

# ASUG

## i.MX Android Security User's Guide

Rev. android-12.1.0\_1.0.0 —

24 October 2022

User guide

### Document information

Information	Content
Keywords	Android, i.MX, android-12.1.0_1.0.0
Abstract	This is a guide of how to do customization work on security features supported by i.MX Android software.



## 1 Preface

---

### 1.1 About This Document

This is a guide of how to do customization work on security features supported by i.MX Android software. It provides an overview of the i.MX Android security features and it focuses on how to configure and use these security features.

The released code can be built into images both with and without Trusty OS integrated while the Trusty OS related contents in this document can only be applied on the images with Trusty OS integrated.

Different SoCs may have different security-related hardware modules. This document has provided detailed information about it. Users of a specific SoC only need to focus on the information about the specific one. If the SoC type is not explicitly specified, refer to the common descriptions for all SoCs.

### 1.2 NXP security disclaimer

- NXP has documented how to correctly configure the security IPs in the documentation for the i.MX family.
- It is not possible to provide a universal secure configuration for Trusty OS that meets all users' requirements and the security of Trusty OS is enhanced release by release.
- Users should therefore customize the Trusty OS security configuration to lock and secure their end products according to their specific security requirements.
- NXP delivers Trusty OS including its Android space secure modules as an open-source software enablement and not as a secure production ready implementation.
- Using Google AOSP Trusty OS source codes instead of NXP Trusty OS source codes might have an impact on the features supported and security level of i.MX platforms.

### 1.3 Conventions

The following conventions are used in this document:

- Software code is shown in `Consolas` font.
- `${MY_ANDROID}` is a reference to the i.MX Android source code root directory.
- `${MY_TRUSTY}` is a reference to i.MX Trusty OS source code root directory.

## 2 Overview of i.MX Android Security Features

---

### 2.1 Introduction of security-related hardware modules

Security features are based on security-related codes, which need to do some cryptographic calculations to protect security data. Security requires that only security-related code is allowed to run on certain hardware resources. Therefore, these hardware resources are called security-related hardware modules. There are some security hardware modules on the i.MX platform, which co-work with the Trusty OS to guarantee security:

- CAAM: Cryptographic Acceleration and Assurance Module, is a hardware component of a System on Chip (SoC) that provides security assurance and hardware acceleration

of cryptographic algorithms, packet encapsulation and decapsulation, and other cryptographic operations.

- **TrustZone:** Arm TrustZone creates an isolated secure world, which can be used to provide confidentiality and integrity to the system. It is used to protect high-value code and data for diverse use cases like authentication. It is frequently used to provide a security boundary for the Trusted Execution Environment, like Trusty OS.
- **TZASC:** TrustZone Address Space Controller, is an Advanced Microcontroller Bus Architecture (AMBA) compliant SoC peripheral. It is a high-performance, area-optimized address space controller to protect security-sensitive software and data in a trusted execution environment against potentially compromised software running on the platform.
- **CSU:** Central Security Unit sets access control policies between the bus masters and bus slaves, enabling the peripherals to be separated into distinct security domains.
- **RDC:** Resource Domain Controller (RDC) provides support for the isolation of destination memory mapped locations such as peripherals and memory to a single core, a bus master, or set of cores and bus masters.
- **xRDC:** On i.MX 8QuadMax and i.MX 8QuadXPlus, the eXtended Resource Domain Controller (xRDC) replaces the RDC and TrustZone components (CSU, TZASC, etc.), which can be found in previous i.MX processors.  
i.MX 8QuadMax and i.MX 8QuadXPlus SoC contain a mix of Cortex-A and Cortex-M CPUs, which frequently operate in an asymmetric mode with different software environments executing on them. To keep these software environments from unintentionally interfering with each other, the SoC contains xRDC to enforce isolation. The xRDC operates in a manner like Arm's TrustZone. Transactions from masters are annotated with user-side band information to indicate their domain and the access control logic allows/disallows accesses to peripherals/memory based on this information.
- **AHAB/HABv4:** The Advanced High Assurance Boot (AHAB) and High Assurance Boot (HABv4) support authentication on the images by using cryptography operations to prevent unauthorized software from being executed during the device boot sequence. Details about how to verify images with HAB can be found in Chapter 2.1.
- **SCU:** The System Controller Unit (SCU) is only for i.MX 8QuadMax and i.MX 8QuadXPlus platforms. It consists of a Cortex-M4 processor and a set of peripherals and interfaces to connect to an external PMIC and to control internal subsystems. The SCU Cortex-M4 is the first processor to boot the chip. The SCU is dedicated to:
  - Boot management
  - Power management
    - External power management by communicating with external PMIC
    - Internal power management of all the subsystems
  - Clock and reset management
  - I/O configuration and muxing
  - Resource partitioning/access control
- **SECO:** The Security Controller Subsystem (SECO) is only for i.MX 8QuadMax and i.MX 8QuadXPlus platforms. It manages several security hardware modules (CAAM, SNVS, OTP, ADM, etc.) to perform cryptography acceleration and ensure the security of the whole system.
- **eMMC RPMB:** RPMB is a separate physical partition in the eMMC device designed for secure data storage. Every access to RPMB is authenticated and it allows the host to store data to this area in an authenticated and replay protected manner. In Trusty OS, the RPMB partition is managed as the secure storage to store all critical data like lock/unlock status, rollback index, etc.

The following table lists the modules on the i.MX 8QuadMax, i.MX 8QuadXPlus, i.MX 8M Mini, i.MX 8M Quad platforms:

**Table 1. Modules on the i.MX 8QuadMax, i.MX 8QuadXPlus, i.MX 8M Mini, i.MX 8M Quad platforms**

Modules	i.MX 8QuadMax and 8Quad XPlus	i.MX 8M Quad, 8M Mini, 8M Nano, and 8M Plus
CAAM	Y	Y
TZASC	N	Y
CSU	N	Y
RDC	N	Y
xRDC	Y	N
AHAB	Y	N
HABv4	N	Y
SCU	Y	N
SECO	Y	N
eMMC	Y	Y

## 2.2 Trusty OS security recommendations for i.MX 8 SoC configuration

### 2.2.1 Secure CSU configuration

The Central Security Unit (CSU) manages the system security policy for peripheral access on the SoC. The CSU allows trusted code to set individual security access privileges on each of the peripherals, using one of the eight security access privilege levels. The CSU may assign bus master security privileges during bus transactions according to the programmed policy. The CSU has two primary security-related features:

- Peripheral Access Policy: The appropriate bus master privilege and identity are required to access each peripheral.
- Masters Privilege Policy: The CSU overrides the bus master privilege signals (secure/non-secure).

On the i.MX 8M platforms, Trusty OS and other software components such as ATF, SPL, and potentially U-Boot (if run in the secure world) have access to CSU registers and potentially configure or overwrite peripheral access and master privilege policies. The secure code (CSU driver) may have a non-secure CSU configuration by default. The CSU configuration should be modified from the default non-secure configuration based on the security requirements of the final application/end product:

- The CSU\_CSln registers that control the Peripheral access policy and the CSU\_SA registers that control the Masters privilege policy should be set appropriately.
- All CSU register configuration should be locked with the appropriate bit fields to prevent any CSU modification at runtime.
- The CSU may need to be re-configured upon power state transitions depending on the i.MX platform and low-power support.

**Note:**

- Since *imx-android-11.0.0\_2.2.0*, the Trusty OS and ATF have made some default CSU configuration for reference.
- For more information, refer to the CSU chapter and Security Integration chapters in the respective SoC i.MX Security Reference Manual.
- An i.MX Security Checklist is available and should be perused to ensure the end product is correctly secured.

### 2.2.2 Secure TZASC configuration

The TrustZone Address Space Controller (TZASC) protects security-sensitive SW and data in a trusted execution environment against potentially compromised SW running on the platform. The i.MX 8M platforms use the TZC-380 controller.

On the i.MX 8M platforms, Trusty OS and other software components such as ATF, SPL, and potentially U-Boot (if run in the secure world) have access to the TZASC registers and potentially configure or overwrite memory regions. The secure code may have a non-secure TZASC configuration by default. The TZASC configuration should be modified from the default non-secure configuration based on the security requirements of the final application/end product:

- Set the memory regions to meet the security requirement.
- The region 0 should be set to secure to prevent any access to secure memory through aliased memory range.
- Memory region configuration must be locked to prevent runtime modification.
- Depending on the platform, the TZASC may need to be re-configured upon power state transitions.

**Note:**

- Since *imx-android-11.0.0\_1.0.0*, the default ATF has the reference code to configure Trusty OS memory region as TZASC protected.
- For more information, refer to the SoC i.MX Security Reference Manual and the CoreLink TrustZone Address Space Controller TZC-380 Technical Reference Manual.
- An i.MX Security Checklist is available and should be reviewed to ensure the end product is correctly secured.

### 2.2.3 Secure OCRAM configuration

The OCRAM (On Chip RAM) is a small memory embedded on the SoC. The OCRAM size varies based on the i.MX SoC. Just like the DRAM, the OCRAM memory can be protected by the TrustZone through configuration. The OCRAM supports TrustZone and non-TrustZone accesses and there is an option to configure a TrustZone-only access region. The CSU can be used to set access permissions to the OCRAM memory and in addition has programmable lock bits to prevent runtime modifications.

On the i.MX 8M platforms, Trusty OS and other software components such as ATF, SPL, and potentially U-Boot (if run in the secure world) can potentially configure or overwrite peripheral access and master privilege policies. However, the secure world and the non-secure world have both access to the GPR registers. The OCRAM configuration should be modified from the default non-secure configuration to a secure configuration based on the security requirements of the final application/end product:

- Check the Secure CSU configuration recommendations.
- The CSU must be set to protect the OCRAM.

- The respective GPR registers must be set to enable the protection of the OCRAM by the TrustZone.
- Lock GPR and CSU registers with the respective lock bits to prevent runtime modifications.

**Note:**

- *i.MX 8M Plus, 8M Quad, and 8M Nano have this configured in ATF.*
- *For more information, refer to the CSU chapter and Security Integration chapters in the respective SoC i.MX Security Reference Manual.*
- *An i.MX Security Checklist is available and should be reviewed to ensure the end product is correctly secured.*

## 2.2.4 Secure RDC configuration

The Resource Domain Controller (RDC) provides support for the isolation of destination memory mapped locations such as peripherals and memory to a single core, a bus master, or set of cores and bus masters. The RDC provides a mechanism to allow boot time configuration code to establish resource domains by assigning cores, bus masters, peripherals and memory regions to domain identifiers. Once configured, bus transactions are monitored to restrict accesses initiated by cores and bus masters to their respective peripherals and memory.

On the i.MX 8M platforms, the secure world and the non-secure world have both access permissions to the RDC registers. The RDC configuration should be modified from the default non-secure configuration to a secure configuration based on the security requirements of the final application/end product:

- Check the code for default and non-secure RDC configuration.
- Set the RDC configuration for your security requirements.
- Ensure that the settings do not conflict with the CSU and AIPSTZ configurations.
- Lock the RDC settings with the respective lock bits to prevent runtime modifications.

**Note:**

- *For more information, refer to the RDC chapter and Security Integration chapters in the respective SoC i.MX Security Reference Manual.*
- *An i.MX Security Checklist is available and should be reviewed to ensure the end product is correctly secured.*

## 2.2.5 Secure AIPSTZ configuration

The AIPSTZ is a peripheral designed as the bridge between AHB bus and peripherals with the lower bandwidth IP Slave (IPS) buses. The AIPSTZ bridge provides programmable access protections for both masters and peripherals. It allows the privilege level of a master to be overridden, forcing it to user mode privilege, and allows masters to be designated as trusted or untrusted.

On the i.MX 8M platforms, the secure world and the non-secure world have both access the AIPSTZ registers. The AIPSTZ configuration should be modified from the default non-secure configuration to a secure configuration based on the security requirements of the final application/end product:

- Check the code for default and non-secure AIPSTZ configuration.
- Set the AIPSTZ configuration for your security requirements.
- Ensure that the settings do not conflict with the CSU and RDC configurations.

- Lock the AIPSTZ settings with lock bits to prevent runtime modifications.

**Note:**

- For more information, refer to the AIPSTZ Chapter in the respective SoC i.MX Reference Manual and the SoC i.MX Security Reference Manual.
- An i.MX Security Checklist is available and should be reviewed to ensure the end product is correctly secured.

### 2.2.6 SCU/SCFW

On the i.MX 8QuadMax and 8QuadXPlus platforms, System Controller Firmware (SCFW) which running on the System Controller Unit (SCU) maintains the power, clock and security components like xRDC and TZASC. Other software components can communicate with SCU by the SCFW APIs.

By default, the Trusty OS is running on the secure partition and this is configured in the ATF. Refer to the SCFW API manual for more information.

## 2.3 i.MX Android security framework

i.MX Android/Android Automotive security framework includes secure enhanced U-Boot, Android/Android Auto, i.MX Trusty OS, and the related hardware.

Secure enhanced U-Boot provides the Android Verified Boot module, keys provisioning interface, and secure storage proxy.

Android Verified Boot assures the end user of the integrity of the software loaded and started by secure-enhanced U-Boot. This is defined by Google, and more details can be found in <https://source.android.com/security/verifiedboot/avb>.

Key provisioning interface provides the RPMB keys, key attestation, and AVB keys provisioning interface. These interfaces can be used to inject the keys into the device to make it secure.

Secure Storage Proxy is the client of Secure Storage service from Trusty OS. It helps to access the RPMB secure storage device by SoC IPs.

Android/Android Auto platform, based on Google's design, integrates the Keymaster HAL, Gatekeeper HAL, and Secure Storage proxy.

Keymaster HAL uses trusty-backed one and supports Keymaster V2 and Keymaster V3 APIs. For more details about keymaster, see <https://source.android.com/security/keystore>.

Gatekeeper also uses the Trusty-backed gatekeeper HAL. For more details about gatekeeper, see <https://source.android.com/security/authentication/gatekeeper>.

i.MX Trusty OS is based on Trusty OS that is released from Google. Secure TAs and services are integrated in it. Trusty OS is a very important module for the whole security of i.MX Android/Android Auto platform.

Trusty OS provides a trusty-ipc, which is used to realize communication between secure and non-secure world. Trusty OS has the hardware driver for CAAM used for keyblob calculation and security algorithm acceleration.

The following figure shows the logic between these components.

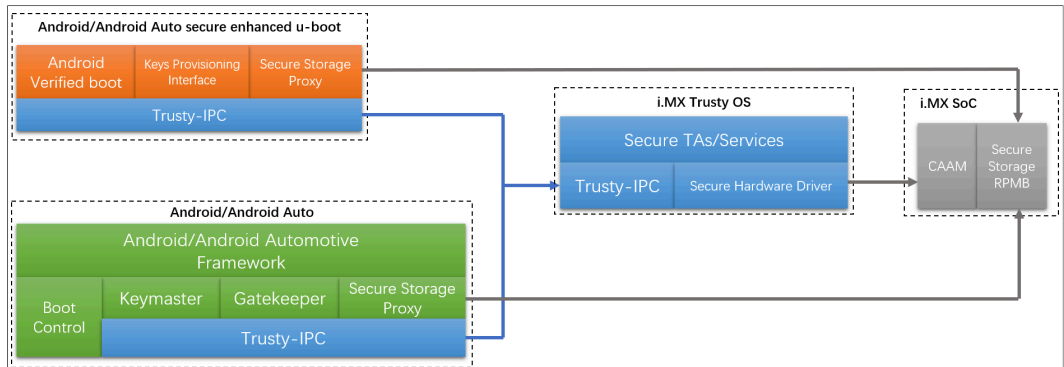


Figure 1. i.MX Android security framework

The following figure shows the i.MX Android/Android Auto security trust chain.

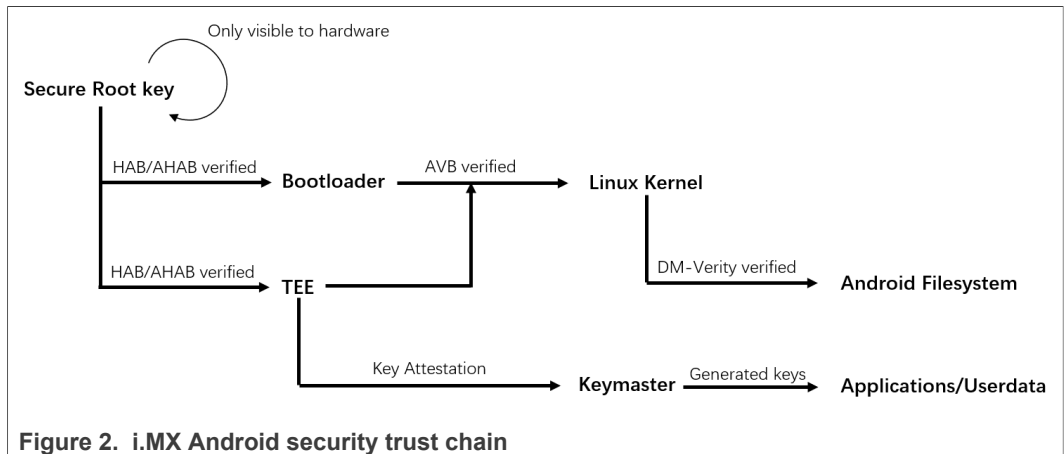


Figure 2. i.MX Android security trust chain

Secure root key is programmed into the One Time Programmable (OTP) eFuse hardware in i.MX chips and work as the root trust of the solution. It is used by CAAM to generate other keys. In the trust chain, the HAB/AHAB, AVB, and DM-Verity are used by a different level to verify the specific images or encrypt user data.

After power-on, the boot process begins, U-Boot and Trusty OS are loaded by ROM code. They are the first to be verified by ROM code with HAB/AHAB. They can only be executed after they pass the verification. U-Boot loads the Linux kernel and uses AVB to verify it before jumping to the Linux kernel. The Linux kernel mounts the Android file system. Data access from Android file system will be verified by DM-Verity to assure integrity. The security chain is formed by these features.

### 3 Customization work on i.MX Android Security Features

#### 3.1 Verifying images with HAB

The i.MX family of application processors provides the High Assurance Boot (HAB) feature in the on-chip ROM. The ROM is responsible for loading the initial image and verifying the image before the image is executed.

Due to the new architecture, multiple firmware and software images are required to boot i.MX 8 family devices. NXP defines "container" to organize images. AHAB for i.MX 8Quad and i.MX 8ULP devices can recognize the format of "container" and verify the



images in a container. For i.MX 8M devices, these images are stored in the format of Flattened Image Tree (FIT) with an appropriate Image Vector Table (IVT) set. HABv4 for i.MX 8M devices can recognize this format and verify the images. By default, HAB verification is enabled and i.MX chip is in open stage, so failure of HAB verification does not block the boot process. After closing the chip, only correctly signed images can be executed.

This section covers AHAB used on i.MX 8Quad and i.MX 8ULP family devices and HABv4 used on i.MX 8M family devices.

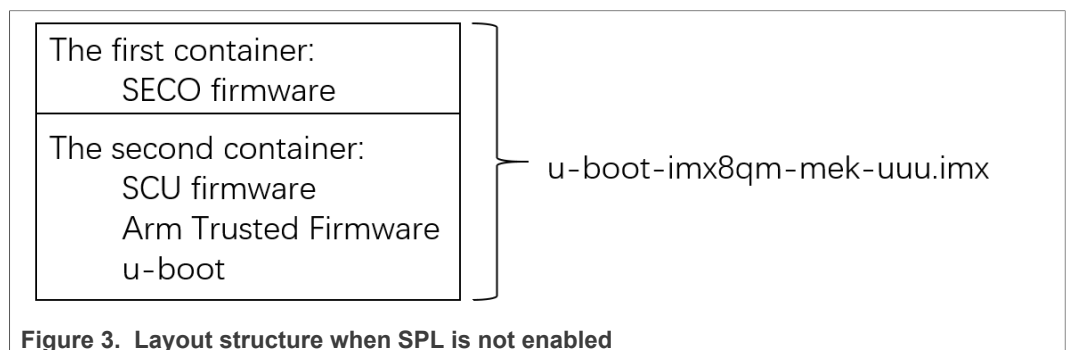
### 3.1.1 Verifying images with AHAB

AHAB is closely bound with "container". Detailed information about "container" can be found in the Reference Manual of specific chips. According to the Reference manual, the hash values of multiple firmware and software components are stored in the container header. The container sign process described below embeds an SRK table in the container and signs the container. The contents described in the sign process is used to verify the container during the boot time.

For i.MX 8QuadMax MEK and i.MX 8QuadXPlus MEK, "container" is used to organize the SECO firmware, SCU firmware, MCU software, Arm Trusted Firmware, Trusty OS, and U-Boot. For i.MX 8ULP, "container" is used to organize the Sentinel firmware, Upower firmware, MCU software, Arm Trusted Firmware, Trusty OS, and U-Boot.

The SECO firmware on i.MX 8Quad or the Sentinel firmware on i.MX 8ULP are always in the first container. When U-Boot has SPL enabled, three containers are used to organize the images. If SPL is not enabled, two containers are used. i.MX 8QuadMax is taken as an example below to introduce the different layout of images.

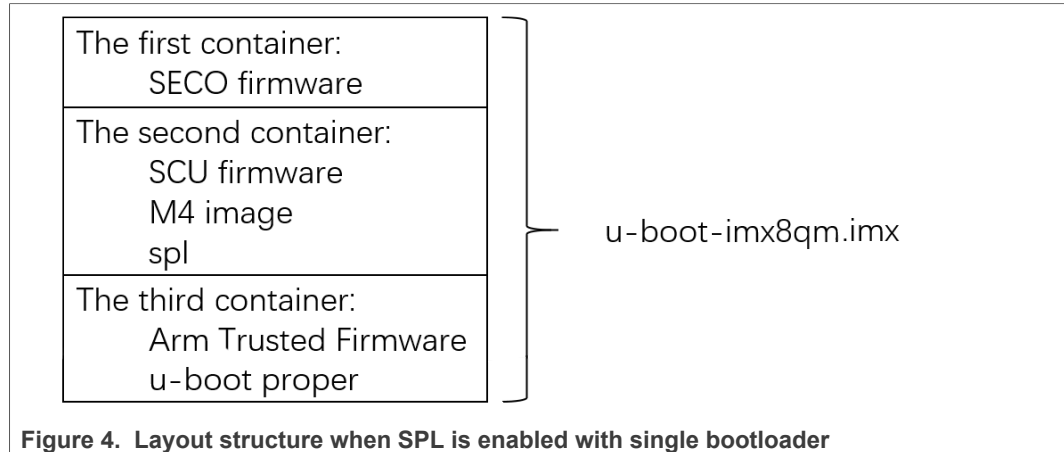
The UUU U-Boot images of i.MX 8Quad platforms do not have SPL enabled, so two containers are used. The first container contains the SECO firmware, which is provided as a binary file and signed by NXP. The second container contains SCU firmware, Arm Trusted Firmware, and U-Boot proper. It is constructed in build time and will be appended to the end of the first container. Taking `u-boot-imx8qm-mek-uuu.imx` as an example, its high-level layout structure is denoted in the following figure.



Unlike i.MX 8Quad, the UUU U-Boot image of i.MX 8ULP enables SPL, so it has three containers included.

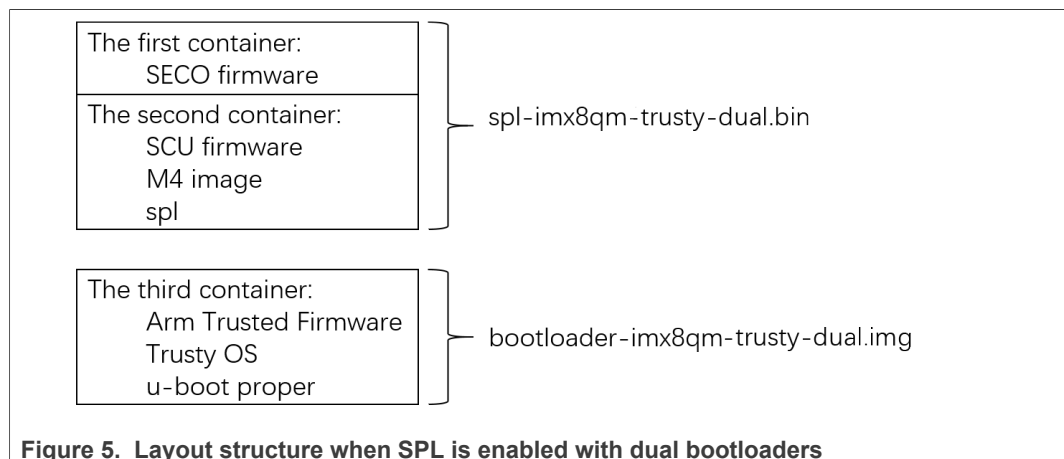
The normal u-boot images to be flashed to the board on i.MX 8Q has SPL enabled, so three containers are used. The first container contains SECO firmware. The second container is constructed in build time and appened to the first container, this container contains SPL. The third container is also constructed in build time, it contains u-boot proper. The two containers constructed at build time are not signed by default. These three containers are in two files if dual bootloader is enabled. For single bootloader

condition, take `u-boot-imx8qm.imx` of standard Android images as an example, layout structure is shown in below figure:



**Figure 4. Layout structure when SPL is enabled with single bootloader**

For dual bootloader condition, take `spl-imx8qm-trusty-dual.bin` and `bootloader-imx8qm-trusty-dual.img` of Android images as an example, the first container and the second container are combined as one image, the third container is combined as another image. The layout structure is well denoted with below figure.



**Figure 5. Layout structure when SPL is enabled with dual bootloaders**

Like i.MX 8Quad, the normal U-Boot image on i.MX 8ULP also has three containers. The single-bootloader and dual-bootloader conditions have the similar layout.

To sign the containers constructed in the process of building Android images, perform the following steps:

1. Download Code Signing Tool (CST) from the [NXP official website](#). Decompress the package using the following command:

```
$ tar zxvf cst-3.1.0.tgz
```

2. Generate AHAB PKI tree. After the tool package is decompressed, enter the directory of `release/keys/`, and execute the following command:

```
$ ./ahab_pki_tree.sh
```

Then enter some parameters based on the output of this script. An example is as follows:

```
Do you want to use an existing CA key (y/n)? : n
```

```
Do you want to use Elliptic Curve Cryptography (y/n)?: y
Enter length for elliptic curve to be used for PKI tree:
Possible values p256, p384, p521: p384
Enter the digest algorithm to use: sha384
Enter PKI tree duration (years): 5
Do you want the SRK certificates to have the CA flag set? (y/n)?: n
```

After the preceding command is successfully executed, the private keys are under the `keys/` directory, and the public key certificates are under the `crts/` directory.

3. Generate AHAB SRK tables.

Enter the directory of `release/crts/`, and execute the following command:

```
$ ../linux64/bin/srktool -a -s sha384 -t
SRK_1_2_3_4_table.bin \
-e SRK_1_2_3_4_fuse.bin -f 1 -c \
SRK1_sha384_secp384r1_v3_usr.crt.pem,\
SRK2_sha384_secp384r1_v3_usr.crt.pem,\
SRK3_sha384_secp384r1_v3_usr.crt.pem,\
SRK4_sha384_secp384r1_v3_usr.crt.pem
```

Additionally, for platforms like i.MX 8ULP, which only has 8 words of SRK fuses, regenerate `SRK_1_2_3_4_fuse.bin` by SHA256 from `SRK_1_2_3_4_table.bin`:

```
$ openssl dgst -binary -sha256 SRK_1_2_3_4_table.bin >
SRK_1_2_3_4_fuse.bin
```

After the command is executed successfully, the SRK table and its SHA512 or SHA256 value are generated and saved respectively in two files under `release/crts/`.

The SRK table is embedded in the container in the process of signing that container. Therefore, during the boot time, it can be used to verify the signature. If the signature is authenticated, the hash value of firmware and software images can be trusted to verify the corresponding firmware and software. The SRK table hash value will be fused to the OTP eFuse hardware and work as the "secure root key". It is used to verify the SRK table embedded in the container.

Files generated in `release/keys/` and `/release/crts/` are very important. If the SRK HASH value is fused to the chip and then the chip is changed from open to close state, the board can only boot with images signed with these files.

It is suggested to use a dedicated copy of the CST package for different device families, since this process will generate files with the same names in the same directory.

4. Build Android images to construct the containers to be signed.

To use AHAB to verify images in SPL, enable `CONFIG_AHAB_BOOT` configurations in corresponding defconfig files in u-boot code. they are not enabled by default. Take `imx8qm_mek_android_trusty_dual_defconfig` as an example:

```
diff --git a/configs/
imx8qm_mek_android_trusty_dual_defconfig b/configs/
imx8qm_mek_android_trusty_dual_defconfig
index 7a8a7a3c1c5..d441bded80d 100644
--- a/configs/imx8qm_mek_android_trusty_dual_defconfig
+++ b/configs/imx8qm_mek_android_trusty_dual_defconfig
@@ -214,3 +214,4 @@ CONFIG_ATTESTATION_ID_MANUFACTURER="nxp"
CONFIG_ATTESTATION_ID_MODEL="MEK-MX8Q"
CONFIG_SHA256=y
CONFIG_DUAL_BOOTLOADER=y
+CONFIG_AHAB_BOOT=y
```

`mkimage_imx8` is a tool used to construct containers. It outputs the layout information of a container on standard output when constructing it. When building the Android images, save the log information of the build system. For example, execute the following command:

```
$ ./imx-make.sh -j12 2>&1 | tee make_android.txt
```

During the build process, the build system output information is saved in `make_android.txt`. The three-example layout mentioned above can be categorized into two when signing the images: the image file directly generated by `mkimage_imx8` and the image file not directly generated by `mkimage_imx8`. As `mkimage_imx8` outputs the container layout information in a file generated by itself, if the image file is directly generated by `mkimage_imx8`, the layout information parameter can be directly used. If the final image is assembled with intermediate files generated by `mkimage_imx8`, the layout information parameters need to be properly processed before being used.

If SPL is enabled, single bootloader image like `u-boot-imx8qm.img` is assembled with two files generated by `mkimage_imx8`. In fact, dual bootloader condition keeps these two files separated, while single bootloader condition appends one file after another at 1KB aligned boundary.

5. Get the layout information of containers in a file.

Layout information is needed when signing the container. To be more specific, it is the offset information of the container and the container signature block in a file. Code signing tool uses these offsets to locate the container in a file. The layout information can be found in `make_android.txt` just generated. With the following principles, the offset of the container to be signed in a file can be easily found. The generated files with containers to be signed mentioned before are built based on different U-Boot defconfig files. Different U-Boot compilation targets with different U-Boot defconfig files can be found in `/${MY_ANDROID}/device/nxp/${PLATFORM}/${BOARD}/BoardConfig.mk`. Taking i.MX 8QuadMax MEK and i.MX 8QuadXPlus MEK as example, the U-Boot configurations are listed below with some annotations.

```
# bootloader-imx8qm.img and spl-imx8qm.bin
TARGET_BOOTLOADER_CONFIG :=
  imx8qm:imx8qm_mek_androidauto_trusty_defconfig
# bootloader-imx8qm-secure-unlock.img and spl-imx8qm-secure-
unlock.bin
TARGET_BOOTLOADER_CONFIG += imx8qm-secure-
unlock:imx8qm_mek_androidauto_trusty_secure_unlock_defconfig
# bootloader-imx8qxp.img and spl-imx8qxp.bin
TARGET_BOOTLOADER_CONFIG +=
  imx8qxp:imx8qxp_mek_androidauto_trusty_defconfig
# u-boot-imx8qxp-c0.img
TARGET_BOOTLOADER_CONFIG += imx8qxp-
c0:imx8qxp_mek_androidauto_trusty_defconfig
# bootloader-imx8qxp-secure-unlock.img and spl-imx8qxp-
secure-unlock.bin
TARGET_BOOTLOADER_CONFIG += imx8qxp-secure-
unlock:imx8qxp_mek_androidauto_trusty_secure_unlock_defconfig
# u-boot-imx8qm.img
TARGET_BOOTLOADER_CONFIG :=
  imx8qm:imx8qm_mek_android_defconfig
# u-boot-imx8qxp.img
TARGET_BOOTLOADER_CONFIG +=
  imx8qxp:imx8qxp_mek_android_defconfig
# bootloader-imx8qm-dual.img and spl-imx8qm-dual.bin
```

```

TARGET_BOOTLOADER_CONFIG += imx8qm-
dual:imx8qm_mek_android_dual_defconfig
# bootloader-imx8qxp-dual.img and spl-imx8qxp-dual.bin
TARGET_BOOTLOADER_CONFIG += imx8qxp-
dual:imx8qxp_mek_android_dual_defconfig
# u-boot-imx8qm-hdmi.imx
TARGET_BOOTLOADER_CONFIG += imx8qm-
hdmi:imx8qm_mek_android_hdmi_defconfig
# u-boot-imx8qxp-c0.imx
TARGET_BOOTLOADER_CONFIG += imx8qxp-
c0:imx8qxp_mek_android_defconfig
# u-boot-imx8qm-md.imx
TARGET_BOOTLOADER_CONFIG += imx8qm-
md:imx8qm_mek_android_hdmi_defconfig
# bootloader-imx8qxp-c0-dual.img and spl-imx8qxp-c0-dual.bin
TARGET_BOOTLOADER_CONFIG += imx8qxp-c0-
dual:imx8qxp_mek_android_dual_defconfig
# u-boot-imx8dx.imx
TARGET_BOOTLOADER_CONFIG +=
imx8dx:imx8dx_mek_android_defconfig
# u-boot-imx8dx-mek-uuu.imx
TARGET_BOOTLOADER_CONFIG += imx8dx-mek-
uuu:imx8dx_mek_android_uuu_defconfig
# bootloader-imx8qm-trusty-dual.img and spl-imx8qm-trusty-
dual.bin
TARGET_BOOTLOADER_CONFIG += imx8qm-trusty-
dual:imx8qm_mek_android_trusty_dual_defconfig
# bootloader-imx8qxp-trusty-dual.img and spl-imx8qm-trusty-
dual.bin
TARGET_BOOTLOADER_CONFIG += imx8qxp-trusty-
dual:imx8qxp_mek_android_trusty_dual_defconfig
# bootloader-imx8qm-trusty-secure-unlock-dual.img and spl-
imx8qm-trusty-secure-unlock-dual.bin
TARGET_BOOTLOADER_CONFIG += imx8qm-trusty-secure-unlock-
dual:imx8qm_mek_android_trusty_secure_unlock_dual_defconfig
# bootloader-imx8qxp-trusty-c0-dual.img and spl-imx8qxp-
trusty-c0-dual.bin
TARGET_BOOTLOADER_CONFIG += imx8qxp-trusty-c0-
dual:imx8qxp_mek_android_trusty_dual_defconfig
# bootloader-imx8qxp-trusty-secure-unlock-dual.img and spl-
imx8qxp-trusty-secure-unlock-dual.bin
TARGET_BOOTLOADER_CONFIG += imx8qxp-trusty-secure-unlock-
dual:imx8qxp_mek_android_trusty_secure_unlock_dual_defconfig
# u-boot-imx8qm-mek-uuu.imx
TARGET_BOOTLOADER_CONFIG += imx8qm-mek-
uuu:imx8qm_mek_android_uuu_defconfig
# u-boot-imx8qxp-mek-uuu.imx
TARGET_BOOTLOADER_CONFIG += imx8qxp-mek-
uuu:imx8qxp_mek_android_uuu_defconfig

```

The defconfig file name can be used to locate the layout information of containers. Search the defconfig file name in `make_android.txt` just generated. A line prompts that the build process for that defconfig file has been finished. Container and container signature block offset can be found in several lines before this line. Search the log file for container layout information. For the condition that images to be signed are directly generated by `mkimage_imx8`, the container layout information of `u-boot-imx8qm-mek-uuu.imx` with the defconfig file of

imx8qm\_mek\_android\_uuu\_defconfig like below can be found in the log file. Unrelated lines are omitted and represented with ellipsis.

```
CST: CONTAINER 0 offset: 0x400
CST: CONTAINER 0: Signature Block: offset is at 0x590
...
===== Finish building imx8qm-mek-
uuu:imx8qm_mek_android_uuu_defconfig =====
```

container layout info of bootloader-imx8qm-trusty-dual.img and spl-imx8qm-trusty-dual.bin with the defconfig file of imx8qm\_mek\_android\_trusty\_dual\_defconfig like below can be found in the log file. The layout info of the first two lines is for bootloader-imx8qm-trusty-dual.img, the info in the following two lines are for spl-imx8qm-trusty-dual.bin.

```
CST: CONTAINER 0 offset: 0x0
CST: CONTAINER 0: Signature Block: offset is at 0x190
...
CST: CONTAINER 0 offset: 0x400
CST: CONTAINER 0: Signature Block: offset is at 0x610
...
===== Finish building
imx8qm:imx8qm_mek_android_trusty_dual_defconfig
=====
```

For the condition that images to be signed are not directly generated by mkimage\_imx8, take u-boot-imx8qm.imx with the defconfig file of imx8qm\_mek\_android\_defconfig as an example, below info should be retrieved in the log file. the first two lines are for the intermediate file u-boot-atf-container.img, This file is the third container which has Arm Trusted Firmware and u-boot proper in it. The layout info in the following two lines are for another intermediate file which has the second container in it, this intermediate file will be padded to 1KB aligned boundary, then u-boot-atf-container.img is appended to this file. So the second two lines can be directly used, while the first two lines should be added with an offset, it's 388KB, 0x61000 in hexadecimal, then the offset of the third container in the final u-boot-imx8qm.imx is 0x61000, its signature offset in the file is 0x61000+0x190=0x61190 .

```
CST: CONTAINER 0 offset: 0x0
CST: CONTAINER 0: Signature Block: offset is at 0x190
...
CST: CONTAINER 0 offset: 0x400
CST: CONTAINER 0: Signature Block: offset is at 0x610
...
append u-boot-atf-container.img at 388 KB
...
===== Finish building
imx8qm:imx8qm_mek_android_defconfig =====
```

A summary of the offsets of container and container signature are listed in the following table.

**Table 2. Container offset information**

Files having container to be signed	Container offset in the file	Container signature block offset
u-boot-imx8qm-mek-uuu.imx	0x400	0x590

**Table 2. Container offset information...continued**

Files having container to be signed	Container offset in the file	Container signature block offset
bootloader-imx8qm-trusty-dual.img	0x0	0x190
spl-imx8qm-trusty-dual.bin	0x400	0x610
u-boot-imx8qm.imx	0x400 0x61000	0x610 0x61190

The same rule can be used on other targets with one exception that the uuu u-boot target for i.MX 8ULP has SPL enabled, so target config `imx8ulp_evk_android_uuu_defconfig` should follow the same rule as `u-boot-imx8qm.imx`.

6. Sign the image files.

Copy the files to be signed to the directory of `release/linux64/bin/` in Code Signing Tool (CST) directory. The binary file named `cst` is used to sign these files. This CST needs the CSF description file to be provided as an input file when it is executed. CSF examples are in the directory of `/${MY_ANDROID}/vendor/nxp-opensource/u-boot-imx/doc/imx/ahab/csf_examples/`. We copy one `cst_uboot_atf.txt` to CST `release/linux64/bin/`.

Make some changes to the `cst_uboot_atf.txt` just copied based on the image to sign. For the condition that images to be signed are directly generated by `mkimage_imx8`, taking `u-boot-imx8qm-mek-uuu.imx` as an example, the modification is as follows:

```
@@ -4,9 +4,9 @@ Version = 1.0
[Install SRK]
# SRK table generated by srktool
-File = "../crts/SRK_1_2_3_4_table.bin"
+File = "../../crts/SRK_1_2_3_4_table.bin"
# Public key certificate in PEM format on this example only
  using SRK key
-Source = "../crts/SRK1_sha384_secp384r1_v3_usr_cert.pem"
+Source = "../../crts/SRK1_sha384_secp384r1_v3_usr_cert.pem"
# Index of the public key certificate within the SRK table
  (0 .. 3)
Source index = 0
# Type of SRK set (NXP or OEM)
@@ -16,6 +16,6 @@ Revocations = 0x0
[Authenticate Data]
# Binary to be signed generated by mkimage
-File = "u-boot-atf-container.img"
+File = "u-boot-imx8qm-mek-uuu.imx"
# Offsets = Container header Signature block (printed out by
  mkimage)
-Offsets = 0x0 0x110
+Offsets = 0x400 0x590
```

Then execute the command below:

```
$ ./cst -i cst_uboot_atf.txt -o signed-u-boot-imx8qm-mek-uuu.imx
```

With preceding command successfully executed, `signed-u-boot-imx8qm-mek-uuu.imx` is generated. Copy it back to the output directory, and change its name as

before, because `uuu_imx_android_flash` script flashes images based on their names.

Based on the description of signing `u-boot-imx8qm-mek-uuu.imx` to sign all other images directly generated by `mkimage_imx8` like `bootloader-imx8qm-trusty-dual.img`, `spl-imx8qm-trusty-dual.bin`, etc. Signing process of images not directly generated by `mkimage_imx8` is a bit different, it should be signed twice, take `u-boot-imx8qm.imx` as an example, firstly make below change to the original `cst_uboot_atf.txt` as below:

```
@@ -4,9 +4,9 @@ Version = 1.0
[Install SRK]
# SRK table generated by srktool
-File = "../crts/SRK_1_2_3_4_table.bin"
+File = "../../crts/SRK_1_2_3_4_table.bin"
# Public key certificate in PEM format on this example only
  using SRK key
-Source = "../crts/SRK1_sha384_secp384r1_v3_usr_cert.pem"
+Source = "../../crts/SRK1_sha384_secp384r1_v3_usr_cert.pem"
# Index of the public key certificate within the SRK table
  (0 .. 3)
Source index = 0
# Type of SRK set (NXP or OEM)
@@ -16,6 +16,6 @@ Revocations = 0x0
[Authenticate Data]
# Binary to be signed generated by mkimage
-File = "u-boot-atf-container.img"
+File = "u-boot-imx8qm.imx"
# Offsets = Container header Signature block (printed out by
  mkimage)
-Offsets = 0x0 0x110
+Offsets = 0x400 0x610
```

Then execute the command below:

```
$ ./cst -i cst_uboot_atf.txt -o first_signed_u-boot-
imx8qm.imx
Apply the following change on the "cst_uboot_atf.txt" as
  below to do the second sign
@@ -16,6 +16,6 @@ Revocations = 0x0
[Authenticate Data]
# Binary to be signed generated by mkimage
-File = "u-boot-imx8qm.imx"
+File = "first_signed_u-boot-imx8qm.imx"
# Offsets = Container header Signature block (printed out by
  mkimage)
-Offsets      = 0x400 0x610
+Offsets      = 0x61000 0x61190
```

And execute the command below:

```
$ ./cst -i cst_uboot_atf.txt -o second_signed_u-boot-
imx8qm.imx
```

With preceding command successfully executed, `second_signed_u-boot-imx8qm.imx` is generated, copy it back to the output directory, and change its name as before. All the images not directly generated by `mkimage_imx8` can be signed with preceding method, here in other words, SPL enabled single bootloader images can be signed with this method.

Images are signed now. When booting with signed images, SRK table embedded in image file is used to verify the signature. Embedded SRK table is verified based on



its hash value. The hash value is programmed in OTP efuse in i.MX chips, so it is not tempered by others. Perform the following steps to fuse the SRK hash value.

7. Dump the SRK hash value.

Change the directory to `release/crts/` in Code Signing Tool (CST). For i.MX 8Quad platforms, execute the following command to dump the SRK hash value:

```
$ od -t x4 SRK_1_2_3_4_fuse.bin
0000000 d436cc46 8eccda9 b89e1601 5fada3db
0000020 d454114a b6cd51f4 77384870 c50ee4b2
0000040 a27e5132 eba887cf 592c1e2b bb501799
0000060 ee702e07 cf8ce73e fb55e2d5 eba6bbd2
```

On i.MX 8ULP, the SRK Hash uses SHA256 and dump 8 words fuses:

```
$ od -t x4 SRK_1_2_3_4_fuse.bin
0000000 db2959f2 90dfc39c 53394566 e0b75829
0000020 85e6f3b1 af00983d e5e804fe 7a451024
```

8. Use the U-Boot fuse command to fuse the hash value to a chip.

Because the fuse command is removed from U-Boot for Android Auto images to shorten the boot time, use UUU to load the U-Boot used by UUU to RAM, and then use the fuse command.

Change the board to serial download mode, and execute the following command to download U-Boot to RAM. It then enters fastboot mode.

For i.MX 8QuadMax, it is as follows:

```
$ sudo ./uuu_imx_android_flash.sh -f imx8qm -i
```

For i.MX 8QuadXPlus, it is as follows:

```
$ sudo ./uuu_imx_android_flash.sh -f imx8qxp -i
```

For i.MX 8ULP, it is as follows:

```
$ sudo ./uuu_imx_android_flash.sh -f imx8ulp -i
```

With the commands above executed, U-Boot used by UUU under the current working directory is loaded to RAM on board and it enters fastboot mode.

On the U-Boot console, it shows that U-Boot is in fastboot mode. Press "CTRL+C" to exit fastboot mode and enter U-Boot command mode.

For i.MX 8QuadMax, execute the following commands on the U-Boot console:

```
=> fuse prog 0 722 0xd436cc46
=> fuse prog 0 723 0x8eccda9
=> fuse prog 0 724 0xb89e1601
=> fuse prog 0 725 0x5fada3db
=> fuse prog 0 726 0xd454114a
=> fuse prog 0 727 0xb6cd51f4
=> fuse prog 0 728 0x77384870
=> fuse prog 0 729 0xc50ee4b2
=> fuse prog 0 730 0xa27e5132
=> fuse prog 0 731 0xeba887cf
=> fuse prog 0 732 0x592c1e2b
=> fuse prog 0 733 0xbb501799
=> fuse prog 0 734 0xee702e07
=> fuse prog 0 735 0xcf8ce73e
=> fuse prog 0 736 0xfb55e2d5
=> fuse prog 0 737 0xeba6bbd2
```

For i.MX 8QuadXPlus, execute the following commands on the U-Boot console:

```
=> fuse prog 0 730 0xd436cc46
```

```

=> fuse prog 0 731 0x8eccdda9
=> fuse prog 0 732 0xb89e1601
=> fuse prog 0 733 0x5fada3db
=> fuse prog 0 734 0xd454114a
=> fuse prog 0 735 0xb6cd51f4
=> fuse prog 0 736 0x77384870
=> fuse prog 0 737 0xc50ee4b2
=> fuse prog 0 738 0xa27e5132
=> fuse prog 0 739 0xeba887cf
=> fuse prog 0 740 0x592c1e2b
=> fuse prog 0 741 0xbb501799
=> fuse prog 0 742 0xee702e07
=> fuse prog 0 743 0xcf8ce73e
=> fuse prog 0 744 0xfb55e2d5
=> fuse prog 0 745 0xeba6bbd2

```

For i.MX 8ULP, execute below commands on U-Boot console:

```

=> fuse prog 15 0 0xdb2959f2
=> fuse prog 15 1 0x90dfc39c
=> fuse prog 15 2 0x53394566
=> fuse prog 15 3 0xe0b75829
=> fuse prog 15 4 0x85e6f3b1
=> fuse prog 15 5 0xaf00983d
=> fuse prog 15 6 0xe5e804fe
=> fuse prog 15 7 0x7a451024

```

Now, images are signed and SRK hash value is fused. The images can be flashed to boards. For how to flash i.MX Android images, see the *Android™ Release Notes (ARN)*.

The chip is now in open stage, and verification failure does not block the boot process. To make sure that SRK hash value is correctly fused and images are correctly signed, check the SECO event during boot. After `CONFIG_AHAB_BOOT` is enabled in the defconfig file of U-Boot, use a U-Boot command to check the SECO events. After images are signed and SRK hash value is programmed, boot the board to U-Boot command mode. On the U-Boot console, execute the following command:

```

=> ahab_status

```

If preceding command outputs the SECO event, use the following code to check whether it is related to AHAB verification.

```

0x0087EE00 = The container image is not signed.
0x0087FA00 = The container image was signed with wrong key
              that is not matching the OTP SRK hashes.

```

For example, if the SRK hash value is programmed, but images are not signed, after `ahab_status` is executed, the following prompt is displayed on the console:

```

=> ahab_status
Lifecycle: 0x0020, NXP closed
SECO Event[0] = 0x0087EE00
CMD = AHAB_AUTH_CONTAINER_REQ (0x87)
IND = AHAB_NO_AUTHENTICATION_IND (0xEE)
SECO Event[1] = 0x0087EE00
CMD = AHAB_AUTH_CONTAINER_REQ (0x87)
IND = AHAB_NO_AUTHENTICATION_IND (0xEE)

```

After it is confirmed that the SRK hash value is correctly fused, and signed images do not cause AHAB-related SECO events, execute the following command on the U-Boot console to close the chip:

```
=> ahab_close
```

Note that this close operation is **irreversible** to the chips and the closed chips does not boot up if AHAB verification fails.

### 3.1.2 Verifying images with HABv4

HABv4 verifies images based on Image Vector Table (IVT) and Flattened Image Tree (FIT). Detailed information of the boot image format can be found in the Reference Manual of specific chips. According to the Reference manual, the sign process described below embeds the Command Sequence File (CSF) generated by code signing tool in the final image. The CSF is used to verify the images during the boot time.

To sign the images for i.MX 8M devices, perform the following steps.

#### 3.1.2.1 Get CST tool and keys configuration

1. Download Code Signing Tool (CST) from [NXP official website](#).

Decompress the package with the following command:

```
$ tar zxvf cst-3.1.0.tgz
```

2. Generate the HABv4 PKI tree.

After the tool package is decompressed, enter the directory of `release/keys/`, and execute the following command:

```
./hab4_pki_tree.sh
```

Then enter some parameters based on the output of this script. An example is as follows:

```
Do you want to use an existing CA key (y/n)? : n
Do you want to use Elliptic Curve Cryptography (y/n)? : n
Enter key length in bits for PKI tree: 2048
Enter PKI tree duration (years): 5
How many Super Root Keys should be generated? 4
Do you want the SRK certificates to have the CA flag set? (y/n)? : y
```

3. Generate AHAB SRK tables and eFuse hash.

Enter the directory of `release/crts/`, and execute the following command:

```
$ ../linux64/bin/srktool -h 4 -t SRK_1_2_3_4_table.bin -e \
SRK_1_2_3_4_fuse.bin -d sha256 -c \
SRK1_sha256_2048_65537_v3_ca.crt.pem, \
SRK2_sha256_2048_65537_v3_ca.crt.pem, \
SRK3_sha256_2048_65537_v3_ca.crt.pem, \
SRK4_sha256_2048_65537_v3_ca.crt.pem
```

After the preceding commands are executed successfully. The SRK table and its SHA256 value are generated and saved respectively in two files under `release/crts/`.

The SRK table is embedded in the CSF. Therefore, during the boot time, it can be used to verify the signature. The SRK table SHA256 value is fused to the OTP eFuse hardware and works as the "secure root key". It is used to verify the SRK table in CSF.

Files generated in `release/keys/` and `/release/crts/` are very important. If the SRK HASH value is fused to the chip and then changes the chip from open to close state, the board can only boot with images signed with these files.

If you are a user of both i.MX 8Quad and i.MX 8M devices, use two copies of this CST respectively for two device families, since this process generates files with the same names in the same directory, while these files should be prevented from being overwritten.

4. Dump the SRK hash value.

Change directory to `release/crts/` in Code Signing Tool (CST). Execute the following command to dump the SRK hash value:

```
hexdump -e '/4 "0x"' -e '/4 "%X"\n"' SRK_1_2_3_4_fuse.bin
0x20593752
0x6ACE6962
0x26E0D06C
0xFC600661
0x1240E88F
0x1209F144
0x831C8117
0x1190FD4D
```

5. Use the U-Boot fuse command to fuse the hash value to a chip.

Flash the image just signed to the board, and then boot to U-Boot command mode, and execute the following command to fuse the SRK hash value. This is the same for i.MX 8M Mini, i.MX 8M Nano, i.MX 8M Quad, and i.MX 8M Plus.

```
fuse prog -y 6 0 0x20593752 0x6ACE6962 0x26E0D06C 0xFC600661
fuse prog -y 7 0 0x1240E88F 0x1209F144 0x831C8117 0x1190FD4D
```

### 3.1.2.2 Sign bootloader images

1. Build Android images to generate the file to be signed.

To support HAB features, enable `CONFIG_IMX_HAB` configurations in the corresponding defconfig files in U-Boot code. They are not enabled by default. Taking i.MX 8M Mini EVK and i.MX 8M Nano EVK as an example, the files are:

```
imx8mm_evk_android_defconfig
imx8mm_evk_android_dual_defconfig
imx8mm_evk_android_trusty_secure_unlock_dual_defconfig
imx8mm_evk_android_trusty_dual_defconfig
imx8mm_evk_android_uuu_defconfig
```

Layout information of the final U-Boot image is needed during the signing process. When building the Android images, save the log information of build system. For example, execute the following command:

```
$ ./imx-make.sh -j12 2>&1 | tee make_android.txt
```

During the build process, the build system output information is also saved in `make_android.txt`.

2. Get the layout information of the file to be signed.

Final U-Boot image files are generated with different U-Boot defconfig files. Taking i.MX 8M Mini as an example, different U-Boot compilation targets with different U-Boot defconfig files can be found in `_${MY_ANDROID}/device/nxp/imx8m/evk_8mm/BoardConfig.mk`. They are listed below with some annotations.

```
TARGET_BOOTLOADER_CONFIG :=
imx8mm:imx8mm_evk_android_defconfig
```

```
TARGET_BOOTLOADER_CONFIG += imx8mm-
dual:imx8mm_evk_android_dual_defconfig
TARGET_BOOTLOADER_CONFIG += imx8mm-trusty-secure-unlock-
dual:imx8mm_evk_android_trusty_secure_unlock_dual_defconfig
TARGET_BOOTLOADER_CONFIG += imx8mm-trusty-
dual:imx8mm_evk_android_trusty_dual_defconfig
TARGET_BOOTLOADER_CONFIG += imx8mm-evk-
uuu:imx8mm_evk_android_uuu_defconfig
```

Search the defconfig file name in the log of `make_android.txt` just generated for a line prompt that the build process for that defconfig file is finished. The layout information needed in the signing process is several lines before that. The layout information needs to be categorized based on whether dual-bootloader is enabled. The defconfig files with "dual" substring in their names have dual-bootloader enabled. On the contrary, the ones without "dual" substring in their names do not enable dual-bootloader.

For the single bootloader condition, taking `imx8mm_evk_android_uuu_defconfig` as an example, its corresponding output file is `u-boot-imx8mm-evk-uuu.imx`. The lines directly related to the layout info of this output file are listed below, and unrelated lines are omitted and represented with ellipsis.

```
Loader IMAGE:
...
csf_off                0x2d600
spl_hab_block:        0x7e0fc0 0x0 0x2d600
...
Second Loader IMAGE:
...
sld_csf_off            0x58c20
sld_hab_block:        0x401fcdc0 0x57c00 0x1020
...
0x40200000 0x5AC00 0xAE710
0x402AE710 0x109310 0x8630
0x920000 0x111940 0x91D0
...
===== Finish building imx8mm-evk-
uuu:imx8mm_evk_android_uuu_defconfig =====
```

For the dual-bootloader condition, taking `imx8mm_evk_android_trusty_dual_defconfig` as an example, its corresponding output files are `spl-imx8mm-trusty-dual.bin` and `bootloader-imx8mm-trusty-dual.img`. The lines directly related to the layout info of this output file are listed below, and unrelated lines are omitted and represented with ellipsis.

```
Loader IMAGE:
...
csf_off                0x2d600
spl_hab_block:        0x7e0fc0 0x0 0x2d600
...
Second Loader IMAGE:
...
sld_csf_off            0x58c20
sld_hab_block:        0x401fcdc0 0x57c00 0x1020
...
0x40200000 0x5AC00 0xAE710
0x402AE710 0x109310 0x8630
0x920000 0x111940 0x91D0
...
=====
```

```
===== Finish building imx8mm-evk-
uuu:imx8mm_evk_android_uuu_defconfig =====
```

The same method of retrieving the layout information can be used on other targets.

### 3. Sign the image files.

Copy the files to be signed to the directory of `release/linux64/bin/` in Code Signing Tool (CST) directory. The binary file named CST is used to sign these files. This CST needs CSF description file to be as an input file when it is executed. CSF examples are in the directory of `${MY_ANDROID}/vendor/nxp-opensource/uboot-imx/doc/imx/habv4/csf_examples/mx8m_mx8mm/`. Copy `csf_fit.txt` and `csf_spl.txt` to Code Signing Tool (CST) `release/linux64/bin/`. The file to be signed also needs to be copied to this directory. The signing process is different between single-bootloader condition and dual-bootloader condition. The examples of the two conditions are described together in this section, but in practice, different U-Boot target files should be signed with the following steps one by one.

For the single bootloader example with `imx8mm_evk_android_uuu_defconfig` `defconfig` file, modify the copied `csf_fit.txt` and `csf_spl.txt` as follows.

```
diff --git a/csf_fit.txt b/csf_fit.txt
index d9218ab..dfd0ded 100644
--- a/csf_fit.txt
+++ b/csf_fit.txt
@@ -8,12 +8,12 @@
 [Install SRK]
 # Index of the key location in the SRK table to be installed
 -   File = "../crts/SRK_1_2_3_4_table.bin"
 +   File = "../../crts/SRK_1_2_3_4_table.bin"
 Source index = 0
 [Install CSFK]
 # Key used to authenticate the CSF data
 -   File = "../crts/CSF1_1_sha256_2048_65537_v3_usr.crt.pem"
 +   File = "../../crts/
CSF1_1_sha256_2048_65537_v3_usr.crt.pem"
 [Authenticate CSF]
@@ -23,14 +23,14 @@
 # Target key slot in HAB key store where key will be
 installed
 Target index = 2
 # Key to install
 -   File = "../crts/IMG1_1_sha256_2048_65537_v3_usr.crt.pem"
 +   File = "../../crts/
IMG1_1_sha256_2048_65537_v3_usr.crt.pem"
 [Authenticate Data]
 # Key slot index used to authenticate the image data
 Verification index = 2
 # Authenticate Start Address, Offset, Length and file
 -   Blocks = 0x401fcdc0 0x057c00 0x01020 "flash.bin", \
 -           0x40200000 0x05AC00 0x9AAC8 "flash.bin", \
 -           0x00910000 0x0F56C8 0x09139 "flash.bin", \
 -           0xFE000000 0xFE804 0x4D268 "flash.bin", \
 -           0x4029AAC8 0x14BA6C 0x06DCF "flash.bin"
 +   Blocks = 0x401fcdc0 0x57c00 0x1020 "u-boot-imx8mm-evk-
uuu.imx", \
 +           0x40200000 0x5AC00 0xAE710 "u-boot-imx8mm-evk-
uuu.imx", \
 +           0x402AE710 0x109310 0x8630 "u-boot-imx8mm-evk-
uuu.imx", \
```

```

+          0x920000 0x111940 0x91D0 "u-boot-imx8mm-evk-
uuu.imx"
diff --git a/csf_spl.txt b/csf_spl.txt
index 39adf7a..80165a8 100644
--- a/csf_spl.txt
+++ b/csf_spl.txt
@@ -8,12 +8,12 @@
[Install SRK]
# Index of the key location in the SRK table to be installed
-   File = "../crts/SRK_1_2_3_4_table.bin"
+   File = "../../crts/SRK_1_2_3_4_table.bin"
Source index = 0
[Install CSFK]
# Key used to authenticate the CSF data
-   File = "../crts/CSF1_1_sha256_2048_65537_v3_usr.crt.pem"
+   File = "../../crts/
CSF1_1_sha256_2048_65537_v3_usr.crt.pem"
[Authenticate CSF]
[Unlock]
# Leave Job Ring and DECO master ID registers Unlocked
Engine = CAAM
-   Features = MID
+   Features = MID, MFG
[Install Key]
# Key slot index used to authenticate the key to be installed
@@ -28,10 +28,10 @@
# Target key slot in HAB key store where key will be
installed
Target index = 2
# Key to install
-   File = "../crts/IMG1_1_sha256_2048_65537_v3_usr.crt.pem"
+   File = "../../crts/
IMG1_1_sha256_2048_65537_v3_usr.crt.pem"
[Authenticate Data]
# Key slot index used to authenticate the image data
Verification index = 2
# Authenticate Start Address, Offset, Length and file
-   Blocks = 0x7e0fc0 0x1a000 0x2a600 "flash.bin"
+   Blocks = 0x7e0fc0 0x0 0x2d600 "u-boot-imx8mm-evk-
uuu.imx"

```

For the dual-bootloader example with `imx8mm_evk_android_trusty_dual_defconfig` `defconfig` file, modify the copied `csf_fit.txt` and `csf_spl.txt` as follows.

```

diff --git a/csf_fit.txt b/csf_fit.txt
index d9218ab..dfd0ded 100644
--- a/csf_fit.txt
+++ b/csf_fit.txt
@@ -8,12 +8,12 @@
[Install SRK]
# Index of the key location in the SRK table to be installed
-   File = "../crts/SRK_1_2_3_4_table.bin"
+   File = "../../crts/SRK_1_2_3_4_table.bin"
Source index = 0
[Install CSFK]
# Key used to authenticate the CSF data
-   File = "../crts/CSF1_1_sha256_2048_65537_v3_usr.crt.pem"
+   File = "../../crts/
CSF1_1_sha256_2048_65537_v3_usr.crt.pem"
[Authenticate CSF]

```

```

@@ -23,14 +23,14 @@
# Target key slot in HAB key store where key will be
  installed
Target index = 2
# Key to install
-   File = "../crts/IMG1_1_sha256_2048_65537_v3_usr crt.pem"
+   File = "../../crts/
IMG1_1_sha256_2048_65537_v3_usr crt.pem"
[Authenticate Data]
# Key slot index used to authenticate the image data
Verification index = 2
# Authenticate Start Address, Offset, Length and file
-   Blocks = 0x401fcdc0 0x057c00 0x01020 "flash.bin", \
-           0x40200000 0x05AC00 0x9AAC8 "flash.bin", \
-           0x00910000 0x0F56C8 0x09139 "flash.bin", \
-           0xFE000000 0xFE804 0x4D268 "flash.bin", \
-           0x4029AAC8 0x14BA6C 0x06DCF "flash.bin"
+   Blocks = 0x401fcdc0 0x0 0x1020 "bootloader-imx8mm-
trusty-dual.img", \
+           0x40200000 0x3000 0xAF2D0 "bootloader-imx8mm-
trusty-dual.img", \
+           0x402AF2D0 0xB22D0 0x8630 "bootloader-imx8mm-
trusty-dual.img", \
+           0x920000 0xBA900 0xA1D0 "bootloader-imx8mm-
trusty-dual.img", \
+           0xBE000000 0xC4AD0 0x164C50 "bootloader-imx8mm-
trusty-dual.img"
diff --git a/csf_spl.txt b/csf_spl.txt
index 39adf7a..80165a8 100644
--- a/csf_spl.txt
+++ b/csf_spl.txt
@@ -8,12 +8,12 @@
[Install SRK]
# Index of the key location in the SRK table to be installed
-   File = "../crts/SRK_1_2_3_4_table.bin"
+   File = "../../crts/SRK_1_2_3_4_table.bin"
Source index = 0
[Install CSFK]
# Key used to authenticate the CSF data
-   File = "../crts/CSF1_1_sha256_2048_65537_v3_usr crt.pem"
+   File = "../../crts/
CSF1_1_sha256_2048_65537_v3_usr crt.pem"
[Authenticate CSF]
[Unlock]
# Leave Job Ring and DECO master ID registers Unlocked
Engine = CAAM
-   Features = MID
+   Features = MID, MFG
[Install Key]
# Key slot index used to authenticate the key to be installed
@@ -28,10 +28,10 @@
# Target key slot in HAB key store where key will be
  installed
Target index = 2
# Key to install
-   File = "../crts/IMG1_1_sha256_2048_65537_v3_usr crt.pem"
+   File = "../../crts/
IMG1_1_sha256_2048_65537_v3_usr crt.pem"
[Authenticate Data]
# Key slot index used to authenticate the image data

```



```
Verification index = 2
# Authenticate Start Address, Offset, Length and file
-   Blocks = 0x7e0fc0 0x1a000 0x2a600 "flash.bin"
+   Blocks = 0x7e0fc0 0x0 0x34200 "spl-imx8mm-trusty-
dual.bin"
```

Execute the following commands to generate CSF with the CSF description file. They are the same for single bootloader condition and dual-bootloader condition.

```
$ ./cst --i=csf_spl.txt --o=csf_spl.bin
$ ./cst --i=csf_fit.txt --o=csf_fit.bin
```

Execute the following commands to embed the CSF into the U-Boot image.

```
# For the single bootloader example with
"imx8mm_evk_android_uuu_defconfig" defconfig file
$ dd if=csf_spl.bin of=u-boot-imx8mm-evk-uuu.imx seek=
$((0x2d600)) bs=1 conv=notrunc
$ dd if=csf_fit.bin of=u-boot-imx8mm-evk-uuu.imx seek=
$((0x58c20)) bs=1 conv=notrunc
# For the dual bootloader example with
"imx8mm_evk_android_trusty_dual_defconfig" defconfig file
$ dd if=csf_spl.bin of=spl-imx8mm-trusty-dual.bin seek=
$((0x34200)) bs=1 conv=notrunc
$ dd if=csf_fit.bin of=bootloader-imx8mm-trusty-dual.img
seek=$((0x1020)) bs=1 conv=notrunc
```

With preceding commands successfully executed, `u-boot-imx8mm-evk-uuu.imx` or `spl-imx8mm-trusty-dual.bin` and `bootloader-imx8mm-trusty-dual.img` are well signed. Then sign other images according to the description of the signing process.

Images are signed now. When booting with signed images, the SRK table embedded in the image file is used to verify the signature. Embedded SRK table is verified based on its hash value. The hash value is programed in OTP eFuse in i.MX chips, so it is not affected by others.

### 3.1.2.3 Sign the MCU firmware

The MCU firmware on i.MX 8M is kicked off by the Cortex-A core. Its signature would be verified by HABv4. Some scripts are provided below to auto-sign the MCU firmware. Save these scripts to `linux64/bin/`.

#### 1. Create a CSF template.

Save the following CSF template and rename it as `mcucsf_template`. This is the CSF file of the MCU firmware.

```
[Header]
  Version = 4.3
  Hash Algorithm = sha256
  Engine Configuration = 0
  Certificate Format = X509
  Signature Format = CMS
[Install SRK]
  File = "../crt/srk/SRK_1_2_3_4_table.bin"
  Source index = 0
[Install CSFK]
  File = "../crt/
CSF1_1_sha256_2048_65537_v3_usr_cert.pem"
[Authenticate CSF]
[Install Key]
  Verification index = 0
```

```

Target index = 2
File = "../../../crts/
IMG1_1_sha256_2048_65537_v3_usr.crt.pem"
[Authenticate Data]
Verification index = 2
Blocks = %addr_addr% 0x0 %auth_len% "mcu-pad-ivt.bin"

```

## 2. Create an IVT generation template.

Save the following IVT template and rename it as `genivt_template`. This script helps generate the IVT.

```

#!/usr/bin/perl -w
use strict;
open(my $out, '>:raw', 'ivt.bin') or die "Unable to open:
$!";
print $out pack("V", 0x402000D1); # Signature
print $out pack("V", %jump_addr%); # Jump Location
print $out pack("V", 0x0); # Reserved
print $out pack("V", 0x0); # DCD pointer
print $out pack("V", 0x0); # Boot Data
print $out pack("V", %self_ptr%); # Self Pointer
print $out pack("V", %csf_ptr%); # CSF Pointer
print $out pack("V", 0x0); # Reserved
close($out);

```

## 3. Create a script `habmcugen_template`.

Save the following template and rename it as `habmcugen_template`. This script helps combine the generated images.

```

#!/bin/bash
echo "extend mcu image to %pad_len%..."
objcopy -I binary -O binary --pad-to %pad_len% --gap-
fill=0x5A mcu.bin mcu-pad.bin
echo "generate IVT"
./genIVT
echo "attach IVT..."
cat mcu-pad.bin ivt.bin > mcu-pad-ivt.bin
echo "generate csf data..."
./cst --o mcu_csf.bin --i mcu.csf
echo "merge image and csf data..."
cat mcu-pad-ivt.bin mcu_csf.bin > mcu-signed.bin
echo "extend final image to %sig_len%..."
objcopy -I binary -O binary --pad-to %sig_len% --gap-
fill=0x5A mcu-signed.bin mcu-signed-pad.bin

```

## 4. Create script `mk_secure_mcu_imx8m`.

Save the following script and rename it as `mk_secure_mcu_imx8m`. This script invokes all the scripts above and generates the final signed MCU image.

```

#!/bin/bash
let ddr_addr=$1
if [ ! -f mcu.bin ]
then
    printf "File \"mcu.bin\" does not exist.\n"
    exit 1
fi
# Calculate the size
let pad_len=$((0x20000)) # padding mcu image to 128KB
let auth_len=$((pad_len + 0x20)) # +0x20 "IVT"
let sig_len=$((auth_len + 0x2000))
let self_ptr=$((ddr_addr + pad_len))

```

```

let csf_ptr=$((ddr_addr + auth_len))
let jump_addr=$((ddr_addr))
# change value to hex string
pad_len=`printf "0x%X" ${pad_len}`
auth_len=`printf "0x%X" ${auth_len}`
sig_len=`printf "0x%X" ${sig_len}`
ddr_addr=`printf "0x%X" ${ddr_addr}`
self_ptr=`printf "0x%X" ${self_ptr}`
csf_ptr=`printf "0x%X" ${csf_ptr}`
jump_addr=`printf "0x%X" ${jump_addr}`
# Create habUimagegen
sed -e s/%pad_len%/${pad_len}/g -e s/%sig_len%/${sig_len}/g
  habmcugen_template > habmcuGen
chmod +x habmcuGen
# Create mcu.csf
sed -e s/%ddr_addr%/${ddr_addr}/g -e s/%auth_len%/
  ${auth_len}/g mcucsf_template > mcu.csf
# Create genIVT
sed -e s/%jump_addr%/${jump_addr}/g -e s/%self_ptr%/
  ${self_ptr}/g -e s/%csf_ptr%/${csf_ptr}/g genivt_template >
  genIVT
chmod +x genIVT
# Generate secure boot
./habmcuGen
# OK
printf "mcu image with Signature \"mcu-signed-pad.bin\" is
  ready to use.\n"

```

#### 5. Generate the signed MCU firmware.

Copy the unsigned MCU firmware to `linux64/bin` and rename it as `mcu.bin`. Execute the following command to generate the final signed MCU firmware `mcu-signed-pad.bin`:

```
./mk_secure_mcu_imx8m <mcu-firmware-load-address>
```

As only TCM is supported to run the MCU firmware on the Android platform, so the `mcu-firmware-load-address` should be the load address in TCM like `0x7e0000`.

#### 3.1.2.4 Close the chip

With above steps, the images are signed and SRK hash value is fused. But as the chip is in open stage, verification failure does not block the boot process. One way to make sure that SRK hash value is correctly fused and images are correctly signed is to check HAB events in the U-Boot log. Flash the images which are signed and reboot to U-Boot console and execute the following command:

```
=> hab_status
```

Make sure there is no HAB EVENT reported. Before SPL invokes HAB to verify the FIT part, it first validates the CSF. Make sure there is no CSF-related error as follows in the boot log.

```
Error: CSF header command not found
```

After it is confirmed that the SRK hash value is correctly fused, and signed images do not cause HAB event, execute the following command on the U-Boot console to close the chip.

```
=> fuse prog -y 1 3 0x2000000
```

This close operation is **irreversible** to the chips and closed chips do not boot up if HABv4 verification fails.

## 3.2 Configurations on TEE

### 3.2.1 Memory region configuration in ATF

The TEE binary is loaded to DRAM at the address of `$BBL32_BASE` by SPL. By default, the load address `$BBL32_BASE` is defined as `0xFE000000`. It is specified during the process of generating the bootloader image with `imx-mkimage`. For example, you can specify the load address as `0xFF000000` for i.MX 8QuadMax and i.MX 8QuadXPlus in `_${MY_ANDROID}/vendor/nxp-opensource/imx-mkimage` as follows:

```
diff --git a/iMX8QM/soc.mak b/iMX8QM/soc.mak
index 355851e..fe70191 100644
--- a/iMX8QM/soc.mak
+++ b/iMX8QM/soc.mak
@@ -82,7 +82,7 @@ u-boot-atf-container.img: bl31.bin u-boot-
hash.bin
fi
if [ -f "tee.bin" ]; then \
    if [ $(shell echo $(ROLLBACK_INDEX_IN_CONTAINER)) ]; then \
        - ./$(MKIMG) -soc QM -sw_version
$(ROLLBACK_INDEX_IN_CONTAINER) -rev B0 -c -ap bl31.bin a53
0x80000000 -ap u-boot-hash.bin a53 0x80020000 -ap tee.bin a53
0xFE000000 -out u-boot-atf-container.img; \
        + ./$(MKIMG) -soc QM -sw_version
$(ROLLBACK_INDEX_IN_CONTAINER) -rev B0 -c -ap bl31.bin a53
0x80000000 -ap u-boot-hash.bin a53 0x80020000 -ap tee.bin a53
0xFF000000 -out u-boot-atf-container.img; \
    else \
        ./$(MKIMG) -soc QM -rev B0 -c -ap bl31.bin a53
0x80000000 -ap u-boot-hash.bin a53 0x80020000 -ap tee.bin a53
0xFE000000 -out u-boot-atf-container.img; \
    fi; \
diff --git a/iMX8QX/soc.mak b/iMX8QX/soc.mak
index 56422e0..d917dc3 100644
--- a/iMX8QX/soc.mak
+++ b/iMX8QX/soc.mak
@@ -73,7 +73,7 @@ u-boot-atf.itb: u-boot-hash.bin bl31.bin
u-boot-atf-container.img: bl31.bin u-boot-hash.bin
if [ -f tee.bin ]; then \
    if [ $(shell echo $(ROLLBACK_INDEX_IN_CONTAINER)) ]; then \
        - ./$(MKIMG) -soc QX -sw_version
$(ROLLBACK_INDEX_IN_CONTAINER) -rev B0 -c -ap bl31.bin a35
0x80000000 -ap u-boot-hash.bin a35 0x80020000 -ap tee.bin a35
0xFE000000 -out u-boot-atf-container.img; \
        + ./$(MKIMG) -soc QX -sw_version
$(ROLLBACK_INDEX_IN_CONTAINER) -rev B0 -c -ap bl31.bin a35
0x80000000 -ap u-boot-hash.bin a35 0x80020000 -ap tee.bin a35
0xFF000000 -out u-boot-atf-container.img; \
```

```

else \
    ./$(MKIMG) -soc QX -rev B0 -c -ap bl31.bin a35
0x80000000 -ap u-boot-hash.bin a35 0x80020000 -ap tee.bin a35
0xFE000000 -out u-boot-atf-container.img; \
fi; \

```

After loading the TEE binary to DRAM, the ATF tries to kick it at the address of \$BL32\_BASE with the size of \$BL32\_SIZE, which are defined in \${MY\_ANDROID}/vendor/nxp-opensource/arm-trusted-firmware/plat/imx/\${PLAT}/include/platform\_def.h. By default, \$BL32\_BASE is defined as 0xFE000000 and \$BL32\_SIZE is 0x02000000, but you can configure them as needed. For example, \$BL32\_BASE can be configured as 0xFF000000 and \$BL32\_SIZE can be configured as 0x03000000 for i.MX 8QuadMax and i.MX 8QuadXPlus as follows:

```

diff --git a/plat/imx/imx8qm/include/platform_def.h b/plat/imx/
imx8qm/include/platform_def.h
index b305bfc..6f9f7d4 100644
--- a/plat/imx/imx8qm/include/platform_def.h
+++ b/plat/imx/imx8qm/include/platform_def.h
@@ -37,8 +37,8 @@
#define BL31_LIMIT 0x80020000
#ifdef TEE_IMX8
-#define BL32_BASE 0xfe000000
-#define BL32_SIZE 0x02000000
+#define BL32_BASE 0xff000000
+#define BL32_SIZE 0x03000000
#define BL32_SHM_SIZE 0x00400000
#define BL32_LIMIT 0x100000000
#endif
diff --git a/plat/imx/imx8qxp/include/platform_def.h b/plat/
imx/imx8qxp/include/platform_def.h
index 24eacc2..cfc0717 100644
--- a/plat/imx/imx8qxp/include/platform_def.h
+++ b/plat/imx/imx8qxp/include/platform_def.h
@@ -33,8 +33,8 @@
#define BL31_LIMIT 0x80020000
#ifdef TEE_IMX8
-#define BL32_BASE 0xfe000000
-#define BL32_SIZE 0x02000000
+#define BL32_BASE 0xff000000
+#define BL32_SIZE 0x03000000
#define BL32_SHM_SIZE 0x00400000
#define BL32_LIMIT 0x100000000
#define PLAT_TEE_IMAGE_OFFSET 0x84000000

```

The following table lists the recommended \$BL32\_BASE and \$BL32\_SIZE for DRAM with different sizes on the i.MX 8Quad platform:

**Table 3. Recommended \$BL32\_BASE and \$BL32\_SIZE for DRAM**

DRAM Size (GB)	\$BL32_BASE	\$BL32_SIZE
6	0xFE000000	0x02000000
4	0xFE000000	0x02000000
3	0xFE000000	0x02000000
2	0xFE000000	0x02000000
1	0xBE000000	0x02000000

### 3.2.2 Basic file and folder construction for Trusty OS

i.MX Trusty OS provides a security solution for Android platform and Android Automotive platform. It also provides a set of development APIs for customer to develop their own TAs.

Trusty OS is based on LittleKernel. i.MX Trusty OS has the following basic file structure.

**Table 4. Basic file structure of i.MX Trusty OS**

Folder name	Folder description
trusty/device/nxp/imx8	This folder contains the script files. Most of the configurations for the build target are defined in this folder, including project configuration files. The Makefile configurations, board configurations, and modules need to be built.
trusty/hardware/nxp/app	NXP-specific TA source code folder. Currently the hwcrypto TA located in this folder that provides security functions depends on the i.MX SoC hardware.
trusty/hardware/nxp/target	NXP reference board target folder. Only <code>rules.mk</code> for the build target in this folder, platform name, and UART information are defined in this file.
trusty/hardware/nxp/platform/imx	NXP SoC specific source codes for Trusty OS. All i.MX SoCs share these codes. It includes platform initialization codes, UART drivers, and registers map definitions.
trusty/kernel/lib	Trusty OS core codes including secure monitor calls management, TIPC/QL-TIPC stack.
external/lk	LittleKernel codes, including all LittleKernel modules like arch codes, interrupt management, task management, and SMP support.
trusty/user/app	Trusty OS TAs are placed here, including AVB, Gatekeeper, and Keymaster user space source codes.

For TAs implementation, see Google Trusty OS reference webpage: <https://source.android.com/security/trusty/trusty-ref>.

### 3.2.3 Applying new build target in Trusty OS

By default, NXP already provides i.MX 8QuadMax/8QuadXPlus and i.MX 8M Mini/8M Quad series template in the i.MX Trusty OS. To add a new platform based on i.MX 8QuadMax/8QuadXPlus or i.MX 8M Mini/8M Quad/8M Plus, add or modify the following file or modules.

In `_${MY_TRUSTY}/trusty/device/nxp/imx8/project`, `imx8-inc.mk` contains all common configurations, such as CPU cores, modules that need to be built. The `imx8-inc.mk` can be overwritten by the build target mk files, such as `imx8qm.mk`.

For example, to add a new build target based on i.MX 8QuadMax SoC called `imx8qm-abc`, which has six CPUs and 1024 RPMB blocks, write a new `.mk` file called `imx8qm-abc.mk` in `_${MY_TRUSTY}/trusty/device/nxp/imx8/project`. The content is as follows:

```
TARGET := imx8q
# imx8q/x use lpuart for UART IP
IMX_USE_LPUART := true
```

```
SMP_MAX_CPUS := 6
STORAGE_RPMB_BLOCK_COUNT := 1024
include project/imx8-inc.mk
```

In the root of Trusty OS codes, execute `$make list`. Then `imx8qm-abc` is displayed.

### 3.2.4 Adding unit tests in Trusty OS and adding CAAM self-tests in Trusty OS

Trusty OS supports two unit tests to test the functionality of Trusty IPC (TIPC) and CAAM. It is only for debug purpose and should not be released with the open unit tests. For i.MX 8QuadMax and i.MX 8QuadXPlus, to include these unit tests, make the following changes in `${MY_TRUSTY}/trusty/device/nxp/imx8/project`:

```
diff --git a/project/imx8-inc.mk b/project/imx8-inc.mk
index 681a223..e7dcfdb 100644
--- a/project/imx8-inc.mk
+++ b/project/imx8-inc.mk
@@ -70,6 +70,7 @@ GLOBAL_DEFINES += APP_STORAGE_RPMB_BLOCK_COUNT=
$(STORAGE_RPMB_BLOCK_COUNT)
GLOBAL_DEFINES += \
WITH_LIB_VERSION=1 \
+ WITH_CAAM_SELF_TEST=1 \
# ARM suggest to use system registers to access GICv3/v4 registers
GLOBAL_DEFINES += ARM_GIC_USE_SYSTEM_REG=1
@@ -98,6 +99,8 @@ TRUSTY_ALL_USER_TASKS := \
trusty/user/app/keymaster \
trusty/user/app/gatekeeper \
trusty/user/app/storage \
+ trusty/user/app/sample/ipc-unittest/main \
+ trusty/user/app/sample/ipc-unittest/srv \
# This project requires trusty IPC
WITH_TRUSTY_IPC := true
```

Rebuild the Trusty OS and copy the output binary to `${MY_ANDROID}/vendor/nxp/fsl-proprietary/u-boot-firmware/imx8q_car`. Make the following changes to build out the TIPC test binary:

```
diff --git a/imx8q/mek_8q/mek_8q_car.mk b/imx8q/mek_8q/mek_8q_car.mk
index 6acc89a..19e8e24 100644
--- a/imx8q/mek_8q/mek_8q_car.mk
+++ b/imx8q/mek_8q/mek_8q_car.mk
@@ -59,7 +59,8 @@ PRODUCT_PACKAGES += \
# Add Trusty OS backed gatekeeper and secure storage proxy
PRODUCT_PACKAGES += \
gatekeeper.trusty \
- storageproxyd \
+ storageproxyd \
+ tipc-test
```

Rebuild the Android project, the TIPC test binary is located at `${MY_ANDROID}/out/target/product/mek_8q/data/nativetest64/vendor/tipc-test/tipc-test`. Flash the images to board, and remount and push the `tipc-test` binary to `vendor/bin` with ADB commands.

Trusty OS runs the CAAM unit test when initializing the CAAM. The following logs are displayed in U-Boot if the CAAM is initialized correctly:

```
hwcrypto: 222: Initializing
caam_drv: 728: caam hwrng test PASS!!!
caam_drv: 761: caam blob test PASS!!!
caam_drv: 843: caam gen kdf root key test PASS!!!
caam_drv: 793: caam AES enc test PASS!!!
caam_drv: 802: caam AES enc test PASS!!!
caam_drv: 830: caam hash test PASS!!!
```

If the TIPC unit test is started correctly, the following logs are displayed in U-Boot:

```
ipc-unittest-main: 2607: Welcome to IPC unittest!!!
unittest: 144: added port com.android.ipc-unittest.ctrl handle, 1001, to
  handleset 1000
unittest: 148: waiting forever
ipc-unittest-srv: 318: Init unittest services!!!
```

Run the following commands to test the TIPC. The correct result is as follows:

```
mek_8q:/vendor/bin # tipc-test -t connect
connect_test: repeat = 1
connect_test: done
mek_8q:/vendor/bin # tipc-test -t connect_foo
connect_foo: repeat = 1
connect_foo: done
mek_8q:/vendor/bin # tipc-test -t echo -r 100
echo_test: repeat 100: msgsz 32: variable false
echo_test: done
mek_8q:/vendor/bin # tipc-test -t echo -r 1000
echo_test: repeat 1000: msgsz 32: variable false
echo_test: done
```

### 3.2.5 Modifying the console port for Trusty OS

Due to different hardware board designs, the debug UART may be different. i.MX Trusty OS supports to configure a different UART port by modifying the configuration file.

To change the debug UART port, see the SoC reference manual to get the specific UART port base address. The debug UART address is defined in `trusty/hardware/nxp/target/$SOC_NAME/rules.mk`.

For example, if LPUART1 is used instead of LPUART0 for i.MX 8QuadMax board, make the following modification on `rules.mk`:

```
diff --git a/target/imx8q/rules.mk b/target/imx8q/rules.mk
index e6239e2..8ea3f37 100644
--- a/target/imx8q/rules.mk
+++ b/target/imx8q/rules.mk
@@ -25,4 +25,4 @@
PLATFORM_SOC := imx8qm
PLATFORM := imx
-CONFIG_CONSOLE_TTY_BASE := 0x5A060000
+CONFIG_CONSOLE_TTY_BASE := 0x5A070000
```

### 3.2.6 Configuring the related TA services

The Trusted Application (TA) is the software running in a secure context. There are several TAs running in the Trusty OS. The following figure shows their relationships.



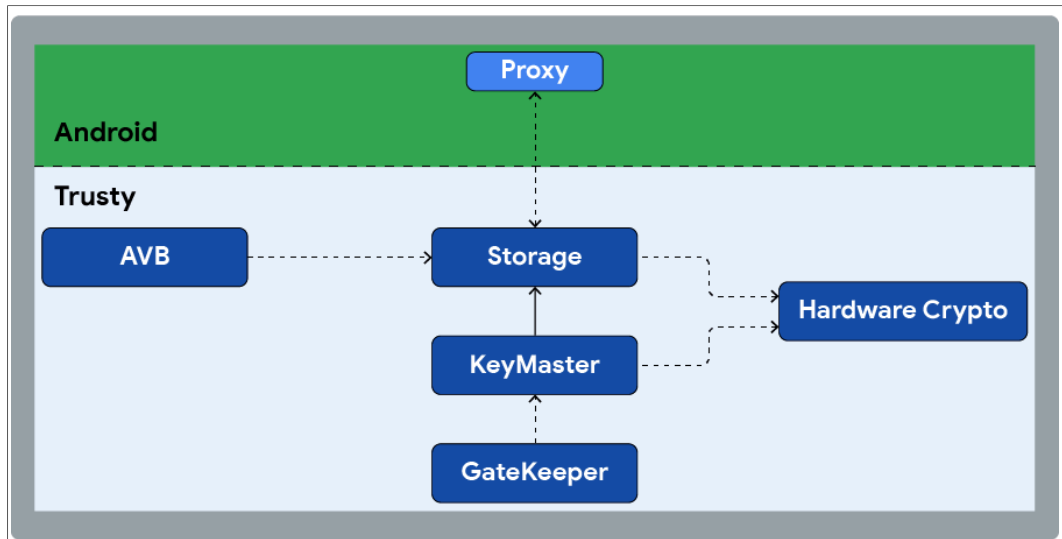


Figure 6. Relationship between TAs

- AVB TA: It provides tamper proof operations for data used during Android Verified Boot (AVB), such as rollback index, lock/unlock state, and vbmeta public key.
- Storage TA: It provides encrypted and tamper proof storage to secure applications, such as AVB TA. All operations that modify the secure storage are transactional.
- Hardware Crypto TA: It provides hardware crypto and accelerates operations based on CAAM, such as RNG generation and SHA1/SHA256 hash calculation.
- Keymaster TA: It provides all secure Keystore operations, with access to the raw key material, validating all of the access control conditions on keys.
- Gatekeeper TA: It authenticates user passwords and generates authentication tokens used to prove to the Keymaster TA that an authentication is done for a particular user at a particular point in time.

### 3.2.7 Specifying apploader encryption and sign keys

Trusted Applications (TAs) in Trusty OS can either be built into the Trusty binary or be built as loadable modules. In the case of building the TAs as loadable modules, the build system would encrypt and sign the modules to ensure its integrity and validity.

#### 1. Specify apploader encryption keys.

The Trusty OS build system would encrypt the loadable modules (in AES-GCM mode) if the encryption key file is specified. The encryption keys can be any valid customized keys. An example of the encryption key is in: `${MY_TRUSTY}/trusty/device/nxp/imx8/project/keys/aeskey.bin`.

It is generated by:

```
openssl rand 16 -out aeskey.bin
```

**Note:** This example encryption key must be replaced by customized key before shipping the products.

In additional, two kinds of encryption keys are supported for different purpose. After generating the encryption key, specify the encryption key files in `${MY_TRUSTY}/trusty/device/nxp/imx8/project/imx8-inc.mk` as follows:

```
APPLoader_ENCRYPT_KEY_0_FILE := <path-to-encryption-key-0>
APPLoader_ENCRYPT_KEY_1_FILE := <path-to-encryption-key-1>
```

To encrypt an application, set the appropriate variable in the application's `rules.mk` as follows:

```
# choose encryption key 0
APPLoader_ENCRYPT_KEY_ID_FOR_${MODULE} := 0
```

or

```
# choose encryption key 1
APPLoader_ENCRYPT_KEY_ID_FOR_${MODULE} := 1
```

## 2. Specify apploader sign keys.

The Trusty OS build system would sign and generate ECDSA signatures for the loadable modules. An example of the ECDSA sign key pairs are in: `/${MY_TRUSTY}/trusty/device/nxp/imx8/project/keys`. They are generated by the script in `/${MY_TRUSTY}/trusty/user/base/app/apploader/generate_ecdsa_keys.sh`:

```
./generate_ecdsa_keys.sh <private key file> <public key file>
```

**Note:** These example sign keys **MUST** be replaced by customized keys before shipping the products.

Similar to encryption keys, two pairs of sign key files can be provided for different purpose. After generating the sign key pairs, specify the key files in `/${MY_TRUSTY}/trusty/device/nxp/imx8/project/imx8-inc.mk` as follows:

```
134 PROJECT_KEYS_DIR := trusty/device/nxp/imx8/project/keys
135
136 APPLoader_SIGN_PRIVATE_KEY_0_FILE := \
137     ${PROJECT_KEYS_DIR}/privateKey0.der
138
139 APPLoader_SIGN_PUBLIC_KEY_0_FILE := \
140     ${PROJECT_KEYS_DIR}/publicKey0.der
141
142 APPLoader_SIGN_PRIVATE_KEY_1_FILE := \
143     ${PROJECT_KEYS_DIR}/privateKey1.der
144
145 APPLoader_SIGN_PUBLIC_KEY_1_FILE := \
146     ${PROJECT_KEYS_DIR}/publicKey1.der
147
148
149
150
151 APPLoader_SIGN_KEY_ID ?= 0
```

If the apploader sign key is not explicitly specified, Trusty OS would sign the loadable modules with `APPLoader_SIGN_PRIVATE_KEY_0_FILE` (as 'APPLoader\_SIGN\_KEY\_ID' is 0). To sign an application with different key, set the appropriate variable in the application's `rules.mk` as follows:

```
# sign with key 0
APPLoader_SIGN_KEY_ID_FOR_${MODULE} := 0
```

or

```
# sign with key 1
APPLoader_SIGN_KEY_ID_FOR_${MODULE} := 1
```

## 3.3 Configurations in U-Boot for security

U-Boot is loaded by SPL and verified with HAB. ATF starts U-Boot. The primary purpose of U-Boot is to load and verify Android images.

### 3.3.1 Overview of security features in U-Boot

Android Verified Boot (AVB) is enabled in i.MX Android images. There is an additional `vbmeta` image used in AVB. This `vbmeta` image does not contain any code that the device will execute. It is used by U-Boot to authenticate its own and other Android images. The other images to be authenticated with the `vbmeta` image include images for boot, dtbo, system, and vendor partitions. If there are other images containing executing code, like product, they are also authenticated based on the data in the `vbmeta` image. The hash value of these images is calculated and the metadata is stored in the `vbmeta` image. The following figure shows the relationship of these images.

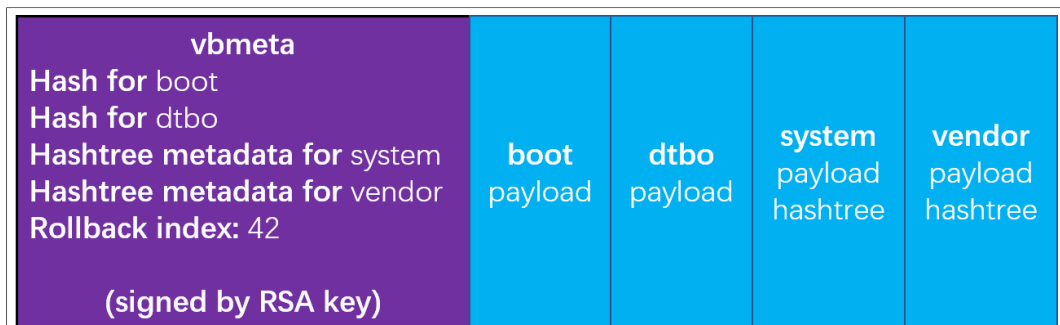


Figure 7. Relationship between `vbmeta` image and related images

To make sure that the `vbmeta` image is trusted, it is signed with the RSA key, and the signature of the `vbmeta` image is verified at boot time before the data in it are used to verify other images.

To prevent rollback attack, there is a rollback index value stored in the `vbmeta` image. The value can increase with the release of images. This rollback index value in the `vbmeta` image is also saved in the RPMB partition of eMMC after all the images are verified as bootable. If the rollback index value in the `vbmeta` image is smaller than the one stored in the RPMB partition of eMMC, U-Boot does not boot with the related images. With dual-bootloader enabled, SPL and U-Boot proper are not in one file, so there is another rollback index value for U-Boot proper.

To prevent the device from getting bricked during OTA, a/b slot feature is provided. Some partitions used to store images have two copies in the boot device. They are called "slot a" and "slot b". The image update process only flashes one slot. An update failure does not affect the other slot.

### 3.3.2 Generating and fusing the eMMC RPMB key

The RPMB partition of eMMC can be fused with the 256-bit secure key. The secure key can only be programmed one time. This secure key can only be accessed by secure world (either TEE or SPL). It is used to sign and verify data transfer between eMMC RPMB and TEE.

There are two ways to generate the RPMB key: CAAM hardware bound key and vendor-specified key. Both ways are associated with CAAM, which uses the value in efuse hardware. If the SRK hash value needs to be programmed into efuse hardware and close the chips, do it first, and only after that can the RPMB key be programmed.

- CAAM hardware bound key  
The RPMB key can be derived from CAAM. This is hardware bound and is unique per device. TEE derives this hardware bound key in every boot from CAAM, and as it is

bound to the CAAM hardware, so it does not need to store one copy of this key. This way is preferred as it is simpler and more secure.

Fastboot command is provided to set the RPMB key from CAAM hardware bound key:

```
$ fastboot oem set-rpmb-hardware-key
```

- Vendor specified key

Vendor can specify any customized RPMB key. One copy of the key is encapsulated with CAAM and the generated key blob is saved into the last block of the eMMC BOOT1 partition. The BOOT1 partition size of eMMC on i.MX 8QuadMax MEK and i.MX 8QuadXPlus MEK is 8 MB. It is 4 MB on i.MX 8M Mini EVK, i.MX 8M Plus EVK, and i.MX8M Nano EVK. To prevent key blob from being tampered when the system is running, the BOOT1 partition is set with power-on write protection when the board boots up.

The location to store the key blob may need to be changed based on the board design. Two macros are used to control the location of the key blob. These two macros are the same for i.MX 8QuadMax MEK and i.MX 8QuadXPlus MEK. Their definitions are as follows:

```
#define KEYSLOT_HWPARTITION_ID 2
#define KEYSLOT_BLKs 0x3FFF
```

While for i.MX 8M Mini EVK and i.MX 8M Nano EVK, the definition is as follows:

```
#define KEYSLOT_HWPARTITION_ID 2
#define KEYSLOT_BLKs 0x1FFF
```

KEYSLOT\_HWPARTITION\_ID represents the eMMC partition. **0** means USERDATA partition, **1** means BOOT0 partition, and **2** means BOOT1 partition. KEYSLOT\_BLKs represents the block in which the key blob is stored.

For i.MX 8QuadMax MEK, they are in the following files. For other platforms, it is in other board-related header files in the same folder.

```
/* Android Automotive */
${MY_ANDROIDID}/vendor/nxp-opensource/uboot-imx/include/configs/
imx8qm_mek_android_auto.h
/* Standard Android */
${MY_ANDROIDID}/vendor/nxp-opensource/uboot-imx/include/configs/
imx8qm_mek_android.h
```

The customized RPMB key file should be started with magic "RPMB" and followed by the raw key. The following steps show how to generate the default key file rpmb\_key\_test.bin:

```
$ touch rpmb_key_test.bin
$ echo -n "RPMB" > rpmb_key_test.bin
$ echo -n -e
'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' >>
rpmb_key_test.bin
```

The \xHH means eight-bit character whose value is the hexadecimal value 'HH'. You can replace above "00" with the key you want to set.

After generating the customized RPMB key file, the following fastboot commands are available to set the RPMB key:

```
$ fastboot stage rpmb_key_test.bin
$ fastboot oem set-rpmb-staged-key
```

After the RPMB key is programmed with either of the two ways, reboot the board. The RPMB service in Trusty OS is then initialized successfully.

### 3.3.3 Generating keys to sign and verify images

Images are signed during the time of building Android platform to ensure their integrity. A pair of asymmetric keys (AVB keys) is used to sign the `vbmeta` struct in the `vbmeta` image. On the standard Android platform, there is another pair of asymmetric keys (AVB boot keys) to sign the `vbmeta` struct in the boot image, which would be built as 'chained partition'. For more information about chained partition, see <https://android.googlesource.com/platform/external/avb/+master/README.md>.

By default, the private AVB key used to sign the `vbmeta` struct in `vbmeta` image is located at:

```

${MY_ANDROIDID}/device/nxp/common/security/testkey_rsa4096.pem

```

Its corresponding public key is:

```

${MY_ANDROIDID}/device/nxp/common/security/
testkey_public_rsa4096.bin.

```

The default private AVB boot key used to sign the boot image is located at:

```

${MY_ANDROIDID}/external/avb/test/data/testkey_rsa2048.pem

```

The private key can be generated with OpenSSL. For example, the following command can generate RSA-4096 private key `test_rsa4096_private.pem`:

```

openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -
outform PEM -out custom_rsa4096_private.pem

```

The corresponding public key can be extracted from the private key with `avbtool`. The `avbtool` can be found in `${MY_ANDROIDID}/external/avb`. Execute the following command to extract the public key from the private key:

```

avbtool extract_public_key --key custom_rsa4096_private.pem --
output custom_rsa4096_public.bin

```

SHA256\_RSA4094 and SHA256\_RSA2048 are recommended algorithm to sign the images for i.MX 8Quad and i.MX 8M devices whose Cryptographic Acceleration and Assurance Module (CAAM) can help accelerate the hash calculation. We can keep it as default.

Custom keys should be used to sign the images for production, examples to set the custom AVB key to sign the `vbmeta` image are as follows:

```

diff --git a/imx8q/mek_8q/BoardConfig.mk b/imx8q/mek_8q/
BoardConfig.mk
index 8e367bb..e1385f9 100644
--- a/imx8q/mek_8q/BoardConfig.mk
+++ b/imx8q/mek_8q/BoardConfig.mk
@@ -207,7 +207,7 @@ BOARD_AVB_ENABLE := true
ifeq ($(PRODUCT_IMX_CAR),true)
BOARD_AVB_ALGORITHM := SHA256_RSA4096
# The testkey_rsa4096.pem is copied from external/avb/test/
data/testkey_rsa4096.pem
-BOARD_AVB_KEY_PATH := device/nxp/common/security/
testkey_rsa4096.pem

```

```
+BOARD_AVB_KEY_PATH := ${your-key-directory}/
custom_rsa4096_private.pem
endif
TARGET_USES_MKE2FS := true
```

Examples to set the custom AVB boot key to sign the boot image are as follows:

```
diff --git a/imx8q/mek_8q/BoardConfig.mk b/imx8q/mek_8q/
BoardConfig.mk
index dbfb2821..744c2086 100644
--- a/imx8q/mek_8q/BoardConfig.mk
+++ b/imx8q/mek_8q/BoardConfig.mk
@@ -253,7 +253,7 @@ BOARD_AVB_ALGORITHM := SHA256_RSA4096
BOARD_AVB_KEY_PATH := device/nxp/common/security/
testkey_rsa4096.pem
ifneq ($(PRODUCT_IMX_CAR), true)
-BOARD_AVB_BOOT_KEY_PATH := external/avb/test/data/
testkey_rsa2048.pem
+BOARD_AVB_BOOT_KEY_PATH := ${your-key-directory}/
custom_rsa2048_private.pem
BOARD_AVB_BOOT_ALGORITHM := SHA256_RSA2048
BOARD_AVB_BOOT_ROLLBACK_INDEX := 0
BOARD_AVB_BOOT_ROLLBACK_INDEX_LOCATION := 2
```

AVB checks the signature and the AVB public key when verifying the `vbmeta` image, so the AVB public key must be stored to the TEE backed RPMB when Trusty is enabled. Make the board enter fastboot mode, and execute the following commands:

```
$ fastboot stage custom_rsa4096_public.bin
$ fastboot oem set-public-key
```

`custom_rsa4096_public.bin` is the public AVB key just generated. If you use the default AVB keys for debug purpose, flash the default public key with the following commands:

```
$ fastboot stage testkey_public_rsa4096.bin
$ fastboot oem set-public-key
```

### 3.3.4 Bypass vbmeta/lock check for development purposes

Bypassing `vbmeta/lock` check is very convenient for development work. To unlock the device after all images are flashed, boot the board to the Android UI, enable "Developer options" in the "Settings" Application, open "OEM unlocking" under "Developer options", and reboot the board to fastboot mode. Execute the following command:

```
$ sudo fastboot oem unlock
```

After the board is unlocked, images can be flashed with the `fastboot` command. To bypass `vbmeta` check, use `fastboot` to flash the `vbmeta` image with the `--disable-verity` option. Take i.MX 8QuadMax MEK as an example, execute the following commands:

```
$ sudo fastboot flash vbmeta_a vbmeta-imx8qm.img --disable-verity
```

```
$ sudo fastboot flash vbmeta_b vbmeta-imx8qm.img --disable-verity
```

### 3.3.5 Changing the value of the rollback index in images

There are three rollback index values involved in this release: bootloader rollback index, boot rollback index, and vbmeta rollback index. The bootloader rollback index is used to prevent bootloader rollback attack. It is only available when the dual-bootloader feature is enabled. The boot rollback index is available when the boot image is built as chained partition. It is only supported on standard Android platform. The vbmeta rollback index is supported on all platforms. By default, the initial value of all kinds of rollback index is zero.

When a version of images is to be released to fix a bug in previous version, which makes previous images under potential attacks, it is suggested to increase the rollback index values by one compared to previous version. For more information about rollback index usage, see <https://android.googlesource.com/platform/external/avb+/master/README.md>.

In this release of i.MX Android platform, some modifications are made to the Android build system. A shell script named `imx-make.sh` is provided to build U-Boot and kernel code independently from the building process of Android images. To make the build process simple, as all images can be built with only one command, `imx-make.sh` starts to build Android images after U-Boot and kernel are built. When executing the `imx-make.sh` script, a shell variable named `BOOTLOADER_RBINDEX` can be specified to change the bootloader rollback index, a shell variable named `AVB_BOOT_RBINDEX` can be specified to change the boot rollback index, and a shell variable named `AVB_RBINDEX` can be specified to change the vbmeta rollback index.

As an example, the following command can be executed to change the rollback index values. Change `${bootloader_rbindex}`, `${avb_rbindex}`, and `${avb_boot_rbindex}` to the value you want to set:

```
BOOTLOADER_RBINDEX=${bootloader_rbindex} AVB_RBINDEX=${avb_rbindex} AVB_BOOT_RBINDEX=${avb_boot_rbindex} ./imx-make.sh -j4
```

If the U-Boot and kernel are already built and they do not need to be updated, `make` can be used to build vbmeta and other related images need to be updated. Change the rollback index value as shown in the following example. Change `${avb_rbindex}` and `${avb_boot_rbindex}` to the value you want to set.

```
make -j4 AVB_RBINDEX=${avb_rbindex} AVB_BOOT_RBINDEX=${avb_boot_rbindex}
```

### 3.3.6 Programming the attestation key

Attestation key is programmed in U-Boot. The keystore key attestation aims to provide a way to strongly determine if an asymmetric key pair is hardware-backed, what the properties of the key are, and what constraints are applied to its usage.

Google provides the attestation "keybox", which contains private keys (RSA and ECDSA) and the corresponding certificate chains to partners from the Android Partner Front End (APFE). After retrieving the "keybox" from Google, you need to parse the "keybox",

provision the keys and certificates to secure storage. Both keys and certificates should be encoded with Distinguished Encoding Rules (DER).

Two ways are provided to provision the attestation keys and certificates, one is provisioning the keys and certificates in plaintext format directly, another is provisioning the keys and certificates in AES-ECB encrypted format which will then be decrypted into plaintext before written into secure storage. Before provisioning start, please make sure the secure storage is properly initialized for Trusty OS.

1. Provision keys and certificates in plaintext format.

Fastboot commands are provided to flash the attestation keys and certificates in plaintext format to device. This way is more simple but has more risk of leaking the keys. Boot the board into fastboot mode and use the following commands.

- Set the RSA private key:

```
$ fastboot stage ${path-to-rsa-private-key}
$ fastboot oem set-rsa-atte-key
```

- Set the ECDSA private key:

```
$ fastboot stage ${path-to-ecdsa-private-key}
$ fastboot oem set-ec-atte-key
```

- Append the RSA certificate chain:

```
$ fastboot stage ${path-to-rsa-atte-cert}
$ fastboot oem append-rsa-atte-cert
```

The second command may need to be executed multiple times to append the whole certificate chain.

- Append the ECDSA certificate chain:

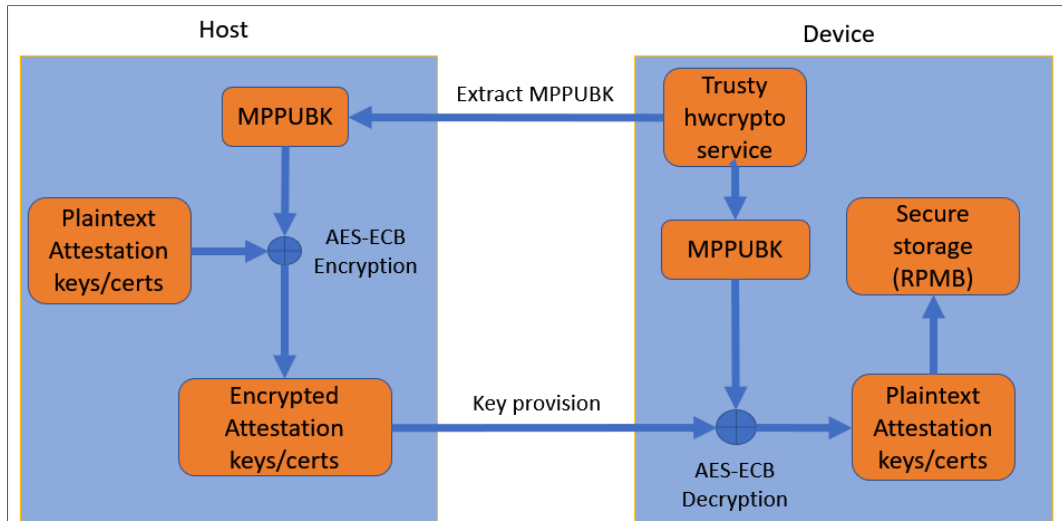
```
$ fastboot stage < path-to-ecdsa-cert >
$ fastboot oem append-ec-atte-cert
```

The second command may need to be executed multiple times to append the whole certificate chain.

2. Provision keys and certificates in AES-ECB encrypted format.

Fastboot commands are provided to flash AES-ECB encoded keys and certificates to device, it can prevent leakage of the plaintext materials. This way will encrypt the keys and certificates with the MPPUBK (Manufacturing Protection Public Key, generated from CAAM module), then the encrypted keys and certificates would be decrypted into plaintext in Trusty OS before they are written into secure storage. Please note this way can only be used on HAB CLOSED board. The workflow is like below:





**Figure 8. Flashing AES-ECB encoded keys and certificates to device**

Boot the board into fastboot mode and perform the following steps:

- a. Get MPPUBK:

```
$ fastboot oem get-mppubk
$ fastboot get_staged mppubk.bin
```

- b. Encrypt the plaintext attestation keys and certificates with the MPPUBK. The attestation keys and certificates should be encrypted in AES-ECB with the generated MPPUBK. The following is a simple encryption python scripts gen\_secure\_atte.py:

```
from Crypto.Cipher import AES
import struct
import argparse
parser = argparse.ArgumentParser(description='Secure
Provision encrypt tool.')
parser.add_argument('key', type=file)
parser.add_argument('plaintext', type=file)
parser.add_argument('blob', type=argparse.FileType('w'))
args = parser.parse_args()
data = args.plaintext.read()
data_len = len(data)
written_len = struct.pack('I', data_len)
#AES need 16 bytes align, so pad the data it
data += '\0' * (((len(data)+15)/16 * 16) - len(data))
key2 = args.key.read()
key = key2[0:16]
magic = "!AT"
pad0 = struct.pack('B', 0)
cipher = AES.new(key, AES.MODE_ECB)
blob = cipher.encrypt(data)
args.blob.write(magic)
args.blob.write(pad0)
args.blob.write(written_len)
args.blob.write(blob)
#blob structure describe as below:
#{
# char magic[4] = "!AT";
# uint32_t len = plaintext_length
# uint8 *encrypted_data
```

```
# }
```

Encrypt the keys and the certificates on the host computer:

```
$ python gen_secure_atte.py mppubk.bin < path-to-plaintext-keys-or-certificates > < encrypted-keys-or-certificates >
```

c. Set encrypted RSA private key:

```
$ fastboot stage ${path-to-encrypted-rsa-private-key}
$ fastboot oem set-rsa-atte-key-enc
```

d. Set encrypted ECDSA private key:

```
$ fastboot stage ${path-to-encrypted-ecdsa-private-key}
$ fastboot oem set-ec-atte-key-enc
```

e. Append encrypted RSA certificate chain:

```
$ fastboot stage ${path-to-encrypted-rsa-atte-cert}
$ fastboot oem append-rsa-atte-cert-enc
```

The second command may need to be executed multiple times to append the whole certificate chain.

f. Append encrypted ECDSA certificate chain:

```
$ fastboot stage < path-to-encrypted-ecdsa-cert >
$ fastboot oem append-ec-atte-cert-enc
```

The second command may need to be executed multiple times to append the whole certificate chain.

### 3.3.7 Programming the hardware identifiers

ID attestation allows the device to provide proof of its hardware identifiers. For more information about ID attestation, see <https://source.android.com/security/keystore/attestation>. All hardware identifiers provisioned should match related system properties. The following table shows a map of all supported identifiers and the system properties.

Identifiers	Build property
ATTESTATION_ID_BRAND	ro.product.brand
ATTESTATION_ID_DEVICE	ro.product.device
ATTESTATION_ID_MANUFACTURER	ro.product.manufacturer
ATTESTATION_ID_MODEL	ro.product.model
ATTESTATION_ID_PRODUCT	ro.product.product.name
ATTESTATION_ID_SERIAL	ro.serialno

The hardware identifiers are provisioned in bootloader and they can be set by configs. Take i.MX 8QM as example, its hardware identifiers are set in `{UBOOT_PATH}/configs/imx8qm_mek_android_trusty_defconfig`:

```
CONFIG_ATTESTATION_ID_BRAND="Android"
CONFIG_ATTESTATION_ID_DEVICE="mek_8q"
CONFIG_ATTESTATION_ID_PRODUCT="mek_8q"
CONFIG_ATTESTATION_ID_MANUFACTURER="nxp"
```

```
CONFIG_ATTESTATION_ID_MODEL="MEK-MX8Q"
```

Note that the serial number is unique for each device, so its value is auto detected in bootloader.

Change the identifiers as needed, then flash updated bootloader image and boot the board into fastboot mode, run the following command to provision the identifiers:

```
$ fastboot oem set-device-id
```

### 3.3.8 Provisioning Widevine L1 keybox

The Widevine is a widely used Digital Rights Management (DRM) technology, which helps to protect media streaming from being copied and redistributed. The L1 level Widevine is now supported on the i.MX 8M Plus EVK board. Before making the L1 level Widevine work, users need to provision the Widevine keybox first.

Just like attestation keys and certificates, two ways are provided to provision the Widevine keybox, one is provisioning the keybox binary in plaintext format, the other is provisioning the keybox binary in AES-ECB encrypted format which will then be decrypted into plaintext before written into secure storage.

- Provisioning keybox binary in plaintext format  
Fastboot commands are provided to flash the keybox in plaintext format to device. This way is simpler but has more risks of leaking the keys.

Boot the board into fastboot mode and run the following commands:

```
$ fastboot stage ${path-to-keybox-binary}
$ fastboot oem provision-wv-keybox
```

- Provisioning keybox binary in AES-ECB encrypted format  
Fastboot commands are provided to flash AES-ECB encoded keybox to device, it can prevent leakage of the plaintext materials. This way will encrypt the keybox with the MPPUBK (Manufacturing Protection Public Key, generated from CAAM module), then the encrypted keybox would be decrypted into plaintext in Trusty OS before it's written into secure storage. Please note this way can only be used on HAB CLOSED board. Perform the following steps to encrypt the keybox and provision it into the device:

1. Get the MPPUBK.

```
$ fastboot oem get-mppubk
$ fastboot get_staged mppubk.bin
```

2. Encrypt the plaintext keybox with the MPPUBK.

The plaintext keybox binary should be encrypted in AES-ECB with the generated MPPUBK. Do the encryption with the python scripts `gen_secure_atte.py`, which is mentioned in [Section 3.3.6](#).

```
$ python gen_secure_atte.py mppubk.bin ${path-to-plaintext-keybox} ${encrypted-keybox}
```

3. Provision the widevine keybox.

```
$ fastboot stage ${path-to-encrypted-keybox}
$ fastboot oem provision-wv-keybox-enc
```

### 3.3.9 Changing the way to store lock status and/or rollback index

For images with TEE enabled, lock status and rollback index values are stored in RPMB. The rollback index value for AVB is written/read by TEE into/from RPMB but the write/read process is initiated by U-Boot. For i.MX Android with dual-bootloader feature, there is a rollback index for bootloader, this rollback index value for bootloader is written/read by SPL into/from RPMB.

Rollback index values and lock status can be used for many purposes as designed by developers, not limited to the usage in i.MX Android code. At this point, it is necessary to know how the lock status and rollback index values are stored on board.

For i.MX Android with dual-bootloader feature, the rollback index value for bootloader is read from RPMB to compare with the one in the bootloader image. If the rollback index value is bigger than the one stored in RPMB and the images are verified as bootable, rollback index value in bootloader image is written into RPMB. This logic is completed in the following function:

```
static int spl_verify_rbidx(struct mmc *mmc, AvbABSslotData
    *slot,
    struct spl_image_info *spl_image)
```

In the following file:

```
 ${MY_ANDROIDID}/vendor/nxp-opensource/uboot-imx/lib/avb/fsl/
    fsl_avb_ab_flow.c
```

For new boards just flashed with images, at their first time of boot, a default rollback index value is written in RPMB in the following function:

```
int rpmb_init(void)
```

In the following file:

```
 ${MY_ANDROIDID}/vendor/nxp-opensource/uboot-imx/lib/avb/fsl/
    fsl_avb_ab_flow.c
```

From the functions listed above, it is known that the rollback index value for bootloader is located by a `kblb_hdr_t` type structure variable. This structure has a magic value. A member with the type of `kblb_tag_t` is used to specify the rollback index value.

Now in i.MX Android Auto, the offset of the rollback index value for bootloader is controlled by a macro named `BOOTLOADER_RBIDX_START` as defined in the following two files respectively for i.MX 8QuadMax MEK and i.MX 8QuadXPlus MEK.

```
 ${MY_ANDROIDID}/vendor/nxp-opensource/uboot-imx/include/configs/
    imx8qm_mek_android_auto.h
    ${MY_ANDROIDID}/vendor/nxp-opensource/uboot-imx/include/configs/
    imx8qxp_mek_android_auto.h
```

The value for `BOOTLOADER_RBIDX_START` is `0x3FF000`, 4KB offset from the end of the RPMB partition.

The read process of the rollback index value for AVB is initiated by U-Boot in the following function:

```
FbLockState fastboot_get_lock_stat(void)
```

In the following file:

```
${MY_ANDROID}/vendor/nxp-opensource/uboot-imx/drivers/usb/  
gadget/fastboot_lock_unlock.c
```

For images with TEE enabled, this function invokes the following function. It uses TIPC to communicate with TEE to get the value.

```
int trusty_read_lock_state(uint8_t *lock_state)
```

The write process of the rollback index value for AVB is initiated by U-Boot in the following function:

```
int fastboot_set_lock_stat(FbLockState lock)
```

In the following file:

```
${MY_ANDROID}/vendor/nxp-opensource/uboot-imx/drivers/usb/  
gadget/fastboot_lock_unlock.c
```

For images with TEE enabled, this function invokes the following function. It uses TIPC to communicate with TEE to save the value.

```
int trusty_write_lock_state(uint8_t lock_state)
```

Rollback index value for AVB is read to compare with the one in vbmeta image and the one in vbmeta image is saved into RPMB if necessary. This logic is completed in the following function:

```
AvbABFlowResult avb_flow_dual_uboot(AvbABOps* ab_ops,  
const char* const* requested_partitions,  
AvbSlotVerifyFlags flags,  
AvbHashtreeErrorMode hashtree_error_mode,  
AvbSlotVerifyData** out_data)
```

In the following file:

```
${MY_ANDROID}/vendor/nxp-opensource/uboot-imx/lib/avb/fsl/  
fsl_avb_ab_flow.c
```

The following two functions are invoked to read and store the rollback index for vbmeta:

```
AvbIOResult fsl_read_rollback_index_rpmb(AvbOps* ops, size_t  
rollback_index_slot,  
uint64_t* out_rollback_index)  
AvbIOResult fsl_write_rollback_index_rpmb(AvbOps* ops, size_t  
rollback_index_slot,  
uint64_t rollback_index)
```

They finally communicate with TEE to finish the work.

### 3.3.10 Choosing to boot a specific slot

With both slots flashed with images, a specific slot can be chosen to boot manually for development purpose. Boot the board into fastboot mode, and execute the following command to boot from "slot a" or "slot b":

```
$ sudo fastboot set_active a
$ sudo fastboot set_active b
```

### 3.3.11 Disabling development options in U-Boot

To facilitate development, some development options are set in U-Boot, which may bring in potential security holes. Before shipping the final products, these options must be closed.

- **Boot delay**

By default, the U-Boot reserves 2 seconds count-down to help developer stop at U-Boot and run some U-Boot commands. This can be disabled by setting `CONFIG_BOOTDELAY` to `-2`. For i.MX 8M Plus EVK, make the following changes. Similar changes need to be made on other platforms that you are working on.

```
diff --git a/configs/imx8mp_evk_android_trusty_defconfig b/
configs/imx8mp_evk_android_trusty_defconfig
index 80a4d45a5e..0ea9e3b9fc 100644
--- a/configs/imx8mp_evk_android_trusty_defconfig
+++ b/configs/imx8mp_evk_android_trusty_defconfig
@@ -185,3 +185,4 @@ CONFIG_ATTESTATION_ID_DEVICE="evk_8mp"
    CONFIG_ATTESTATION_ID_PRODUCT="evk_8mp"
    CONFIG_ATTESTATION_ID_MANUFACTURER="nxp"
    CONFIG_ATTESTATION_ID_MODEL="EVK_8MP"
+CONFIG_BOOTDELAY=-2
```

- **Bootargs appending**

The bootargs may need to be changed frequently during development. NXP U-Boot supports appending the U-Boot variable `append_bootargs` to the default bootargs, which will be passed to kernel. However, this feature can be used by hackers to compromise the device and should be disabled in any formal release. To disable the bootargs appending feature, you need to disable `CONFIG_APPEND_BOOTARGS`. For For i.MX 8M Plus EVK, make the following changes. Similar changes need to be made on other platforms that you are working on.

```
diff --git a/configs/imx8mp_evk_android_trusty_defconfig b/
configs/imx8mp_evk_android_trusty_defconfig
index 80a4d45a5e..62ab4afaa0 100644
--- a/configs/imx8mp_evk_android_trusty_defconfig
+++ b/configs/imx8mp_evk_android_trusty_defconfig
@@ -171,7 +171,6 @@ CONFIG_CMD_BMP=y
    CONFIG_LZ4=y
    CONFIG_FLASH_MCUFIRMWARE_SUPPORT=y
-CONFIG_APPEND_BOOTARGS=y
    CONFIG_SPL_MMC_SUPPORT=y
    CONFIG_AVB_WARNING_LOGO=y
    CONFIG_AVB_WARNING_LOGO_COLS=0x320
```

- **Disable manufacturing production public key extraction**

The manufacturing production public key can be used as encryption/decryption keys in some cases, leaking the key has potential risk of leaking other important materials. The extraction of the manufacturing production public must be disabled for shipping

images. For i.MX 8M Plus EVK, you can make below changes. Similar changes need to be made on other platforms that you are working on.

```
diff --git a/configs/imx8mp_evk_android_trusty_defconfig b/
configs/imx8mp_evk_android_trusty_defconfig
index 80a4d45a5e..61be3f5ebe 100644
--- a/configs/imx8mp_evk_android_trusty_defconfig
+++ b/configs/imx8mp_evk_android_trusty_defconfig
@@ -185,3 +185,4 @@ CONFIG_ATTESTATION_ID_DEVICE="evk_8mp"
CONFIG_ATTESTATION_ID_PRODUCT="evk_8mp"
CONFIG_ATTESTATION_ID_MANUFACTURER="nxp"
CONFIG_ATTESTATION_ID_MODEL="EVK_8MP"
+CONFIG_GENERATE_MPPUBK=n
```

### 3.3.12 Secure unlock

Secure unlock is designed to prevent unauthorized unlock. It requires the unlock credential. The unlock operation can only occur after a valid unlock credential is provided.

An example is provided, which generates the unlock credential with serial number and MPPUBK (Manufacturing Protection Public Key, generated from CAAM module). To enable the secure unlock feature, enable `CONFIG_SECURE_UNLOCK`. For i.MX 8M Plus EVK, make the following changes. Similar changes need to be made on other platforms that you are working on:

```
diff --git a/configs/imx8mp_evk_android_trusty_defconfig b/
configs/imx8mp_evk_android_trusty_defconfig
index 80a4d45a5e..b3da5c6ea3 100644
--- a/configs/imx8mp_evk_android_trusty_defconfig
+++ b/configs/imx8mp_evk_android_trusty_defconfig
@@ -185,3 +185,4 @@ CONFIG_ATTESTATION_ID_DEVICE="evk_8mp"
CONFIG_ATTESTATION_ID_PRODUCT="evk_8mp"
CONFIG_ATTESTATION_ID_MANUFACTURER="nxp"
CONFIG_ATTESTATION_ID_MODEL="EVK_8MP"
+CONFIG_SECURE_UNLOCK=y
```

Perform the following steps to verify the secure unlock feature. These operations can only be executed on the HAB closed boards:

1. Get MPPUBK.

```
$ fastboot oem get-mppubk
$ fastboot get_staged mppubk.bin
```

2. Get the serial number.

```
$ fastboot oem get-serial-number
$ fastboot get_staged serial.bin
```

3. Generate the unlock credential. Encrypt `serial.bin` with `mppubk.bin` on the host PC. For the encryption script, see Section [Section 3.3.6](#).

```
$ python gen_secure_atte.py mppubk.bin serial.bin serial-
enc.bin
```

4. Verify the secure unlock feature.

```
$ fastboot stage serial-enc.bin
$ fastboot oem unlock
```

You may need to lock the device first if the device is already in unlocked state.

### 3.4 Configurations in Linux/Android platform for security features

#### 3.4.1 DM-Verity relationship with vbmeta

The Device Mapper verity (DM-verity) kernel feature supports transparent integrity checking of block devices. This feature helps Android users be sure that when booting a device, it is in the same state as when it is flashed. The `vbmeta` image contains a kernel command-line descriptor for setting up DM-verity for `system.img`, together with hashtree descriptors for `system.img` and `vendor.img`. The hash tree descriptor in the `vbmeta` image contains the root hash, salt and the offset of the hashtree, which are essential to do the DM-verity check for system and vendor partitions.

When the DM-verity is enabled for system and vendor partition, any operations that break the consistency of the `system.img`, `vendor.img`, and `vbmeta.img` will cause DM-verity check failure, and thus cause the system boot failure.

#### 3.4.2 Trusty OS Linux driver configuration

The Trusty OS supports to output the logs to UART or TIPC log channel. The Trusty OS Linux driver supports to carry the logs from the Trusty OS by TIPC channel. By default, this feature is enabled in the reference image.

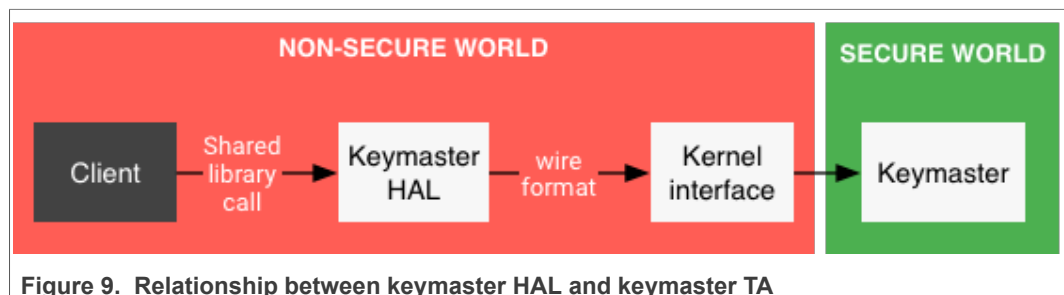
In the Trusty OS Linux driver `trusty-log`, when it is enabled, the Trusty OS shuts down the UART output log port. The UART driver in the Trusty OS outputs characters synchronously and it costs much IO time.

The `trusty-log` driver is configured in the device tree as follows:

```
trusty-log {
    compatible = "android,trusty-log-v1";
};
```

#### 3.4.3 Introductions of trusty based keymaster, gatekeeper, and secure storage proxy

The trusty backed keymaster HAL is a dynamically loadable library used by the keystore service to provide hardware-backed cryptographic services. It does not provide any sensitive operations in user space, or even in kernel space. All sensitive operations are delegated to the keymaster TA in the Trusty OS (secure world). The relationship is shown in the following figure.





The trusty backed keymaster HAL 3.0 is designed for Android Pie 9 or later, which cannot work for Android Oreo 8.1. Instead, the Android Oreo 8.1 is running trusty backed keymaster HAL 2.0.

The Gatekeeper subsystem performs device pattern/password authentication. It enrolls and verifies passwords through an HMAC with a secret key. Additionally, the Gatekeeper throttles consecutive failed verification attempts and refuses to service requests based on a given timeout and a given number of consecutive failed attempts. The trusty backed gatekeeper sends all critical operations to the gatekeeper TA in trusty.

The secure storage proxy is running in the Linux end to communicate with the storage TA in trusty to perform secure storage read/write operations, for example, reading/writing data from/to RPMB partition of the eMMC device.

Trusty backed keymaster, gatekeeper, and secure storage proxy all depend on secure storage, which can only be accessed by trusty, but users may not want to set the secure storage properly (like the key of RPMB), because in some instances, security is not so important and can even be neglected. In this case, both keymaster and gatekeeper fall back to software backed version, and they are chosen by the `androidboot.keystore` variable in the kernel command line.

When the trusty and associated trusted applications (such as keymaster TA and storage TA) are initialized properly, U-Boot sets `androidboot.keystore` to `trusty`, otherwise to `software`, and then passes it to the kernel through the kernel command line. The `androidboot.keystore` is translated to `ro.boot.keystore` Android property, and then the initialization program chooses the keymaster and gatekeeper version (trusty backed or software backed) and starts the secure storage proxy according to this property. The following figure shows the workflow.

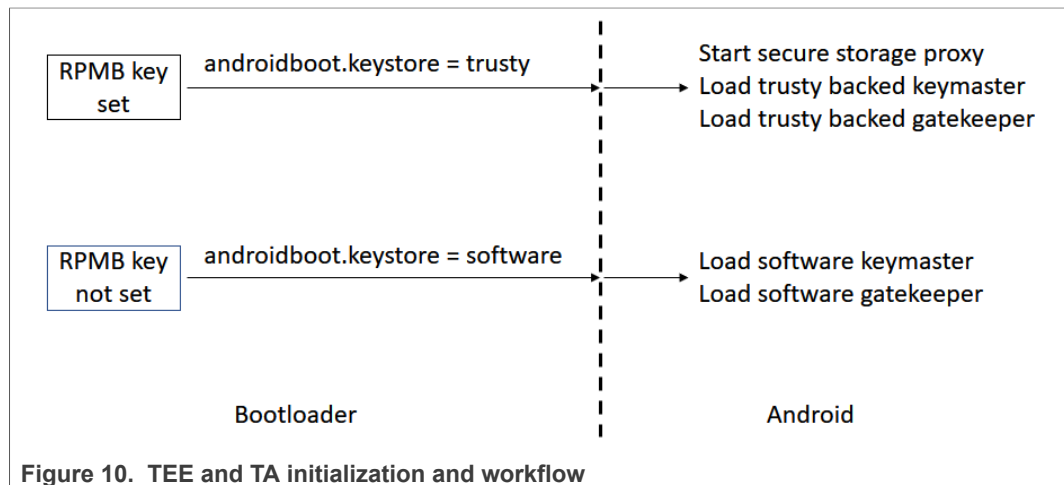


Figure 10. TEE and TA initialization and workflow

### 3.4.4 Disabling GPU flat mapping

When GPU flat mapping is enabled, hackers may be able to snoop privileged memory. The GPU flat mapping is enabled by default, but the user can disable it by adding bootargs as follows:

```
galcore.baseAddress=<dram-base-address> galcore.physSize=0
galcore.mmuException=0
```

The `dram-base-address` varies on different platforms. The value should be decided based on each platform. Take i.MX 8M Plus as an example. The user can disable the

GPU flat mapping by adding the following bootargs in {MY\_ANDROID}/imx8m/evk\_8mp/BoardConfig.mk:

```
diff --git a/imx8m/evk_8mp/BoardConfig.mk b/imx8m/evk_8mp/BoardConfig.mk
index 4732374b..83089631 100644
--- a/imx8m/evk_8mp/BoardConfig.mk
+++ b/imx8m/evk_8mp/BoardConfig.mk
@@ -119,6 +119,9 @@ BOARD_KERNEL_CMDLINE := init=/init
 androidboot.console=ttymxc1 androidboot.hardware
 BOARD_KERNEL_CMDLINE += transparent hugepage=never
 BOARD_KERNEL_CMDLINE += swiotlb=65536
+# disable GPU flat mapping
+BOARD_KERNEL_CMDLINE += galcore.baseAddress=0x40000000
galcore.physSize=0 galcore.mmuException=0
+
# display config
BOARD_KERNEL_CMDLINE += androidboot.lcd_density=240
androidboot.primary_display=imx-drm
```

## 4 Revision History

### Revision history

Revision number	Date	Substantive changes
P9.0.0_1.0.0-beta	11/2018	Initial release
P9.0.0_1.0.0-ga	01/2019	i.MX 8M, i.MX 8QuadMax, i.MX 8QuadXPlus GA release.
P9.0.0_2.0.0-ga	04/2019	i.MX 8M, i.MX 8QuadMax, i.MX 8QuadXPlus GA release.
P9.0.0_2.0.0-ga	08/2019	Updated the location of the SCFW porting kit.
android-10.0.0_1.0.0	02/2020	i.MX 8M Mini, i.MX 8M Quad, i.MX 8QuadMax, and i.MX 8QuadXPlus GA release.
android-10.0.0_1.0.0	03/2020	Deleted the Android 10 image.
android-10.0.0_2.1.0	04/2020	i.MX 8M Plus Alpha and i.MX 8QuadXPlus Beta release.
android-10.0.0_2.0.0	05/2020	i.MX 8M Mini, i.MX 8M Nano, i.MX 8M Quad, i.MX 8QuadMax, and i.MX 8QuadXPlus GA release.
android-10.0.0_2.3.0	07/2020	i.MX 8M Plus EVK Beta1 release, and all the other i.MX 8 GA release.
android-11.0.0_1.0.0	12/2020	i.MX 8M Plus EVK Beta release, and all the other i.MX 8 GA release.
android-11.0.0_1.2.0	03/2021	i.MX 8M Plus EVK GA release.
android-11.0.0_2.0.0	04/2021	i.MX 8M Mini, i.MX 8M Nano, i.MX 8M Plus, and i.MX 8M Quad GA release.
android-11.0.0_2.2.0	07/2021	i.MX 8M Mini, i.MX 8M Nano, i.MX 8M Plus, and i.MX 8M Quad GA release.
android-11.0.0_2.4.0	10/2021	i.MX 8ULP EVK Alpha release, i.MX 8M Mini, i.MX 8M Nano, i.MX 8M Plus, and i.MX 8M Quad GA release.

**Revision history...continued**

Revision number	Date	Substantive changes
android-11.0.0_2.6.0	01/2022	i.MX 8ULP EVK Beta release, i.MX 8M Mini, i.MX 8M Nano, i.MX 8M Plus, and i.MX 8M Quad GA release.
android-11.0.0_2.6.0	01/2022	Corrected a typo in Section 3.3.2.
android-12.0.0_1.0.0	03/2022	i.MX 8ULP EVK Beta release, i.MX 8M Mini, i.MX 8M Nano, i.MX 8M Plus, and i.MX 8M Quad GA release.
android-12.0.0_2.0.0	07/2022	i.MX 8ULP EVK Beta release, i.MX 8M Mini, i.MX 8M Nano, i.MX 8M Plus, and i.MX 8M Quad GA release.
android-12.1.0_1.0.0	10/2022	i.MX 8ULP EVK Beta release, i.MX 8M Mini, i.MX 8M Nano, i.MX 8M Plus, i.MX 8M Quad, i.MX 8QuadMax, and i.MX 8QuadXPlus GA release.

## 5 Legal information

### 5.1 Definitions

**Draft** — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

### 5.2 Disclaimers

**Limited warranty and liability** — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

**Right to make changes** — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

**Suitability for use** — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

**Applications** — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

**Terms and conditions of commercial sale** — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at <http://www.nxp.com/profile/terms>, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

**Export control** — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

**Suitability for use in non-automotive qualified products** — Unless this data sheet expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

**Translations** — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

**Security** — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at [PSIRT@nxp.com](mailto:PSIRT@nxp.com)) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

### 5.3 Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

**NXP** — wordmark and logo are trademarks of NXP B.V.

## Contents

<b>1</b>	<b>Preface</b> .....	<b>2</b>	3.3.12	Secure unlock .....	47
1.1	About This Document .....	2	3.4	Configurations in Linux/Android platform for security features .....	48
1.2	NXP security disclaimer .....	2	3.4.1	DM-Verity relationship with vbmeta .....	48
1.3	Conventions .....	2	3.4.2	Trusty OS Linux driver configuration .....	48
<b>2</b>	<b>Overview of i.MX Android Security</b>		3.4.3	Introductions of trusty based keymaster, gatekeeper, and secure storage proxy .....	48
	<b>Features</b> .....	<b>2</b>	3.4.4	Disabling GPU flat mapping .....	49
2.1	Introduction of security-related hardware modules .....	2	<b>4</b>	<b>Revision History</b> .....	<b>50</b>
2.2	Trusty OS security recommendations for i.MX 8 SoC configuration .....	4	<b>5</b>	<b>Legal information</b> .....	<b>52</b>
2.2.1	Secure CSU configuration .....	4			
2.2.2	Secure TZASC configuration .....	5			
2.2.3	Secure OCRAM configuration .....	5			
2.2.4	Secure RDC configuration .....	6			
2.2.5	Secure AIPSTZ configuration .....	6			
2.2.6	SCU/SCFW .....	7			
2.3	i.MX Android security framework .....	7			
<b>3</b>	<b>Customization work on i.MX Android</b>				
	<b>Security Features</b> .....	<b>8</b>			
3.1	Verifying images with HAB .....	8			
3.1.1	Verifying images with AHAB .....	9			
3.1.2	Verifying images with HABv4 .....	19			
3.1.2.1	Get CST tool and keys configuration .....	19			
3.1.2.2	Sign bootloader images .....	20			
3.1.2.3	Sign the MCU firmware .....	25			
3.1.2.4	Close the chip .....	27			
3.2	Configurations on TEE .....	28			
3.2.1	Memory region configuration in ATF .....	28			
3.2.2	Basic file and folder construction for Trusty OS .....	30			
3.2.3	Applying new build target in Trusty OS .....	30			
3.2.4	Adding unit tests in Trusty OS and adding CAAM self-tests in Trusty OS .....	31			
3.2.5	Modifying the console port for Trusty OS .....	32			
3.2.6	Configuring the related TA services .....	32			
3.2.7	Specifying apploader encryption and sign keys .....	33			
3.3	Configurations in U-Boot for security .....	34			
3.3.1	Overview of security features in U-Boot .....	35			
3.3.2	Generating and fusing the eMMC RPMB key .....	35			
3.3.3	Generating keys to sign and verify images .....	37			
3.3.4	Bypass vbmeta/lock check for development purposes .....	38			
3.3.5	Changing the value of the rollback index in images .....	39			
3.3.6	Programming the attestation key .....	39			
3.3.7	Programming the hardware identifiers .....	42			
3.3.8	Provisioning Widevine L1 keybox .....	43			
3.3.9	Changing the way to store lock status and/or rollback index .....	44			
3.3.10	Choosing to boot a specific slot .....	46			
3.3.11	Disabling development options in U-Boot .....	46			

Please be aware that important notices concerning this document and the product(s) described herein, have been included in section 'Legal information'.