

---

# **Security Manual Kirkstone**

**PHYTEC Messtechnik GmbH**

**Oct 02, 2025**



# CONTENTS

<b>1</b>	<b>SECURiPhy Overview</b>	<b>3</b>
1.1	Short Crypto Refresher . . . . .	4
1.2	Recommended Security Requirements . . . . .	4
<b>2</b>	<b>Enable Security</b>	<b>7</b>
2.1	Distro ampliphy-secure and ampliphy-vendor-secure . . . . .	7
2.2	Distro ampliphy-provisioning and ampliphy-vendor-provisioning . . . . .	7
2.3	The different machines for TI K3 controller . . . . .	7
2.4	Enable Security Features in your own Distro . . . . .	8
<b>3</b>	<b>Secure Boot</b>	<b>9</b>
3.1	Chain of Trust . . . . .	9
3.2	Boot Process . . . . .	9
3.3	Image Signing . . . . .	11
3.4	Flattened Image Tree (FIT) . . . . .	13
3.5	Configuration Class for Signing images . . . . .	13
3.6	Building a signed image . . . . .	15
<b>4</b>	<b>Activate Secure Boot on the Device</b>	<b>17</b>
4.1	eMMC Boot Partition to Enable Boot . . . . .	17
4.2	Activate Secure Boot for NXP SOC . . . . .	18
4.3	Activate Secure Boot for TI K3 SOC . . . . .	20
4.4	Next Steps after Activation of Secure Boot . . . . .	22
4.5	Key Revocation . . . . .	22
<b>5</b>	<b>Kernel Module Signing</b>	<b>25</b>
5.1	Enable kernel module signing facility . . . . .	25
<b>6</b>	<b>Device Tree Overlay and Secure Boot</b>	<b>27</b>
6.1	Device Tree Overlay for i.MX6UL and i.MX6 . . . . .	27
6.2	Device Tree Overlay for the other NXP source . . . . .	27
<b>7</b>	<b>Secure Key Storage</b>	<b>29</b>
7.1	NXP i.MX CAAM . . . . .	29
7.2	Trusted Execution Environment: OP-TEE . . . . .	30
7.3	Trusted Platform Module (TPM) 2.0 . . . . .	32
7.4	Kernel Key Retention Service for filesystem Encryption . . . . .	32
7.5	Cryptographic Token Interface PKCS#11 . . . . .	33
<b>8</b>	<b>Secure Storage</b>	<b>35</b>
8.1	Filesystem with Integrity vs Authenticated filesystem . . . . .	36

8.2	Requirements for Filesystem Encryption . . . . .	36
8.3	Boot Process Flow . . . . .	36
8.4	Starting the Build Process . . . . .	36
8.5	Setup Secure Storage on your Device . . . . .	37
<b>9</b>	<b>Hardening of the System</b>	<b>41</b>
<b>10</b>	<b>Physical security</b>	<b>43</b>
10.1	Secure JTAG . . . . .	43
10.2	Disable Serial Downloader . . . . .	46
10.3	Force Internal Boot . . . . .	47
10.4	Disable Boot from External Memory . . . . .	47
<b>11</b>	<b>Keys and Certificates Management</b>	<b>49</b>
11.1	Public Key Infrastructure Tree (PKI tree) . . . . .	49
11.2	PHYTEC Development Keys (phytec-dev-ca) . . . . .	49
11.3	Create Your Own PKI Tree . . . . .	51
11.4	Create NXP AHAB / HABV4 Keys . . . . .	51
11.5	Create TI K3 keys . . . . .	51
11.6	Create Kernel FIT-Image key . . . . .	58
11.7	Create Kernel Module Signing key . . . . .	58
11.8	Create rauc update certificates . . . . .	58

Security Manual	
Document Title	Security Manual Kirkstone
Document Type	Security Manual
Last modified	2025/08/27
Is Branch of	Security Manual

Compatible BSPs	BSP Release Type	BSP Release Date	Security Support Status
BSP-Yocto-Ampliphy-i.MX6-PD22.1.0	Major	14.12.2022	full
BSP-Yocto-Ampliphy-i.MX6-PD22.1.1	Minor	20.06.2023	full
BSP-Yocto-Ampliphy-i.MX6UL-PD22.1.0	Major	11.08.2022	full
BSP-Yocto-Ampliphy-i.MX6UL-PD22.1.1	Minor	23.05.2023	full
BSP-Yocto-NXP-i.MX8MM-PD23.1.0	Major	12.12.2023	full
BSP-Yocto-NXP-i.MX8MP-PD23.1.0	Major	12.12.2023	full
BSP-Yocto-Ampliphy-AM62x-PD23.2.0	Major	28.09.2023	partly
BSP-Yocto-Ampliphy-AM62Ax-PD23.1.0	Major	28.09.2023	partly
BSP-Yocto-Ampliphy-AM64x-PD23.2.0	Major	28.09.2023	partly
BSP-Yocto-Ampliphy-AM68x-PD24.1.0	Major	12.03.2024	keywriter

This manual applies to all Kirkstone based PHYTEC releases.

PHYTEC's Yocto distribution Ampliphy (former Yogurt) supports different Security mechanism. The security features have impact to the bootloader, the Linux kernel, Device Tree, and root filesystem. This manual describes how Security features is used and implemented on various PHYTEC platforms. Note, that different modules use different bootloaders and flash storage devices, which affects the way things are handled. Make sure to read the correct sections fitting your platform.
















#### Note

This manual contains machine-specific paths and variable contents. Make sure you are using the correct machine and device names for your application when executing any commands.



## SECURIPHY OVERVIEW

SECURiPhy is the PHYTEC secure linux distribution and a part of the security packages phyKNOX.

SECURITY PACKAGES	Basic PRE-Configured For free	Advanced FULLY-configured Ready to use secure	Maintained FULLY-CYCLIC audited Security for the entire lifecycle	Individually Adaptable to your product
 Software Lifecycle Management			 CYCLIC Maintenance	Selectable
 Common Vulnerabilities and Exposures (CVE) Correction			 CYCLIC Review	Selectable
 Device Initialisation Client Certificates, Secure Boot		 Onboarded Device	 Onboarded Device	Selectable with / without
 Secure management and update your IoT devices	 Update-Basic	 Update-Advanced	 Update-Advanced	
 SECURiPhy-Distro	 SECURiPhy-Basic All Linux security tools ready for configuration	 SECURiPhy-Advanced With professional support	 SECURiPhy-Maintained audited security for the future	Levels selectable: SL1 - SL3 (according to IEC 62443 standard)
 Consulting / Support	1 hour <b>FOR FREE</b>	 Cost per hour	 Cost per hour	 Cost per hour

With increasing digitization and networking, the protection of embedded systems against unauthorized access and targeted attacks is more important than ever. Guaranteeing this type of security, along with functional security, is a major challenge in electronics design. PHYTEC supports you in minimizing risks by considering security requirements during the development of our hardware and board support packages. On top of these deployment-ready solutions, we support you with individual project consulting on complex security principles.

Security is a process encompassing all parts of a device and all development phases of its lifetime.

## 1.1 Short Crypto Refresher

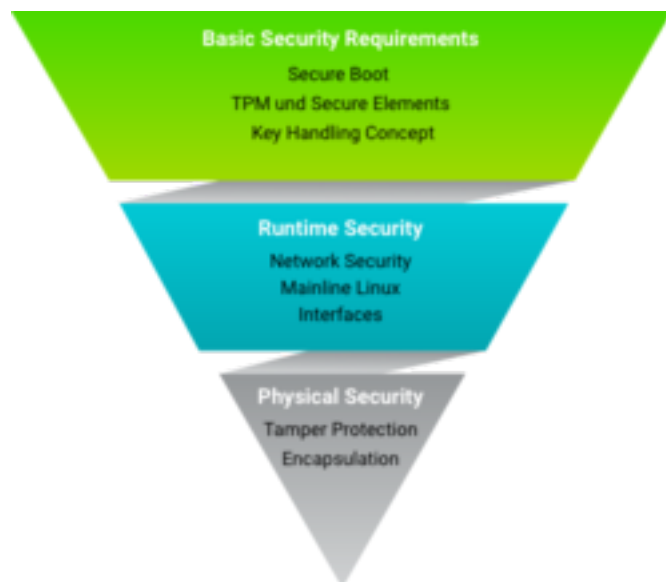
Function	Description
Symmetric cryptography	The same key for encryption or decryption
Public key cryptography	Two mathematically dependent keys for encryption or decryption. The public key is used for encryption while the private key is used for decryption.
Hash	One-way function, fixed output size (SHA*)
HMAC	Data authentication using hash and shared secret
Signature	Data authentication using public-key cryptography (keys & certificates, RSA & ECDSA)
Unauthenticated encryption	Attackers can't read private data but could modify it (AES-CBC, AES-XTS, ...)
Authenticated encryption	Attacker can't read private data and modification is detected (AEAD: AES GCM, AEGIS)
Trusted Keys	Symmetric key with variable length is a key type of the existing kernel keyring service. Require the availability of a Trust Source for greater security like a TPM, NXP CAAM or TEE
Encrypted Keys	Symmetric key with variable length is a key type of the existing kernel keyring service.

## 1.2 Recommended Security Requirements

As of the writing of this manual, recommendations apply to key lengths, certificates, and hash values. These recommendations come from BSI (Bundesamt für Sicherheit in der Informationstechnik) and NIST (National Institute of Standards and Technology).



In the technical and connected world, it is important to build a “security by design” approach that thwarts intrusion into your product, data, and intellectual property at multiple levels.



Here are the Security features from the Standard BSP.

- **Basic Security**

Basic Security is the fundament of the security measures implementations and includes support for basic modules such as:

- True Number Generator and Cryptographic support
- Secure Boot
- Secure Key Storage and Usage
- Secure Storage
- Secure Updates

- **Parts from Access Control**

Access Control regulates the access of users and services to the device and components in the device according to the least privilege access principle.

- Secure Console
- Secure Shell
- User and Role Management

- **Provisioning**

Provisioning includes the activation of hardware security features like Secure Boot and the generation of specific keys and X509 certificates on the device in secure manufacturing like the PHYTEC secure production area.

More Additional Security Features (not part of this BSP)

- **Network Security**

Network Security enables secure connections to connected devices or servers via Ethernet, WLAN, and LTE, but also secures access to the device from outside.

- Remote Access
- Server, Cloud Integration Tools

- Intrusion Protection
  - Firewall
  - Container
- **Interface Security**

Interface Security secures the interfaces against third-party access and enables the secure connection of intended devices.

  - USB
  - Field Bus
- **Hardening**

Hardening refers to the reduction of software components and kernel configuration to a necessary minimum.
- **Physical Security**

Physical Security secures the device from direct physical access to protect the corresponding application and data from external access.

  - Secure Debug
  - Tamper Protection
  - Housing
  - Encapsulation of the circuit board

## ENABLE SECURITY

### Note

Distro `ampliphy-secure`, `ampliphy-vendor-secure`, `ampliphy-provisioning` and `ampliphy-vendor-provisioning` are sample Yocto distros like `ampliphy` with additional security pre-configurations. Additional security measurements for production usage are necessary and depend on your threat model. PHYTEC services can support your implementation.

## 2.1 Distro `ampliphy-secure` and `ampliphy-vendor-secure`

The distro `ampliphy-secure` or `ampliphy-vendor-secure` with the `phytec-security-image` is an example of a production image with secure-update support. The `phytec-security-image.rootfs.wic` or `phytec-security-image.rootfs.partup` can boot only from an eMMC!

For devices based on the TI K3 controller (AM6 series) the `MACHINE` variable in the `$BUILDDIR/conf/local.conf` should be set to HS-SE machine variant.

## 2.2 Distro `ampliphy-provisioning` and `ampliphy-vendor-provisioning`

The distro `ampliphy-provisioning` or `ampliphy-vendor-provisioning` with the `phytec-provisioning-image` is for the production or the first initialization of your device based on a NXP controller in a secure area. The `phytec-provisioning-image.rootfs` can boot directly from an SD card to a Kernel with a minimal `initramfs` to

- install the `phytec-security-image.rootfs` as `wic` or `partup` to the eMMC
- initialize the secure key storage on the device
- initialize the secure storage on the device

For devices based on the TI K3 controller (AM6 series) use the distro `ampliphy` and build the `phytec-headless-image` to boot from sd-card. The `MACHINE` variable in the `$BUILDDIR/conf/local.conf` should be set to the HS-SE machine variant.

## 2.3 The different machines for TI K3 controller

The HS-SE machine variant is the machine with secure boot enabled, so it will be built with signed bootloaders. For the TI K3 controller exist different machines for

- General Purpose (GP): The device is not capable of secure operation
- High Secure - Field Securable (HS-FS): is the state of a K3 device before it has been eFused with customer security keys.

- High Secure - Security Enforced (HS-SE): devices enforce an authenticated boot flow for secure boot.

Board	HS-FS device	HS-SE device
phyCORE-AM62Ax	phyboard-lyra-am62axx-2	
phyCORE-AM62x	phyboard-lyra-am62xx-3	phyboard-lyra-am62xx-4
phyCORE-AM64x	phyboard-electra-am64xx-2	phyboard-electra-am64xx-3
phyCORE-AM68x	phyboard-izar-am68x-3	phyboard-izar-am68x-4

For NXP controller based boards do not exists different machines for devices with activated and not activated Secure boot, because signed images can be booted independence of the device state.

## 2.4 Enable Security Features in your own Distro

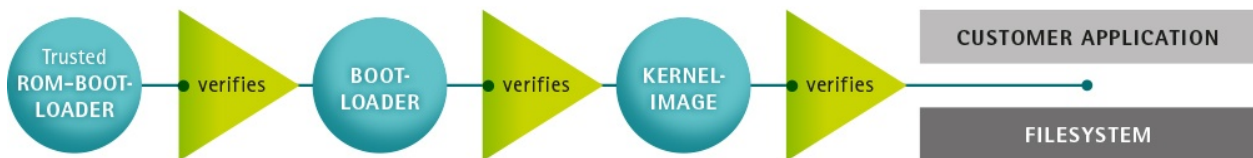
Activate the following DISTRO\_FEATURES in your distribution

DISTRO_FEATURES	Description
secureboot	for building a signed bootloader and kernel FIT-Image
securestorage	All necessary tools and configurations for file encryption and integrity initialization on the board
protectionsshield	<ul style="list-style-type: none"> <li>• Three levels low, medium, and high</li> <li>• Four examples: users root, phyadmin, phyuser, phyread</li> <li>• Password protection for bootloader and kernel serial and ssh</li> </ul>
hardening	Example kernel reduction for machine features
kernelmodsign	Enabled Linux kernel module signing, so only modules signed with a specific key can be loaded.
update	Activate rauc A/B update system

## SECURE BOOT

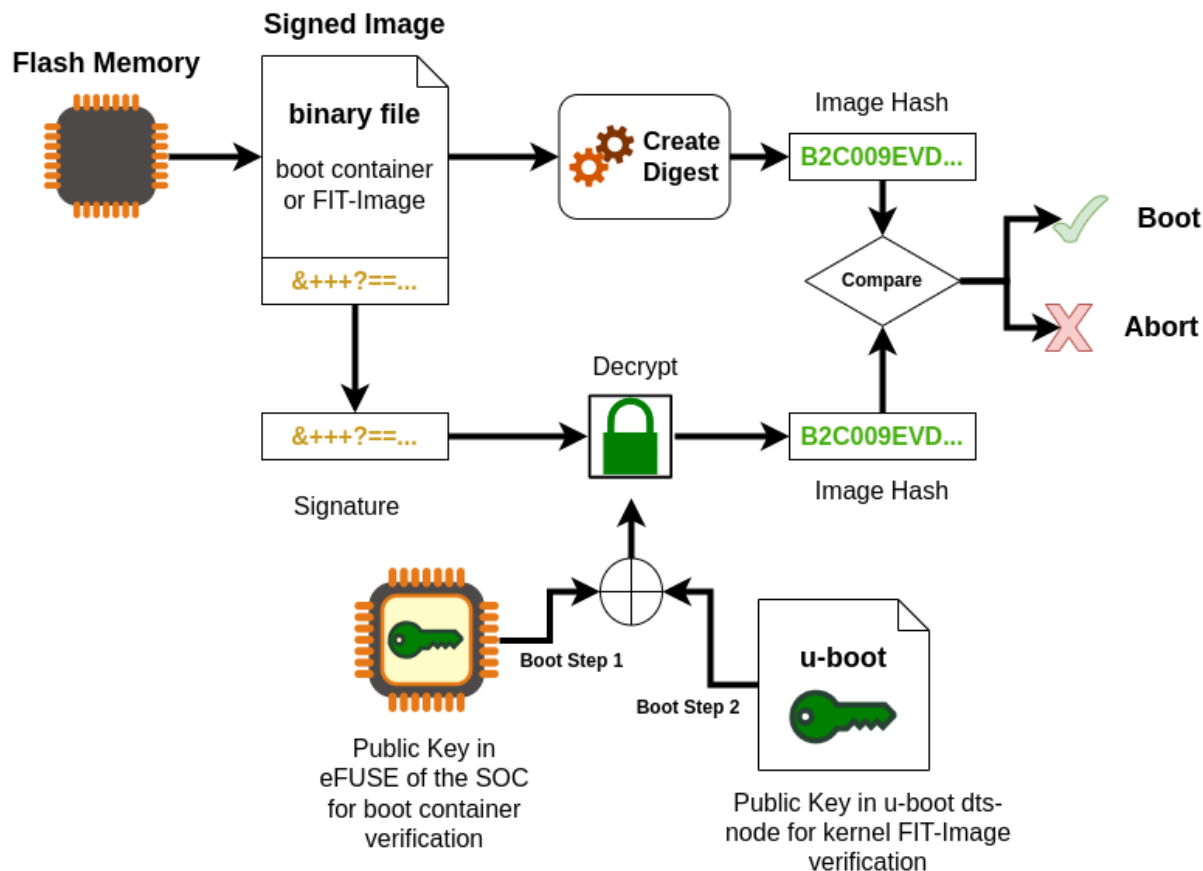
### 3.1 Chain of Trust

Secure boot is used to ensure that only trustworthy, signed software can be executed on the controller. This is the first stage of the Chain-of-Trust. With the Chain-of-Trust, signed programs are always started by other previously verified programs. This ensures that even the end application is at the highest layer of trustworthiness.



### 3.2 Boot Process

The boot process differs between the SOC and the vendor of the SOC's. The main boot process is the following



1. The Trusted ROM-bootloader, which is part of the SOC, verifies the boot container (U-boot spl, ATF, firmware, op-tee) with the internal unit (**Boot Step 1**)

SOC	Unit	Boot mode
NXP i.MX6, i.MX6UL, NXP i.MX8M Mini/Plus	NXP HABV4	Single: Cortex-A
NXP i.MX93, i.MX91, i.MX95	NXP AHAB + EdgeLock secure enclave	LPboot: M33 Single: A35 -> M33 DUAL: M33 + A35
TI AM62x, AM64x, AM68x	R5	

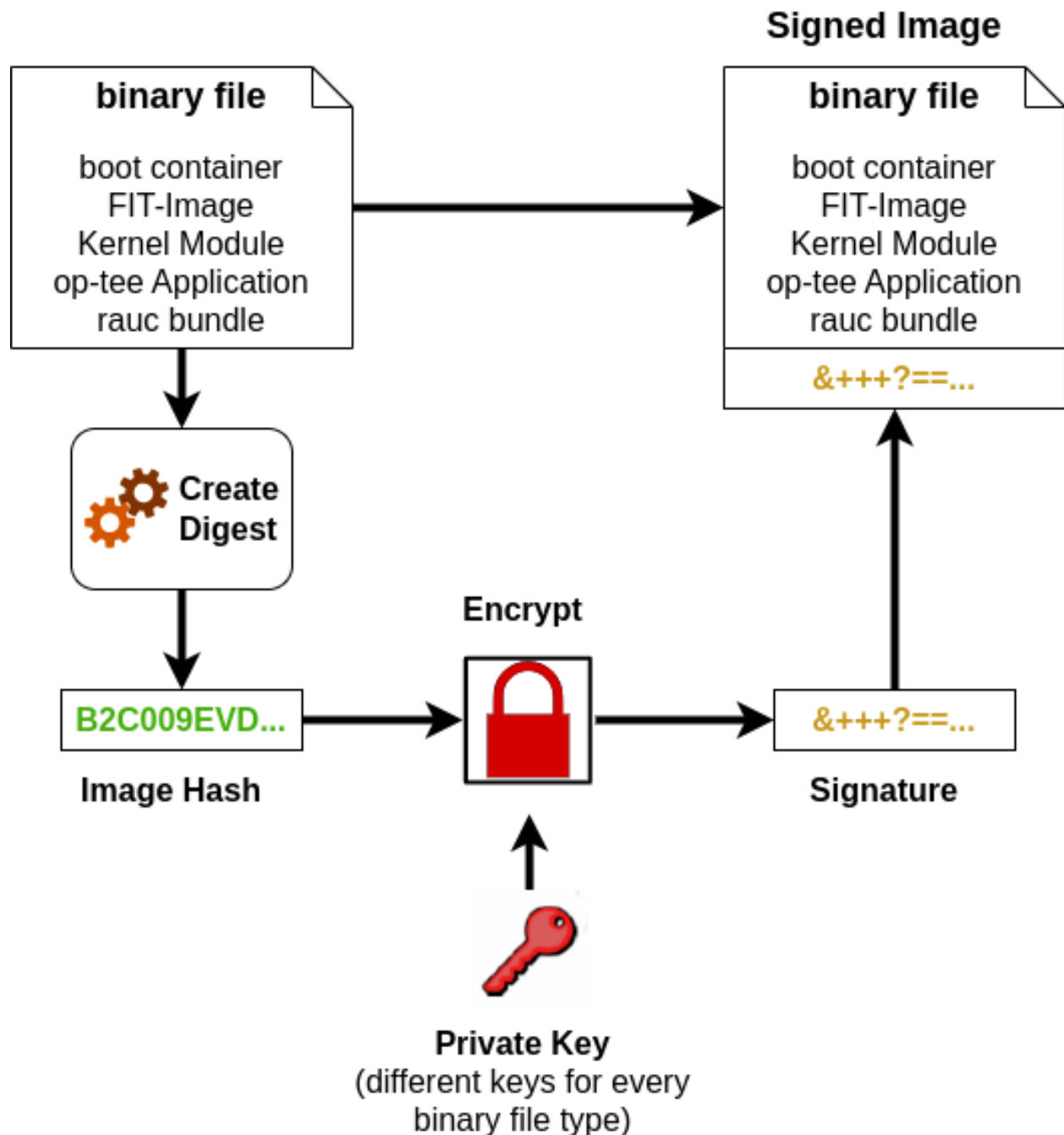
2. u-boot SPL loads u-boot proper from the FIT image and ATF (ARM Trusted Firmware) and optionally OP-TEE.
3. Then, u-boot loads and verifies the FIT image containing a Linux kernel, DTB, and ramdisk. (**Boot Step 2**)
4. If built with signed kernel modules (standard), Linux will only load kernel modules verifies with a kernel compiled in public key

If you use the DISTRO\_FEATURE secureboot or a TI HS-SE machine variant, then the bootloader is configured additionally:

1. The bootloader is signed and is used in the images (wic, partup and bmap).
2. The bootloader boots only signed kernel fitImage after a verification.
3. The bootloader use only the built-in environment and only loads some necessary variables for RAUC update mechanism.

### 3.3 Image Signing

In the following flowchart you can see the signing process for different parts of an image.



1. A Hash is created for the binary file like the boot container, Kernel FIT image, kernel modules, op-tee applications or the rauc update bundle.
2. A signature is from the hash with a private key.
3. The signature is added to the binary file.



## 3.4 Flattened Image Tree (FIT)

FIT images are used with Secure Boot as standard format for packing kernel, device-tree and optional initramfs. The FIT image are signed in the yocto build with a Private Key. The public key is on the target, compiled in the bootloader or the NXP HAB keys are used. Documentation about FIT is available in the [Flattened Image Tree project](#).

There are two different yocto classes for creation of a signed FIT image.

- PHYTEC *sources/meta-phytec/classes/fitimage.bbclass*
  - With FIT image recipes you can define more separate FIT images.
  - Example for FIT image recipes are in *sources/meta-ampliphy/recipes-images/fitimage/*
  - To create a own FIT image, you need to specify some variables in the recipe:
    - \* FITIMAGE\_SLOTS: Use this to list all slot classes for which the FIT image should contain images. A value of “kernel fdt fdtapply”, for example, will create a manifest with images for two slot classes - kernel and devicetree.
    - \* FITIMAGE\_SLOT\_<slotclass>: For each slot class, set this to the image (recipe) name which builds the artifact you intend to place in the slot class.
    - \* FITIMAGE\_SLOT\_<slotclass>[type]: For each slot class, set this to the type of image you intend to place in this slot. Possible types are the kernel, fdt, fdto, fdtapply, or ramdisk.
    - \* FITIMAGE\_SLOT\_<slotclass>[file]: For slot type kernel, fdt, fdto and fdtapply set this to the file of the image you intend to place in this slot.
    - \* FITIMAGE\_SLOT\_<slotclass>[fstype]: For slot type ramdisk, set this to the filesystem type of image you intend to place in this slot.
    - \* FITIMAGE\_SLOT\_<slotclass>[name]: For slot type fdtapply, set this to the final device tree and configuration name.
- poky *sources/poky/meta/classes-recipe/kernel-fitimage.bbclass*
  - This is the standard upstream FIT image class in yocto mainly for u-boot, which built one FIT image with initramfs and without initramfs.

At first the PHYTEC FIT image class is used to create the FIT images, because it supports barebox and u-boot and you can define more separate FIT images. Since security has increasingly become an integral part of the SoC manufacturer’s BSPs, which use the kernel-fitimage, PHYTEC has decided to gradually switch to this class, too.

## 3.5 Configuration Class for Signing images

All variables to adjust the bootloader and kernel fitImage signing process can be found in the *source/meta-ampliphy/secureboot.bbclass*

At first the necessary variables for signing the bootloader for the different SOC types. The variable `BOOTLOADER_SIGN` is obsolete, because the `DISTRO_FEATURES` secureboot includes the Bootloader signing.

```
BOOTLOADER_SIGN ??= "true"
BOOTLOADER_SIGN[type] = "boolean"

CERT_PATH ??= "${OEROOT}/../..../phytec-dev-ca"
# for NXP HABv4 based systems
BOOTLOADER_SIGN_IMG_PATH ??= "${CERT_PATH}/nxp_habv4_pki/crts/IMG1_1_sha256_4096_65537_v3_usr_
```

(continues on next page)

(continued from previous page)

```

↪ crt.pem"
BOOTLOADER_SIGN_CSF_PATH ??= "${CERT_PATH}/nxp_habv4_pki/crts/CSF1_1_sha256_4096_65537_v3_usr_
↪ crt.pem"
BOOTLOADER_SIGN_SRKFUSE_PATH ??= "${CERT_PATH}/nxp_habv4_pki/crts/SRK_1_2_3_4_table.bin"
BOOTLOADER_HABV4_SRK_INDEX ??= "0"

# AHAB
AHAB_SRK_TABLE_BIN ?= "${CERT_PATH}/nxp_ahab_pki/crts/SRK_1_2_3_4_table.bin"
AHAB_SRK_PUB_CERT ?= "${CERT_PATH}/nxp_ahab_pki/crts/SRK1_sha512_secp521r1_v3_usr_crt.pem"
AHAB_SRK_INDEX ?= "0"

# for TI K3
BOOTLOADER_TI_K3_MPK_KEY ??= "${CERT_PATH}/ti_k3/keys/phytecSMPK.pem"
BOOTLOADER_TI_K3_DEGENERATE_KEY ??= "${CERT_PATH}/ti_k3/keys/ti-degenerate-key.pem"

```

In the following view you can see the necessary variables for signing with the PHYTEC FIT image class. The FITIMAGE\_PUBKEY\_SIGNATURE\_PATH is only important, when using the FITIMAGE\_SIGN\_ENGINE="software". This means, that the u-boot validates the kernel fitImage und use the compiled in public key. The alternative is that the NXP HAB unit validates the kernel fitImage, then the FITIMAGE\_SIGN\_ENGINE="nxphab" must be set. This is only possible for NXP SOC's with HAB unit and u-boot as bootloader. The following configuration are in the *sources/meta-ampliphy/secureboot.bbclass*

```

FITIMAGE_SIGN ?= "true"
FITIMAGE_SIGN[type] = "boolean"

FITIMAGE_NO_DTB_OVERLAYS ?= "false"
FITIMAGE_NO_DTB_OVERLAYS[type] = "boolean"

FITIMAGE_SIGNER ?= "customer"
FITIMAGE_PUBKEY_SIGNATURE_PATH ?= "${WORKDIR}/signature_node.dtsi"

FITIMAGE_SIGN_ENGINE ?= "software"

FITIMAGE_SIGN_KEY_PATH ?= "${CERT_PATH}/fit/FIT-4096.key"
FITIMAGE_HASH ?= "sha256"
FITIMAGE_SIGNATURE_ENCRYPTION ?= "rsa4096"
FITIMAGE_SIGNER_VERSION ?= "vPD20.0.0"

```

The signing with the poky kernel-fitimage class needs the following configuration in

- machine configuration in *sources/meta-phytec/conf/machine* for the kernel, initrd, device-tree and device-tree overlay LOADADDRESS and ENTRYPOINT addresses
- *source/meta-ampliphy/secureboot.bbclass* for the signing key parameter

```

UBOOT_SIGN_KEYDIR = "${CERT_PATH}/fit"
UBOOT_SIGN_KEYNAME = "FIT-4096"
FIT_SIGN_ALG = "rsa4096"
FIT_HASH_ALG = "sha256"

```

- *sources/meta-ampliphy/conf/distro* file for the INITRAMFS\_IMAGE.

```

— ampliphy-secure and ampliphy-vendor-secure: INITRAMFS_IMAGE =
  "phytec-secureboot-initramfs"

```

```
– ampliphy-provisioning and ampliphy-vendor-provisioning: INITRAMFS_IMAGE =  
  "phytec-provisioning-initramfs"
```

## 3.6 Building a signed image

To build a signed provisioning image for the configuration of the device, which can boot from sd-card or Serial Downloader, then the `DISTRO` needs to be set to `ampliphy-vendor-secure` or `ampliphy-secure`. The main parts for the provisioning-image are the bootloader and the `fitImage`, which includes an `initramfs` with all necessary tools.

```
# for NXP SOC  
host:~$ bitbake phytec-provisioning-image  
  
# for TI K3 SOC  
host:~$ bitbake phytec-headless-image
```

To build the `phytec-security-image` for the eMMC or ubifs with `rauc` update support, then the `DISTRO` needs to be set to `ampliphy-vendor-secure` or `ampliphy-secure`.

```
# for all SOC  
host:~$ bitbake phytec-security-image
```

### Note

If you have some boot warnings or errors like `/initrd.image: incomplete write` or the kernel boot fails, then please check the size for `cma` (kernel boot parameter, setting in bootloader). The allocate RAM for `cma` can be too much, which is import for systems with 256 MByte or 512 Mbyte RAM.



## ACTIVATE SECURE BOOT ON THE DEVICE

The final step to activate secure boot on your device is to burn the secure eFuse configuration.

### Warning

The secure eFuse configuration can only be written once and is irreversible!

For Secure Boot are only public information are burned to SOC from NXP and TI. When building the yocto-secure distro for the first time, the bootloader image is signed with PHYTEC's development keys. Yocto stores these development keys to `yocto/phytec-dev-ca`

### Note

Create and use your own keys and certificates for signing your images. Burn the right key into the Controller eFuse. Please refer to the chapter *Secure Key Storage*

## 4.1 eMMC Boot Partition to Enable Boot

If you install your eMMC with the partup image, then the eMMC is configured with the right configuration. If you install the bootloader standalone on the eMMC, then please check the eMMC configuration for the right partition.

	barebox	u-boot
Set eMMC as an active device	<code>barebox\$ detect mmc3</code>	<code>u-boot=&gt; mmc dev 2</code>
Show active boot partition	<code>barebox\$ devinfo mmc3</code>	<code>u-boot=&gt; mmc partconf 2</code>
Set user area for boot	<code>barebox\$ mmc3.boot=disabled</code>	<code>u-boot=&gt; mmc partconf 2 0 7 0</code>
	disabled: user partition	0x7: user partition
	boot0: Boot partition 0	0x1: Boot partition 0
	boot1: Boot partition 1	0x2: Boot partition 1

Active boot output for barebox:

```
...
Parameters:
boot: disabled (type: enum) (values: "disabled", "boot0", "boot1", "user")
```

(continues on next page)

(continued from previous page)

```
nt_signature: 9a54880c (type: uint32)
probe: 0 (type: bool)
```

### Active boot output for u-boot

```
EXT_CSD[179], PARTITION_CONFIG:
BOOT_ACK: 0x0
BOOT_PARTITION_ENABLE: 0x1
PARTITION_ACCESS: 0x7
```

## 4.2 Activate Secure Boot for NXP SOC

For NXP SOC you can burn the Fuses with u-boot or with the tool crucible in the kernel userspace. The necessary SRK fuses contain the hash value of the SRK public keys. They are never used on open devices! In closed devices, they are used to validate the public key contained in signed firmware images. Before closing the device, you must store the hash of the public keys in the SRK OTP bits on the device. This will allow the ROM loader to validate the public key included in signed firmware images.

- NXP i.MX with HAB: example `SRK_1_2_3_4_fuse.bin` file in `yocto/phytec-dev-ca/nxp_habv4_pki/crts/SRK_1_2_3_4_fuse.bin`

If you build the signed bootloader, then the following tools are available in the bootloader.

### 4.2.1 Check the current state of your device

- NXP i.MX6 with HAB and bootloader barebox

```
barebox$ hab -i
Current SRK hash:
0000000000000000000000000000000000000000000000000000000000000000
devel mode
```

- NXP i.MX8M Series with HAB and bootloader u-boot

```
u-boot=> hab_status
Secure boot disabled

HAB Configuration: 0xf0, HAB State: 0x66
No HAB Events Found!
```

### 4.2.2 Burn the SRK

- NXP i.MX6 with HAB and bootloader barebox you can copy the `SRK_1_2_3_4_fuse.bin` to the device with e.g. `tftp` and burn directly with

```
barebox$ hab -p -s SRK_1_2_3_4_fuse.bin
```

to check

```
barebox$ hab -i
Current SRK hash:
3425849ab41a49b07ba0b6d5e7dc92fd7cc80dc1a904bdd8e49f4e705953029b
devel mode
```

- SOC with u-boot you must write every word to the Fuses

NXP i.MX8M Series with HAB	
host:~\$ od -t x4 SRK_1_2_3_4_fuse.bin	0000000 9a842534 b0491ab4 d5b6a07b fd92dce7 0000020 c10dc87c d8bd04a9 704e9fe4 9b025359
burn the fuses	u-boot=> fuse prog 6 0 0x9a842534 u-boot=> fuse prog 6 1 0xb0491ab4 u-boot=> fuse prog 6 2 0xd5b6a07b u-boot=> fuse prog 6 3 0xfd92dce7  u-boot=> fuse prog 7 0 0xc10dc87c u-boot=> fuse prog 7 1 0xd8bd04a9 u-boot=> fuse prog 7 0 0x704e9fe4 u-boot=> fuse prog 7 0 0x9b025359
read and check the fuses	u-boot=> fuse read 6 0 4 0x00000000: 9a842534 b0491ab4 d5b6a07b fd92dce7 u-boot=> fuse read 7 0 4 0x00000000: c10dc87c d8bd04a9 704e9fe4 9b025359
reset the board	u-boot=> reset
check the state	u-boot=> hab_status  No Events Found!

### 4.2.3 Lock the device

#### Warning

This step is irreversible and could brick your device. Before closing the device:

- Verify you have built a signed bootloader image.
- Reset your board and verify there are no HAB.
- Verify the SRK eFuses have been burned correctly.

- NXP i.MX6 with HAB and bootloader barebox:

```
barebox$ hab -p -l
Device successfully locked down
```

The device is directly locked and the SRK is write protected, too.

- SOC with u-boot:

NXP i.MX8M Series with HAB	
Lock your device Secure Boot active	u-boot=> fuse prog 1 3 0x2000000
Set Read protection	not available
Set Over-ride protection for shadow register	not available
Set Write protection for SRK	u-boot=> fuse prog 0 0 0x200

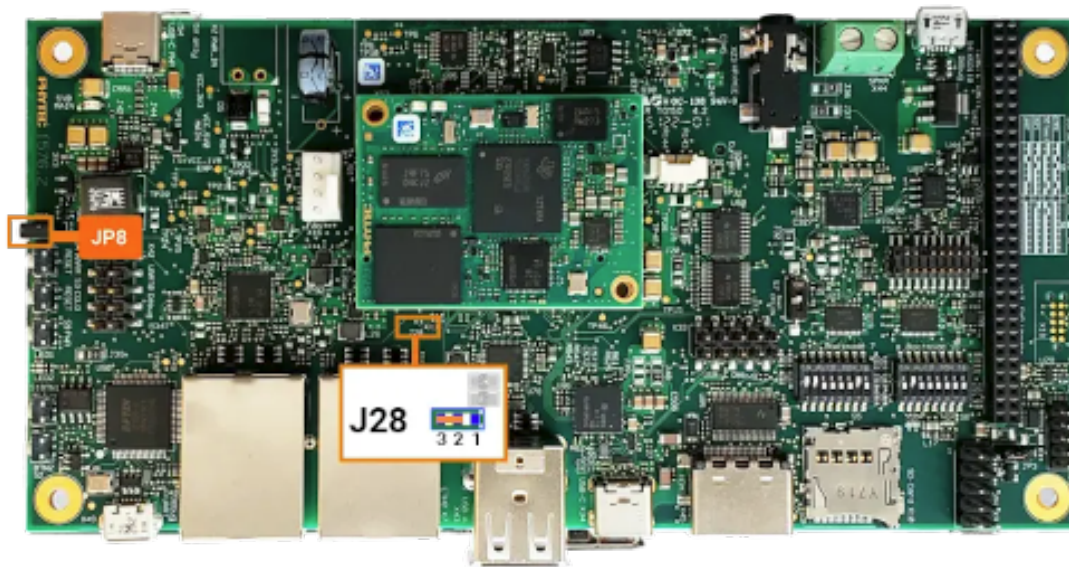
## 4.3 Activate Secure Boot for TI K3 SOC

You can only burn the Fuses with the OTP-Keywriter, which you have create in the chapter *Keys and Certificates Management*. To run the keywriter on your hardware we recommend starting with a regular SD card that has an unsigned image on it. Once you have your bootable SD card, copy the `tiboot3.bin` you generated into the boot partition of the SD card, replacing the previous version of the binary.

- AM62x

Now you must set JP8 on the development kit for AM62x in order to flash the keys.

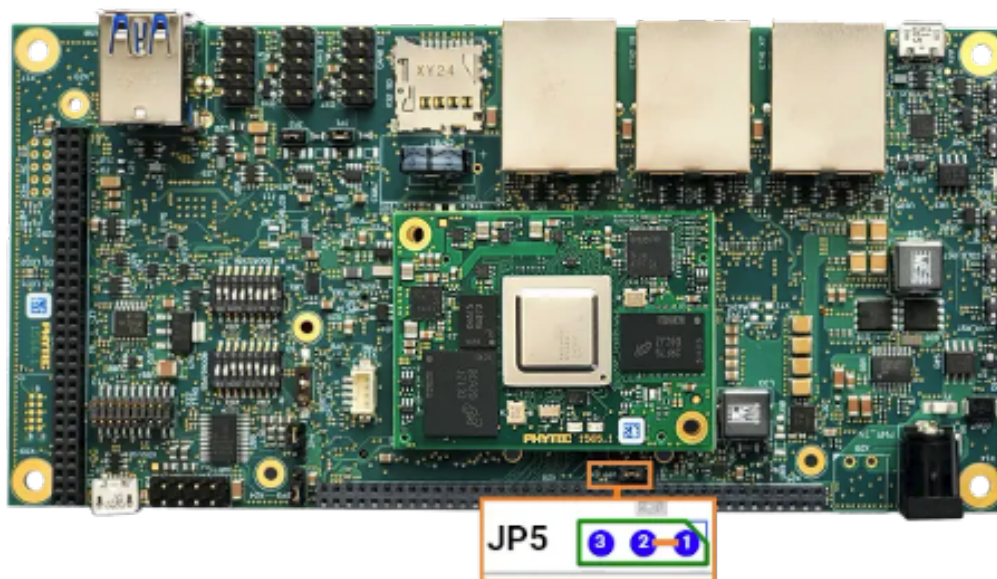


**Note**

For some older AM62x boards you also need to verify that the resistor on J28 is set to position 2+3.

- AM64x

Now you must set JP5 to pins 1 and 2 on the development kit in order to flash the keys.



Once this jumper is set, plug the SD card into the kit and boot as you normally would. You should see a message that keywriting was successful. The keywriter will only successfully write one time.

If you are using the incremental approach to programming your keys, it is essential that you run your Key Revision binary after all the other binaries have been successfully run. Writing the key revision is what converts the device to a secure boot device, so you will not be able to run your other binaries after the key revision is set.

## 4.4 Next Steps after Activation of Secure Boot

### Warning

After you have closed the device, consider the following points with regard to how firmware authentication can potentially be skipped:

- JTAG could be used to boot the processor and avoid the secure boot. See Secure JTAG
- The bootloader will drop to a console after an unsuccessful firmware authentication for debugging purposes. That console can still be used to boot, so it should be disabled in the production firmware. See Protection Shield Level
- please check the NXP and TI websides for more information

## 4.5 Key Revocation

- NXP SOC: You have four keys from which you can revoke until 3 keys.
- TI K3 SOC: You have 2 keys, a SMPK and BMPK (Backup Key)

### 4.5.1 Revoke NXP SRK Key

Although securing the device involves programming the hash of four public keys into the eFuses, only one key (number 1 by default) is used in the secure boot process. If the key gets compromised, it can be revoked and a different key used.

To use a different key for the signature of bootloader images, change the following variables in `sources/meta-ampliphy/classes/secureboot.bbclass`:

```
# for NXP with HABV4
BOOTLOADER_SIGN_IMG_PATH ??= "${CERT_PATH}/nxp_habv4_pki/crts/IMG1_1_sha256_4096_65537_v3_usr_
↳ crt.pem"
BOOTLOADER_SIGN_CSF_PATH ??= "${CERT_PATH}/nxp_habv4_pki/crts/CSF1_1_sha256_4096_65537_v3_usr_
↳ crt.pem"
BOOTLOADER_HABV4_SRK_INDEX ??= "0"
```

The following keys are available:

key Slot	IMG Certificate	CSF Certificate	SRK_REVOLE[2:0]
0	IMG1_1_sha256_4096_*	CSF1_1_sha256_4096_*	001
1	IMG2_1_sha256_4096_*	CSF2_1_sha256_4096_*	010
2	IMG3_1_sha256_4096_*	CSF3_1_sha256_4096_*	100
3	IMG4_1_sha256_4096_*	CSF4_1_sha256_4096_*	not revocable

Example for Revoke Key Slot 0 on NXP SOC with HABV4

barebox i.MX6, i.MX6UL	u-boot i.MX8M series
barebox\$ mw -l -d /dev/imx-ocotp 0xBC 0x0001	u-boot=> fuse prog 9 3 0x1

#### Note

- The SRK Revocation does not modify the SRK hash values, only the SRK\_REVOKE fuse has to be programmed.
- In a closed configuration, HAB, by default, sets the SRK\_REVOKE\_LOCK sticky bit in the OCOTP controller to write protect this eFuse field.
- To instruct HAB not to lock the SRK\_REVOKE field, the CSF commands in the bootloader need to be reconfigured.



## KERNEL MODULE SIGNING

When the [kernel module signing facility](#) is enabled, Linux can enforce that only modules that have been signed with a specific key can be loaded. Keys with invalid signatures won't be allowed to load. This makes it harder for attackers to load malicious or manipulated modules.

This is enforced by the kernel and does not require userland support.

### 5.1 Enable kernel module signing facility

To enable the kernel module signing facility, add the following `DISTRO_FEATURE` to your configuration file in `conf/distro/xyz.conf`

```
DISTRO_FEATURES += "kernelmodsign"
```

#### Warning

By default, the kernel modules will be signed with PHYTEC's public, for example, the development key. Unless you create your own key, this feature does not offer any protection.



## DEVICE TREE OVERLAY AND SECURE BOOT

Device Tree overlays are device tree fragments that can be merged into a device tree during boot time. These are for example hardware descriptions of an expansion board. They are instead of being added to the device tree as an extra include, now applied as an overlay. They also may only contain setting a node's status depending on whether it is mounted or not.

### 6.1 Device Tree Overlay for i.MX6UL and i.MX6

#### Warning

The Device Tree Overlay support is generally deactivated and not supported for i.MX6UL and i.MX6 with Secure Boot in the security distro and image

The new ADIN1300 Ethernet PHY is supported in the standard BSP as devicetree overlay for the phyBOARD-Mira and phyBOARD-Nunki. In the security distro and image, a new device tree is created with the FIT-image recipes in the `sources/meta-ampliphy/recipes-images/fitimages/` and the `fdtap-apply` mechanism from the `source/meta-phytec/classes/fitimage.bbclass`. More information in the chapter Linux Kernel in the FIT-Image In the barebox is an Ethernet PHY detection, which boots the correct configuration from the FIT-image.

### 6.2 Device Tree Overlay for the other NXP source

#### 6.2.1 Build Time

The overlays set in the `$KERNEL_DEVICETREE` Yocto machine variable will be automatically added as a node to the signed FIT-Image.

#### Note

Only Device Tree Overlays in the FIT-Image can be used on the device.

#### 6.2.2 Run Time

The `${overlays}` variable can be either set directly in the U-Boot environment. Or be a part of the external `bootenv.txt` environment.

#### Warning

Manipulation Risk! The external bootenv.txt is not signed and protected against manipulation, so overlays can be changed and deleted in the bootenv.txt.

The `${overlays}` variable loaded from the external environment will always overwrite the value from the environment saved directly in the flash. By default, the `${overlays}` variable is not set directly in the U-Boot environment but comes from the external bootenv.txt environment file. It is also located in the boot partition of the SD card image.

#### Note

Please use Device Tree Overlay only in the development stage of your product. Create a final Device Tree for your device for the production phase.

### 6.2.3 Deactivate Device Tree Overlay Support

To disable the Device Tree Overlay support set the following variable in `sources/meta-ampliphy/classes/secureboot.bbclass` to true

```
FITIMAGE_NO_DTB_OVERLAYS ?= "true"
```

All the machine-defined Device Tree Overlays will be added to the FIT-Image. If you do not want Device Tree Overlays in the FIT-Image, please remove `fdto` in the `sources/meta-ampliphy/recipes-image/fitimage/phytec-secureboot-ramdisk-fitimage.bb` or in your own FIT-Image recipe.

```
FITIMAGE_SLOTS ?= "kernel fdt fdto ramdisk"
```



## SECURE KEY STORAGE

A fundamental aspect of security is integrity and confidentiality. Many applications require an embedded device to keep sensitive data. The standard solution to this problem is to use encryption to protect the data and ensure that only authorized users have access to the encryption key. When a user interacts directly with a system, the encryption key can be protected with a password, pin code, or fingerprint that is provided by the user. However, many embedded devices work without user interaction, so this is not an option in those cases.

In the BSP, three different variants of Secure Key Storage can be implemented, depending on hardware support. The available hardware support is activated with `MACHINE_FEATURE`.

Type of Secure Key Storage	Hardware Support	MACHINE_FEATURE
NXP CAAM	* all NXP i.MX6, i.MX6UL * all i.MX8M series	caam
Trusted Execution Environment TEE	* all NXP i.MX SOC * all TI K3 SOC	optee
Trusted platform Module TPM	* on base boards for i.MX8M series * on phyGATE-Tauri-S / L	tpm

Machines built with the `MACHINE_FEATURE` have all necessary prerequisites enabled.

### 7.1 NXP i.MX CAAM

The NXP i.MX6, i.MX6UL and i.MX8M series processors include hardware encryption through NXP's Cryptographic Accelerator and Assurance Module (CAAM, also known as SEC4). The CAAM combines functions to create a modular and scalable acceleration and assurance engine.

More information about the CAAM module can be found in the corresponding NXP reference Manual: [i.MX Reference Manual](#)

### 7.1.1 Prerequisites and Caveats

Secureboot is required for trusted CAAM Key blob functionality. If Secure Boot Keys are burned, the keys are locked. After a reset, the CAAM unit creates internal keys for the signing and encryption CAAM blobs. These keys are internal in the CAAM and can not be read out and overwritten.

### 7.1.2 Test and using

You can use the CAAM unit accelerator with the cryptodev driver.

```
target:~$ openssl rand -engine devcrypto -hex 30
target:~$ openssl ecparam -engine devcrypto -genkey -out eckey.pem -name prime256v1
```

## 7.2 Trusted Execution Environment: OP-TEE

OP-TEE is a Trusted Execution Environment (TEE) designed as a companion to a non-secure Linux kernel running on Arm; Cortex-A cores using the TrustZone technology.

OP-TEE is supported for the NXP i.MX8M series, NXP i.MX9 series and TI K3 SOC. This allows users who are interested in utilizing [OP-TEE](#) to use and test it on their devices.

### Warning

If you want to use OP-TEE in production, then you must configure the complete isolation between the normal and secure TrustZone world. [For more information](#)

OP-TEE is divided into the following components:

- OP-TEE kernel: The kernel acts as a secure world OS. This kernel is signed by HABv4.
- tee-supplciant: Helper daemon allowing OP-TEE to read/write from/to secure storage. In practice, this means OP-TEE will save encrypted and authenticated data in the filesystem.
- xtest: Utilities to test OP-TEE.

### 7.2.1 Prerequisites and Caveats

- Secure Boot is required for OP-TEE to prevent a malicious OP-TEE kernel from loading.
- It is furthermore required to allow the generation of a hardware unique key that OP-TEE can use to derive a key for secure storage encryption and other use cases.
- Trusted Application Key-Pair: [OP-TEE signs trusted applications](#) in order to ensure their authenticity and integrity. By default, OP-TEE uses a pre-generated key, which you must replace with your own before using OP-TEE in production.

### 7.2.2 Testing OP-TEE

**xtest**

- When OP-TEE is enabled during the build, the “xtest” utility will be shipped.
- Executing “xtest” will run a couple of tests supplied by the OP-TEE project to ensure it is working as intended.

## Memory Isolation: devmem2

- OP-TEE will load itself into a defined region in RAM. This region is reserved in Linux and does not attempt to allocate memory in this area.
- OP-TEE modifies the device tree of Linux during startup to ensure this.
- During runtime, the following nodes will be visible in the device tree:

```
target:~$ dtc -I dtb -O dts /proc/device-tree
reserved-memory {
    #address-cells = <0x02>;
    #size-cells = <0x02>;
    ranges;

    linux,cma {
        linux,cma-default;
        alloc-ranges = <0x00 0x40000000 0x00 0x40000000>;
        compatible = "shared-dma-pool";
        size = <0x00 0x28000000>;
        reusable;
    };

    optee_shm@0x57c00000 {
        reg = <0x00 0x57c00000 0x00 0x400000>;
        no-map;
    };

    optee_core@0x56000000 {
        reg = <0x00 0x56000000 0x00 0x1c00000>;
        no-map;
    };
};
```

- optee\_core denotes the secure world memory region. It is not accessible, even to the Linux kernel.
- optee\_shm is the shared region between the normal and secure world, allowing normal-world client applications to exchange data with OP-TEE-trusted applications.
- Memory access policy enforcement can be tested using the “devmem2” utility.

```
target:~$ devmem2 0x56000000

Memory mapped at address 0xfffff88e2c000.
Bus error

target:~$ $?
135

target:~$ devmem2 0x57c00000
/dev/mem opened.
Memory mapped at address 0xfffffb4f3c000.
Read at address 0x57C00000 (0xfffffb4f3c000): 0xA0A28501
```

- In this example the 0x56000000 address is the optee\_core region. Access is currently being blocked by the TZASC policy set up by OP-TEE, which causes a “Bus error”. The shared region, on the other hand, is accessible.

## 7.3 Trusted Platform Module (TPM) 2.0

The Trusted Platform Module (TPM) is an international standard for a secure cryptoprocessor, a dedicated microcontroller designed to secure hardware through integrated cryptographic keys. The TPM 2.0 is:

- specified from the Trusted Computing Group (TCG)
- TCG and Common Criteria (CC) certified EAL4+
- updateable for the Firmware
- available from different manufacturers
- used to create and store keys and certificates that can be used for filesystem encryption, device identification, and authentication
- a safe on the device, because the persistent keys are in the TPM and the key blobs can only be encrypted with the specific TPM

The Linux kernel has driver support for the TPM. TPM is the standard trusted key in the kernel keyring service. The [middleware for the TPM](#) is Open Source and supports OpenSSL, PKCS#11, and more. [More information about the software stack for the TPM 2.0: A practical guide for using the TPM 2.0:](#)

The TPM is not on the SOM, it is located on the carrier board.

### 7.3.1 Initialization of the TPM

The TPM 2.0 must be initialized at first with the command *tss2\_provision*. This command is used in the tool *physecurekeystorage-install*, when you use the *trustedtpm* key type.

## 7.4 Kernel Key Retention Service for filesystem Encryption

“The Linux key-management facility is primarily a way for various kernel components to retain or cache security data, authentication keys, encryption keys, and other data in the kernel.” Linux kernel is a kernels facility for “password caching”, which stores them in a computers memory (RAM) during an active users/system session. The Linux keyring accessing is via syscalls from the user space into the kernel space. Applications to access are keyctl, systemd-ask-password and others.

The documentation about the Kernel Key Retention service can be found at <https://www.kernel.org/doc/html/latest/security/keys/core.html> The following description and implementation are based on the <https://www.kernel.org/doc/html/latest/security/keys/trusted-encrypted.html>

- The kernel standard trusted key types are trusted tpm, trusted tee and trusted caam. The encrypted blobs are stored in the file *trusted\_key.blob* in the first boot partition and in the third partition with name *config*.
- The secure caam is only supported in the NXP vendor based BSP and used the black key blob mechanism and used the kernel key type *logon*. The encrypted blobs are stored in the file *tksecure\_key*.

The following table list the supported key types for the different SOC's.

Key Type	depend on the MACHINE_FEATURE	NXP i.MX6 (UL)	NXP i.MX8M MNP	TI AM6 Series
trustedtpm	tpm2	x	x	x
trustedtee	optee	x	x	x
trustedcaam	caam	x (not ULL)	x	
securecaam	caam		x	

### 7.4.1 Secure Key Storage Initialization with phySecureKeyStorage Tool

The tool `physecurekeystorage-install` is part of the ramdisk userspace of phytec-provisioning-initramfs and included in the meta-ampliphy layer of the PHYTEC Standard BSP.

The `physecurekeystorage-install` tool can initialize all supported secure key storages of your machine, but always only one can be active. For example, the phyBOARD-Polis-imx8mm supports Trusted TEE, Trusted TPM, Trusted CAAM and Secure CAAM, but initialized is only Trusted TPM.

```
target:~$ physecurekeystorage-install -h
```

```
PHYTEC Install Script v1.7 for Secure Key Storage
```

```
Usage: physecurekeystorage-install [PARAMETER] [ACTION]
```

```
Example:
```

```
physecurekeystorage-install --newkeystorage trustedtpm
physecurekeystorage-install --deletekeystorage
physecurekeystorage-install --loadkeystorage
physecurekeystorage-install --pkcs11testkey
```

```
One of the following action can be selected:
```

```
-n | --newkeystorage <value> Create new Secure Key Storage
                                trustedcaam (only NXP controller)
                                trustedtee
                                trustedtpm
                                securecaam (black blob only NXP Vendor BSP)
-d | --deletekeystorage Erase the existing Secure Key Storage
-l | --loadkeystorage Load the existing Secure Key Storage
-p | --pkcs11testkey Create an ECC testkey with user pin 1234
-h | --help This Help
-v | --version The version of physecurekeystorage-install
```

## 7.5 Cryptographic Token Interface PKCS#11

Also known as “Cryptoki”. PKCS#11 specifies a number of standard calls to relay cryptographic requests (such as a signing operation) to a third party module. Such a module may be a TPM or OP-TEE, it is a software PKCS#11 trusted application that appears to the userland as one.

The library or pkcs11-module-path for PKCS#11 depend on the device: \* TPM 2.0: /usr/lib/libtpm2\_pkcs11.so.0 \* OP-TEE: /usr/lib/libckteec.so.0 \* SmartCards: /usr/lib/opensc-pkcs11.so

The following provider.conf is for the usage with openssl 3.0 and a TPM 2.0. Please set the pkcs11-module-path to your selected Secure key storage.

```
openssl_conf = openssl_init

[openssl_init]
providers = provider_sect

[provider_sect]
default = default_sect
pkcs11 = pkcs11_sect

[default_sect]
activate = 1

[pkcs11_sect]
module = /usr/lib/openssl-modules/pkcs11.so
pkcs11-module-path = /usr/lib/pkcs11/libtpm2_pkcs11.so
activate = 1
```

If the TPM 2.0 is initialized e.g. with the tool physecurekeystorage, then you can create a device certificate.

```
#set TPM Pin
target:~$ TPM_PIN=1234
# Create self-signed certificate
target:~$ OPENSSL_CONF=provider.cnf openssl req -new -x509 -provider tpm2 -days 100 -subj '/
↳CN=my_key/' -key "pkcs11:model=SLB9670;manufacturer=Infineon;token=test;object=test-keypair;
↳type=private;pin-value=${TPM_PIN}" -out ecc.crt
# Write Device Cert to TPM
target:~$ pkcs11-tool -module /usr/lib/pkcs11/libtpm2_pkcs11.so -w ecc.crt -y cert -a iotdm-
↳cert --pin ${TPM_PIN} -d 2
```

#### Note

For device identification on a server or cloud provider, you need a Certificate Authority to sign the device certificate.

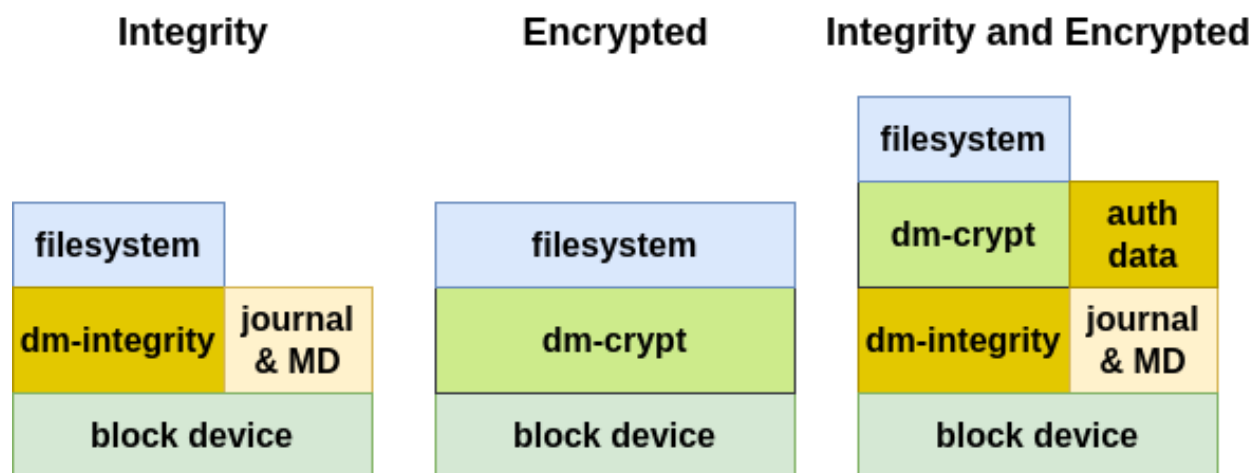
- You can find a more detailed example with Op-tee [https://optee.readthedocs.io/en/latest/building/userland\\_integration.html](https://optee.readthedocs.io/en/latest/building/userland_integration.html)
- Examples with openssl for the TPM 2.0: <https://github.com/tpm2-software/tpm2-tss-engine>

OpenSSL is a robust, commercial-grade, full-featured software library for general-purpose cryptography and secure communication.

## SECURE STORAGE

Secure storage is a combination of the authenticated and encrypted filesystem that adds another layer of security to your product. It uses the kernel's cryptographic support to encrypt all the data you store in the root filesystem. Attempting to access this data without the correct encryption key returns random, meaningless bytes.

The default implementation of secure storage in the PHYTEC BSP is the root filesystem encryption with integrity support:



This manual describes the integrity or/and encryption of the complete root filesystem. Note that on-the-fly encryption and decryption do introduce a small performance penalty in read and write speeds.

Alternatives for the complete root filesystem with integrity and encryption are:

- Partition encryption: To protect some sensitive files but not pay the cost of encrypting the complete rootfs, you can keep the rootfs partition authenticated unencrypted and set up a specific authenticated encrypted partition where the sensitive files will be stored.
- File-specific encryption: Only separate folders and files will be encrypted.

### Note

- ampliphy-secure is an example of how integrity and encryption on embedded devices work. It uses encryption with integrity for a complete partition on eMMC.
- Encrypting the entire root partition should be considered. However, this can only be done on the device.
- An integrity check with dm-integrity is a highly recommended addition to the filesystem encryption.

## 8.1 Filesystem with Integrity vs Authenticated filesystem

The actual standard BSP includes integrity support with hash sha-256, which has protection against data error. An authenticated file system should use HMAC with signed hashes, which have protection against device-turned-off data manipulation from attackers. For this variant, an additional symmetric key is necessary.

## 8.2 Requirements for Filesystem Encryption

- File integrity and encryption support for block devices (SD card, eMMC) or MTD device (NAND, NOR)
- Secure Key Storage to securely store the authentication and encryption key
- Secure Boot must be activated and the device must be locked for proper secure key storage.
- A user login should be activated for access control on runtime.

## 8.3 Boot Process Flow

- bootloader verifies FIT-Image with linux-kernel image, device tree, and ramdisk before they are executed
- Linux kernel executes the ramdisk (read-only filesystem)
- The bootscript loads the authenticated encrypted filesystem encryption key with the CAAM, TEE or TPM unit in the RAM and encrypts the filesystem. After the encryption, the root filesystem will be switched and the boot process continues.

## 8.4 Starting the Build Process

Filesystem integrity and encryption are included in the *DISTRO\_FEATURE* secureboot and securestorage.

You can choose in the *sources/meta-ampliphy/conf/distro/common-secure.inc* between

- fileauthorenc: use integrity or encrypted filesystem
- fileauthandenc: use integrity and encrypted filesystem

This configuration is important for the RAUC update system because the use of integrity and encrypted filesystem are stacked and the number of device-mappers is doubled to use integrity or encrypted filesystem.

```
DISTRO_FEATURES += "securestorage"
#possible types: fileauthorenc , fileauthandenc
SECURE_STORAGE_TYPE = "fileauthandenc"
OVERRIDES_append = ":securestorage:${SECURE_STORAGE_TYPE}"
```

This configuration changes the rauc system.conf configuration in the rootfs image for the target, too. The device change from the /dev/mtdblockX to the device mapper /dev/dm-x. With this changes the integrity and the encryption are retained during an update.



## 8.5 Setup Secure Storage on your Device

The filesystem encryption ensures the target has a unique key or an equal key per device.

The filesystem encryption process flow:

- The filesystem encryption key is generated and stored encrypted with CAAM, TEE, or TPM.
- Encryption is initialized.
- The partition is formatted.
- Data is copied to the encrypted partition.

### 8.5.1 First Boot

From a high-level point of view, an eMMC device is like an SD card. Therefore, it is possible to flash the image phytec-provisioning-image from the Yocto build system directly to the SD card. The image contains the signed bootloader and signed FIT-Image with an initramfs.

If your filesystem is not initialized, is damaged, or the key blob is deleted, then you can reinstall the encrypted filesystem with the following instructions.

- Boot the phytec-provisioning-image from the SD card or load the provisioning fitImage with tftp to the memory in the bootloader
- The device stops with the following message because there is no encrypted key stored in the folder /secrets:

The default user is root with the password root:

#### Note

If there is no login in 60s, then the system goes to power off

```
Login timed out after 60 second
[ERROR] Key and Filesystem Initialization
The system will poweroff in 10 seconds
reboot: Power down
```

- If this is your first boot from the device and no image is on the eMMC, please flash an image to the eMMC.

### 8.5.2 Key Generation for Secure Storage

Please follow the instructions in the chapter *Secure Key Storage*

### 8.5.3 Secure Storage Initialization with phySecureStorage tool

The tool physecurestorage-install is part of the initramfs userspace.

The physecurestorage-install tool can initialize the filesystem with encryption, integrity, or both methods together.

```
target:~$ physecurestorage-install -h

PHYTEC Install Script v1.5 for Secure Storage
```

(continues on next page)

(continued from previous page)

Usage: physecurestorage-install [PARAMETER] [ACTION]

Example:

```
physecurestorage-install --flashpath /dev/mmcblk0
--filesystem /media/phytec-security-image.ext4
--flashlayout 5,6
--newsecurestorage intenc
```

One of the following action can be selected:

```
-n | --newsecurestorage <value>   Create new Secure Storage of type
                                   int    Root File System with integrity
                                   enc    Encrypted root file system
                                   intenc  Encrypted root file system with integrity
-h | --help                        This Help
-v | --version                     The version of the physecurestorage-install
```

The following PARAMETER must be set for new Secure Storage:

```
-p | --flashpath <flash device>
-s | --filesystem <path to root as tgz or ext4>
-l | --flashlayout <value>        partition number for the rootfs partitions
                                   5,6    rootfs partitions are 5 and 6
-L | --labelname <value>         label name for the partition
```

- The parameter <flashpath> is the eMMC device.
- The parameter <filesystem> is the path to tar.gz archive of the filesystem, which should be installed on the flash device.
- Please copy the filesystem image, <IMAGENAME>-<MACHINE>.tar.gz, to a USB or MMC drive so that it can be installed on the target. If partup packages are used for initial flashing, then mount the partup package as type squashfs first and find the root filesystem there.
- The parameter <flashlayout> contains the rootfs partition.
- The parameter RAUC initializes both RAUC rootfs partitions.
- After the installation, power off the system:

```
kernel$ poweroff -f
```

- Restart the system. After a successful installation, the system will boot to
- the kernel login console.

### 8.5.4 Recover an Initialized Device

If your filesystem is damaged or the key blob is deleted, then you can reinstall the encrypted filesystem with the following options.

1. Reinitialize your device with the phytec-provisioning-image from the SD card (Boot in ramdisk)
2. Boot in rescue mode of the existing flash image with minimal tools support

The following commands are for starting the rescue mode with a booted device from eMMC:

- Stop booting in the bootloader. The Protection Shield Level low is in default with password: root
- Add Linux bootargs in the bootloader and boot the fitImage from the eMMC:

- for barebox (i.MX6 and i.MX6UL)

```
barebox$ global linux.bootargs.rescue="rescue=1"
barebox$ boot
```

- for u-boot:

```
u-boot=> run loadraucimage
u-boot=> run raucargs
u-boot=> setenv bootargs ${bootargs} rescue=1
u-boot=> bootm ${loadaddr}
```



## **HARDENING OF THE SYSTEM**

The `DISTRO_FEATURE` hardening activates the kernel reduction with deselect fragments. The name of the deselection variable is `KERNEL_FEATURES_DESELECT`.

The deselect fragment selection for bluetooth, can, optee, pci and wifi depend on `MACHINE_FEATURES` with the same name. If these features are not set in `MACHINE_FEATURES`, then the deselect fragment with the same name is active, but can be selected independently from the `MACHINE_FEATURE` too. The fragments debug, kvm, media and xen are selected by default and are independent from the machine feature.

Overwriting the initial definition of the variable `KERNEL_FEATURES_DESELECT` is possible.

Kernel Fragment	Description	Selection with KERNEL_FEATURES _DESELECT	Selection with MACHINE_FEATURE
hardening.cfg	Activate some hardening features in the kernel. This fragment is the default active with the distro feature hardening.	NO	NO
deselect-bluetooth.cfg	Disable the Bluetooth support.	yes	yes
deselect-can.cfg	Disable the CAN support.	yes	yes
deselect-debug.cfg	Disable kernel debug support.	yes initial set	no
deselect-kvm.cfg	Disable kernel-based virtual machine support.	yes initial set	no
deselect-media.cfg	Disable the ANALOG / DIGITAL TV, RADIO and SDR support	yes initial set	no
deselect-optee.cfg	Disable optee support.	yes	yes
deselect-pci.cfg	Disable PCI interface support.	yes	yes

## PHYSICAL SECURITY

To further protect your device, it is important to reduce attack vectors. Start by securing development features like JTAG and serial downloader. For activation or deactivation of controller features, is necessary to write and read eFuses.

### Warning

The secure eFuse configuration can only be written once and is irreversible.

### 10.1 Secure JTAG

Most embedded devices provide a JTAG interface for debugging purposes. However, if left unprotected, this interface can become an important attack vector on the systems in series production. The most controllers allows you to regulate JTAG access with three security modes using OTP (One Time Programmable) eFuses:

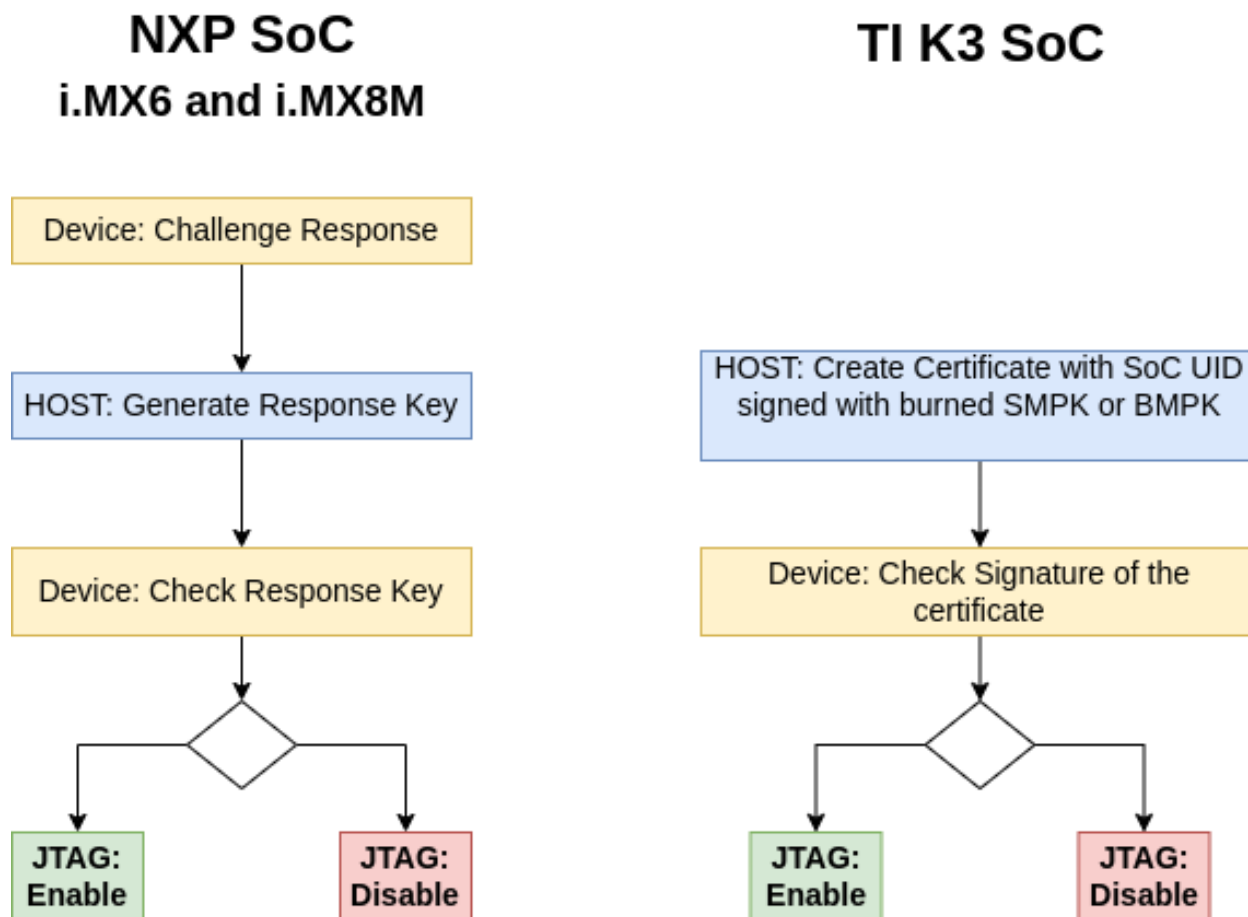
Mode	Security level	Description	NXP SOC	TI SOC
Enabled	low	This is the default mode of operation and you have full access to JTAG.	yes	yes
Disabled debugging	medium	This mode disables debugging but leaves the boundary scan functionality enabled.	yes	
Secure	high	This mode provides high security. JTAG use is regulated by a challenge response authentication mechanism	Secret response key	X509 certificate
Disabled	high	This mode provides maximum security. All security-sensitive JTAG features are permanently blocked, preventing any debugging.	yes	yes

The NXP Soc support different authentication depend on the SoC or the state of the SoC

- NXP i.MX6/UL/ULL and NXP i.MX8M MNP: Secret response key is supported and can be activate independent of the lifecycle



The Secure Debug Mechanism with authentication differs between NXP and TI.



#### Note

The i.MX9 family supports additionally the asymmetric signed message based debug enablement, which has better security compared to the password based mechanism (Secret response key). Secure debug can only be enabled when the device is in OEM\_CLOSED lifecycle. In this life cycle, only authenticated debug is allowed.

Additional information about JTAG Security can be found:

- NXP: [Secure Debug in i.MX6/7/8M Family of Application Processors AN4686](#)
- TI: [Secure Debug User Guide](#) or in the restricted security resources for your SoC type.

### 10.1.1 Disable Debugging Mode only for NXP SoC

Set JTAG to “Disabled debugging” mode:

- i.MX6 and i.MX6UL/ULL with barebox:

```
barebox$ mw -l -d /dev/imx-ocotp 0x18 0xC00000
```

- i.MX8M MNP with u-boot:

```
u-boot=> fuse prog 1 3 0xC00000
```

### 10.1.2 Disable JTAG Mode

#### Note

##### only for NXP i.MX6 family and NXP i.MX8M MNP:

The HAB can normally enable JTAG debugging with the HAB\_JDE-bit in the OCOTP SCS register. The JTAG\_HEO-bit can override this behavior. If this feature is not required, it is highly recommended this be disabled.

- NXP i.MX6 and i.MX6UL/ULL with barebox:

```
# Disable JTAG Mode
barebox$ mw -l -d /dev/imx-ocotp 0x18 0x00100000
# To prevent HAB from Enabling JTAG
barebox$ mw -l -d /dev/imx-ocotp 0x18 0x08000000
```

- NXP i.MX8M MNP with u-boot:

```
# Disable JTAG Mode
u-boot=> fuse prog 1 3 0x200000
# To prevent HAB from Enabling JTAG
u-boot=> fuse prog 1 3 0x4000000
```

## 10.2 Disable Serial Downloader

Disabling the serial download support is recommended for security-enabled configurations:

- NXP i.MX6 with barebox:

```
# Disable only Read Access for SDP
barebox$ mw -l -d /dev/imx-ocotp 0x18 0x0004
# Disable SDP Mode Completely
barebox$ mw -l -d /dev/imx-ocotp 0x18 0x0001
```

- NXP i.MX6UL/ULL with barebox:

```
# Disable only Read Access for SDP
barebox$ mw -l -d /dev/imx-ocotp 0x18 0x40000
# Disable SDP Mode Completely
barebox$ mw -l -d /dev/imx-ocotp 0x18 0x20000
```

- NXP i.MX8M MNP with u-boot:

```
# Disable SDP Mode Completely
u-boot=> fuse prog 2 0 0x200000
```

## 10.3 Force Internal Boot

Ensure the device always boots in INTERNAL BOOT (FORCE\_BT\_FROM\_FUSE) mode, ignoring BOOT\_MODE pins. This setting is recommended for security-enabled configurations.

At first you should burn the Boot Fuses.

- NXP i.MX6 with barebox:

```
barebox$ mw -l -d /dev/imx-ocotp 0x18 0x80000000
```

- NXP i.MX6UL/ULL with barebox:

```
barebox$ mw -l -d /dev/imx-ocotp 0x18 0x10000
```

- NXP i.MX8M MNP with u-boot:

```
u-boot=> fuse prog 2 0 0x100000
```

## 10.4 Disable Boot from External Memory

By writing to the DIR\_BT\_DIS FUSE, we can disable boot from external memory.

- NXP i.MX6 and i.MX6UL/ULL with barebox:

```
barebox$ mw -l -d /dev/imx-ocotp 0x18 0x0008
```

- NXP i.MX8M MNP with u-boot:

```
u-boot=> fuse prog 1 3 0x8000000
```



## KEYS AND CERTIFICATES MANAGEMENT

### 11.1 Public Key Infrastructure Tree (PKI tree)

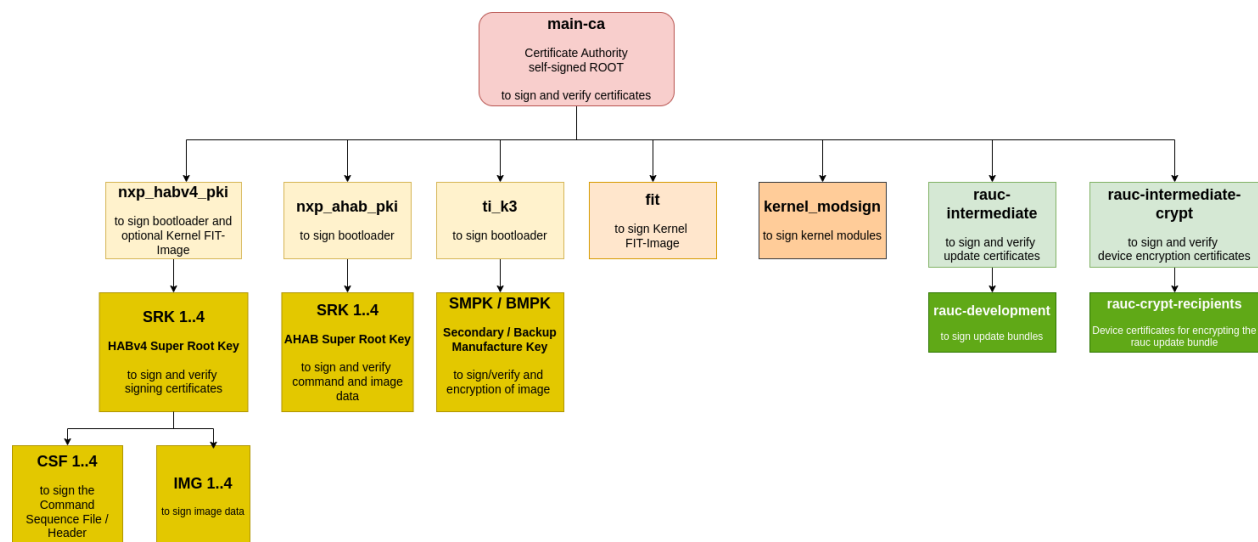
To use a secure boot with a signed bootloader and a signed kernel image, several keys and certificates are required to sign the images. The key and certificate creation is a manual process and the public key infrastructure (PKI) tree must be in place before you start your build. This BSP includes the PHYTECD development pki-tree as an example. You are obligated to create your own pki-tree with your own keys and certificates.

#### Note

It is highly recommended to use different keys for different parts of your system to avoid a single point of failure regarding your security concept.

### 11.2 PHYTEC Development Keys (phytec-dev-ca)

The included phytec-dev-ca example consists of a self-signed main-ca and three derived sub-ca's for bootloader, Fit-Image, and RAUC updates.



The recipes for Bootloader, FIT-Image, and RAUC depend on the recipe phytec-dev-ca. If you build the BSP for the first time, the PHYTEC development keys are downloaded from <https://github.com/phytec/phytec-dev-ca> to yocto/phytec-dev-ca. They are used to sign the Bootloader, FIT-Image, Kernel modules, and the RAUC bundles.

Name	Description	Key Type
main-ca	self-signed Certificate authority	RSA-4096
nxp_ahab_pki	NXP HABv4 Key Authority for i.MX93	NIST P-521
nxp_habv4_pki	NXP HABv4 Key Authority for i.MX6/UL/ULL and i.MX8M Nano/Mini/Plus	RSA-4096
ti_k3	TI K3 Key Authority for AM62 / AM64 / AM68	RSA-4096
fit	Kernel FIT-Image signing key and certificate	RSA-4096
kernel-modsign	Key for the Linux kernel module signing facility, independent of CA	RSA-4096
rauc-intermediate	RAUC ca (intermediate ca) and RAUC ca sign development key for signing the bundles	RSA-2048
rauc-intermediate-crypt	RAUC ca for device certificates to encrypt update bundles	RSA-4096

The SOC specific nxp\_ahab\_pki, nxp\_habv4\_pki and ti\_k3 are for signing the boot container files, which are verified with the SOC internal unit and SOC rom loader or dedicated controllers in the SOC.

All keys and certificates are stored in an XCA database phytec-dev-ca.xdb, which can be configured with the open-source application XCA from <https://hohnstaedt.de/xca/>. The password for the phytec-dev-ca.xdb is: phytec-dev-ca

Only the necessary keys and certificates for the build process are exported to the directory.

```
The phytec-dev-ca is installed in the directory
/<path to>/yocto
|--build
|--phytec-dev-ca
|--source
```

All keys and certificates are in an XCA database and are not copied to different paths from the packages in the build folder. The directory contains only the necessary certificates and keys for the building process.

### Warning

- Use the PHYTEC development keys only for the first test.
- The PHYTEC development keys are not secure!
- Create and use your own keys and certificates!

## 11.3 Create Your Own PKI Tree

Please create your PKI offline with a separate system. For example, boot a read-only system from USB which you only use to create the PKI. The phytec-dev-ca is created with XCA from <https://hohnstaedt.de/xca/> , but you can use any other tool, too.

### 11.3.1 Change PKI-Tree from phytec-dev-ca to Your Own PKI

In the configuration class *sources/meta-ampliphy/classes/secureboot.bbclass*, the path to your PKI tree is initially defined:

```
CERT_PATH ??= "${OEROOT}/../../phytec-dev-ca"
```

If you want to change the path, then reinit the `CERT_PATH ?=` in your layer or overwrite the `CERT_PATH` in the `conf/local.conf`

The name of your PKI tree must have a name other than phytec-dev-ca. The recipe for phytec-dev-ca uses the name “phytec-dev-ca” as a parameter for the `clean` command.

After the `CERT_PATH` has been changed, you must clean and rebuild the bootloader, FIT-Image, RAUC bundles, and the rootfs!

## 11.4 Create NXP AHAB / HABV4 Keys

NXP provided scripts to create keys and certificates for NXP AHAB or NXP HABV4. The scripts are from the [imx-code-signing-tool repository](#)

You can use this script or a PKI application like the XCA to create the keys and certificates to sign the bootloader or boot container.

For creation, the SRK table and SRK Fuses from the SRK certificates are scripts in the `imx-code-signing-tool` repository in the folder `add-ons` which used the `srktool`. You can install the `srktool` with

```
host:~$ apt install imx-code-signing-tool
# or build from source
host:~$ make -C code/obj.linux64 OSTYPE=linux64 ENCRYPTION=yes
```

More information about `cst` and HAB4 API you can find in the `doc` folder of the `imx-code-signing-tool` repository.

## 11.5 Create TI K3 keys

Key Types Involved in Secure Boot

- MEK (Manufacture Encryption Key): TI-provided, permanently fused, validates TI-signed artifacts.
- SMPK (Secondary Manufacture Public Key): User-generated, fused into the device, used to validate user-signed bootloader images.

- BMPK (Backup Manufacture Public Key): Optional, user-generated backup key. If the SMPK is ever lost or compromised, you can reconfigure the system to use the BMPK for bootloader signature verification. The BMPK is strongly recommended for robust device security lifecycle management.

### 11.5.1 Installing the SDK

To create a copy of the OTP Keywriter that includes your own keys, you will need TI's MCU Plus SDK, CCS, SYSCONFIG, and the OTP keywriter source code.

#### For the TI AM62x

- MCU Plus SDK for AM62x (11.01.00.16): <https://www.ti.com/tool/download/MCU-PLUS-SDK-AM62X/11.01.00.16>
- CCS (20.1.0): <https://www.ti.com/tool/download/CCSTUDIO/20.1.0>
- SYSCONFIG (1.24): <https://www.ti.com/tool/download/SYSCONFIG/1.24.0.4150>
- ARM-CGT-CLANG (4.0.1): <https://www.ti.com/tool/de-de/download/ARM-CGT-CLANG/4.0.1.LTS>
- Keywriter (11.01.00) source code must be requested from the AM62X-RESTRICTED-SW section of the AM62x downloads page: <https://www.ti.com/tool/PROCESSOR-SDK-AM62X>
- OTP keywriter firmware supports a maximum certificate length of 6144 bytes
- Once you have the MCU Plus SDK set up, install the keywriter source to <MCU\_PLUS\_SDK\_DIRECTORY>/source/security.

#### For the TI AM64x

- MCU Plus SDK for AM64x (09.00.00.35): <https://www.ti.com/tool/download/MCU-PLUS-SDK-AM64X/09.00.00.35>
- CCS (12.4.0): <https://www.ti.com/tool/download/CCSTUDIO/12.4.0>
- SYSCONFIG (1.17): <https://www.ti.com/tool/download/SYSCONFIG/1.17.0.3128>
- Keywriter (09.00.00.35) source code must be requested from the AM64X-HS-RESTRICTED-SW section of the AM64x downloads page: <https://www.ti.com/secureresources/AM64X-HS-RESTRICTED-SW>
- OTP keywriter firmware supports a maximum certificate length of 5400 bytes
- Once you have the MCU Plus SDK set up, install the keywriter source to <MCU\_PLUS\_SDK\_DIRECTORY>/source/security.

#### For the TI AM68x/TDA4x

- TI Processor SDK RTOS for J721S2 (10.01.00.04): <https://www.ti.com/tool/download/PROCESSOR-SDK-RTOS-J721S2/10.01.00.04>
- GCC Compiler (9.2-2019.12): [https://developer.arm.com/-/media/Files/downloads/gnu-a/9.2-2019.12/binrel/gcc-arm-9.2-2019.12-x86\\_64-aarch64-none-linux-gnu.tar.xz](https://developer.arm.com/-/media/Files/downloads/gnu-a/9.2-2019.12/binrel/gcc-arm-9.2-2019.12-x86_64-aarch64-none-linux-gnu.tar.xz)
- CCS (12.4.0): <https://www.ti.com/tool/download/CCSTUDIO/12.4.0>
- OTP keywriter add-on package (10.01.00) must be requested from the J7X-RESTRICTED-SECURITY section: <https://www.ti.com/secureresources/J7X-RESTRICTED-SECURITY>
- OTP keywriter firmware supports a maximum certificate length of 6144 bytes
- Follow the Readme instructions of the OTP keywriter add-on package
- Additional installation step



```

host:~$ cd <MCU_PLUS_SDK_DIRECTORY>
host:~$ ./sdk_builder/scripts/setup_psd_rtos.sh
host:~$ cd <MCU_PLUS_SDK_DIRECTORY>/<PDK_j721s2_DIRECTORY>/packages/ti/boot/keywriter/
↪scripts/
host:~$ mkdir tifek
host:~$ cp ti_fek_public.pem tifek/

```

### Note

Wait for your access request approval by TI, this usually takes 1-3 days. Building the keywriter has only been tested with the specified versions!

Before we begin programming keys, we need to make changes to the source code.

#### For the TI AM62x

In <MCU\_PLUS\_SDK\_DIRECTORY>/source/security/sbl\_keywriter/am62x-sk/r5fss0-0\_nortos/main.c disable or remove line 76:

Disable the voltage Vpp setting:

```
//keywriter_setVpp();
```

This is because there is a pin on the SoC that needs to be set high to write keys, and TI does this using I2C on their boards which requires this function to run. We will set this pin using a jumper on our board.

#### For the TI AM64x

In <MCU\_PLUS\_SDK\_DIRECTORY>/source/security/sbl\_keywriter/am64x-evm/r5fss0-0\_nortos/main.c disable or remove line 61:

Disable the voltage Vpp setting:

```
//keywriter_setVpp();
```

This is because there is a pin on the SoC that needs to be set high to write keys, and TI does this using I2C on their boards which requires this function to run. We will set this pin using a jumper on our board.

#### For the TI AM68x/TDA4x

In <MCU\_PLUS\_SDK\_DIRECTORY>/<PDK\_j721s2\_DIRECTORY>/packages/ti/boot/keywriter/soc/j721s2/keywriter\_utils.c

Activate the voltage Vpp on the phyCORE-AM68x with this patch patch < diff.patch

```

--- keywriter_utils.c      2024-12-12 18:18:53.000000000 +0100
+++ keywriter_utils.c      2025-08-25 11:41:59.935563549 +0200
@@ -40,6 +40,9 @@

#include "keywriter_utils.h"
#include "board_utils.h"
#include "board_internal.h"
+
+#define MAIN_CTRL_BASE      (0x00100000U)

static void keywr_leo_pmicb_set_params(Pmic_CoreCfg_t *pmicConfigData)

```

(continues on next page)

(continued from previous page)

```

{
@@ -164,37 +167,21 @@
*/
void OTP_VppEn(void)
{
-   Board_I2cInitCfg_t i2cCfg;
-   Board_IDInfo_v2    info;
-   Board_STATUS       status;
-   bool               skBoardDet = BFALSE;
-
-   i2cCfg.i2cInst      = BOARD_I2C_EEPROM_INSTANCE;
-   i2cCfg.socDomain    = BOARD_SOC_DOMAIN_WKUP;
-   i2cCfg.enableIntr   = BFALSE;
-   Board_setI2cInitConfig(&i2cCfg);
-
-   /* Check if the board is SK */
-   status = Board_getIDInfo_v2(&info, KEYWRITER_SK_EEPROM_SLAVE_ADDR);
-   if(BOARD_SOK == status)
-   {
-       if(!(strcmp(info.boardInfo.boardName,
-                   "AM68-SK-SOM",
-                   BOARD_BOARD_NAME_LEN)))
-       {
-           UART_printf("AM68 SK Detected!!\n");
-           skBoardDet = BTRUE;
-       }
-   }
+   uint32_t regVal;
-
-   if(BTRUE == skBoardDet)
-   {
-       /* Enable VPP for AM68 SK board */
-       OTP_VppEn_SK();
-   }
-   else
-   {
-       /* Enable VPP for J721S2 EVM or a Custom board */
-       OTP_VppEn_EVM();
-   }
+   UART_printf("OTP_VppEn_phyCORE-AM68x/TDV4 \n");
+
+   /* pinmux padconfig*/
+   mmr_unlock(MAIN_CTRL_BASE,7);
+   HW_WR_REG32(BOARD_MAIN_PMUX_CTRL_ADDR+PIN_MCAN12_RX, PIN_PULL_DISABLE | PIN_MODE(7));
+
+   /* Set the MAIN GPIO 0 Pin 2 direction to output */
+   regVal = (HW_RD_REG32(CSL_GPIO0_BASE+0x10)) & ~(0x1 << 0x02);
+   HW_WR_REG32(CSL_GPIO0_BASE+0x10, regVal);
+
+   /* Set the MAIN GPIO 0 Pin 2 value to high */
+   regVal = (HW_RD_REG32(CSL_GPIO0_BASE+0x14)) | (0x1 << 0x02);
+   HW_WR_REG32(CSL_GPIO0_BASE+0x14, regVal);

```

(continues on next page)

(continued from previous page)

```
+  
+   UART_printf("OTP Vpp is Enabled!\n");  
}
```

This is because there is a pin on the SoC that needs to be set high to write keys and a GPIO is on use of the phyCORE-AM68x.

## 11.5.2 Generating keys

The keywriter source comes with a tool to help generate your own keys. To generate keys, go to

- AM62x: <MCU\_PLUS\_SDK\_DIRECTORY>/source/security/sbl\_keywriter/scripts/cert\_gen/am62x
- AM64x: <MCU\_PLUS\_SDK\_DIRECTORY>/source/security/sbl\_keywriter/scripts/cert\_gen/am64x
- AM68x: <MCU\_PLUS\_SDK\_DIRECTORY>/<PDK\_j721s2\_DIRECTORY>/packages/ti/boot/keywriter/scripts

and run

```
host:~$ ./gen_keywr_cert.sh -g
```

This will create a set of five keys in the `keys/` directory.

Alternatively you can copy the PHYTEC dummy keys to this folder, which you can get from [https://github.com/phytec/phytec-dev-ca/tree/main/ti\\_k3](https://github.com/phytec/phytec-dev-ca/tree/main/ti_k3)

You can use the keywriter to flash these keys to your hardware and you will need to keep them safe to use for signing your images as well.

## 11.5.3 Building the Keywriter

There are two methods for creating the keywriter. You can create one keywriter that contains all of your keys and configuration (e.g. JTAG configuration), or you can make one keywriter per key and configurations. The all-at-once approach is more straightforward, but if your key certificates end up with a certificate exceeding the maximum certificate length you may need to use the incremental approach.

### Generate Incremental Certificates

If you end up with a certificate exceeding the maximum certificate length while trying to build and program all the keys at once, you may need to flash the keys incrementally. To do this you will need separate certificates for each key. Starting in

- AM62x: <MCU\_PLUS\_SDK\_DIRECTORY>/source/security/sbl\_keywriter/scripts/cert\_gen/am62x,
- AM64x: <MCU\_PLUS\_SDK\_DIRECTORY>/source/security/sbl\_keywriter/scripts/cert\_gen/am64x,
- AM68x: <MCU\_PLUS\_SDK\_DIRECTORY>/<PDK\_j721s2\_DIRECTORY>/packages/ti/boot/keywriter/scripts,

#### Note

Each certificate results in one programming step with a fresh boot of the device. Until the KEYREV value is set to either 1 or 2, the device is considered an HS-FS device, and key values can continue being programmed incrementally. So, programming the KEYREV should be left to the final step.

- generate the first certificate for the Model Specific Value (MSV), the SMPK and the SMEK key:

Listing 1: MSV+SMPK+SMEK

```
host:~$ ./gen_keywr_cert.sh --msv 0xC0FFE -s keys/smpk.pem --smek keys/smek.key --aes256_
↳ keys/aes256.key -t tifik/ti_fek_public.pem
```

- Override parameter: --msv-ovrd, -s-ovrd, --smek-ovrd
- Readout Protection: --msv-rp, -s-rp, --smek-rp
- Overwrite Protection --msv-wp, -s-wp, --smek-wp
- generate a certificate for the backup key BMPK and BMEK.

Listing 2: BMPK+BMEK

```
host:~$ ./gen_keywr_cert.sh -b keys/bmpk.pem --bmek keys/bmek.key --aes256 keys/aes256.key_
↳ -t tifik/ti_fek_public.pem
```

- Override parameter: -b-ovrd, --bmek-ovrd
- Readout Protection: --b-rp, --bmek-rp
- Overwrite Protection --b-wp, --bmek-wp
- generate a certificate to disable JTAG

Listing 3: Disable JTAG

```
host:~$ ./gen_keywr_cert.sh --jtag-disable --aes256 keys/aes256.key -t tifik/ti_fek_public.
↳ pem
```

You can add this directly to the first certificate.

- Override parameter: --jtag-disable-ovrd
- Readout Protection: --jtag-disable-rp
- Overwrite Protection --jtag-disable-wp
- generate a certificate for Secure Board Config

#### The Secure Board Configuration

Listing 4: Secure Board Config

```
host:~$ ./gen_keywr_cert.sh --sr-bcfg <board configuration> --aes256 keys/aes256.
↳ key -t tifik/ti_fek_public.pem
```

- Override parameter: --sr-bcfg-ovrd
- Readout Protection: --sr-bcfg-rp
- Overwrite Protection --sr-bcfg-wp
- Additionally, if the extended OTP needs to be programmed via keywriter (for USB/PCIE VID/PID), make sure to program the extended OTP before converting the device to an HS-SE device!

Listing 5: Extended OTP

```
host:~$ ./gen_keywr_cert.sh --ext-otp ext_otp_data.bin --ext-otp-indx 1 --ext-otp-size 3 --
↪aes256 keys/aes256.key -t tifik/ti_fek_public.pem
```

- Readout and Overwrite Protection: `---ext-otp-wprp`
- generate certificate to set keycnt and keyrev to enable the HS-SE device with secure boot
  - keycnt = 1 for only SMPK, keycnt = 2 for using SMPK and BMPK
  - keyrev = 1 for active SMPK, keyrev = 2 for active BMPK

Listing 6: Activate Secure Boot

```
host:~$ ./gen_keywr_cert.sh --keycnt 2 --keyrev 1 --aes256 keys/aes256.key -t tifik/ti_fek_
↪public.pem
```

- Override parameter: `--keycnt-ovrd`, `--keyrev-ovrd`
- Readout Protection: `--keycnt-rp`, `--keyrev-rp`
- Overwrite Protection `--keycnt-wp`, `--keyrev-wp`

If you create one certificate, then generate the keywriter and run it on the device.

### Generate One Shot Certificate

Using the keys generated in the previous step, we can now generate a certificate to sign our hardware and enable secure boot. Go to

- AM62x: `<MCU_PLUS_SDK_DIRECTORY>/source/security/sbl_keywriter/scripts/cert_gen/am62x`
- AM64x: `<MCU_PLUS_SDK_DIRECTORY>/source/security/sbl_keywriter/scripts/cert_gen/am64x`
- AM68x: `<MCU_PLUS_SDK_DIRECTORY>/<PDK_j721s2_DIRECTORY>/packages/ti/boot/keywriter/scripts`

and run the following:

```
host:~$ ./gen_keywr_cert.sh -t tifik/ti_fek_public.pem --msv 0xC0FFE -s keys/smpk.pem --smek_
↪keys/smek.key -b keys/bmpk.pem --bmek keys/bmek.key --jtag-disable --keycnt 2 --keyrev 1
```

#### Note

Please add Readout and Overwrite protection for the field.

This generates a certificate containing our keys (primary\_cert.bin). Please note the maximum certificate length limit, which requires incremental certificates writing.

### Build the keywriter

For the TI AM62x

```
host:~$ cd <MCU_PLUS_SDK_DIRECTORY>/source/security/sbl_keywriter/am62x-sk/r5fss0-0_nortos/
↪ti-arm-clang/
host:~$ make -sj PROFILE=debug clean
host:~$ make -sj PROFILE=debug
```

The keywriter for the certificate has now been built and is in the `tiboot3.bin` file in `<MCU_PLUS_SDK_DIRECTORY>/source/security/sbl_keywriter/am62x-sk/r5fss0-0_nortos/ti-arm-clang`.

For the TI AM64x

```
host:~$ cd <MCU_PLUS_SDK_DIRECTORY>/source/security/sbl_keywriter/scripts/cert_gen/x509cert
host:~$ python3 ../../../../tools/bin2c/bin2c.py final_certificate.bin keycert.h KEYCERT
host:~$ cd ../../am64x-evm/r5fss0-0_nortos/ti-arm-clang/
host:~$ make -sj PROFILE=debug clean
host:~$ make -sj PROFILE=debug
```

The keywriter for the certificate has now been built and is in the `tiboot3.bin` file in `<MCU_PLUS_SDK_DIRECTORY>/source/security/sbl_keywriter/am64x-evm/r5fss0-0_nortos/ti-arm-clang`

For the TI AM68x/TDA4x

```
host:~$ cd <MCU_PLUS_SDK_DIRECTORY>/<PDK_j721s2_DIRECTORY>/packages/ti/build
host:~$ make keywriter_img_clean SOC=j721s2 BOARD=j721s2_evm
host:~$ make keywriter_img SOC=j721s2 BOARD=j721s2_evm PROFILE=debug -sj
```

The keywriter for the certificate has now been built and is in the `tiboot3.bin` file is `<MCU_PLUS_SDK_DIRECTORY>/<PDK_j721s2_DIRECTORY>/packages/ti/boot/keywriter/binary/j721s2/keywriter_img_combined_j721s2_release.tiimage`

Save it elsewhere so that we can build the other keywriters without overwriting this one. Make sure that you keep track of the binaries so that you can flash them in the correct order later.

## 11.6 Create Kernel FIT-Image key

You can create the Kernel FIT-Image Key with a PKI Tool or openssl.

For the signing of the Kernel FIT-Image the private key is used. The public key is build into as device-tree node in the bootloader and used for the FIT-Image verification.

The certificate is not necessary for the signing and verification of the FIT-Image.

## 11.7 Create Kernel Module Signing key

You can create the Key and certificate for kernel module signing with a PKI tool or openssl. You must combine the private key and the certificate to one file.

```
host:~$ cat private.key certificate.pem > kernel_modsign.pem
```

## 11.8 Create rauc update certificates

You can create the Key and certificate for rauc with a PKI tool or openssl. More details on the [rauc documentation](#)