Application Note

Jacinto7 High Security Device Development

Neo Wang, Joe Shen, Brijesh Jadav, and Keerthy J

ABSTRACT

The Jacinto7 (J7) SoC has two types of silicon: the general purpose (GP) and high security (HS). The HS device also has two sub-types that represent the state of the HS device: high security-field securable (HS-FS) and high security-security enforced (HS-SE). All security features are enabled in the HS-SE device to ensure integrity, confidentiality, and anti-cloning protection to the end-customer system and software. Before starting volume production with the HS-SE device, you should make sure the correctness of each step in the HS device. This document introduces you to how to complete the security process in the TIDK device and the customer product HS device.

Table of Contents

List of Figures

List of Tables

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

In the different customer stages from development to volume production, the J7 series SoC provides several device types according to the different requirements of security, as shown in Figure 1-1.

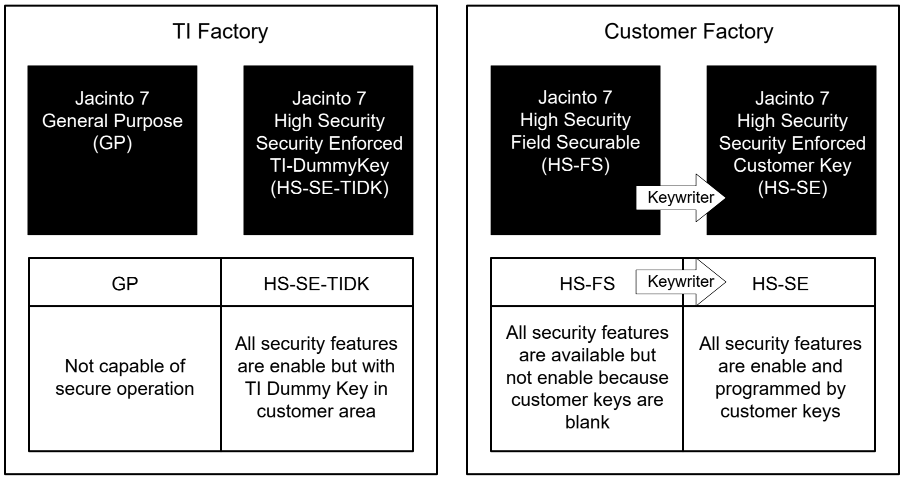
TI Factory
Jacinto 7 General Purpose (GP)
Jacinto 7 High Security Security Enforced TI-DummyKey (HS-SE-TIDK)
GP
HS-SE-TIDK
Not capable of secure operation
All security features are enable but with TI Dummy Key in customer area
Customer Factory
Jacinto 7 High Security Field Securable (HS-FS)
Keywriter
Jacinto 7 High Security Security Enforced Customer Key (HS-SE)
HS-FS
Keywriter
HS-SE
All security features are available but not enable because customer keys are blank
All security features are enable and programmed by customer keys

Figure 1-1. Jacinto7 SoC Device Type

- **GP device**: General-Purpose device type, can be production device, but normally used in development considering security.
- **HS-SE-TIDK device**: High Security device contains TI keys, and TI dummy keys in the customer area. TIDK devices can never be production devices.
- **HS-FS device**: High Security-Field Securable device, the customer area is blank. The HS-FS device needs to program keys before delivery to the end-customer.
- **HS-SE device**: High Security-Security Enforced device. The customer has programmed their keys into the customer area. The HS-SE devices can be production devices.

Different device types bring different security features, but also different restrictions. The GP device has no safety features, so the JTAG port is unlocked and all the binaries do not need to be signed and encrypted, so it is usually used as a development device. The HS device enforces the security feature, but the JTAG port is locked and all the binaries must be signed and encrypted, so it is usually used as a production device. The customer can see the development flow in Figure 1-2 to complete the HS process.

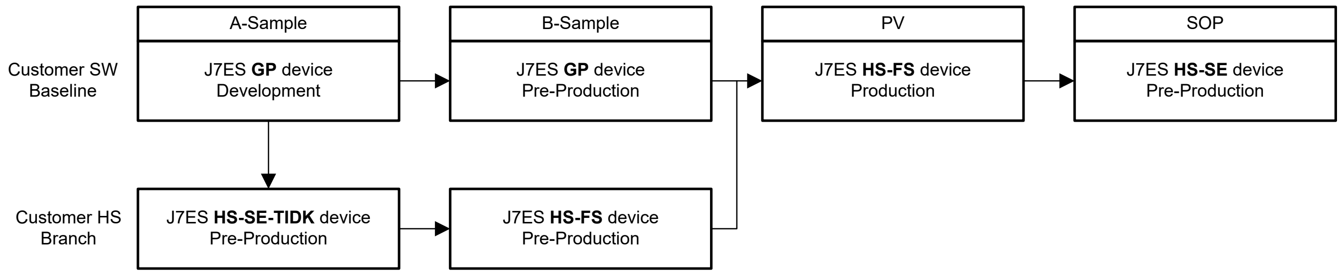
A-Sample
B-Sample
PV
SOP
Customer SW Baseline
J7ES GP device Development
J7ES GP device Pre-Production
J7ES HS-FS device Production
J7ES HS-SE device Pre-Production
Customer HS Branch
J7ES HS-SE-TIDK device Pre-Production
J7ES HS-FS device Pre-Production

Figure 1-2. Jacinto7 High Security Development

The recommended process is:

- Customer engineers do their development on GP device; meanwhile the HS engineers start their security development on HS-SE-TIDK device, to ensure the correctness of the HS sign and encrypt operation.
- The HS engineers purchase HS-FS device and use TI dummy keys or random dummy keys to double check the HS process. In order to confirm the process of keys generation, Keywriter setup, and eFuse programming.
- Synthesize the work of GP and the security branch of HS-FS, also the customer needs to setup their HSM and program customer keys into eFuse, and use the same customer keys to sign and encrypt the system image.

Lot of keys are used during the security process in the J7 device, as shown in Table 1-1. The notes describe important keys during HS development. The SMPKH, SMEK, BMPKH, and BMEK are programmed into eFuse to authenticate and decrypt the system image. The AES-256 key and TI FEK public key are used to protect the key programming process.

TI also provides test keys in the Keywriter package. Customers can use these test keys to go through the complete HS process, then setup up their HSM and use customer keys to complete these tasks before production.

Table 1-1. Keys in High Security Development

Acronym	Name	Status	Owner	Notes
KEK	Key Encryption Key	Necessary	Device	256-bit statistically unique random number per device
MPK hash	Manufacturer Public Key hash	Necessary	TI	512-bit SHA2 hash of MPK. MPK is a 4096-bit key programmed by TI in factory.
MEK	Manufacturer Encryption Key	Necessary	TI	256-bit initial encryption key for the device, used for encrypted boot, programmed by TI in factory.
SMPK hash	Secondary Manufacturer Public Key hash	Necessary	Customer	512-bit SHA2 hash of SMPK. SMPK is a 4096-bit key used to authenticate the signed binary.
SMEK	Secondary Manufacturer Encryption Key	Necessary	Customer	256-bit customer encryption key for encrypted boot used to decrypt the encrypted binary.
BMPK hash	Back up Manufacturer Public Key hash	Optional	Customer	Back up 512-bit SHA2 hash of SMPK. SMPK is a 4096-bit key used to authenticate the signed binary.
BMEK	Back up Manufacturer Encryption Key	Optional	Customer	Back up 256-bit customer encryption key for encrypted boot used to decrypt the encrypted binary.
AES-256	Advanced Encryption Standard 256-bit Key	Optional	Customer	Random 256-bit number to be used as a temporary AES encryption key for protecting the OTP extension data.
TI FEK Pub	TI Factory Encryption Key	Necessary	TI	RSA 4K encryption key to protect the customer key material before they are written to the eFuses.

2 TIDK Device Verification

HS-SE-TIDK device is security enforced silicon, which has TI dummy key programmed in customer area. The customer needs to use the TI dummy key to sign and encrypt their system image, if they want to verify the functionality in the HS-SE-TIDK device. This step is necessary to help customers familiarize the process of signing and encrypting binaries with security keys. But the TI dummy key is public for all the customers, so the HS-SE-TIDK device can never be a production device.

The TI dummy keys are included in the default RTOS SDK. You can download the SDK from RTOS SDK for DRA829 & TDA4VM Jacinto™ Processors. You can find TI dummy keys in the following folder. All the processes tested in SDK7.1 software work fine.

```
# cp ${PSDKRA_PATH}/pdk/packages/ti/build/makerules/k3_dev_mpk.pem ~/TIDummyKey/smpk.pem
# cp ${PSDKRA_PATH}/pdk/packages/ti/build/makerules/k3_dev_mek.txt  ~/TIDummyKey/smek.txt
```

2.1 Sign and Encrypt Second BootLoader (SBL)

The SBL (Second BootLoader) is loaded by MCU R5F ROM and authenticated by DMSC. The MCU R5F runs this code and starts the boot flow for other cores. Follow below command to build the signed SBL for HS device.

```
# cd ~/ti-processor-sdk-rtos-j721e-evm-07_01_00_11/pdk_jacinto_07_01_00_45/packages/ti/build
# make -j BOARD=j721e_evm CORE=mcu1_0 BUILD_PROFILE=release sbl_mmcsd_img_hs
#ls ${PSDKRA_PATH}/pdk/packages/ti/boot/sbl/binary/j721e_evm_hs/mmcsd/bin/
sbl_mmcsd_img_mcu1_0_release.tiimage
```

The default makefile only sign the SBL but does not encrypt that, apply below patch and rebuild the SBL_HS to get the sign and encrypt SBL image.

```
diff --git a/packages/ti/build/makerules/common.mk b/packages/ti/build/makerules/common.mk
index f56e069..e9ec0d9 100644
--- a/packages/ti/build/makerules/common.mk
+++ b/packages/ti/build/makerules/common.mk
@@ -635,7 +635,7 @@ else ifeq ($(SOC),$(filter $(SOC), am65xx am64x j721e j7200))
        $(CHMOD) a+x $(SBL_CERT_GEN)
 endif
-       $(SBL_CERT_GEN) -b $(SBL_BIN_PATH) -o $(SBL_TIIMAGE_PATH) -c R5 -l $(SBL_RUN_ADDRESS) -k $($
(APP_NAME)_SBL_CERT_KEY) -d DEBUG -j DBG_FULL_ENABLE -m $(SBL_MCU_STARTUP_MODE)
+       $(SBL_CERT_GEN) -b $(SBL_BIN_PATH) -o $(SBL_TIIMAGE_PATH) -c R5 -l $(SBL_RUN_ADDRESS) -k
$($(APP_NAME)_SBL_CERT_KEY)  -y ENCRYPT -e $(SBL_ENCRYPT_KEY_HS) -d DEBUG -j DBG_FULL_ENABLE -m $
(SBL_MCU_STARTUP_MODE)
diff --git a/packages/ti/build/makerules/platform.mk b/packages/ti/build/makerules/platform.mk
index cc6b905..381f1dd 100644
--- a/packages/ti/build/makerules/platform.mk
+++ b/packages/ti/build/makerules/platform.mk
@@ -200,7 +200,7 @@ endif
export SBL_CERT_KEY=$(ROOTDIR)/ti/build/makerules/rom_degenerateKey.pem
-
+export SBL_ENCRYPT_KEY_HS=~/TIDummyKey/smek.txt
```

The previous makefile and compile command will generate the GP SBL binary first, then use TI dummy key to sign and encrypt it. Also, during building the SBL_HS, it will also sign the board configuration, security configuration, RM (Resource Management) and PM (Power Management), then integrate it into SBL. So, the finally SBL can be authenticate and decrypt by HS-SE-TIDK device.

Before starting the encryption of the binary, there is a known bug in SDK8.0 and previous SDK versions. Apply the following patch first then encrypt the binary.

```
diff --git a/packages/ti/build/makerules/x509CertificateGen.sh b/packages/ti/build/makerules/
x509CertificateGen.sh
index 20fe23b..4c906e5 100755
--- a/packages/ti/build/makerules/x509CertificateGen.sh
+++ b/packages/ti/build/makerules/x509CertificateGen.sh
@@ -116,7 +116,7 @@ image_encrypt() {
        truncate -s %16 enc_tmp.bin
        xxd -r -p $ENC_RS enc_rs.bin
        cat enc_tmp.bin  enc_rs.bin > enc_bin_rs.bin
-       ENC_BIN=$CERT_SIGN"-ENC-"$BIN
+       ENC_BIN=$BIN"-ENC-"$CERT_SIGN
        echo "$ENC_BIN"
        if [ "$IMG_ENC" == "ENCRYPT" ];then
```

2.2 Sign and Encrypt System Image

The bootloader demonstration in the GP device can be found in CAN Response and Bootloader Demo Application. The required binaries to start up the system are shown in Table 2-1.

Table 2-1. Required Binaries to Start up the System

	Image Needs to be Signed and Encrypted		
System Architecture	**Boot RTOS on Multicore**	**Boot Linux on A72 and RTOS on Other Cores**	**Boot QNX on A72 and RTOS on Other Cores**
Binaries	tiboot3.bin tifs.bin app lateapp1 lateapp2 lateapp3	tiboot3.bin tifs.bin app lateapp1 lateapp2 atf_optee.appimage tidtb_linux.appimage tikernelimage_linux.appimage	tiboot3.bin tifs.bin app lateapp1 lateapp2 atf_optee.appimage ifs_qnx.appimage

The difference is the ATF (Arm Trusted Firmware) image, DTB (Device Tree Binary) image, and Linux kernel image should be signed and encrypted if running Linux in A72 core, and the file system does not need to be signed. Also, only ATF and IFS (Image FileSystem) images need to be signed, if implementing the QNX in A72 core.

For the binaries in Table 2-1, they can sign and encrypt by x509Certificate script in the default SDK using the following command.

```
# ${PSDKRA_PATH}/pdk/packages/ti/build/makerules/x509CertificateGen.sh -b binary_need_sign&encrypt
-o signed_encrypted_binary -c R5 -l 0x0 -k ${PSDKRA_PATH}/pdk/packages/ti/build/makerules/
k3_dev_mpk.pem -y ENCRYPT -e ${PSDKRA_PATH}/pdk/packages/ti/build/makerules/k3_dev_mek.txt  -d
DEBUG -j DBG_FULL_ENABLE -m SPLIT_MODE
```

Except for SBL and system images, the TIFS also need to sign and encrypt by the TI dummy key. The TIFS signed with the TI dummy key already provided in SDK, can be used directly in the HS-SE-TIDK device.

```
$ cp ${PSDKRA_PATH}/pdk/packages/ti/drv/sciclient/soc/V1/tifs-hs-enc.bin   /media/user/boot/tifs.bin
```

3 Keys Programming

Keywriter is a software package to fuse the customer keys into the HS-FS device and convert it to an HS-SE device. Keywriter builds on top of the PDK and provides special TIFS binary, TI keys generation tools, and related source code. The customer can use this Keywriter package to create a binary that runs as a bootloader on the target, which will automatically finish the key programming.

3.1 Install Keywriter

Once the keywriter package is installed, it consists of two folders as shown in Figure 3-1. You need to merge and replace these two original folders or files in PDK with the keywriter folder.

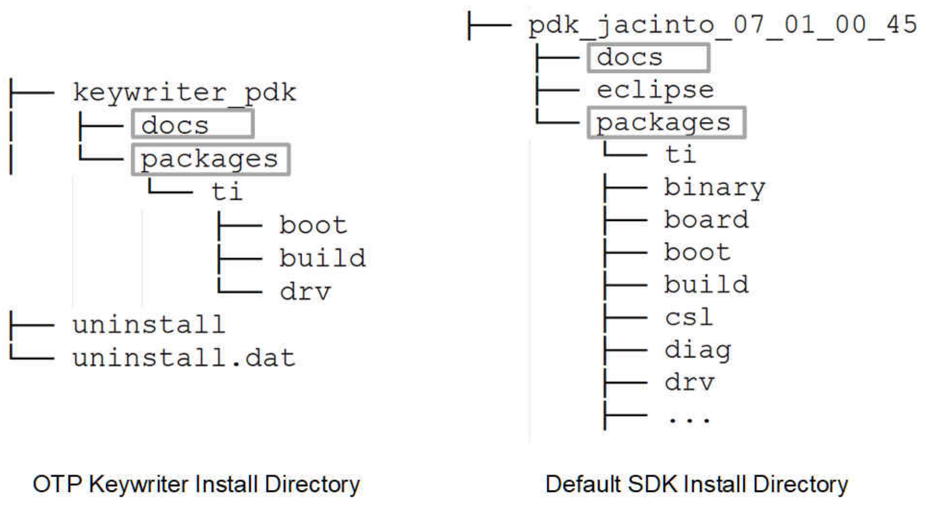

Figure 3-1. Keywriter Install Directory

OpenSSL is required for building the OTP Keywriter, make sure your Linux OS is installed the OpenSSL before next step.

```
# sudo apt-get install openssl
```

3.2 Keys Generation

After installing the SDK, the Keywriter package should include the keys to generate the x509 certificate before building the Keywriter application, then append this x509 certificate to the final keywriter binary, as shown in Figure 3-2.

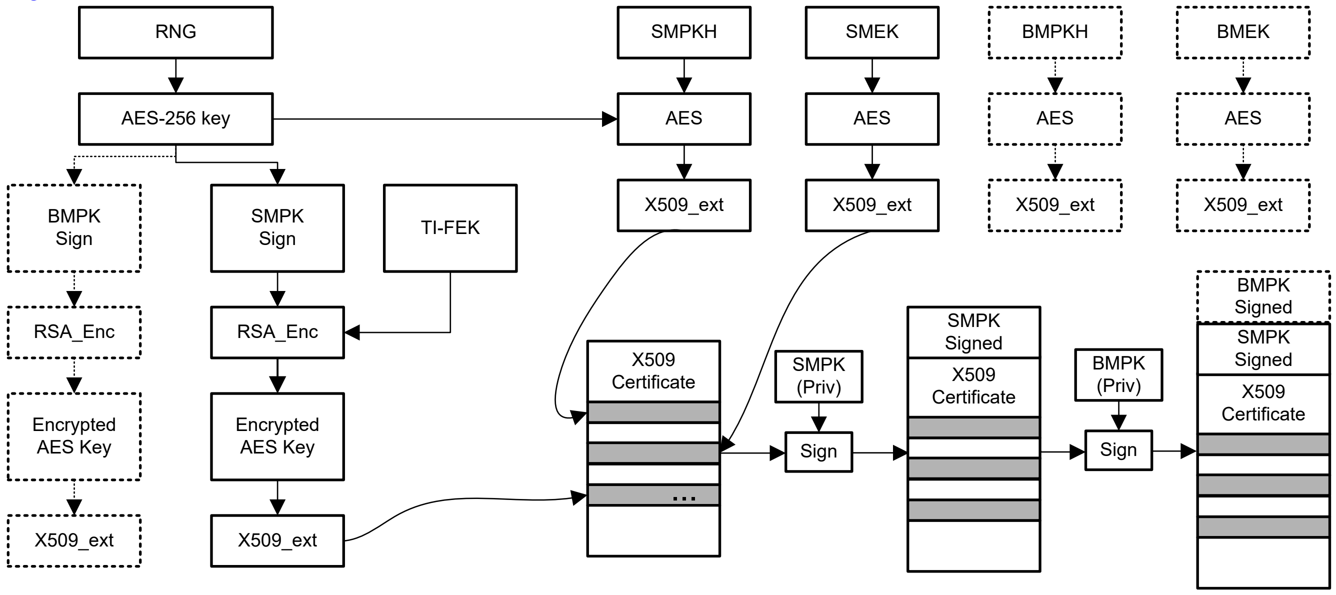

Figure 3-2. X509 Certificate Generation Procedure

- The RNG (Random Numeral Generator) in the customer HSM generates the random aes-256 key. This aes-256 key is temporary and used to encrypt each field. For each field, the aes-256 key is the same but called with a different IV (initialization vector). This aes-256 key also is encrypted by the TI FEK public key and included in the x509 extension.
- The SMPKH, SMEK, BMPKH, and BMEK keys are encrypted by the aes-256 key and included in a different x509 extension. These keys will be programmed into eFuses in plain text.
- The x509 configuration is signed by the SMPK private key and the BMPK private key (optional), then append these filed to the original x509 extension.

The process is irreversible once the eFuse is programmed by customer keys. In order to make sure the correctness of the operation and process, before the customer starts to program their customer keys into HS-FS device, it is a good choice to first use the TI dummy keys or random dummy keys as test keys; however, this step is not mandatory.

The Keywriter package provides one script that can generate random dummy keys to help the customer to test. The TI dummy keys are public, while the customer keys should be kept securely in the customer HSM so the customer can choose to use this random dummy keys for test before production. To generate the random dummy key, follow these steps:

```
# cd ${PSDKRA_PATH} /pdk/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/scripts
# ./gen_keywr_cert.sh -g
# cp ${PSDKRA_PATH} /pdk/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/scripts/
ti_fek_public.pem
${PSDKRA_PATH} /pdk/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/scripts/keys/tifekpub.pem
# ls ${PSDKRA_PATH} /pdk/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/scripts/keys
aes256.key bmek.key bmpk.pem smek.key smpk.pem tifekpub.pem
```

Also, the TI dummy keys are published in the TI SDK. Once the customer uses the TI dummy keys to complete the key programming, the device would convert to an HS-SE-TIDK device. So, the customer can use the method in Section 2 to sign and encrypt their system image, then verify these binaries in the device programmed by the TI dummy keys.

```
# cd ${PSDKRA_PATH}/pdk/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/scripts
# ./gen_keywr_cert.sh -g
# rm ${PSDKRA_PATH}/pdk/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/scripts/keys/bmek.key
bmpk.pem
smek.key smpk.pem
# cp ${PSDKRA_PATH}/pdk/packages/ti/build/makerules/k3_dev_mpk.pem
${PSDKRA_PATH}/pdk/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/scripts/keys/smpk.pem
# xxd -p -r ${PSDKRA_PATH}/pdk/packages/ti/build/makerules/k3_dev_mek.txt ${PSDKRA_PATH}/pdk
/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/scripts/keys/smek.key
# cp ${PSDKRA_PATH} /pdk/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/scripts/
ti_fek_public.pem
${PSDKRA_PATH} /pdk/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/scripts/keys/tifekpub.pem
# ls ${PSDKRA_PATH} /pdk/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/scripts/keys
aes256.key smek.key smpk.pem tifekpub.pem
```

The k3_dev_mek.txt is a 32-byte data file, which is used to encrypt the binary as a symmetric key, as shown in Section 2. The corresponding decryption key needs to be programmed into eFuse by Keywriter, but the keywriter requires the file to be in binary form. So the “xxd” command used to convert the format is shown below. Also, the k3_dev_mpk.pem is a secret key of the asymmetric key used to sign the binary, and the corresponding public key calculates the hash value and programmed into eFuse by Keywriter.

It is more reliable to check the smek.key using the following command. Before moving to the next step, make sure these two results are totally the same.

```
# cat ${PSDKRA_PATH}/pdk/packages/ti/build/makerules/k3_dev_mek.txt
c143f03568798964d4a5769bd5a27d3adc0d6bdd8f3cc47b84229e50a54ab043
# xxd -p /home/wangli/ti-processor-sdk-rtos-j721e-evm-
07_01_00_11/pdk_jacinto_07_01_00_45/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/scripts/
keys/smek.key
c143f03568798964d4a5769bd5a27d3adc0d6bdd8f3cc47b84229e50a54ab043
```

If the customer decides to use their customer keys, they also need to copy their keys to the following folder: ${PSDKRA_PATH}/pdk/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/scripts/keys.

3.3 Build Keywriter Application

After generating the keys to be programmed into the SoC, perform the following command to generate the x509 certificate.

```
# cd ${PSDKRA_PATH}/pdk/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/scripts
# ./gen_keywr_cert.sh -s keys/smpk.pem --smek keys/smek.key -t keys/tifekpub.pem -a keys/aes256.key
```

You can find the generated certificate in the folder: x509cert/final_certificate.bin. Perform the following command to compile the keywriter source code and append the x509 certificate to the keywriter application.

```
# cd ${PSDKRA_PATH}/pdk/packages/ti/build
# make keywriter_img -j8
```

The TIFS in GP silicon and HS-FS silicon are different, because the TI production key is already programmed to HS-FS silicon in the TI factory, but it is empty in the GP device. However, while building the Keywriter application, the keywriter.mk in PDK will load the HS-FS TIFS binary and convert it to array forms, which can load by the keywriter application source code. All of these steps are executed automatically while compiling the keywriter source code, so no extra operation required.

The generated keywriter application is in folder:
${PSDKRA_PATH}/pdk/packages/ti/boot/sbl/example/k3MulticoreApp/binary/keywriter_img_j721e_release.bin

3.4 Program Keys in HS-FS Device

After the software is ready, the following hardware requirements must be met when programming keys in the SoC OTP eFuses:

- The VPP_CORE and VPP_MCU power supplies must be disabled when not programming OTP registers.
- The VPP_CORE and VPP_MCU power supplies must be ramped up after the proper device power-up sequence, also the voltage value needs to be set within the following range, as listed in Table 3-1.

Table 3-1. Recommended Operating Conditions for OTP eFuse Programming

Parameter	Description	Minimum	Typical	Maximum	Unit
VPP_CORE	Supply voltage range for the eFuse ROM domain during **normal operation**		N/A		
	Supply voltage range for the eFuse ROM domain during **OTP programming**	1.71	1.8	1.89	V
VPP_MCU	Supply voltage range for the eFuse ROM domain during **normal operation**		N/A		
	Supply voltage range for the eFuse ROM domain during **OTP programming**	1.71	1.8	1.89	V
Tj	Temperature	0	25	85	℃

For steps to start flashing work, see Figure 3-3.

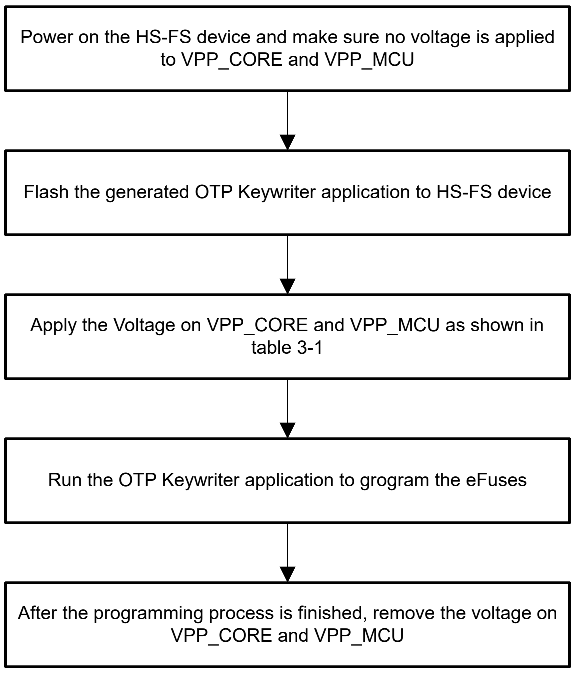
Power on the HS-FS device and make sure no voltage is applied to VPP_CORE and VPP_MCU
Flash the generated OTP Keywriter application to HS-FS device
Apply the Voltage on VPP_CORE and VPP_MCU as shown in table 3-1
Run the OTP Keywriter application to grogram the eFuses
After the programming process is finished, remove the voltage on VPP_CORE and VPP_MCU

Figure 3-3. eFuses Program Process

The OTP Keywriter runs as a bootloader in device boot medium. Once the Keywriter application starts the keys programming, the log can be found in MCU domain UART1 as shown:

```
$ \0OTP Keywriter Revision: 01.00.00.00 (Mar 7 2021 - 20:15:01)
$ OTP Keywriter ver: 20.8.5-w2020.23-am64x-14-g7409e
$ Beginning key programming sequence
$ Taking OTP configuration from 0x41c7e000
$ Debug response: 0x0
$ Key programming is complete
```

After the previous steps, the HS-FS device should be converted to an HS-SE device with specific keys; however, the binaries require signature and encrypt with the same keys if the customer wants to run these binaries in the HS-SE device.

4 Key Programming Verification

After programing the key into eFuse, the customer can verify the programming result and check the device status through the following steps before start production in the factory:

1. Configure boot mode of the board to UART boot and connect second MCU UART serial port of the board to the host PC, refer to the EVM Setup for J721E, and power on the EVM.
2. The terminal prints some log as shown below. You need to remove the extra CCC at the end and save as a log file.

```
# cat default_uart_hs.log
0200000011a00006a37657300000000000000000048535345020001000200010002a600000100010033c74f0c8631aa67a
56d53b06f250d75cb2a9cf7a52d6eb5e21b5e824250d7e09c22d997f09dc9389ecaa3f7d2b64d3a76d6163aa09e928ea0
50e1da9550
7e661f6002b07cd9b0b7c47d9ca8d1aae57b8e8784a12f636b2b760d7d98a18f189760dfd0f23e2b0cb10ec7edc7c6eda
c3d9bdfefe0eddc3fff7fe9ad875195527df02f2a23c0ed9d5fcf6dfb3a097ee4207cb1e2a5956e07ba144b73fe711439
82
```

3. Copy the following code and save it as a python file, which is used to parse the log in step 2.

```
#!/usr/bin/env python3
import binascii
import struct
import string
import sys
filename=sys.argv[1]
fp = open(filename, 'rt')
lines= fp.readlines()
fp.close()
bin_arr = [ binascii.unhexlify(x.rstrip()) for x in lines ]
bin_str = b"".join(bin_arr)
pubInfoStr='BB2B12B4B4B4B'
secInfoStr='BBHHH64B64B32B'
numBlocks = list(struct.unpack('I', bin_str[0:4]))
pubROMInfo = struct.unpack(pubInfoStr, bin_str[4:32])
if numBlocks > 1:
    secROMInfo = struct.unpack(secInfoStr, bin_str[32:200])
print ('------------------------')
print ('SoC ID Header Info:')
print ('------------------------')
print "NumBlocks            :", numBlocks
print ('------------------------')
print ('SoC ID Public ROM Info:')
print ('------------------------')
print "SubBlockId           :", pubROMInfo[0]
print "SubBlockSize         :", pubROMInfo[1]
tmpList = list(pubROMInfo[4:15])
hexList = [hex(i) for i in tmpList]
deviceName = ''.join(chr(int(c, 16)) for c in hexList[0:])
print "DeviceName           :", deviceName
tmpList = list(pubROMInfo[16:20])
hexList = [hex(i) for i in tmpList]
deviceType = ''.join(chr(int(c, 16)) for c in hexList[0:])
print "DeviceType           :", deviceType
dmscROMVer = list(pubROMInfo[20:24])
dmscROMVer.reverse()
print "DMSC ROM Version     :", dmscROMVer
r5ROMVer = list(pubROMInfo[24:28])
r5ROMVer.reverse()
print "R5 ROM Version       :", r5ROMVer
print ('------------------------')
print ('SoC ID Secure ROM Info:')
print ('------------------------')
print "Sec SubBlockId       :", secROMInfo[0]
print "Sec SubBlockSize     :", secROMInfo[1]
print "Sec Prime            :", secROMInfo[2]
print "Sec Key Revision     :", secROMInfo[3]
print "Sec Key Count        :", secROMInfo[4]
tmpList = list(secROMInfo[5:69])
tiMPKHash = ''.join('{:02x}'.format(x) for x in tmpList)
print "Sec TI MPK Hash      :", tiMPKHash
tmpList = list(secROMInfo[69:133])
custMPKHash = ''.join('{:02x}'.format(x) for x in tmpList)
print "Sec Cust MPK Hash    :", custMPKHash
tmpList = list(secROMInfo[133:167])
```

```
    uID = ''.join('{:02x}'.format(x) for x in
    print "Sec Unique ID        :", uID
```

4. Use the following command to parse the log after getting the above two files. The parsed information is as shown:

```
    # python uart_boot_socid.py default_uart_hs.log
    -----------------------
    SoC ID Header Info:
    -----------------------
    NumBlocks             : [2]
    -----------------------
    SoC ID Public ROM Info:
    -----------------------
    SubBlockId            : 1
    SubBlockSize          : 26
    DeviceName            : j7es
    DeviceType            : HSSE
    DMSC ROM Version      : [0, 1, 0, 2]
    R5 ROM Version        : [0, 1, 0, 2]
    -----------------------
    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       :
    33c74f0c8631aa67a56d53b06f250d75cb2a9cf7a52d6eb5e21b5e824250d7e09c22d997f09dc9389ecaa3f7d2b64d3a7
    6d6163aa09e928ea050e1da95507e66
    Sec Cust MPK Hash     :
    1f6002b07cd9b0b7c47d9ca8d1aae57b8e8784a12f636b2b760d7d98a18f189760dfd0f23e2b0cb10ec7edc7c6edac3d9
    bdfefe0eddc3fff7fe9ad875195527d
    Sec Unique ID         : f02f2a23c0ed9d5fcf6dfb3a097ee4207cb1e2a5956e07ba144b73fe71143982
```

The log reports the device type has already been converted to HS-SE, and also both Key Revision and Key Count are 1, which means only programing and using SMPK, not BMPK. While the customer SMPK Hash value can be obtained through the log, the customer can use the following method to check the consistency with the customer own key.

```
# openssl rsa -in k3_dev_mpk.pem -pubout -outform DER -out /tmp/k3_dev_mpk_pub.der
writing RSA key
# sha512sum /tmp/k3_dev_mpk_pub.der
1f6002b07cd9b0b7c47d9ca8d1aae57b8e8784a12f636b2b760d7d98a18f189760dfd0f23e2b0cb10ec7edc7c6edac3d9bdf
efe0eddc3fff7fe9ad875195527d  /tmp/k3_dev_mpk_pub.der
```

After comparison, the hash value of the key customer programmed is exactly the same as the hash value read out from the device. So, we can verify the device has been successfully converted to HS-SE according to the specific customer key.

5 Building and Booting on HS Devices Using Linux SDK

You can build and boot on TDA4VM D samples (Dummy key samples) using the Linux SDK starting 8.0. The HS support was added on SDK 8.0. For J721e HS build/boot instructions, see: https://e2e.ti.com/support/processors/f/791/t/1061584.

6 Summary

Be careful during the process of the HS device because some steps are irreversible. This document can help the customer complete the HS development.