMv7

In ARMv7 the exception handling is done by the Secure World, which registers an instruction to which the processor will jump to when a SMC instruction executes. This instruction corresponds to the address stored in the Monitor Vector Base Address Register (MVBAR) + 0x8.

2.1.2 Exception handling in ARMv8

ARMv8 introduced Exception Levels (ELn) which determine the Privilege Level(PLn) and processor mode (Secure or Non-Secure) at which the current process is running. When an exception occurs the processor jumps to an exception vector table and runs the corresponding handler. Each EL has it's own exception vector table.

3 OP-TEE

OP-TEE is a Trusted Execution Environment designed to run on ARM processors in parallel with a non-secure Linux kernel. It exposes two APIs:

- **TEE Internal Core API**: exposed to Trusted Applications
- **TEE Client API**: allows communication with the TEE

OP-TEE was designed to take advantage of the isolation provided by ARM's TrustZone technology but is compatible with other forms of isolation such as VMs or separate processors.

3.1 OP-TEE Components

OP-TEE's main components are:

- secure privileged layer running at EL-1 (v8-A) or PL-1(v7-A) level
- secure userspace libraries to be used by Trusted Applications
- Linux kernel TEE framework
- Linux userspace library designed upon the TEE Client API
- Linux userspace supplicant daemon (tee-supplicant) responsible for remote services

3.2 Architecture

Interrupt handling is an important factor for context switching between world execution. As we've seen before, ARMv8 uses an exception vector table to map exceptions to their respective handler based on the current Exception Level (ELn). World switching, both to and from the secure world, is done by the Secure Monitor, which traps SMC's. In OPTEE OS, interrupts such as IRQ/FIQ exceptions might also involve world switching.

The type of interrupt, secure or non-secure, determines which OS handles it. When a secure interrupt is signaled by the ARM GIC(Generic Interrupt Controller), if the secure world is executing it handles the interrupt from it's exception vector. If, however, the normal world is executing, the exception is handled by the Monitor which invokes the secure world OS to serve it.

When a non-secure interrupt is signaled in the secure world, the Monitor will transfer execution to the normal world to serve it and then switch back to secure world. In case the normal world is executing, it will handle the exception via it's exception vector.

The Monitor can be reached, from secure or normal, by invoking SMC's or via interrupts such as IRQ/FIQ however, any interrupt raised while in Monitor Mode is trapped by it. In ARMv7 the Monitor's vector is MVBAR and in ARMv8 it's VBAR_{EL3}.

OPTEE OS provides it's own secure monitor which should be used by ARMv7 processors. ARMv8 can use the monitor provided by TrustedFirmware-A.

All SMC interrupts are trapped by the Monitor exception vector while IRQ/FIQ can be trapped in the executing world.

In the normal world:

- non-secure interrupts are trapped by the normal world exception vector
- secure interrupts are trapped by the Monitor which will forward them to the secure world

In the secure world all interrupts are trapped by the secure exception vector and forwarded according to their type.

These leads to two distinct definitions/categories of interrupts:

- **Native Interrupt**: handled by the OPTEE OS
- **Foreign Interrupt**: not handled by the OPTEE OS

The Secure Monitor is responsible for managing all entries/exits in/out of the secure world. When entering the secure world, the Secure Monitor saves the execution state of the normal world (general registers and system registers) and restores the previous state of the secure world. Some registers are not saved as they're used for argument passing.

When entering and exiting the secure world each CPU has it's own entry stack and disables IRQ/FIQ interrupts. An invoke to an SMC can be categorized in two flavours:

- **Fast SMC**: secure world will execute on entry stack with IRQ/FIQ blocked
- **Standard SMC**: secure world will execute the requested service with IRQ/FIQ unblocked. To enable this, a trusted thread that handles SMC call is created and, when scheduled, IRQ/FIQ are unblocked. Whenever the secure world needs to forward a foreign interrupt, it suspends the trusted thread and blocks IRQ/FIQ. The trusted thread stores the execution context of the requested service and is released only when the service returns with an exit status.
- **Both**: end on the entry stack with IRQ/FIQ blocked, with the secure world invoking the Monitor through an SMC, to return to the normal world

OPTEE OS doesn't implement any thread scheduling as such, trusted threads are scheduled by the normal world operating system. In the case of OPTEE, the Linux kernel driver makes it so that, when the Linux kernel invokes OP-TEE, the thread that executed the SMC is assigned a (trusted) thread on the secure world **i.e.** 1:1 mapping. Whenever the OPTEE OS has to handle a foreign interrupt, the context of the trusted thread is saved and, on the normal world, the "mapped" thread is scheduled. Therefore, the trusted thread only resumes execution when the normal thread invokes an SMC again.

In SMP processors, if the normal world kernel supports it, there can be multiple trusted threads running in parallel.

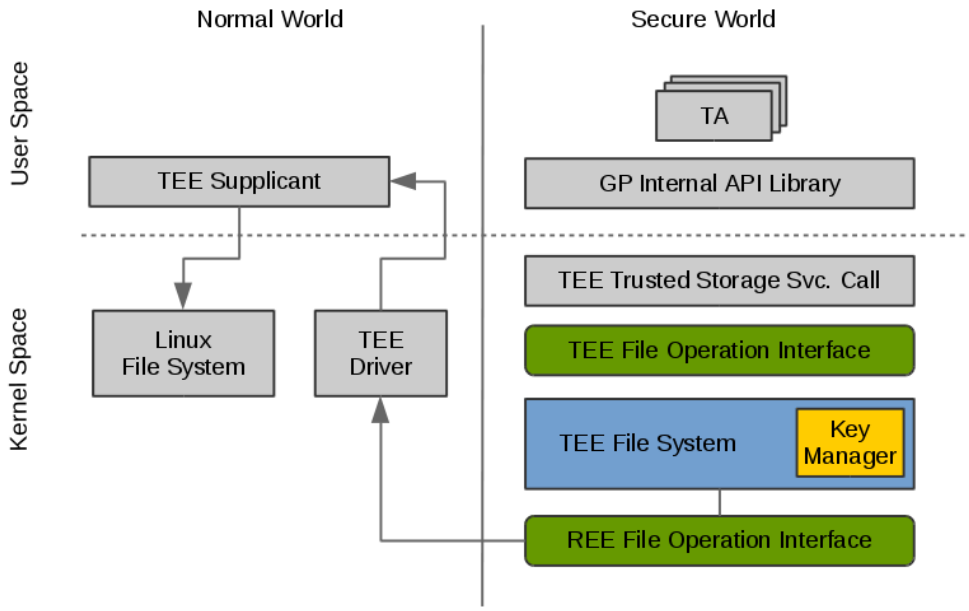
3.2.1 Secure Storage

Secure Storage, or Trusted Storage in GlobalPlatform terminology, is used to store general-purpose data and key material while assuring its confidentiality and integrity as well as the atomicity of the operations performed on the storage. Each TA has it's own secure storage, not accessible to other TAs nor the REE, whose applications and kernel can't read, modify or delete data in secure storage. OPTEE secure storage implements GlobalPlatform's API for secure object manipulation.

OP-TEE implements two different secure-storages:

- REEFS secure storage which can be enabled at compile time
- RPMB(Replay Protected Memory Block) partition on an eMMC device

It's possible to use both implementations simultaneously using their respective identifiers `TEE_STORAGE_PRIVATE_REE`, and `TEE_STORAGE_PRIVATE_RPMB` (`TEE_STORAGE_PRIVATE` refers to REEFS when available).

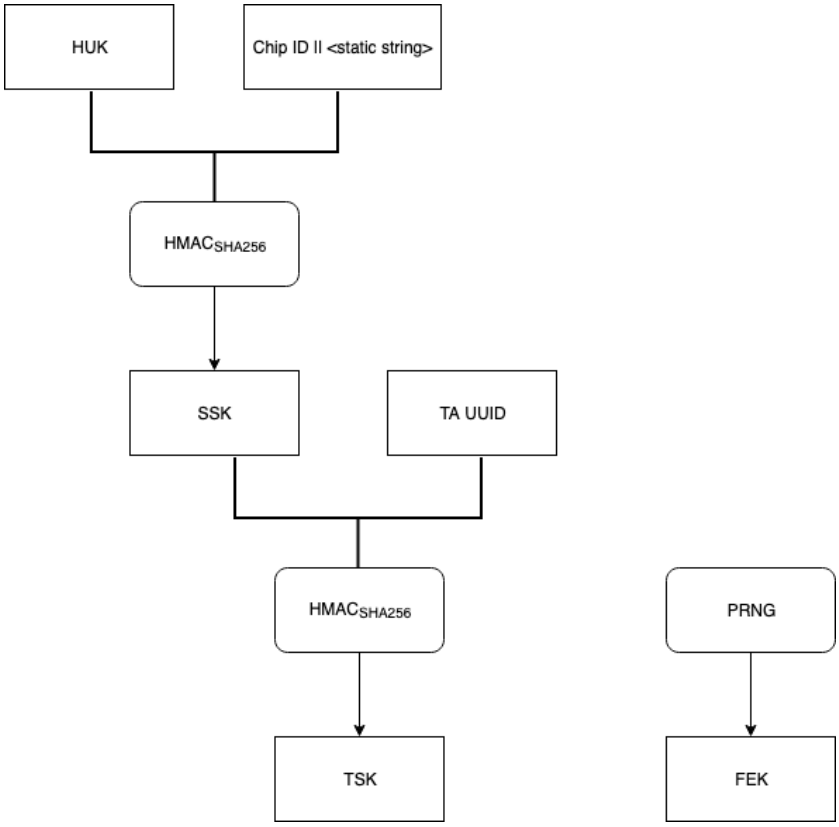


The Key Manager is a component of OPTEE which is responsible for handling encryption/decryption of data and management of the keys used in these operations. There are three types of keys involved in secure storage.

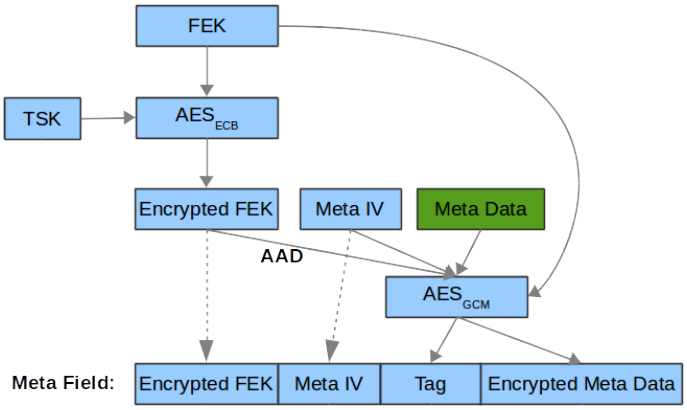
1. The SSK, Secure Storage Key, is used to derive the TSK and is unique to each device.
2. The TSK, Trusted Application Storage Key, is used to encrypt/decrypt the FEK.

3. The FEK, File Encryption Key, is generated for each TEE file/secure object and is used for encrypting/decrypting it and is stored in metadata

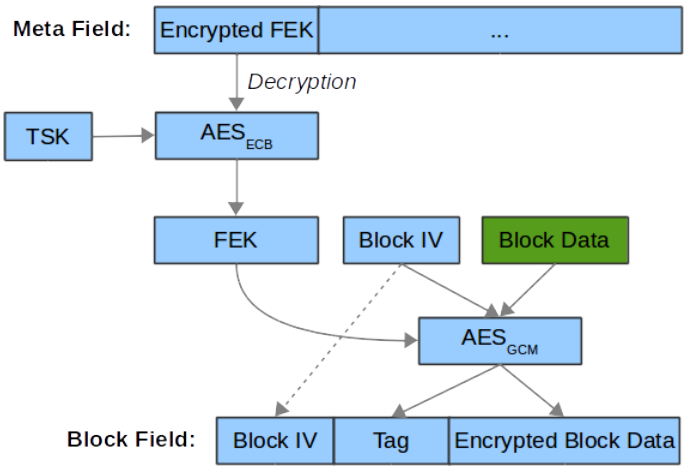
Each of this keys is generated from a set of parameters shown in the following diagram:



1. **Hash Tree** Secure objects can span several data blocks, each encrypted with it's IV and returning a unique tag. To ensure the integrity of each block and the metadata associated with it, a hash tree is used for each secure object. The hash tree, a complete binary tree, is made up of nodes each storing information relative to a single data block, namely it's IV and tag. To ensure the integrity of the tree, each node also stores the hash of it's children nodes. The metadata associated with the secure object is stored in the hash tree header, which contains the encrypted FEK, a counter for rollback protection and metadata (number of nodes and payload length) AES-GCM encrypted using FEK:x



When a block of data is encrypted, the FEK is decrypted and used as input, along with a Block IV generated by a PRNG, to AES_{GCM}:



and a new node is added to the hash tree.

To ensure rollback protection only the hash of node 1 of each secure object needs to be stored.

2. **Atomic updates** The atomicity of operations done in secure storage is achieved by using two backup version for each object: 0 and 1. At any given moment, one version is active and the inactive version is used for updating the object. When the update is finished (**i.e.** hash of node 1 is written to the object list database) the active version is toggled.

3. **Object list database** To keep track of all the secure objects created OPTEE uses a special type of secure object called a object list database, which is stored in the secure world by the `dirf.db` file. Each secure object is indexed by:

   - UUID of TA and object ID (uniquely identifies the object)

- Hash of node 1
- File number (used by the REE kernel)

4. **Considerations**

   (a) The SSK is derived from the HUK and a constant string however, there's no way to use the crypto module for this derivation currently.

   (b) Secure storage is only available to TAs, which must work as a proxy to REE applications

   (c) Using normal world for secure storage makes it vulnerable to DoS as there's no way of recovering data when/if it is corrupted

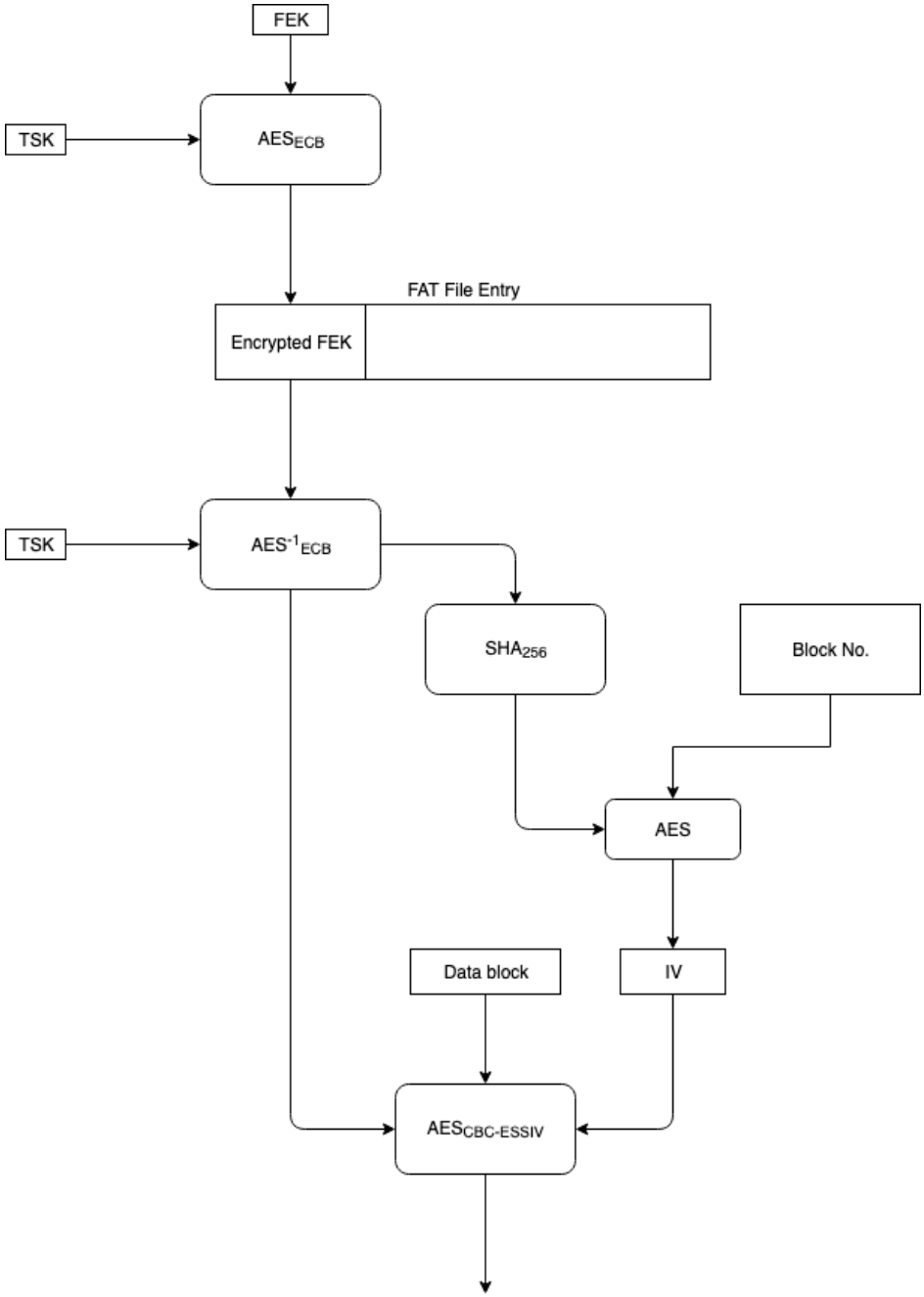
   (d) RPMB prevents rollbacks but comes with a performance penalty

5. **Notes on development** To enable the RPMB FS the flag `CFG_RPMB_FS=y` must be set at compile time. When performing IO operations on secure storage, the `TEE_STORAGE_PRIVATE_RPMB` identifier or the `TEE_STORAGE_PRIVATE`, when REEFS is disabled, should be used when invoking calls available in `tee_file_operations` interface to ensure that RPM set as the storage. The calls to this interface are done via RPC to the `tee_supplicant` running in normal world, which in turn uses `ioctl` to interact with the eMMC device. The RPMB filesystem operations must be routed to the normal world because the Linux kernel assumes a full ownership model of flash memory. As required by the GlobalPlatform Trusted Storage specification, all file operations performed on this FS are atomic. Memory buffers for the requests to perform IO operations are stored in shared memory and use HMAC authentication. The encryption mechanism for files in RPMB FS is as follows:

   (a) 128 bit SSK is generated when OPTEE boots; a TSK is generated, using the SSK and UUID of the TA

   (b) For each file a FEK is generated and encrypted using the TSK before being stored in the FAT entry of the file

   (c) Each 256-byte block of data is encrypted using AES in CBC-ESSIV with an IV obtained from the ESSIV **i.e.** the result of AES encrypting the block number with the first 128 bits of the SHA256 hash of FEK as key



The RPMB FS is made up of three parititions:

- 0-128 bytes: partition data
- 512- bytes : FAT which grows dinamically with the number of files (one entry per file)
- -EOP bytes: file data which grows downwards

3.3 Trusted Applications

Trusted Applications (TA), commonly referred to as trustlets, are programs that run in the secure world. This programs run isolated from the normal world and from each other guaranteeing both data and code integrity. To allow for interaction with the normal world, each TA exposes an "interface" based on commands. When a command is received, the TA parses it and executes the expected computation, returning the result to the client. Clients run in the normal world and have access to both the TA interface as well as the TEE Client API. TAs can act as clients to other TAs by requesting their services. TAs are uniquely identified by a UUID.

3.3.1 Instances, Sessions and Commands

Clients that issue TA commands do so via an instance of that TA. TA instances have their own physical memory space which holds the instance's heap and writable global and static data. Code in instances is executed by Tasks managed by the Trusted OS. An instance can have several Sessions which represent sequences of commands issued by a client, each having their own state. Commands implemented by TAs to express funcionality, each identified by an ID and taking up to four parameters as arguments, called Operation Parameters. This parameters can be either values or memory references to "client space".

4 Trusted Firmware

ARM-TF, or ARM Trusted Firmware, is responsible for implementing the boot and runtime services used by OPTEE OS. In ARMv7 processors, the secure boot mechanism doesn't follow a common interface as such, this process is device dependent. ARMv8 solves this by implementing secure boot at the firmware level.

ARM-TF provides a standard for different firmware components that make up ARM CPUs, ranging from the SMC calling convention to power management and secure boot.

4.1 Trusted Board Boot - Secure Boot

Secure boot is responsible for preventing unauthorized code from running on the CPU and is a key component in guaranteeing the integrity of the code that runs from the very first instruction to the loading of the kernel.

In ARMv7-A and ARMv8-A processors using TF-A this assurance is achieved by the Trusted Board Boot component via a Chain of Trust (CoT), where each stage of the boot process checks the integrity of the following stage before loading it, using x509 certificates. At the root of a CoT are a set of (implicitly) trusted components, which in ARM are:

- SHA256 hash of the **Root of Trust Public Key** (ROTPK), stored in the trusted root-key storage registers
- the **BL1 image**, called ROM boot code as it resides in the ROM and cannot be tampered with, signed by the manufacturer

Every other image and certificate hash is integrity checked before being used. One obvious conclusion from this process is that, if the root/first stage is compromised, so are the following stages and thus, the isolation provided by the TEE is nullified.

The x509.v3 certificates used in the verification of firmware images are categorised in "Key" and "Content" certificates, with the latter being used to verify public keys used to sign the former, which store the hash of the image they authenticate. The authentication process loads the image, calculates it's SHA256 hash and compares it with the hash stored in the corresponding Content Certificate. According to the TBB specification, the signing of images can be done either with ECDSA or RSA (used by TF-A), with the ECDSA approach being more efficient memory and computationally wise, for the same level of security. To allow firmware patches to be applied and prevent downgrade attacks, when an image is loaded, a Non-Volatile (NV) counter present in the certificate is compared against the one stored in hardware, to prevent against rollbacks to a vulnerable version of the firmware image. Each time a patch is applied, the NV stored in hardware is incremented and only images whose certificate NV counter is greater than or equal to the one stored in hardware are allowed.

The keys used to establish the CoT are:

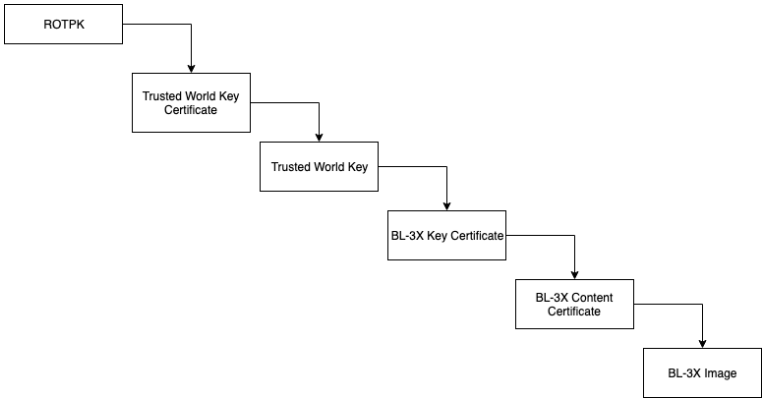
- **Root of Trust Key** (pair) whose private part is used to sign BL2's content certificate and public part is ROTPK

- **Trusted World Key** used to **sign key certificates** for the secure world images SCP$\backslash_{BL2}$, BL31 and BL32
- **Non-Trusted World Key** used to **sign the key certificate** for the non-secure world image BL33
- **BL3-X Keys** used to **sign the content certificates** for BL3-X image, for each of the previous BL images

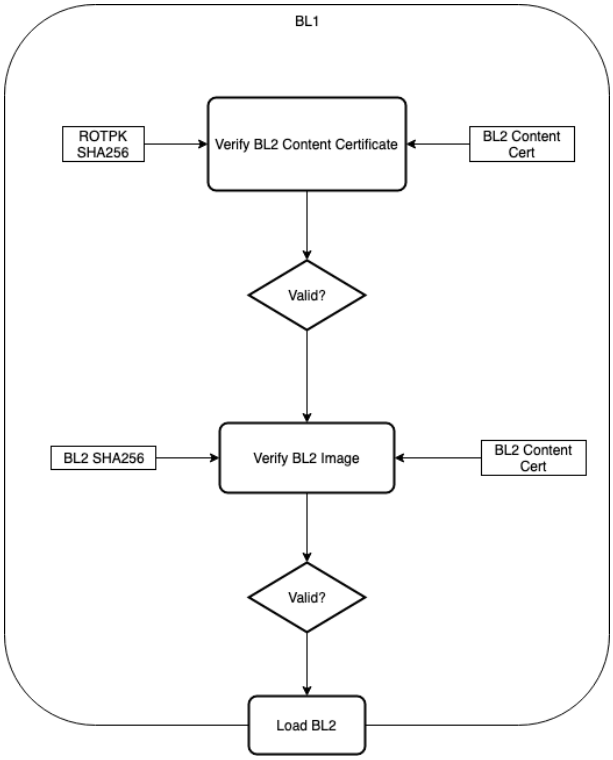
Associated with this keys are the certificates used to sign either content certificates or firmware images:

- **BL2 Content Certificate**: stores the hash of the BL2 image
- **Trusted Key Certificate**: signed by the ROT key, stores the public part of the Trusted/Non-Trusted World keys
- **BL-3X Key Certificate**: signed by the Trusted/Non-Trusted World key, stores the public part of the BL-3X key
- **BL-3X Content Certificate**: signed by the BL-3X key, stores the hash of the BL-3X image

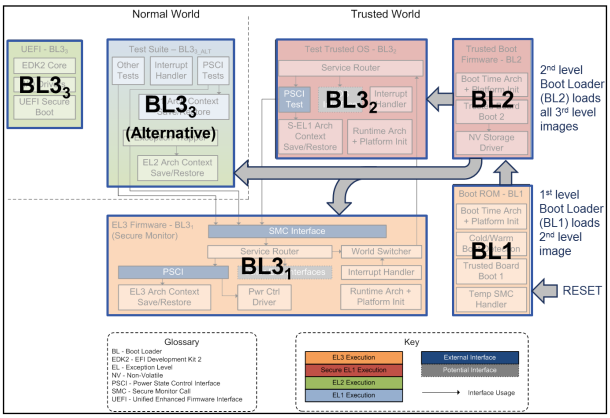
The certificates and keys listed form a verification chain for each image:



Each of the BL-3X images are loaded by BL2 which in turn is loaded by BL1:



The trusted board boot process can therefore be represented as:



5 Terms

Term	Definition
AP	Application Processor (Primary Processor)
EE	Trusted Execution Environment (ARM TrustZone + Trusted OS)
UEFI	Unified Extensible Firmware Interface
RPMB	Replay Protected Memory Block
REEFS	Rich Execution Environmnt File System
eMMC	Embedded MultiMediaCard
SSK	Secure Storage Key
TSK	Trusted Application Storage Key
FEK	File Encryption Key
PRNG	Pseudo-Random Number Generator
RPC	Remote Procedure Call
ESSIV	Encrypted Salt-Sector Initialization Vector