Security Manual Wrynose

PHYTEC Messtechnik GmbH

Nov 25, 2026

CONTENTS

1 1 Introduction **3**
- 1.1 1.1 Short Crypto Refresher . . . 5
- 1.2 1.2 Chain of Trust . . . 6
- 1.3 1.3 Phytec Yocto Distributions . . . 6

2 2 Provisioining **7**
- 2.1 2.1 Prepare Provisioning . . . 7
- 2.2 2.2 Activate Secure Boot . . . 7
- 2.3 2.3 Provisioning the eMMC . . . 7

3 3 Booting securiPHY **9**
- 3.1 3.1 SSH Certificate Authentication Login . . . 9

4 4 Using your own Keys and Certificates **11**
- 4.1 4.1 Key Settings in Yocto . . . 11

5 5 Build securiPHY image **13**
- 5.1 5.1 Building a Signed Image . . . 13
- 5.2 5.2 Configure the securiPHY build . . . 13

6 6 Updating securiPHY **15**
- 6.1 6.1 Build update bundle . . . 15
- 6.2 6.2 Migration Steps . . . 15

7 7 Secure Boot **17**
- 7.1 7.1 Chain of Trust . . . 17
- 7.2 7.2 Boot Process . . . 17
- 7.3 7.3 Image Signing . . . 19
- 7.4 7.4 Flattened Image Tree (FIT) . . . 21

8 8 SecuriPHY Provisioning Scripts **23**
- 8.1 8.1 phyprovisioning-install-emmc . . . 23
- 8.2 8.2 physecurekeystorage-install . . . 23
- 8.3 8.3 physecurestorage-install . . . 24

9 9 Activate Secure Boot on the Device **27**
- 9.1 9.1 Activate Secure Boot for NXP SoC . . . 27
- 9.2 9.2 Activate Secure Boot for TI K3 SoC . . . 32
- 9.3 9.3 Next Steps after Activation of Secure Boot . . . 36
- 9.4 9.4 Key Revocation . . . 36

10 10 Kernel Module Signing **39**
10.1 10.1 Enable Kernel Module Signing Facility 39

11 11 Device Tree Overlay and Secure Boot **41**
11.1 11.1 Device Tree Overlay for i.MX6UL and i.MX6 41
11.2 11.2 Device Tree Overlay for other SoCs 41

12 12 Secure Key Storage **43**
12.1 12.1 NXP i.MX CAAM 44
12.2 12.2 Trusted Execution Environment: OP-TEE 44
12.3 12.3 Trusted Platform Module (TPM) 2.0 46
12.4 12.4 Kernel Key Retention Service for Filesystem Encryption 47
12.5 12.5 Cryptographic Token Interface PKCS#11 48

13 13 Secure Storage **51**
13.1 13.1 Filesystem with Integrity vs Authenticated Filesystem 52
13.2 13.2 Requirements for Filesystem Encryption 52

14 14 Hardening of the System **53**

15 15 Physical Security **55**
15.1 15.1 Secure JTAG 55
15.2 15.2 Disable Serial Downloader 59
15.3 15.3 Force Internal Boot 60
15.4 15.4 Disable Boot from External Memory 60

16 16 Keys and Certificates Management **63**
16.1 16.1 Public Key Infrastructure Tree (PKI tree) 63
16.2 16.2 PHYTEC Development Keys (phytec-dev-ca) 63
16.3 16.3 Create Custom PKI Tree 64
16.4 16.4 Create NXP AHAB / HABV4 Keys 64
16.5 16.5 Create TI K3 Keys 65
16.6 16.6 Create Kernel FIT-Image Key 71
16.7 16.7 Create Kernel Module Signing Key 72
16.8 16.8 Create RAUC Update Certificates 72
16.9 16.9 Create SSH Keys and Certificates 72

17 17 Security Vulnerabilities **73**
17.1 17.1 CycloneDX SBOM 73
17.2 17.2 SPDX 3.0 SBOM 73

18 18 SoC Specific Configuration Tools **75**
18.1 18.1 NXP Universal Update Utility (UUU) 75
18.2 18.2 Snagboot Recover and Reflashing Tool 75
18.3 18.3 partup - the System Initialization Program 75
18.4 18.4 NXP i.MX6 and i.MX8M MNP eFuse Tool Crucible 75
18.5 18.5 Secure Provisioning SDK (SPSDK) 76
18.6 18.6 TPM Infineon Firmware Update Tool 77

Compatible BSPs	BSP Release Type	BSP Release Date	BSP Status
BSP-Yocto-Ampliphy-AM62x-PD26.1.0	Major	2026/09/29	Released
BSP-Yocto-Ampliphy-AM64x-PD26.1.0	Major	2026/09/30	Released
BSP-Yocto-NXP-i.MX95-PD26.1.0	Major	2026/10/01	Released

This manual applies to all Wrynose based PHYTEC releases.

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.

CHAPTER

ONE

1 INTRODUCTION

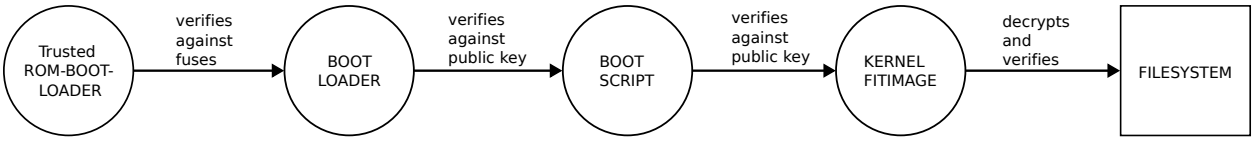
PHYTEC's Yocto distribution securiPHY supports different security mechanism. The security features have impact on the bootloader, the Linux kernel, Device Tree, and root filesystem. This manual describes how security features are used and implemented on various PHYTEC platforms. Note that different modules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

1.1 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 detetcted (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.
Post-Quantum Cryptography (PQC)	Algorithms resistant to classical and quantum attacks, replacing RSA/ECDSA (NIST ML-DSA, ML-KEM). On securiPHY used for AHAB secure boot on the i.MX95 (ML-DSA keys).

1.2 1.2 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.



1.3 1.3 Phytec Yocto Distributions

	ampliphy	securiPHY-provisioning	securiPHY
Use case	early development	for production => provisioning initialize a device for the field	development with security focus
Image	phytec-headless-image	phytec-provisioning-image	phytec-securiphy-image
Update support	no	no	yes
Secure boot chain	no	yes	yes
serial console	yes	yes	no
SSH access	root without password	root without password	root with client certificate

CHAPTER

TWO

2 PROVISIOINING

Note

Create and use your own keys and certificates for signing your images. Burn the right key into the Controller eFuse.

2.1 2.1 Prepare Provisioning

Flash the image `phytec-provisioning-image-*.partup` to the SD card using partup. Set the boot mode to SD card according to the BSP manual for the used board. Power the device and stop in the U-Boot shell.

2.2 2.2 Activate Secure Boot

The first step to activate Secure Boot on your device is to burn the secure eFuse configuration according to *9 Activate Secure Boot on the Device*.

Warning

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

2.3 2.3 Provisioning the eMMC

After the fuses are burned we can continue with provisioning the eMMC. Only the eMMC should be used for the production image. Boot into the provisioning image and log in as root. There is no password needed at this point. Login over SSH is also possible. The next three steps install the security image to the eMMC. The scripts are already included in the provisioning image. More information about the provisioning scripts is available in *8 SecuriPHY Provisioning Scripts*.

2.3.1 2.3.1 phyprovisioning-install-emmc

The tool `phyprovisioning-install-emmc` is used to initialize the eMMC and to create all the required partitions. It requires the `phytec-securiphy-image-*.partup` package as input. Provide the file on the SD card or through the network.

```
target:~$ phyprovisioning-install-emmc \
--filesystem /tmp/phytec-securiphy-image-phyboard-nash-imx93-1.rootfs.partup \
--flashpath /dev/mmcblk0 \
--newemmc
```

This command also mounts the partup package at `/media/data_partup`.

2.3.2 2.3.2 physecurekeystorage-install

The `physecurekeystorage-install` tool can initialize all supported secure key storages of your machine, but only one can be active at a time. For example, the phyBOARD Pollux i.MX8MP supports Trusted TEE, Trusted TPM, Trusted CAAM and Secure CAAM, one of them has to be selected for the secure key storage.

The following command creates the trusted and encrypted keys with Trusted TEE.

```
target:~$ run0 physecurekeystorage-install --newkeystorage trustedtee
```

2.3.3 2.3.3 physecurestorage-install

The tool `physecurestorage-install` is used to create and populate the encrypted filesystem.

```
target:~$ run0 physecurestorage-install \
--filesystem /media/data_partup/phytec-securiphy-image-phyboard-nash-imx93-1.rootfs.ext4 \
--flashpath /dev/mmcblk0 \
--flashlayout 5,6 \
--newsecurestorage intenc
```

The provisioning is now finished. Shutdown the provisioning system with `shutdown 0` and turn off the power to the device.

CHAPTER

THREE

3 BOOTING SECURIPHY

After successfully provisioning the device you can use the freshly installed securiPHY system.

Set the boot jumper to eMMC and power the device.

3.1 3.1 SSH Certificate Authentication Login

Once the `phytec-securiphy-image` is running, the serial console only shows the kernel log. Login is only possible with SSH now.

3.1.1 3.1.1 Login with development certificates

The keys and certificates for SSH access are included in phytec-dev-ca. Download and extract the latest release.

```
host:~$ cd phytec-dev-ca/ssh-ca/user-root
host:~$ ssh-add user_root_ed25519 # enter password sshtest
```

Check if private key and certificate have been added successfully.

```
host:~$ ssh-add -l
256 SHA256:/Cw8W1Z0N/XFCmLvBLDPDuwh+aFYWKWWinLxX/Rx918 root (ED25519)
256 SHA256:/Cw8W1Z0N/XFCmLvBLDPDuwh+aFYWKWWinLxX/Rx918 root (ED25519-CERT)
```

After that, connect to the board.

```
host:~$ ssh -v -o IdentitiesOnly=yes -i user_root_ed25519 root@192.168.3.11
```

3.1.2 3.1.2 Create user certificate

The development user certificate *user_root_ed25519-cert.pub* allows unrestricted access to the target device. A certificate with restricted access can be created as follows:

```
host:~$ ssh-keygen -s user_ca -I root -n root \
-V "YYYYMMDDHHMMSS:YYYYMMDDHHMMSS" \
-O source-address=192.168.0.0/16 \
-O clear \
-O permit-pty \
user_root_ed25519.pub
```

CHAPTER

FOUR

4 USING YOUR OWN KEYS AND CERTIFICATES

See chapter *16 Keys and Certificates Management* to create your own keys and certificates for secure boot.

4.1 4.1 Key Settings in Yocto

The created keys and certificates can be used in the yocto build by setting specific variables. A list of all variables is in `meta-ampliphy/classes/secureboot.bbclass` Set the variables corresponding to the used SoC in `conf/local.conf` of the Yocto build directory.

4.1.1 4.1.1 U-Boot signing

4.1.1.1 NXP i.MX8MP using 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_SIGN_SRKFUSE_PATH ??= "${CERT_PATH}/nxp_habv4_pki/crts/SRK_1_2_3_4_table.bin"
BOOTLOADER_HABV4_SRK_INDEX ??= "0
```

4.1.1.2 NXP i.MX91/93 using 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"
```

4.1.1.3 NXP i.MX95 using AHAB

```
AHAB_SRK0 ?= "${CERT_PATH}/nxp_ahab_pki/crts/SRK1_sha512_secp521r1_v3_usr_crt.pem"
AHAB_SRK1 ?= "${CERT_PATH}/nxp_ahab_pki/crts/SRK2_sha512_secp521r1_v3_usr_crt.pem"
AHAB_SRK2 ?= "${CERT_PATH}/nxp_ahab_pki/crts/SRK3_sha512_secp521r1_v3_usr_crt.pem"
AHAB_SRK3 ?= "${CERT_PATH}/nxp_ahab_pki/crts/SRK4_sha512_secp521r1_v3_usr_crt.pem"
AHAB_SIGN_SRK ?= "${CERT_PATH}/nxp_ahab_pki/keys/SRK1_sha512_secp521r1_v3_usr_key.pem"
AHAB_SIGN_SRK_PASS ?= "${CERT_PATH}/nxp_ahab_pki/keys/key_pass.txt"

AHAB_PQC_SRK0 ?= "${CERT_PATH}/nxp_ahab_pki_pqc/keys/SRK0_mldsa87_key.pub.pem"
AHAB_PQC_SRK1 ?= "${CERT_PATH}/nxp_ahab_pki_pqc/keys/SRK1_mldsa87_key.pub.pem"
AHAB_PQC_SRK2 ?= "${CERT_PATH}/nxp_ahab_pki_pqc/keys/SRK2_mldsa87_key.pub.pem"
AHAB_PQC_SRK3 ?= "${CERT_PATH}/nxp_ahab_pki_pqc/keys/SRK3_mldsa87_key.pub.pem"
```

(continues on next page)

(continued from previous page)

```
AHAB_PQC_SIGN_SRK ?= "${CERT_PATH}/nxp_ahab_pki_pqc/keys/SRK0_mldsa87_key.pem"
AHAB_PQC_SIGN_SRK_PASS ?= "${CERT_PATH}/nxp_ahab_pki_pqc/keys/key_pass.txt"
```

Additional configuration can be done in the configuration file `imx95_ahab_sign.yaml.in` in `meta-spsdk/dynamic-layers/fsl-bsp-release/recipes-bsp/imx-mkimage/imx-boot/`. The SRK index is set in `imx95_ahab_sign.yaml.in` with the default `used_srk_id: 0`.

4.1.1.4 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"
```

4.1.2 4.1.2 FitImage signing

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

4.1.3 4.1.3 Kernel module signing

```
MODSIGN_KEY ?= "${CERT_PATH}/kernel_modsign/kernel_modsign.pem"
MODSIGN_CERT ?= "${CERT_PATH}/kernel_modsign/kernel_modsign.pem"
```

4.1.4 4.1.4 SSH CA public key

```
SSH_CA_PUBKEY ?= "${CERT_PATH}/ssh-ca/user-client-ca/user_client_ca.pub"
```

CHAPTER

FIVE

5 BUILD SECURIPHY IMAGE

Building an image requires a working Yocto build setup according to the Yocto Reference Manual.

5.1 5.1 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, the `DISTRO` needs to be set to securiphy-vendor-provisioning or securiphy-provisioning. The main parts for the provisioning-image are the bootloader and the FIT-image, 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-securiphy-image for the eMMC or ubifs with RAUC update support, `DISTRO` needs to be set to securiphy-vendor or securiphy.

```
# for all SoC
host:~$ bitbake phytec-securiphy-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 Contiguous Memory Allocation (CMA; kernel boot parameter, setting in bootloader). The allocated RAM for CMA can be too much, which is important for systems with 256 MiB or 512 MiB RAM.

5.2 5.2 Configure the securiPHY build

Configuration options in the Yocto build:

- `UBOOT_SIGN_ENABLE` controls if the fitImage will be signed. The default is `1` enabled.
- The `DISTROOVERRIDE secureboot` enables security features and bootloader signing for NXP i.MX SoCs. To enable bootloader signing on TI K3 SoCs use a machine with the `MACHINEOVERRIDE secureenforced`.
- `UBOOT_DISABLE_CONSOLE` disables the U-Boot console. The default is `1`.
- Setting `SERIAL_CONSOLES` to empty disables the Linux console.

- `DISTRO_FEATURE ssh-authentication` requires SSH certificate authentication.
- `DISTRO_FEATURE ssh-hardening` enables additional SSH hardening options.

CHAPTER

SIX

6 UPDATING SECURIPHY

The update mechanism used in securiPHY is RAUC (Robust Auto-Update Controller). The Phytec RAUC Manual describes its integration and general usage in the Phytec BSP.

6.1 6.1 Build update bundle

The update bundle can be build with the command.

```
host:~$ bitbake phytec-securiphy-bundle
```

6.2 6.2 Migration Steps

6.2.1 6.2.1 Yocto scarthgap to walnascar or newer

The optee-client `tee-supplicant` uses a newly added user `teesuppl`. This user has no access to data previously stored by optee in the config partition (`/mnt/config/tee`).

There are 2 ways to handle this change:

1. Set optee client user back to root in the build process Add the following bbappend to your layer.

Listing 1: recipes-security/optee/optee-client_%.bbappend

```
EXTRA_OECMAKE += " -DCFG_TEE_SUPPL_USER=root -DCFG_TEE_SUPPL_GROUP=root"
```

2. Change ownership of the optee data after the update from `root` to `teesuppl`

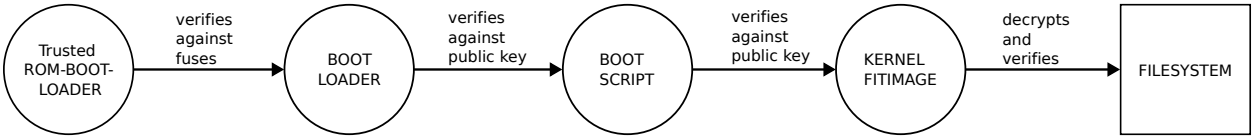
CHAPTER

SEVEN

7 SECURE BOOT

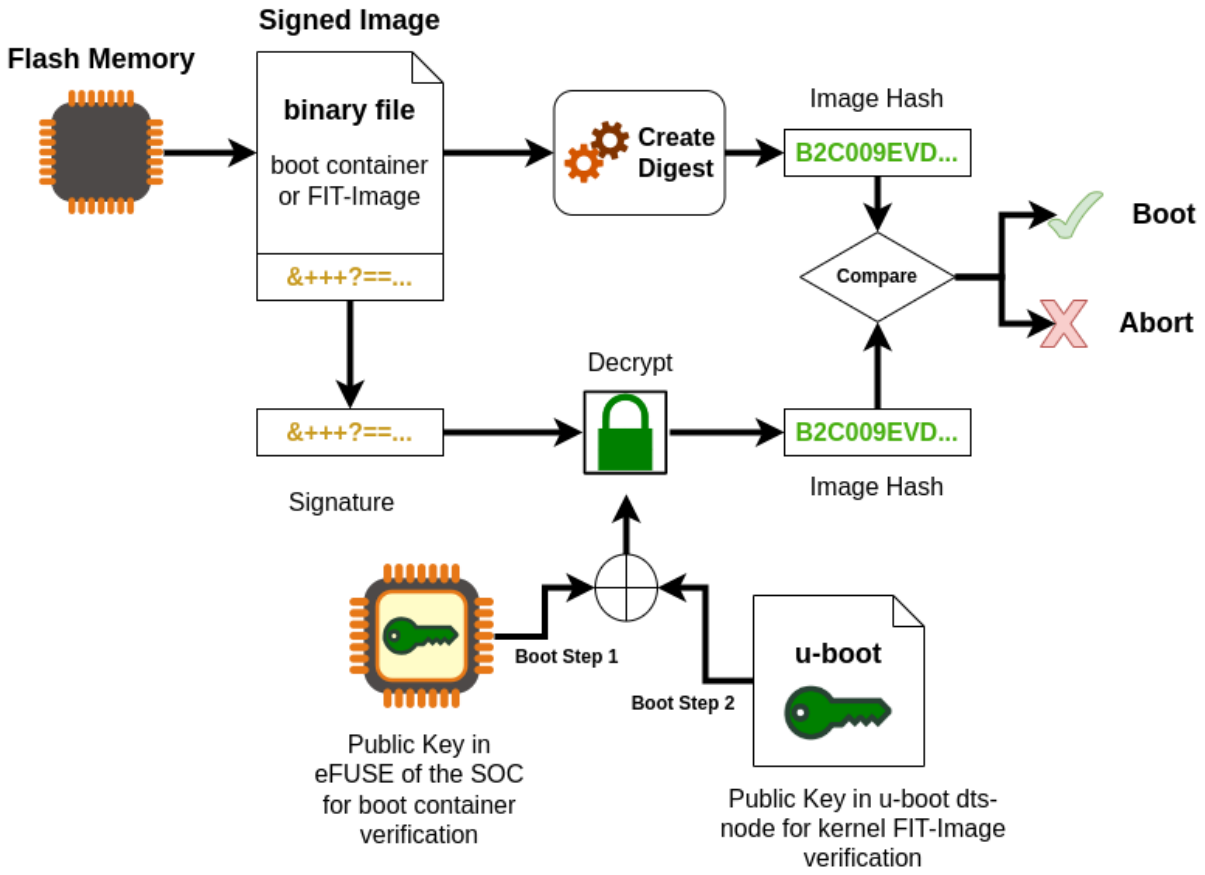
7.1 7.1 Chain of Trust

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.



7.2 7.2 Boot Process

The boot process differs between the different vendors of SoCs and even between SoCs of the same vendor. 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 boot script. (**Boot Step 2**)
4. Next, U-Boot executes the boot script, which loads and verifies the FIT-image containing a Linux kernel, DTB, and ramdisk. (**Boot Step 3**)

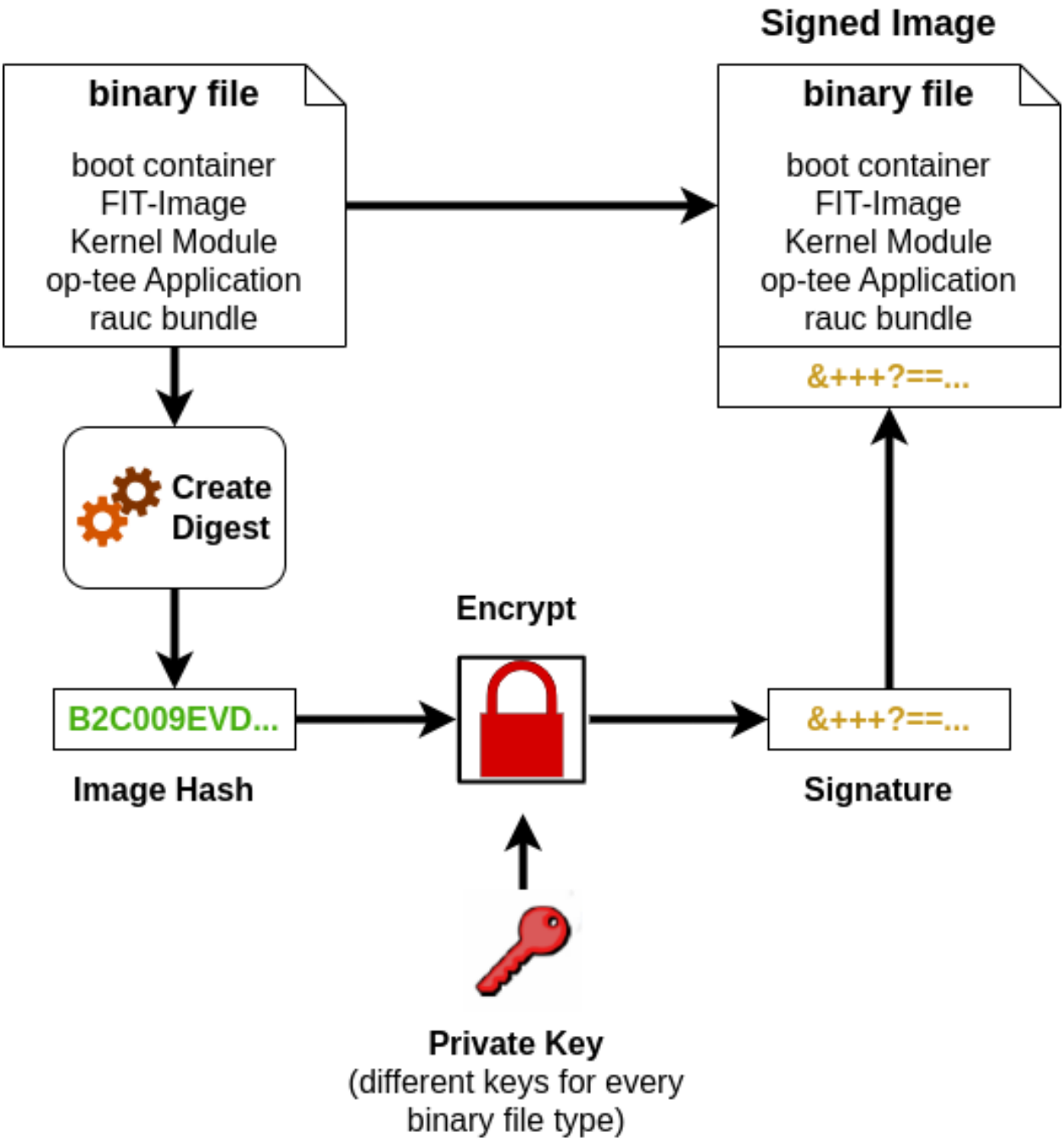
5. If built with signed kernel modules (standard), Linux will only load kernel modules verified with a kernel compiled-in public key

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 uses only the built-in environment and only loads some necessary variables for RAUC update mechanism.

7.3 7.3 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 fitImage, kernel modules, op-tee applications or the RAUC update bundle.
2. A signature is derived from the hash with a private key.
3. The signature is added to the binary file.

7.4 7.4 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.

CHAPTER

EIGHT

8 SECURIPHY PROVISIONING SCRIPTS

The `phytec-securiphy-image` needs to be installed with the following provisioning scripts. They are included in the `phytec-provisioning-image`.

8.1 8.1 phyprovisioning-install-emmc

The phyprovisioning-install-emmc script creates the partitions on the eMMC and installs bootloader and kernel.

```
target:~$ phyprovisioning-install-emmc -h

PHYTEC Install Script v0.7 for eMMC

Usage:  phyprovisioning-install-emmc [PARAMETER] [ACTION]

Example:
    phyprovisioning-install-emmc --filesystem /media/phytec-security-image.rootfs.partup --
↪newemmc

One of the following action can be selected:
    -n | --newemmc    Copy filesystem image to eMMC

The following PARAMETER must be set for eMMC provisioning:
    -p | --flashpath <flash device> = /dev/mmcblk2
    -s | --filesystem <path to partup package>
```

8.2 8.2 physecurekeystorage-install

The physecurekeystorage-install tool initializes all supported secure key storages of your machine, but only one can be active at a time.

```
root@imx8mp-phyflex-libra-rdk-2:~# 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
```

(continues on next page)

(continued from previous page)

```
    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
```

8.3 8.3 physecurestorage-install

The physecurestorage-install tool initializes the filesystem with encryption or integrity and encryption combined.

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

PHYTEC Install Script v1.5 for Secure Storage

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
                           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 the filesystem as tgz or ext4, 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 numbers on the target device. Set all partitions used for RAUC.

CHAPTER

NINE

9 ACTIVATE SECURE BOOT ON THE DEVICE

Warning

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

For Secure Boot only public information is burned to SoCs from NXP and TI. When building the securiphy or securiphy-vendor 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 *12 Secure Key Storage*

9.1 9.1 Activate Secure Boot for NXP SoC

For NXP SoCs 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/`
- NXP i.MX91/93 with AHAB: example `imx93_SRK_1_2_3_4_fuse.bin` file in `yocto/phytec-dev-ca/nxp_ahab_pki/fuses/`
- NXP i.MX95 with AHAB classic: example `imx95_ahab_oem_srk_hash_nxpele.bcf` file in `yocto/phytec-dev-ca/nxp_ahab_pki/fuses/`
- NXP i.MX95 with AHAB PQC: example `imx95_ahab_oem_srk_pqc_hash_nxpele.bcf` file in `yocto/phytec-dev-ca/nxp_ahab_pki_pqc/fuses/`

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

9.1.1 9.1.1 i.MX6 with HAB

Check the current state of your device

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

Burn the SRK

First, 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 the result, use again:

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

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 or AHAB events.
- Verify the SRK eFuses have been burned correctly.

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

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

9.1.2 9.1.2 i.MX8M Mini/Nano/Plus with HAB

Check the current state of your device

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

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

Burn the SRK

Extract the fuse values from `SRK_1_2_3_4_fuse.bin` on the host:

```
host:~$ od -t x4 SRK_1_2_3_4_fuse.bin
0000000 9a842534 b0491ab4 d5b6a07b fd92dce7
0000020 c10dc87c d8bd04a9 704e9fe4 9b025359
```

Burn the fuses from the U-Boot shell:

```
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 2 0x704e9fe4
u-boot=> fuse prog 7 3 0x9b025359
```

Reset the board:

```
u-boot=> reset
```

Read the fuses and check the programmed values:

```
u-boot=> fuse read 6 0 4
0x00000000: 9a842534 b0491ab4 d5b6a07b fd92dce7
u-boot=> fuse read 7 0 4
0x00000000: c10dc87c d8bd04a9 704e9fe4 9b025359
```

Check the HAB state:

```
u-boot=> hab_status
No Events Found!
```

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 or AHAB events.
- Verify the SRK eFuses have been burned correctly.

```
u-boot=> fuse prog 1 3 0x2000000
```

Set the SRK write protection:

```
u-boot=> fuse prog 0 0 0x200
```

9.1.3 9.1.3 i.MX91/93 with AHAB

Check the current state of your device

```
u-boot=> ahab_status
0x0287fad6
IPC = MU APD (0x2)
CMD = ELE_OEM_CNTN_AUTH_REQ (0x87)
IND = ELE_BAD_KEY_HASH_FAILURE_IND (0xFA)
STA = ELE_SUCCESS_IND (0xD6)
```

(continues on next page)

(continued from previous page)

```
0x0287fad6
IPC = MU APD (0x2)
CMD = ELE_OEM_CNTN_AUTH_REQ (0x87)
IND = ELE_BAD_KEY_HASH_FAILURE_IND (0xFA)
STA = ELE_SUCCESS_IND (0xD6)
```

The reason for the `ELE_BAD_KEY_HASH_FAILURE_IND` is the zero hash in the fuses, which is wrong for the signature of the boot container.

Burn the SRK

Extract the fuse values from `imx93_SRK_1_2_3_4_fuse.bin` on the host:

```
host:~$ od -t x4 imx93_SRK_1_2_3_4_fuse.bin
0000000 baaf5d2c e92e0323 23c0ba08 10e7973f
0000020 678de0d5 966d3584 a541dfbe 6ea06dba
```

Burn the fuses from the U-Boot shell:

```
u-boot=> fuse prog 16 0 0xbaaf5d2c
u-boot=> fuse prog 16 1 0xe92e0323
u-boot=> fuse prog 16 2 0x23c0ba08
u-boot=> fuse prog 16 3 0x10e7973f
u-boot=> fuse prog 16 4 0x678de0d5
u-boot=> fuse prog 16 5 0x966d3584
u-boot=> fuse prog 16 6 0xa541dfbe
u-boot=> fuse prog 16 7 0x6ea06dba
```

Reset the board:

```
u-boot=> reset
```

Read the fuses back and compare them to the written values:

```
u-boot=> fuse read 16 0 8
0x00000000: baaf5d2c e92e0323 23c0ba08 10e7973f
0x00000004: 678de0d5 966d3584 a541dfbe 6ea06dba
```

Check the state of AHAB:

```
u-boot=> ahab_status
Lifecycle: 0x00000008, OEM Open
No Events Found!
```

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 or AHAB events.

- Verify the SRK eFuses have been burned correctly.

```
u-boot=> ahab_close
```

Set SRK read protection:

```
u-boot=> fuse prog 0 4 0x4000
```

Set the override protection for the shadow register:

```
u-boot=> fuse prog 0 4 0x2000
```

Set write protection for the SRK fuses:

```
u-boot=> fuse prog 0 4 0x1000
```

9.1.4 9.1.4 i.MX95 with AHAB

The i.MX95 supports classic (RSA, ECC), PQC (ML-DSA), and hybrid (classic + PQC) key configurations. The commands below burn the SRK hashes for either the classic or the PQC configuration. The example files are located in `yocto/phytec-dev-ca/nxp_ahab_pki/fuses/` (classic) and `yocto/phytec-dev-ca/nxp_ahab_pki_pqc/fuses/` (PQC). For a hybrid configuration, burn both the classic and the PQC SRK hashes.

Check the current state of your device

```
u-boot=> ahab_status
0x0287fad6
IPC = MU APD (0x2)
CMD = ELE_OEM_CNTN_AUTH_REQ (0x87)
IND = ELE_BAD_KEY_HASH_FAILURE_IND (0xFA)
STA = ELE_SUCCESS_IND (0xD6)

0x0287fad6
IPC = MU APD (0x2)
CMD = ELE_OEM_CNTN_AUTH_REQ (0x87)
IND = ELE_BAD_KEY_HASH_FAILURE_IND (0xFA)
STA = ELE_SUCCESS_IND (0xD6)
```

The reason for the `ELE_BAD_KEY_HASH_FAILURE_IND` is the zero hash in the fuses, which is wrong for the signature of the boot container.

Burn the SRK

Note

The `.bcf` files are generated during the Yocto build in the build directory: `tmp-securiphy-vendor/work/<machine>-phytec-linux/imx-boot/1.0/build-sign/fuse-script/`

1. Disconnect from U-Boot shell
2. Burn fuses from the host using `nxpele` from `SPSDK`:

```
# classic
host:~$ nxpele -p /dev/ttyUSB0 -d uboot_serial -f mimx9596 batch imx95_ahab_oem_srk_hash_
↪nxpele.bcf
# PQC
host:~$ nxpele -p /dev/ttyUSB0 -d uboot_serial -f mimx9596 batch imx95_ahab_oem_srk_pqc_
↪hash_nxpele.bcf
```

3. Reconnect to U-Boot shell and `u-boot=> reset`
4. Check the state of AHAB:

```
u-boot=> ahab_status
Lifecycle: 0x00000010, OEM Open

No Events Found!
```

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 or AHAB events.
- Verify the SRK eFuses have been burned correctly.

```
u-boot=> ahab_close
```

Set SRK read protection:

```
u-boot=> fuse prog 0 4 0x4000
u-boot=> fuse prog 1 0 0x4000
```

Set the override protection for the shadow register:

```
u-boot=> fuse prog 0 4 0x2000
u-boot=> fuse prog 1 0 0x2000
```

Set write protection for the SRK fuses:

```
u-boot=> fuse prog 0 4 0x1000
u-boot=> fuse prog 1 0 0x1000
```

9.2 9.2 Activate Secure Boot for TI K3 SoC

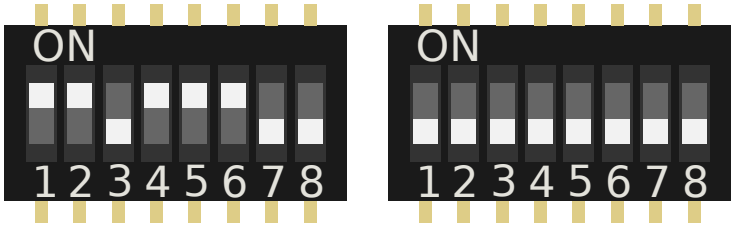
The fuses can only be burnt with the OTP-Keywriter, which can be built as described in chapter *16 Keys and Certificates Management*. The public key is burnt in the SoC. Keys may be overwritten, if the incremental approach is used. If the KEYREV value is set to either 1 or 2, then the device is an HS-SE device with active Secure Boot. After that, certificates cannot be written neither with the keywriter tool nor otherwise anymore.

9.2.1 9.2.1 Check the State of your Device

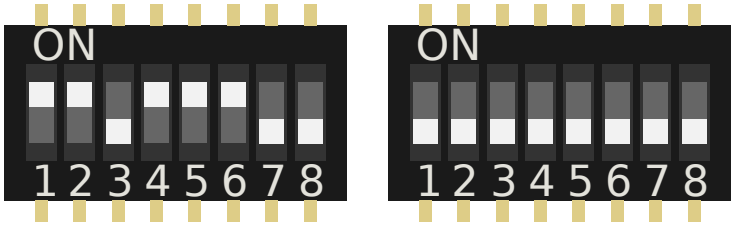
When UART boot mode is selected, boot ROM reports the SoC ID via UART, for both GP and HS devices. Additionally, this can determine the device configuration, which can be used to check if device was converted from HS-FS to HS-SE after key programming.

1. Set the boot mode switches into UART boot mode.

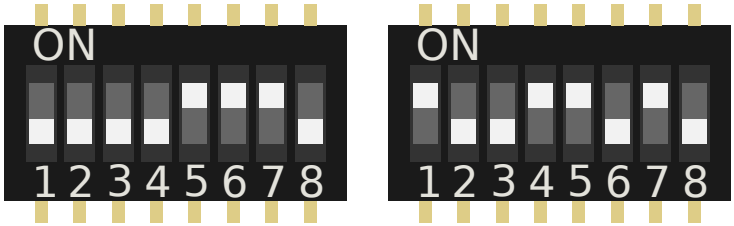
   AM62x: S5 & S6

   AM64x: S3 & S4

   AM68x: S8 & S7

2. Open your serial terminal on `/dev/ttyUSB0`
3. Power up your board
4. Copy the entire string up until "CCC" (ping characters) and paste the string in `<MCU_PLUS_SDK_INSTALL_DIR>/source/security/sbl_keywriter/tools/socid.txt` (For AM68x please download the otp_keywriter from AM62 to use this Python script.)
5. Run in `<MCU_PLUS_SDK_INSTALL_DIR>/source/security/sbl_keywriter/tools/`

```
host:~$  python parse_uart_boot_socid.py socid.txt
-----------------------
SoC ID Public ROM Info:
-----------------------
SubBlockId           :
SubBlockSize         :
DeviceName           : am62x
DeviceType           : HSFS
```

6. After successful activation of Secure Boot the DeviceType is `HSSE`
7. Additionally, you can see the SHA512 hash of the used keys in the output with build profile DEBUG.

This feature depends on the keywriter version. Then you can compare it with the hash of your signing key:

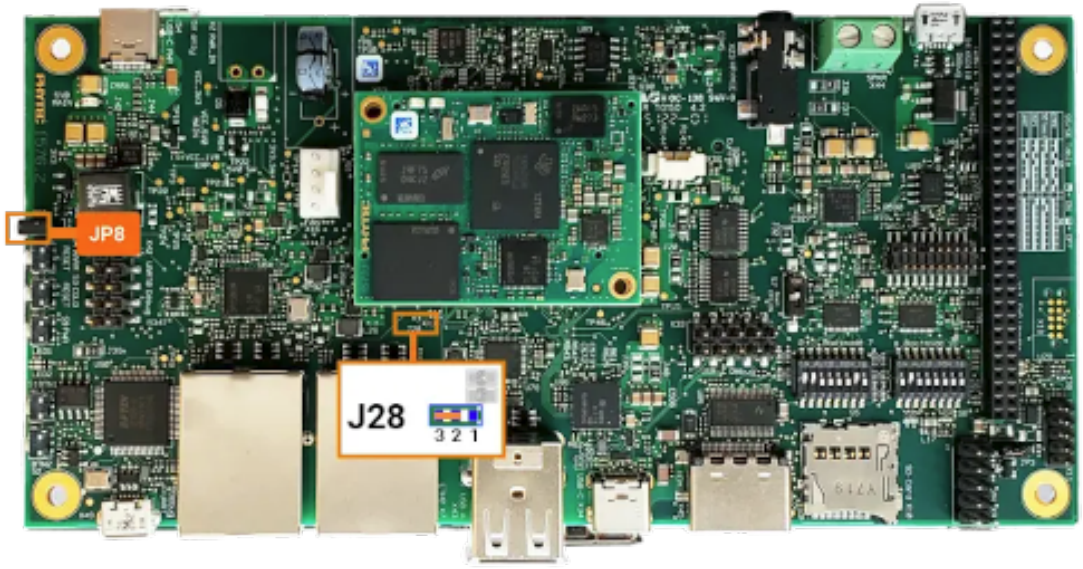
```
host:~$  python parse_uart_boot_socid.py socid.txt
...
----------------------
SoC ID Secure ROM Info:
----------------------
Sec SubBlockId       : 2
Sec SubBlockSize     : 166
Sec Prime            : 0
Sec Key Revision     : 1
Sec Key Count        : 1
Sec TI MPK Hash      :␣
↪d68ecb2c055dff11ade95bd927e837d2a53bc23b0a2800cebce4f106bcf309df2213912d77a157a8b7c2df40672a06a918034aa4c
Sec Cust MPK Hash    :␣
↪99adc5d401188641b1187d28b8e3f7bf94defcf48b503f6b45d6fe677da161a98a27956ecbc1c2f4af43867f56398bd147f54ce24
Sec Unique ID        : 2b8638ba724a82d7c6c0d1d19b56a13bd52da90bfcfa2efb6d6f8d26ed8830f5
```

9.2.2 9.2.2 Write the Certificate and Lock the Device

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. Alternatively you can use UART Boot or USBDFU, too.

For the TI AM62x

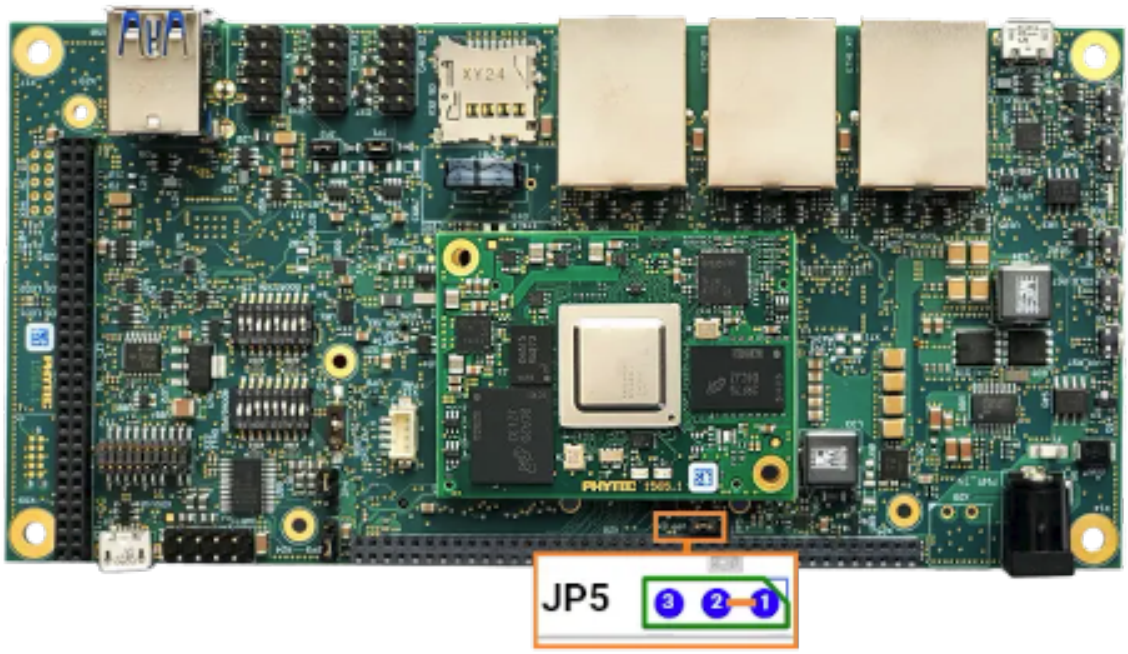
Now you must set JP8 on the development kit for AM62x in order to flash the keys.



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

For the TI AM64x

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



For the TI AM68x/TDA4x

You do not need to set any external jumper to flash the keys because a GPIO on phyCORE-AM68x is used to switch the additional voltage.

Once this configuration is set, plug the SD card into the kit and boot as you normally would. You should see the output messages from the keywriting:

- AM62x and AM64x on Standard Debug Port:
- AM68x on MCU UART Port (Header X33) with Serial Converter

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.

In case something does not went as intended, check the OTP key writer error codes

9.3 9.3 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 at *15 Physical Security*
- 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.
- please check the NXP and TI websites for more information

9.4 9.4 Key Revocation

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

9.4.1 9.4.1 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.

The following keys are available for HABV4:

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 Mini/Nano/Plus
`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.

CHAPTER

TEN

10 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.

10.1 10.1 Enable Kernel Module Signing Facility

To enable the kernel module signing facility, add the following `DISTRO_FEATURES` 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 development key. Unless you create your custom key, this feature does not offer any protection.

CHAPTER

ELEVEN

11 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.

11.1 11.1 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 fdtapply mechanism from the `source/meta-phytec/classes/fitimage.bbclass`. More information can be found in FIT-image section in *7 Secure Boot*.

The barebox contains an Ethernet PHY detection, which boots the correct configuration from the FIT-image.

11.2 11.2 Device Tree Overlay for other SoCs

11.2.1 11.2.1 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.

11.2.2 11.2.2 Run Time

Configuring device tree overlays through `overlays.txt` is disabled in secure context as this file can't be signed and verified.

Device tree overlays specified in the U-Boot variable `fit_overlay_conf` will be used. This variable can either be defined in the default U-Boot environment or in the ampliphy-boot bootscript.

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.

CHAPTER

TWELVE

12 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 * on i.MX9 phyBOARD-Nash	tpm

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

12.1 12.1 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

12.1.1 12.1.1 Prerequisites and Caveats

Secure Boot 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.

12.1.2 12.1.2 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
```

12.2 12.2 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-supplicant: 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.

12.2.1 12.2.1 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.

12.2.2 12.2.2 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 0xffff88e2c000.
Bus error

target:~$ $?
135
```

(continues on next page)

(continued from previous page)

```
target:~$ devmem2 0x57c00000
/dev/mem opened.
Memory mapped at address 0xffffb4f3c000.
Read at address  0x57C00000 (0xffffb4f3c000): 0xA0A28501
```

- In this example the 0x5600000 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.

12.3 12.3 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.

12.3.1 12.3.1 Initialization of the TPM

At first, the TPM 2.0 must be initialized with the command `tss2_provision`. This command is used in the tool `physecurekeystorage-install`, when choosing the *trustedtpm* key type.

12.3.2 12.3.2 Error Codes of the TPM

If you want decode some TPM error codes, then use the command tpm2_rc_decode on the device.

```
target:~$ tpm2_rc_decode 0x00060025
fapi:No certificate
```

- ErrorCode (0x00060025) Verify EK certificate

```
target:~$ tss2_provision
ERROR:fapi:../tpm2-tss-4.0.2/src/tss2-fapi/api/Fapi_Provision.c:944:Fapi_Provision_
↪Finish() ErrorCode (0x00060025) Verify EK certificate
ERROR:fapi:../tpm2-tss-4.0.2/src/tss2-fapi/api/Fapi_Provision.c:174:Fapi_
↪Provision() ErrorCode (0x00060025) Provision
```

Reason: For the check of the manufacture TPM certificate a internet connection is necessary.

Please check:

- your internet connection
- or deactivate the certificate check with adding `"ek_cert_less": "yes"` to /etc/tpm2-tss/fapi-config.json

- **ErrorCode (0x00060001) Failed to verify intermediate certificate / CRL is not yet valid**
  Please check if the date/clock is set correctly
- ErrorCode (0x98E) authorization HMAC check failed

```
ERROR: Esys_DictionaryAttackLockReset(0x98E) - tpm:session(1):the authorization␣
↪HMAC check failed and DA counter incremented
ERROR: Failed DictionaryLockout Reset
```

Reason: The TPM initialisation of the TPM with FAPI use HMAC with a symmetric key. If the TPM configuration on the device is deleted or damaged, then symmtric key for authtentication is lost.

To Reset the TPM completely:

```
target:~$ tpm2_clear -c platform
```

12.4 12.4 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 SoCs.

Key Type	depend on the MACHINE_FEAT	NXP i.MX6 (UL)	NXP i.MX8M MNP	NXP i.MX93/91	TI AM6 Series
trustedtpm	tpm2	x	x	x	x
trustedtee	optee	x	x	x	x
trustedcaam	caam	x (not ULL)	x		
securecaam	caam		x		

12.5 12.5 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/ossl-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-install`, then you can create a device certificate.

```
#set TPM Pin
target:~$ TPM_PIN=1234
# Create self-signed certificate
target:~$ OPENSSL_CONF=provider.conf 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.

The `pkcs11-tool` parameter `--private` can protect the public key and other public objects like certificates for access to delete and read out, because a PIN (password) is necessary for every access.

- You can find a more detailed example with OP-TEE 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.

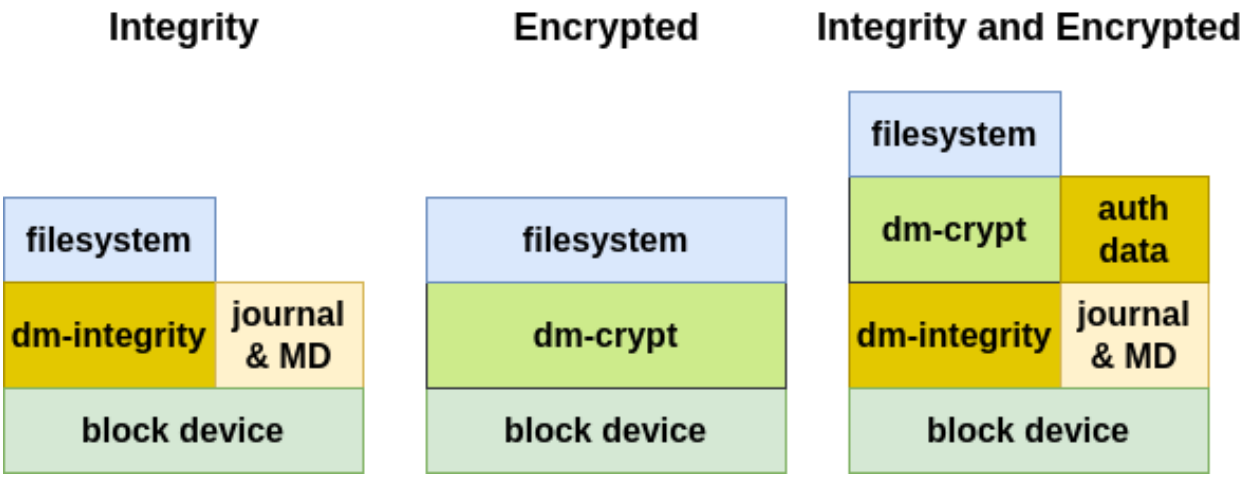
CHAPTER

THIRTEEN

13 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

- securiphy 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.

13.1 13.1 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.

13.2 13.2 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.

13.2.1 13.2.1 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 initramfs)

CHAPTER

FOURTEEN

14 HARDENING OF THE SYSTEM

The `DISTRO_FEATURES="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_FEATURES` 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

CHAPTER

FIFTEEN

15 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, it is necessary to write and read eFuses.

Warning

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

15.1 15.1 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:

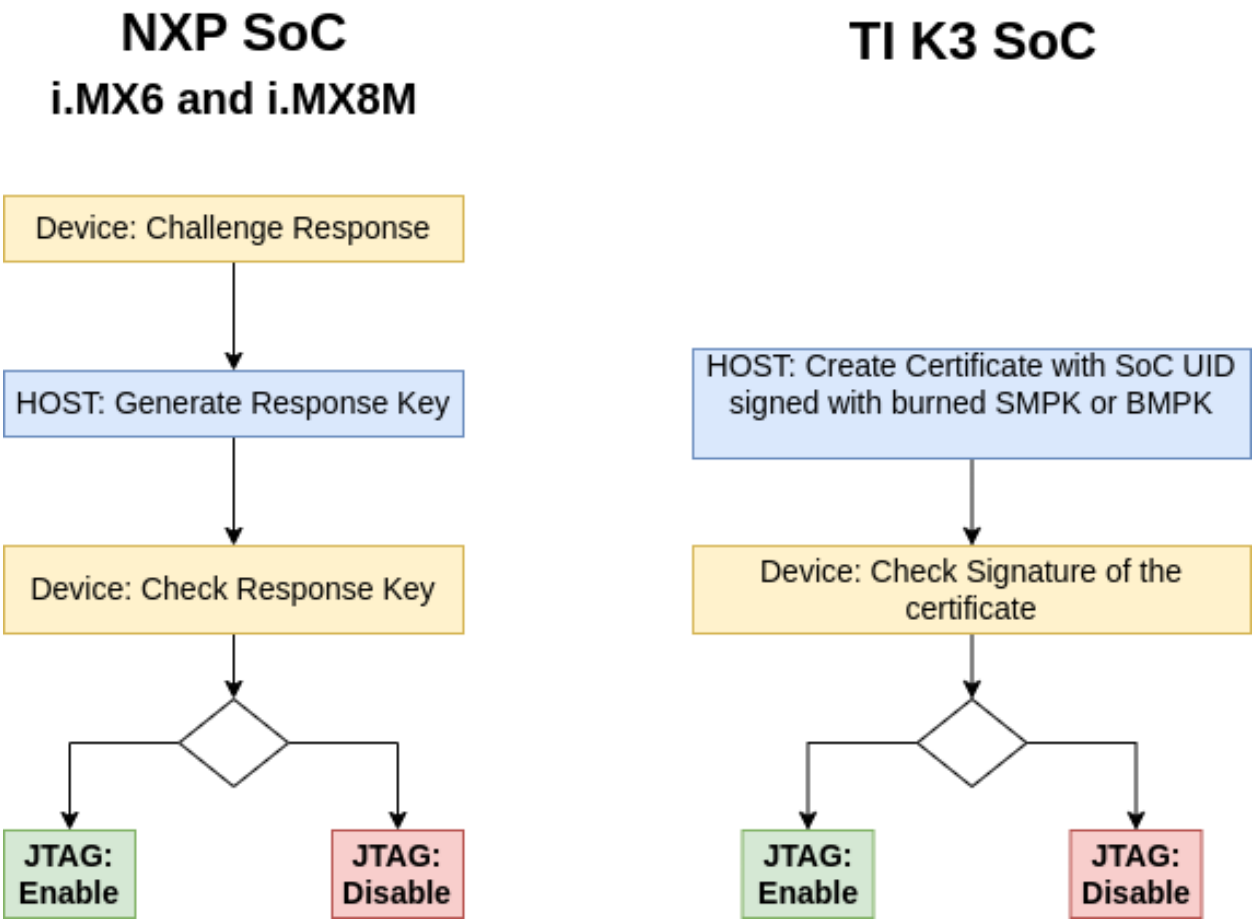
Mode	Security level	Description	NXP SOC	TI SOC
Enabled	low	This is the default mode of operation and you have full access to JTAG.	yes	yes
Disabled debugging	medium	This mode disables debugging but leaves the boundary scan functionality enabled.	yes	
Secure	high	This mode provides high security. JTAG 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

- NXP i.MX93/i.MX91:
  - In the OEM_CLOSED lifecycle mode is the authentication debug mode enabled.
  - In the OEM_LOCKED the ELE Debug and JTAG is disabled.

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 Secure Debug on ELE-AP based i.MX SoCs AN14579
- TI: Secure Debug User Guide or in the restricted security resources for your SoC type.

15.1.1 15.1.1 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
```

- i.MX9 family do not support this mode

15.1.2 15.1.2 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
```

- NXP i.MX93/i.MX91 with U-Boot:

```
#read status for OEM closed
u-boot=> ahab_status
Lifecycle: 0x00000020, OEM Closed
# Set in OEM_LOCKED mode to disable JTAG and debug
u-boot=> ele_message 0x20480000 0x20000 0602951780000000
# Reset the board to activate
u-boot=> reset
#read Status
u-boot=> ahab_status
Lifecycle: 0x00000100, OEM Locked
```

or with the NXP i.MX9 EdgeLock Enclave (ELE) nxpele tool

```
# read status for OEM_CLOSED
host:~$ nxpele -p /dev/ttyUSB0 -f mx93 get-info
```

(continues on next page)

(continued from previous page)

```
ELE get info ends successfully:
Command:          0xda
Version:          2
Length:           160
SoC ID:           9300
SoC version:      A100
Life Cycle:       OEM_CLSD - 0x0040

# Set in OEM_LOCKED mode to disable JTAG and debug
host:~$ nxpele -p /dev/ttyUSB0 -f mx93 -vv forward-lifecycle-update -l OEM_LOCKED

INFO:spsdk.ele.ele_comm:Sent message information:
Command:         ELE_FWD_LIFECYCLE_UP_REQ - (0x95)
Command words:   2
Command data:    False
Response words:  2
Response data:   False
Response status: Success

Forward Lifecycle update ends successfully.

#reset the board
host:~$ nxpele -p /dev/ttyUSB0 -f mx93 reset
# read status
host:~$ nxpele -p /dev/ttyUSB0 -f mx93 get-info
ELE get info ends successfully:
Command:          0xda
Version:          2
Length:           160
SoC ID:           9300
SoC version:      A100
Life Cycle:       OEM_LCKD - 0x0200
```

- TI K3 with keywriter:

  JTAG can be disabled with the OTP keywriter tool by using the parameter `--jtag-disable` for certificate creation and setting the respective part of the eFuse field. See chapter *16 Keys and Certificates Management* for details and further steps.

15.2 15.2 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
```

- NXP i.MX93/i.MX91 with U-Boot:

```
# Disable SDP Mode Completely
in the process of clarification
```

15.3 15.3 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
```

- NXP i.MX93/i.MX91 with U-Boot:

Boot Device	BOOT_CFG0
eMMC	0x20020002
SD Card	0x20000103

```
# set boot mode for eMMC with eMMC Bus width to 0b01 (8 bit)
# and BT_FUSE_SEL (Boot fuses already programmed) bit
u-boot=> fuse prog 3 0 0x20020002
# set the FORCE_BT_FROM_FUSE bit
u-boot=> fuse prog 3 0 0x40000000
```

15.4 15.4 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
```

CHAPTER

SIXTEEN

16 KEYS AND CERTIFICATES MANAGEMENT

16.1 16.1 Public Key Infrastructure Tree (PKI tree)

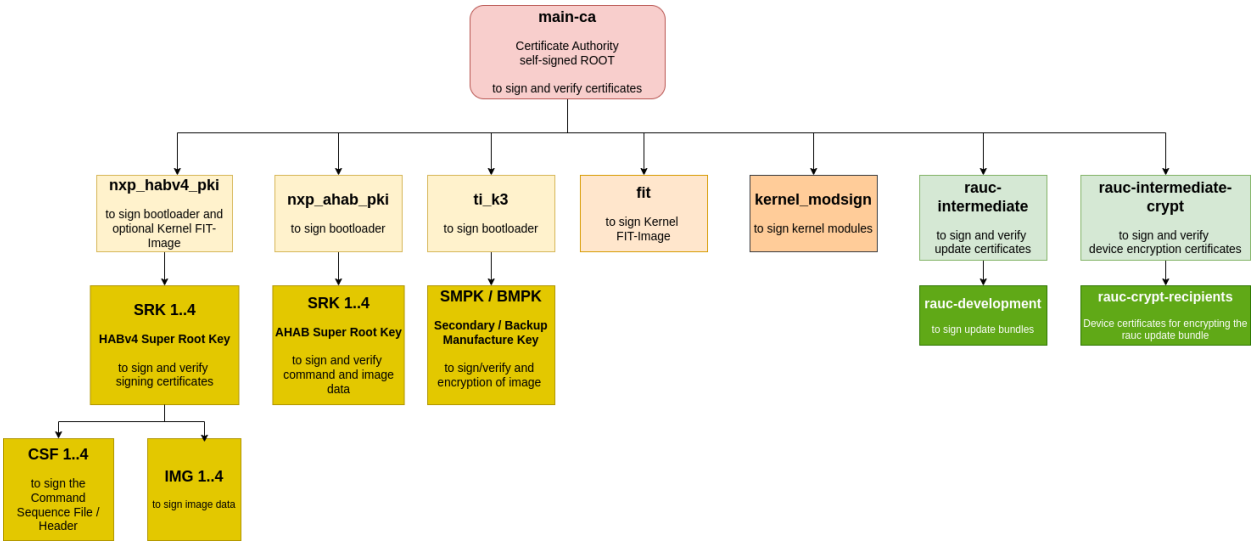
To use a secure boot with a signed bootloader and a signed kernel image, several keys and certificates are required to sign the images. The key and certificate creation is a manual process and the public key infrastructure (PKI) tree must be in place before you start your build. This BSP includes the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

16.2 16.2 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.

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!

16.3 16.3 Create Custom 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.

16.3.1 16.3.1 Change PKI Tree from phytec-dev-ca to Custom 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!

16.4 16.4 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 the HABv4 API can be found in the doc folder of the imx-code-signing-tool repository.

16.4.1 16.4.1 AHAB PQC keys

The i.MX95 supports three key configurations for AHAB secure boot:

- **classic**: only classical keys (RSA or ECC)
- **PQC**: only post-quantum keys (ML-DSA)
- **hybrid**: classical and post-quantum keys combined

Classic keys (RSA, ECC) can be created in the same way as for other NXP SoCs with the `imx-code-signing-tool`, as described above or with NXP Secure Provisioning SDK (SPSDK, see *18 SoC Specific Configuration Tools*). PQC and hybrid keys must be generated with SPSDK.

Generate the PQC key tree (ML-DSA) with:

```
host:~$ nxpcrypto pki-tree ahab-pqc -k mldsa87 -p test -o mldsa87 -e pem -n 4
```

16.5 16.5 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.

16.5.1 16.5.1 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
- 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/secureresources/AM62X-RESTRICTED-SECURITY
- 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
- 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 <PROCESSOR_SDK_RTOS_DIRECTORY>
host:~$ ./sdk_builder/scripts/setup_psdk_rtos.sh
host:~$ cd pdk_j721s2_<VERSION>/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 `<PROCESSOR_SDK_RTOS_DIRECTORY>/pdk_j721s2_<VERSION>/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)
{
@@ -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(!(strncmp(info.boardInfo.boardName,
-                     "AM68-SK-SOM",
-                     BOARD_BOARD_NAME_LEN)))
-        {
-            UART_printf("AM68 SK Detected!!\n");
-            skBoardDet = BTRUE;
-        }
-    }
+    uint32_t regVal;

-    if(BTRUE == skBoardDet)
```

(continues on next page)

(continued from previous page)

```
-    {
-        /* 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);
+
+    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.

16.5.2 16.5.2 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: `<PROCESSOR_SDK_RTOS_DIRECTORY>/pdk_j721s2_<VERSION>/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

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.

16.5.3 16.5.3 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.

16.5.3.1 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: `<PROCESSOR_SDK_RTOS_DIRECTORY>/pdk_j721s2_<VERSION>/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 tifek/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 tifek/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 tifek/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 tifek/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 tifek/ti_fek_public.pem
```

  - Readout and Overwrite Protection: `---ext-otp-wprp`
- generate certificate to set keycnt and keyref 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 tifek/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.

16.5.3.2 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: `<PROCESSOR_SDK_RTOS_DIRECTORY>/pdk_j721s2_<VERSION>/packages/ti/boot/keywriter/scripts`

and run the following:

```
host:~$ ./gen_keywr_cert.sh -t tifek/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.

16.5.3.3 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:~$ export MCU_PLUS_SDK_PATH="<MCU_PLUS_SDK_DIRECTORY>"
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 <PROCESSOR_SDK_RTOS_DIRECTORY>/pdk_j721s2_<VERSION>/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 `<PROCESSOR_SDK_RTOS_DIRECTORY>/pdk_j721s2_<VERSION>/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.

16.6 16.6 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.

16.7 16.7 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
```

16.8 16.8 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

16.9 16.9 Create SSH Keys and Certificates

Note

Available from Yocto Walnascar

First, create the user CA.

```
host:~$ ssh-keygen -t ed25519 -f user_ca -C "User CA"
```

This creates `user_ca`, which should be kept save, and `user_ca.pub`, which will be installed on the target.

Second, create the user private key and a corresponding public key, which is signed by the user CA, both will be installed on the client accessing the target.

```
host:~$ ssh-keygen -t ed25519 -f user_root_ed25519 -a 32 -C "root"
# example with 24h access to the target
host:~$ ssh-keygen -s user_ca -I root -n root -V +24h user_root_ed25519.pub
```

Note

The `user_root` key pair grants access to all devices where `user_ca.pub` is installed. Install a different `user_ca` for each device if this is not wanted.

CHAPTER

SEVENTEEN

17 SECURITY VULNERABILITIES

The used software can be affected by security vulnerabilities. A security vulnerability generally is a bug in software code that could allow an attacker to gain control of a system. It is essential to check the software regularly against published security flaws (Common Vulnerabilities and Exposures).

The OpenEmbedded Core layer has a cve-check class to check the recipes against public CVEs. With this CVE check you get a list of all CVEs for the version of the package. Only the CVEs with a patch with the name of the CVE in the recipe or CVEs in a whitelists can be marked as fixed. There is no code analysis and you must start a new Yocto build for every check.

17.1 17.1 CycloneDX SBOM

CycloneDX is an international Standard (ECMA-424) for Bill of Materials and makes it simple to detect, triage, and report security vulnerabilities. The CycloneDX is in JSON Format and the component identifiers CPE and PURL enables the detection of known vulnerabilities. In the PHYTEC BSP a CycloneDX SBOM is created for the `securiphy` distro. To add CycloneDX SBOM generation to your distro, add the following line:

```
require conf/distro/include/common-cyclonedx.inc
```

17.2 17.2 SPDX 3.0 SBOM

The SPDX 3.0 SBOM can include all compiled files in the kernel by setting `SPDX_INCLUDE_COMPILED_SOURCES = "1"` in the kernel recipe.

CHAPTER

EIGHTEEN

18 SOC SPECIFIC CONFIGURATION TOOLS

This chapter gives an overview of tools, which can be used for a better configuration of features of your SoM. Mostly, you can use the bootloader commands, but the parameters and addresses differ between versions.

18.1 18.1 NXP Universal Update Utility (UUU)

The Universal Update Utility (UUU) from NXP is a program to be executed on the host to load and run the bootloader onto the board through SDP (Serial Download Protocol). For detailed information visit https://github.com/nxp-imx/mfgtools or download the Official UUU-tool documentation.

More information and examples can be found in the SoC specific manuals.

18.2 18.2 Snagboot Recover and Reflashing Tool

An alternative to the NXP UUU tool is snagboot, which supports TI k3, STM32 and NXP SoC's.

18.3 18.3 partup - the System Initialization Program

partup is the PHYTEC default program for partitioning and image installation on an embedded device. The partup package is created in the Yocto build and is smaller than a wic image.

18.4 18.4 NXP i.MX6 and i.MX8M MNP eFuse Tool Crucible

You can use the tool crucible to burn eFuses from kernel userspace. The tool is Go based and is integrated in our *phytec-provisioning-image*.

It can be used to e.g. burn the SRK hash for the i.MX8MP:

```
SRK_HASH=( 0x9A842534 0xB0491AB4 0xD5B6A07B 0xFD92DCE7 0xC10DC87C 0xD8BD04A9 0x704E9FE4␣
↪0x9B025359 )
# Read the SRK hash at first to e.g. check if it is empty or the target SRK hash
for (( x=0; x<${#SRK_HASH[@]}; x++ ))
    do
   # Read the SRK hash at first
   rv=$(crucible -m IMX8MP -r 0 -b 16 -s read ${SRK_HASH[$x]})
   #check if the SRK is zero
   if [[ "$rv" =~ ^0x[0]*$ ]]; then
      # write the SRK hash
      crucible -m IMX8MP -r 0 -b 16  -Y -e big blow "OCOTP_SRK$x" ${SRK_HASH[$x]}
```

(continues on next page)

(continued from previous page)

```
   fi
done
```

To burn the SRK hash to an other SoC, just replace `IMX8MP` with another supported SoC identifier. The supported SoCs can be found on https://github.com/usbarmory/crucible/tree/master/cmd/crucible/fusemaps.

18.5 18.5 Secure Provisioning SDK (SPSDK)

NXP provides the Secure Provisioning SDK (SPSDK). You can install it as a Python package on your host:

```
host:~$ python3 -m venv venv
host:~$ source venv/bin/activate
host:~$ python -m pip install --upgrade pip
host:~$ pip install spsdk
host:~$ spsdk --help
```

The following preparatory steps are necessary for the `nxpele` tool to work in the uboot_serial mode:

- use the bootloader with CONFIG_AHAB_BOOT=y and CONFIG_CMD_MEMORY=y support (provisioning variant)
- open a terminal with the serial port of your board e.g. /dev/ttyUSB0
- power up your board
- stop and login to the bootloader
- close your serial connection

Then you can use the `nxpele` tool to communicate with and to configure the EdgeLock Enclave:

```
host:~$ nxpele --help
Commands:
nxpele                          Utility for communication with the EdgeLock Enclave on target over␣
↪BLHOST.
├── batch                       Invoke nxpele commands defined in command file.
├── commit                      Commit information.
├── derive-key                  Derive key.
├── dump-debug-data             Dump ELE debug buffer data of EdgeLock Enclave firmware.
├── ele-fw-auth                 Authenticate and execute EdgeLock Enclave firmware.
├── enable-apc                  Send request to enable APC to EdgeLock Enclave.
├── enable-rtc                  Send request to enable RTC to EdgeLock Enclave.
├── forward-lifecycle-update    Forward Lifecycle update to Closed or Locked state.
├── generate-keyblob            Group of sub-commands related to generate Keyblob.
│   ├── DEK                     Generate DEK keyblob on EdgeLock Enclave.
│   ├── IEE                     Generate IEE keyblob atomic command on EdgeLock Enclave.
│   ├── IEE-KEYBLOB             Generate IEE keyblob on EdgeLock Enclave.
│   ├── OTFAD                   Generate OTFAD keyblob atomic command on EdgeLock Enclave.
│   └── OTFAD-KEYBLOB           Generate OTFAD keyblob on EdgeLock Enclave.
├── get-ele-fw-status           Get status of EdgeLock Enclave firmware.
├── get-ele-fw-version          Get version of EdgeLock Enclave firmware.
├── get-ele-trng-state          Get status of EdgeLock Enclave TRNG.
├── get-events                  Get stored events in EdgeLock Enclave.
├── get-info                    Get information from EdgeLock Enclave.
```

(continues on next page)

(continued from previous page)

```
├── load-keyblob              Load EdgeLock Enclave keyblob to hardware.
├── oem-cntn-auth             Authenticate OEM container.
├── ping                      Send general EdgeLock Enclave PING message.
├── read-common-fuse          Read common fuse from EdgeLock Enclave.
├── read-shadow-fuse          Read shadow fuse from EdgeLock Enclave.
├── release-container         Release EdgeLock Enclave firmware message.
├── reset                     Send general EdgeLock Enclave RESET message.
├── reset-apc-context         Send request to reset APC context in EdgeLock Enclave.
├── signed-message            Send signed message to EdgeLock Enclave.
├── start-trng                Start True Random Number Generator in EdgeLock Enclave message.
├── verify-image              Verify OEM image.
├── write-fuse                Write one fuse by specifying index and data to be written.
└── write-shadow-fuse         Write one shadow fuse by specifying index and data to be written.

host:~$ nxpele -p /dev/ttyUSB0 -d uboot_serial -f mx93 get-info
ELE get info ends successfully:
Command:          0xda
Version:          2
Length:           160
SoC ID:           9300
SoC version:      A100
Life Cycle:       OEM_LCKD - 0x0200
SSSM state:       4
UUID:             2db3725f3951488ca5a51a8f575fdff5
SHA256 ROM PATCH: 725e3348349e8664b79b93020e8d2e2273d83f33fd9ea7381cce9999df6fbe9a
SHA256 FW:        49240dc9eb7f228738fc0d614539c2378d58f92be5df56d1547043deed8cd915
Advanced information:
   OEM SRKH:      ␣
↪2c5dafba23032ee908bac0233f97e710d5e08d6784356d96bedf41a5ba6da06e0000000000000000000000000000000000000000000000000000
   IMEM state:     202
   CSAL state:     EdgeLock secure enclave random context initialization succeed
   TRNG state:     TRNG entropy is valid and ready to be read
```

18.6 18.6 TPM Infineon Firmware Update Tool

On PHYTEC Boards there are usually TPMs from Infineon embedded and this chapter describes the firmware update process for these devices.

At first, you need access to the myInfineon Collaboration Platform (ICP) for downloading the

- TPM SLB 96xx Update Tool
- Firmware for the TPM 2.0 from Infineon

For the registration on the myInfineon please follow the ICP Guide. If you have access to the ICP, then login on https://mycases.infineon.com/ and "Create new case" to request access to the update tool and the firmware for your device.

If your case is accepted, then you can download the `TPM_FU_v2.03.4733.00_ToolsOnly_Linux_SourceCode.tar.gz` or newer. For the installation of the TPM update tool in your image, you need a recipe with the name *recipes-devtools/infineon/infineon-tpm-updater_v2.03.4733.bb*

```
SUMMARY = "INFINEON TPM FIRMWARE UPDATER"
LICENSE = "Proprietary"
LIC_FILES_CHKSUM = "file://../../License.txt;md5=36ca2c0688532edca7708132cbd7585e"

DEPENDS = "openssl"

SRC_URI = "file://TPM_FU_v2.03.4733.00_ToolsOnly_Linux_SourceCode.tar.gz"

# Uncomment, because compiler set it internally before
# In Makefile is a set for FORTIFY_SOURCE
CFLAGS += "-U_FORTIFY_SOURCE"
S = "${WORKDIR}/Source/TPMFactoryUpd"
B = "${WORKDIR}/Source/TPMFactoryUpd"

EXTRA_OEMAKE = " \
   ARCH='${TARGET_ARCH}' \
   CROSS_COMPILE='${TARGET_SYS}-' \
   CC='${CC}' \
   LIB='${AR}' \
"

do_install () {
   #oe_runmake install DESTDIR=${D} INCLUDEDIR=${includedir}
   install -d -m 755 ${D}${sbindir}
   install ${B}/TPMFactoryUpd ${D}${sbindir}
}

RDEPENDS:${PN} += " \
   openssl \
   libcrypto \
   libgcc \
"

INSANE_SKIP:${PN} += "already-stripped"
FILES:${PN} += "{sbindir}"
```

After that, you can include the `infineon-tpm-updater` in your image recipe, build your image, install and boot it.

Warning

Infineon Attention from Users Manual It is recommended to always restart the system directly after the TPM Firmware Update, since certain system hardware and software components might not be aware of a TPM Firmware Update without a restart (especially in case the TPM family has been changed with the update.)

The configuration of the TPM should not be changed by the firmware update. The generated keys with `keyctl` and `pkcs11` should be available after the update of the firmware.

- Read the TPM information on the device:

```
target:~$ TPMFactoryUpd -info
```

(continues on next page)

(continued from previous page)

```
************************************************************************
*    Infineon Technologies AG   TPMFactoryUpd   Ver 02.03.4733.00    *
************************************************************************
   TPM information:
   ----------------
   TPM family                        :    2.0
   TPM firmware version              :    7.85.4555.0
   TPM firmware recovery support     :    No
   TPM firmware valid                :    Yes
   TPM operation mode                :    Operational
   TPM platformAuth                  :    Empty Buffer
   Remaining updates                 :    64
```

- Update the TPM 2.0

```
target:~$ TPMFactoryUpd -update tpm20-emptyplatformauth -firmware TPM20_7.85.4555.0_to_TPM20_7.
↪86.19393.2.BIN
************************************************************************
*    Infineon Technologies AG   TPMFactoryUpd   Ver 02.03.4733.00    *
************************************************************************

  TPM update information:
  -----------------------
  TPM family                        :    2.0
  TPM firmware version              :    7.85.4555.0
  TPM firmware valid                :    Yes
  TPM operation mode                :    Operational
  TPM platformAuth                  :    Empty Buffer
  Remaining updates                 :    64
  New firmware valid for TPM        :    Yes
  TPM family after update           :    2.0
  TPM firmware version after update :    7.86.19393.2

  Preparation steps:
  TPM2.0 policy session created to authorize the update.

  DO NOT TURN OFF OR SHUT DOWN THE SYSTEM DURING THE UPDATE PROCESS!

  Updating the TPM firmware ...
  Completion: 100 %
  TPM Firmware Update completed successfully.

  A system restart is required before the TPM can enter operational mode again.
```