NXP Semiconductors
User's Guide

Document Number: ASUG
Rev. P9.0.0_2.1.1-AUTO-ga, 06/2019

i.MX Android™ Security User's Guide

Contents

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.

This guide can be applied to the following platforms:

- i.MX 8QuadMax MEK with Android Auto platform running on it.
- i.MX 8QuadXPlus MEK with Android Auto platform running on it.

Trusty OS is not enabled on the released images of other platforms currently. Content in this document can be partly used on them.

1.2 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.
- 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 contains 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 8QuadXPlus	i.MX 8M Quad and 8M Mini
CAAM	Y	Y
TZASC	N	Y
xRDC	Y	N
AHAB	Y	N
HABv4	N	Y
SCU	Y	N
SECO	Y	N
eMMC	Y	Y

2.2 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.

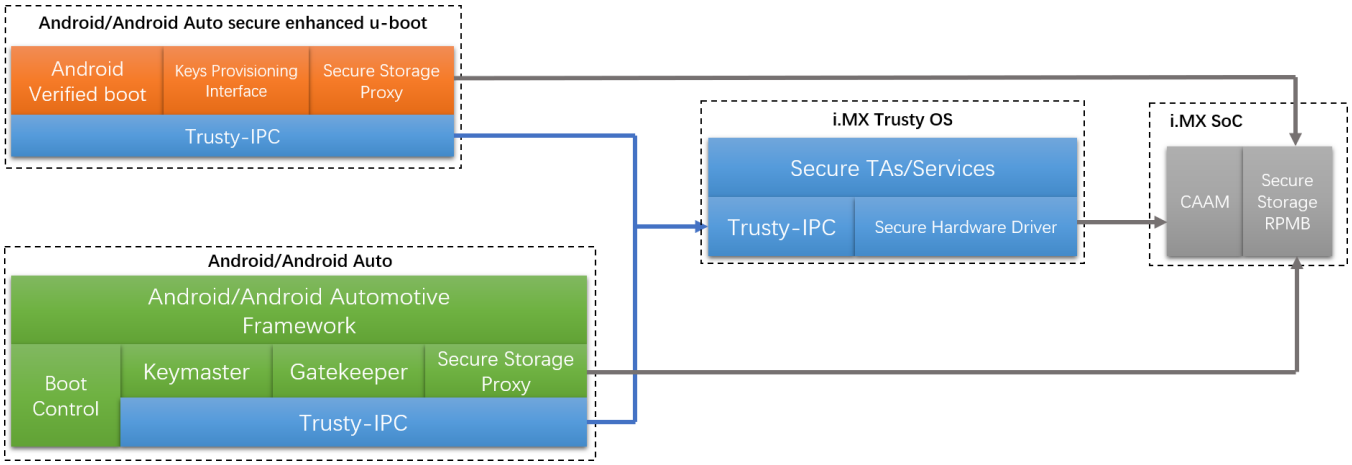

Figure 1. i.MX Android security framework

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

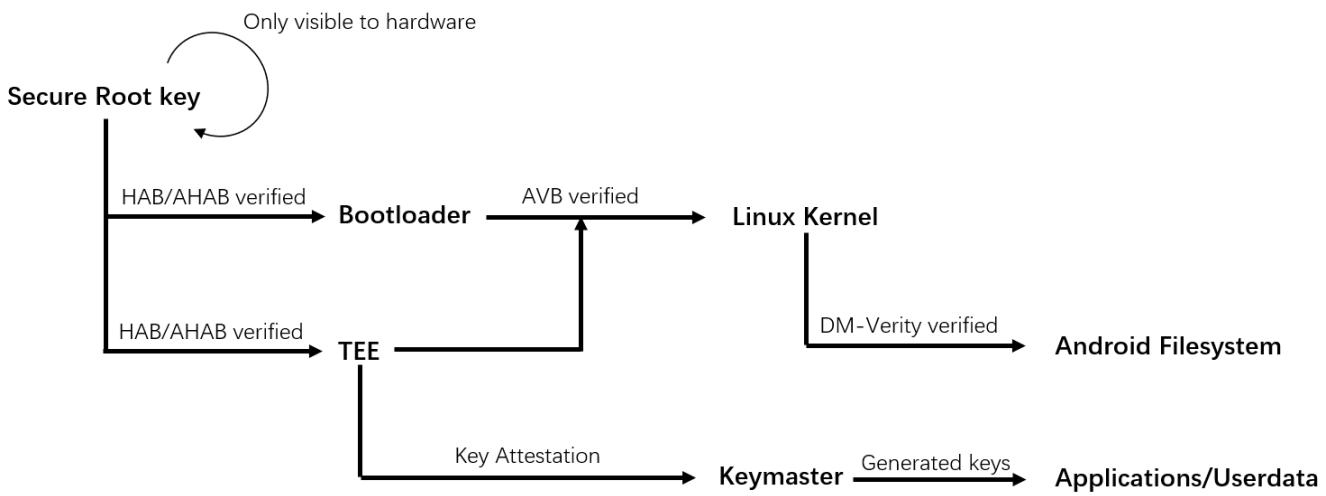

Figure 2. i.MX Android security trust chain

Secure root key is programed 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 How to use HAB to verify images

This part only describes AHAB, which is used in the i.MX 8 family devices. It does not describe verifying images with HABv4.

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 and store them in one file. AHAB can recognize the format of "container" and verify the images in a container. The i.MX chip is in open stage by default, so failure of this verification does not block the boot process. Detailed information about "container" can be found in the Reference Manual of specific chips. From the Reference Manual, it is known that the hash values of multiple firmwares and softwares are stored in the container header. The container signing process described below embeds an SRK table in the container and signs the container. The contents introduced in the signing process are used to verify the container during the boot time.

In Android Auto images for mek_8qm and mek_8qxp, a "container" is used to organize the SECO firmware, SCU firmware, Cortex-M4 software, Trusty OS, and U-Boot.

SPL is enabled for the U-Boot to be flashed to the board. Three containers are used in this condition. Containers are distinguished by their order of being loaded. The first container only has SECO firmware in it, and this container is provided as a binary file by NXP. It is signed by the key owned by NXP. In the Android build process, the second container is appended after the first container in one image file. The second container contains SCU firmware, a Cortex-M4 image, and SPL. The third container has ATF, Trusty OS, and U-Boot proper in it. The second and third containers are constructed when building the Android platform, and they are not signed by default.

SPL is not enabled for the U-Boot to be used by UUU. It is loaded to the RAM through the USB port by UUU. Before being executed, it is also verified by AHAB. Two containers are used in this condition. In a similar fashion to the way the images are flashed to boards, the container containing the SECO firmware is the first one to be loaded and verified. U-Boot is contained in the second container. By default, the second container is not signed in the Android build process.

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
   ```

3. Generate AHAB SRK tables and efuse hash.
   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
   ```

   After the command is executed successfully, the SRK table and its SHA512 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 SHA512 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.

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 the corresponding defconfig files in U-Boot code. They are not enabled by default. For Android Auto platform, the files are as follows:

```
imx8qm_mek_androidauto_trusty_defconfig
imx8qm_mek_androidauto2_trusty_defconfig
imx8qm_mek_android_uuu_defconfig
imx8qxp_mek_androidauto_trusty_defconfig
imx8qxp_mek_androidauto2_trusty_defconfig
imx8qxp_mek_android_uuu_defconfig
```

   "mkimage_im8" is used to construct containers. It outputs the layout information of a container on standard output when constructing it. Android build system redirects it to `/dev/null`. To remove this redirection, make the following modification on the repository in the directory of `${MY_ANDROID}/ device/fsl/`.

```
        diff --git a/imx8q/mek_8q/AndroidUboot.mk b/imx8q/mek_8q/AndroidUboot.mk
        index 518af15..f7256e7 100644
        --- a/imx8q/mek_8q/AndroidUboot.mk
        +++ b/imx8q/mek_8q/AndroidUboot.mk
        @@ -127,7 +127,7 @@ define build_imx_uboot
                fi; \
                cp $(UBOOT_OUT)/tools/mkimage $(IMX_MKIMAGE_PATH)/imx-mkimage/$
$MKIMAGE_PLATFORM/mkimage_uboot; \
                $(MAKE) -C $(IMX_MKIMAGE_PATH)/imx-mkimage/ clean; \
                - $(MAKE) -C $(IMX_MKIMAGE_PATH)/imx-mkimage/ SOC=$$MKIMAGE_PLATFORM $
$FLASH_TARGET 1&gt;/dev/null || exit 1; \
                + $(MAKE) -C $(IMX_MKIMAGE_PATH)/imx-mkimage/ SOC=$$MKIMAGE_PLATFORM $
$FLASH_TARGET || exit 1; \
                if [ "$(PRODUCT_IMX_CAR)" != "true" ] || [ `echo $(2) | rev | cut -d
'-' -f1` == "uuu" ] || [ "$(strip $(2))" == "imx8qm-xen-dom0" ]; then \
                   cp $(IMX_MKIMAGE_PATH)/imx-mkimage/$$MKIMAGE_PLATFORM/flash.bin $
(PRODUCT_OUT)/u-boot-$(strip $(2)).imx; \
                else \
        @@ -140,7 +140,7 @@ define build_imx_uboot
                cp $(UBOOT_OUT)/spl/u-boot-spl.bin $(PRODUCT_OUT)/spl-$(strip
$(2)).bin; \
                cp $(UBOOT_OUT)/tools/mkimage $(IMX_MKIMAGE_PATH)/imx-mkimage/$
$MKIMAGE_PLATFORM/mkimage_uboot; \
                $(MAKE) -C $(IMX_MKIMAGE_PATH)/imx-mkimage/ clean; \
                - $(MAKE) -C $(IMX_MKIMAGE_PATH)/imx-mkimage/ SOC=$$MKIMAGE_PLATFORM $
$FLASH_TARGET 1&gt;/dev/null || exit 1; \
                + $(MAKE) -C $(IMX_MKIMAGE_PATH)/imx-mkimage/ SOC=$$MKIMAGE_PLATFORM $
$FLASH_TARGET || exit 1; \
                cp $(IMX_MKIMAGE_PATH)/imx-mkimage/$$MKIMAGE_PLATFORM/u-boot-xen-
container.img $(PRODUCT_OUT)/bootloader-$(strip $(2)).img; \
                rm $(PRODUCT_OUT)/u-boot-$(strip $(2)).imx; \
            fi;
```

   When building the Android images, save the log information of the build system. For example, execute the following command:

```
$ make -j12 | tee make_android.txt
```

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

   After Android Auto images for i.MX 8QuadMax MEK and i.MX 8QuadXPlus MEK are built, the following image files need to be signed.

Table 2. Image files for i.MX 8QuadMax and i.MX 8QuadXPlus

Image file	Remark
spl-imx8qm.bin	Two containers, one is signed by NXP, and one is not signed.

Table continues on the next page...

Table 2. Image files for i.MX 8QuadMax and i.MX 8QuadXPlus (continued)

Image file	Remark
bootloader-imx8qm.img	One container, not signed.
u-boot-imx8qm-mek-uuu.imx	Two containers, one is signed by NXP, and one is not signed.
spl-imx8qxp.bin	Two containers, one is signed by NXP, and one is not signed.
bootloader-imx8qxp.img	One container, not signed.
u-boot-imx8qxp-mek-uuu.imx	Two containers, one is signed by NXP, and one is not signed.
bootloader-imx8qm-xen.img	One container, not signed.

In preceding files, `spl-imx8q*` and `bootloader-imx8q*.img` are flashed to the board. `u-boot-imx8q*-mek-uuu.imx` is used by UUU.

For images to be flashed to boards, `spl-imx8q*.bin` is composed of two containers and one is signed by NXP. The third container is in `bootloader-imx8q*.img`, so each file needs to be signed one time.

For images to be used by UUU, `u-boot-imx8q*-mek-uuu.imx` is composed of two containers and one is signed by NXP, so it needs to be signed one time.

5. Get the layout information of containers in a file.
   For `make_android.txt` newly generated, execute the following command:

```
$ grep "CST: CONTAINER" ./make_android.txt
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
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
CST: CONTAINER 0 offset: 0x0
CST: CONTAINER 0: Signature Block: offset is at 0x90
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
CST: CONTAINER 0 offset: 0x400
CST: CONTAINER 0: Signature Block: offset is at 0x590
CST: CONTAINER 0 offset: 0x400
CST: CONTAINER 0: Signature Block: offset is at 0x610
```

There are 18 lines of output. Every two lines describe one container constructed in the build process. Nine containers are constructed when building an Android Auto image.

This output is related to the makefile variable "TARGET_BOOTLOADER_CONFIG" defined in `${MY_ANDROID}/device/fsl/imx8q/mek_8q/BoardConfig.mk`. From this file, the final value of this variable for Android Auto with Cortex-M4 image is:

```
imx8qm:imx8qm_mek_androidauto_trusty_defconfig \
imx8qxp:imx8qxp_mek_androidauto_trusty_defconfig \
imx8qm-xen:imx8qm_mek_androidauto_xen_dual_defconfig \
imx8qm-xen-dom0:imx8qm_mek_spl_defconfig \
imx8qm-mek-uuu:imx8qm_mek_android_uuu_defconfig \
imx8qxp-mek-uuu:imx8qxp_mek_android_uuu_defconfig
```

There are six targets. The table below lists the files generated and the number of containers constructed when building each of these targets. The offset information of containers in these files corresponds to the preceding Android build system output information except "u-boot-imx8qm-xen-dom0.imx". Other files are direct output of "mkimage_imx8" while "u-boot-imx8qm-xen-dom0.imx" is not, so the corresponding lines of the offset output does not represent the real offset of the containers. This has no impact on customers who do not use XEN.

Table 3. Files generated and constructed container number

Target	File(s) generated	Constructed container number
imx8qm	bootloader-imx8qm.img spl-imx8qm.bin	2
imx8qxp	bootloader-imx8qxp.img spl-imx8qxp.bin	2
imx8qm-xen	bootloader-imx8qm-xen.img spl-imx8qm-xen.bin	1
imx8qm-xen-dom0	u-boot-imx8qm-xen-dom0.imx	2
imx8qm-mek-uuu	u-boot-imx8qm-mek-uuu.imx	1
imx8qxp-mek-uuu	u-boot-imx8qxp-mek-uuu.imx	1

Then you can get the container offset information for each file to be signed as listed in the following table.

Table 4. Container offset information

Files having container to be signed	Container offset in the file	Container signature block offset
bootloader-imx8qm.img	0x0	0x190
spl-imx8qm.bin	0x400	0x610
bootloader-imx8qxp.img	0x0	0x190
spl-imx8qxp.bin	0x400	0x610
u-boot-imx8qm-mek-uuu.imx	0x400	0x590
u-boot-imx8qxp-mek-uuu.imx	0x400	0x610

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 as an input file when it is executed. CSF examples are in the directory of `${MY_ANDROID}/vendor/nxp-opensource/uboot-imx/doc/imx/ahab/csf_examples/`. We copy one `cst_uboot_atf.txt` to CST `release/linux64/bin/`.

   Make some changes to `cst_uboot_atf.txt` just copied based on the image to sign. For example, sign `u-boot-imx8qm-mek-uuu.imx`. 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_crt.pem"
+Source = "../../crts/SRK1_sha384_secp384r1_v3_usr_crt.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, since `uuu_imx_android_flash` script flashes images based on their names.

Based on the description of signing `u-boot-imx8qm-mek-uuu.imx`, sign all the other images that need to be signed.

Images are signed now. When booting with signed images, 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 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). Execute the following command to dump the SRK hash value:

```
$ od -t x4 SRK_1_2_3_4_fuse.bin
0000000 d436cc46 8ecccda9 b89e1601 5fada3db
0000020 d454114a b6cd51f4 77384870 c50ee4b2
0000040 a27e5132 eba887cf 592c1e2b bb501799
0000060 ee702e07 cf8ce73e fb55e2d5 eba6bbd2
```

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, we 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
```

   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 0x8ecccda9
=> 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 0x8ecccda9
=> 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
```

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 programed, 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 programed, 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.

3.2 Configurations on TEE

3.2.1 Memory region configuration in ATF

The TEE binary is loaded to DRAM at the address of `$BL32_BASE` by SPL. By default, the load address `$BL32_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, `$BL32_BASE` can be configured as 0xFF000000 and `$BASE_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 5. 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 fully 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 6. 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.

Table continues on the next page...

Table 6. Basic file structure of i.MX Trusty OS (continued)

Folder name	Folder description
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 `rules.mk` 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, 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/uboot-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 are 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.

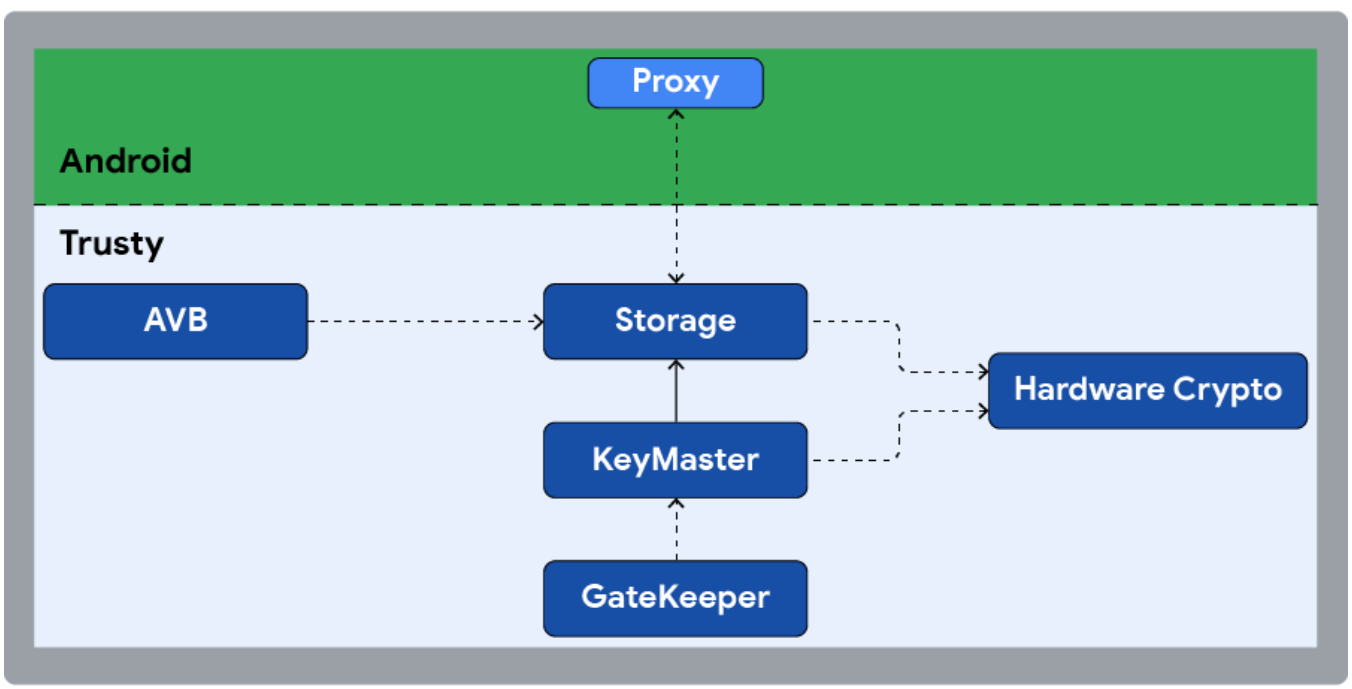

Figure 3. Relationship between TAs

- 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.

3.3 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. 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.

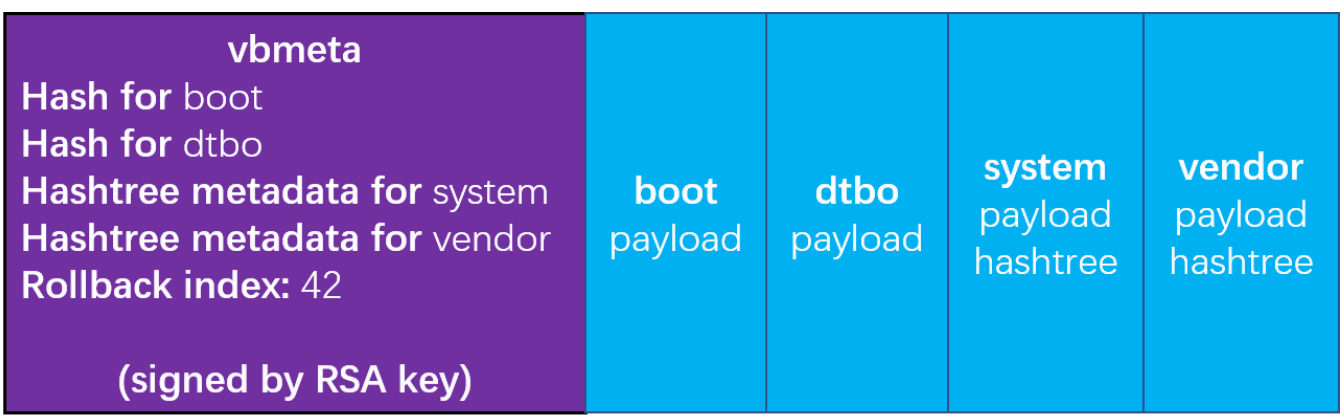

Figure 4. Relationship between vbmeta image and related images

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.

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 is 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. This secure key also needs to be saved in the format that only TEE can parse, so TEE can use this key to communicate with RPMB. This 256-bit secure key is used to sign and verify data transferred between eMMC RPMB and TEE.

The RPMB key can only be programed one time. The saved copy of RPMB key is encapsulated with CAAM, and CAAM 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.

Two ways are provided to set the RPMB key:

- Manually specify a 256-bit key and program it.
  a. A file containing the key needs to be generated. In the default key file "rpmb_key_test.bin", all 256 bits are zero. It can be generated with the following commands:

```
$ touch rpmb_key.bin
$ echo -n "RPMB" &gt; rpmb_key.bin
$ echo -n -e
'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' &gt;&gt; rpmb_key.bin
```

  The '\xHH' means 8-bit character whose value is the hexadecimal value 'HH'. You can replace above "00" with the key you want to set.

  b. Program the key with the file just generated.
  c. Make the board enter fastboot mode, and then execute the following commands on the host side:

```
$ fastboot stage rpmb_key.bin
$ fastboot oem set-rpmb-key
```

- Program a random key.
  Make the board enter fastboot mode, and then execute the following commands on the host side:

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

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

The two ways above program the key to eMMC fuse. A key blob is generated base on the key value and the blob is saved for TEE to use. In the default condition, this key blob is saved in the 16383rd block of BOOT1 partition in eMMC for i.MX 8QuadMax and i.MX 8QuadXPlus. The BOOT1 partition size of eMMC on i.MX 8QuadMax and i.MX 8QuadXPlus is 8 MB. The key blob is in the last block in BOOT1 partition. To prevent key blob from being tampered when the system is running, BOO1 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. The two macros are the same for i.MX 8QuadMax and i.MX 8QuadXPlus. Their definitions are as follows:

```
#define KEYSLOT_HWPARTITION_ID 2
#define KEYSLOT_BLKS 0x3FFF
```

"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. 0x3FFF equals to 16383 mentioned above. The definition of "KEYSLOT_BLKS" may need to change based on the eMMC capacity on customized boards.

For i.MX 8QuadMax, they are in the following file:

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

For i.MX 8QuadXPlus, they are in the following file:

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

3.3.3 Generating AVB key to sign and verify images

The vbmeta image is signed during the time of building Android platform. By default, it is signed with a test private key as follows:

```
${MY_ANDROID}/device/fsl/common/security/testkey_rsa4096.pem
```

Its corresponding public key is:

```
${MY_ANDROID}/device/fsl/common/security/testkey_public_rsa4096.bin.
```

The default algorithm used to sign the image is "SHA256_RSA4096".

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_ANDROID}/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" is recommended for i.MX 8QuadMax/8QuadXPlus, whose Cryptographic Acceleration and Assurance Module (CAAM) can help accelerate the hash calculation. We can keep it as default.

To use the private key just generated to sign the vbmeta image, make the following changes on the repository in `${MY_ANDROID}/device/fsl`.

```
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/fsl/common/security/testkey_rsa4096.pem
+BOARD_AVB_KEY_PATH := ${your-key-directory}/custom_rsa4096_private.pem
endif
TARGET_USES_MKE2FS := true
```

To enable U-Boot to verify image signature with the public key just generated, save the public key in TEE backed RPMB for Android Auto platform. 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 key just generated. If there is no change made to the private key used to sign vbmeta image, you still need to store the default public key with the following commands:

```
$ 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 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 two rollback index values in i.MX 8QuadMax and i.MX 8QuadXPlus Android Auto images after the dual-bootloader feature is enabled. One is for the image that contains U-Boot proper, and one is for vbmeta images and other images whose hash metadata is stored in the vbmeta image. In default condition, these two rollback index values are both zero.

When a version of images is to be released to fix a bug in previous version that makes previous images under potential attacks, it is recommended to increase the rollback index values by one compared to the previous version.

To change the rollback index value for the image that contains U-Boot proper, specify a variable named "BOOTLOADER_RBINDEX" for the `make` command to build the images.

To change the rollback index value for vbmeta image and other related images, specify a variable named "AVB_RBINDEX" for the `make` command to build the images.

For example, you can execute the following command to build Android images, change that `$(avb_rbindex)` and `$(bootloader_rbindex)` to the value you want to set.

```
make AVB_RBINDEX=$(avb_rbindex) BOOTLOADER_RBINDEX=$(bootloader_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).

Fastboot commands are provided to provision the attestation keys and certificates. Make sure that the secure storage is properly initialized for Trusty OS. Boot the board information 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.

3.3.7 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, AvbABSlotData *slot,
                            struct spl_image_info *spl_image)
```

In the following file:

```
${MY_ANDROID}/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_ANDROID}/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 and i.MX 8QuadXPlus.

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

3.3.8 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
```

3.3.9 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 1 second 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 8QuadMAX and i.MX 8QuadXPlus, make the following changes:

```
        diff --git a/configs/imx8qm_mek_androidauto_trusty_defconfig b/configs/
imx8qm_mek_androidauto_trusty_defconfig
        index 0a8c3cb..8150b2b 100644
        --- a/configs/imx8qm_mek_androidauto_trusty_defconfig
        +++ b/configs/imx8qm_mek_androidauto_trusty_defconfig
        @@ -40,7 +40,7 @@ CONFIG_USB_GADGET_DUALSPEED=y

         CONFIG_DM_GPIO=y
         CONFIG_DM_PCA953X=y
        -CONFIG_BOOTDELAY=1
        +CONFIG_BOOTDELAY=-2
         CONFIG_CMD_MMC=y
         CONFIG_DM_MMC=y
         CONFIG_MMC_IO_VOLTAGE=y
        diff --git a/configs/imx8qxp_mek_androidauto_trusty_defconfig b/configs/
imx8qxp_mek_androidauto_trusty_defconfig
        index 0611773..a424e31 100644
        --- a/configs/imx8qxp_mek_androidauto_trusty_defconfig
        +++ b/configs/imx8qxp_mek_androidauto_trusty_defconfig
        @@ -41,7 +41,7 @@ CONFIG_USB_GADGET_DUALSPEED=y

         CONFIG_DM_GPIO=y
         CONFIG_DM_PCA953X=y
        -CONFIG_BOOTDELAY=1
        +CONFIG_BOOTDELAY=-2
         CONFIG_CMD_MMC=y
         CONFIG_DM_MMC=y
         CONFIG_MMC_IO_VOLTAGE=y
```

- 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 i.MX 8QuadMAX and i.MX 8QuadXPlus, make the following changes:

```
        diff --git a/configs/imx8qm_mek_androidauto_trusty_defconfig b/configs/
imx8qm_mek_androidauto_trusty_defconfig
        index 0a8c3cb..bc6a97d 100644
        --- a/configs/imx8qm_mek_androidauto_trusty_defconfig
        +++ b/configs/imx8qm_mek_androidauto_trusty_defconfig
        @@ -120,4 +120,3 @@ CONFIG_NOT_UUU_BUILD=y
         CONFIG_SHA256=y
         CONFIG_SPL_MMC_WRITE=y
         CONFIG_DUAL_BOOTLOADER=y
        -CONFIG_APPEND_BOOTARGS=y
        diff --git a/configs/imx8qxp_mek_androidauto_trusty_defconfig b/configs/
imx8qxp_mek_androidauto_trusty_defconfig
        index 0611773..e501c40 100644
        --- a/configs/imx8qxp_mek_androidauto_trusty_defconfig
        +++ b/configs/imx8qxp_mek_androidauto_trusty_defconfig
        @@ -121,4 +121,3 @@ CONFIG_NOT_UUU_BUILD=y
         CONFIG_SHA256=y
         CONFIG_SPL_MMC_WRITE=y
         CONFIG_DUAL_BOOTLOADER=y
        -CONFIG_APPEND_BOOTARGS=y
```

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 Configuration of the RSA keys for DM-verity

RSA keys are used to sign the DM_verity table to produce a table signature. When verifying a partition, the table signature is validated first. This is done against a key on your boot image in a fixed location. Keys are typically included in the `/verity_key` directory.

The 2048-bit private RSA key that is used to sign the table is generated by OpenSSL, which is included in `${MY_ANDROID}/build/target/product/security/verity_private_dev_key`.

The RSA public key used for verification needs to be in mincrypt format. Converting an OpenSSL RSA public key to mincrypt format requires some modular operations and is not simply a binary format conversion. You can convert the PEM key using the pem2mincrypt tool. The public key is included in `${MY_ANDROID}/build/target/product/security/verity_key`.

You can change the default RSA key using the following commands:

```
    cd build/target/product/security/
    openssl genrsa -out verity_private_dev_key_tem 2048
    openssl pkcs8 -topk8 -inform PEM -in verity_private_dev_key_tem -
outverity_private_dev_key -outform PEM –nocrypt
    pem2mincrypt verity_private_dev_key_tem verity_key
```

NOTE

- Install libssl0.9.8 using the following command:

  ```
  $sudo apt-get install libssl0.9.8
  ```
- The tool pem2mincrypt's source code is under https://github.com/nelenkov/verity.

3.4.3 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";
    };
```

3.4.4 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.

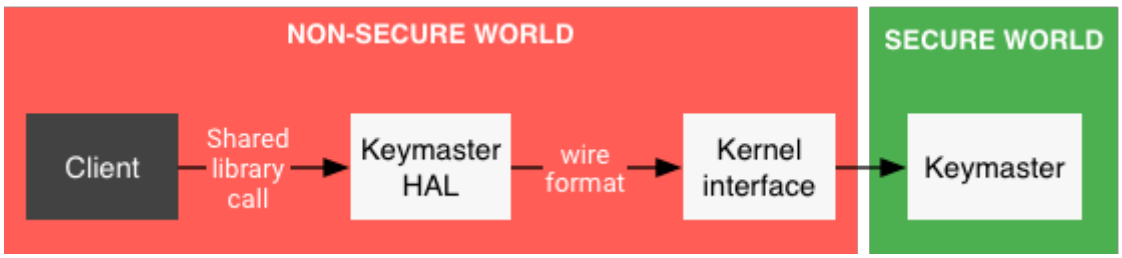

Figure 5. Relationship between keymaster HAL and keymaster TA

The trusty backed keymaster HAL 3.0 is designed for Android Pie 9 or later, which can not 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.

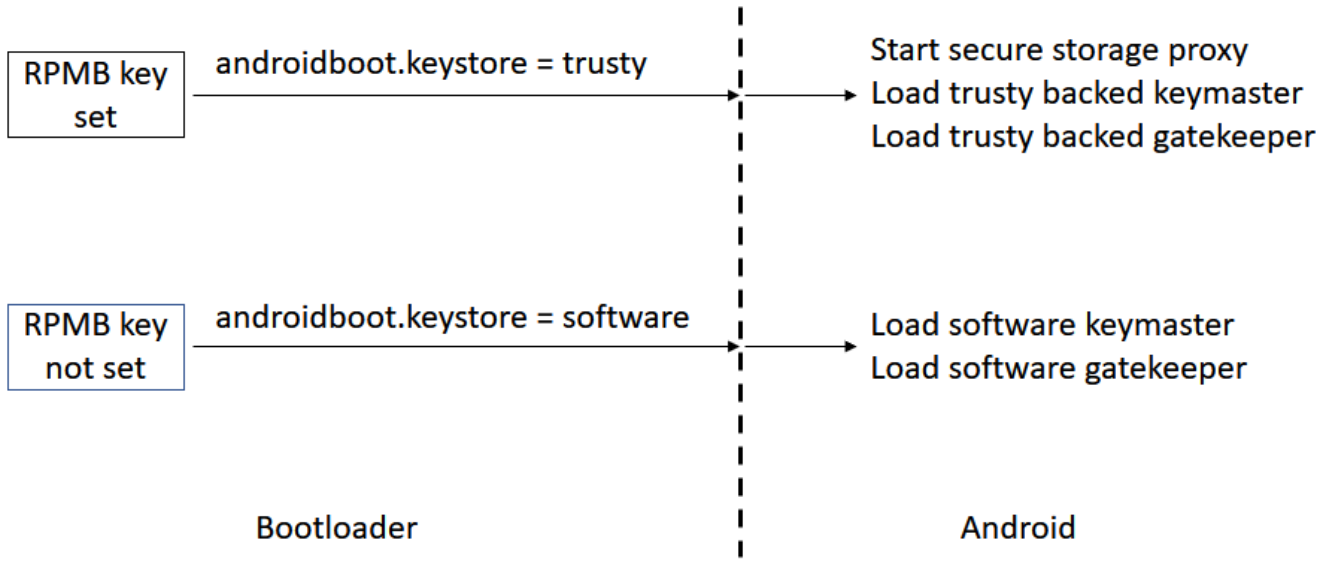

Figure 6. TEE and TA initialization and workflow

4 Revision History

Table 7. Revision history

Revision number	Date	Substantive changes
P9.0.0_2.1.0-AUTO-ga	04/2019	Initial release
P9.0.0_2.1.1-AUTO-ga	06/2019	i.MX 8QuadMax Automotive GA release

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer’s technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

Document Number ASUG
Revision P9.0.0_2.1.1-AUTO-ga, 06/2019