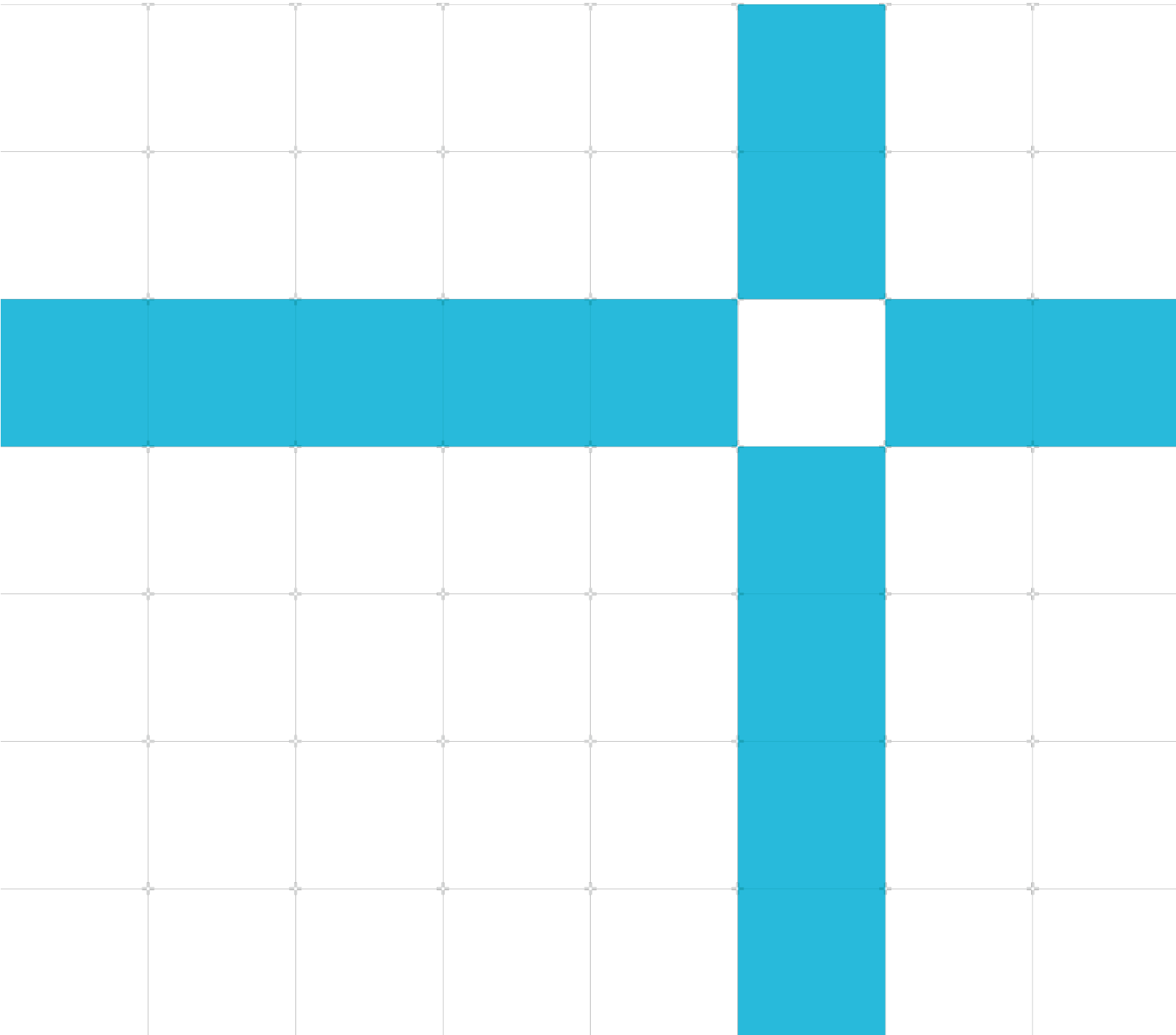
Arm® CryptoCell-312

Guide to Generate and Verify Secure Boot and Secure Debug Certificate Chains

Non-Confidential

Issue 01
107635



Arm CryptoCell-312

Guide to Generate and Verify Secure Boot and Secure Debug Certificate Chains

Release information

Document history

Issue	Date	Confidentiality	Change
0100	1 July 2022	Non-Confidential	First release Version 1.0

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.
(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Feedback

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

Contents

1 Overview

Secure Boot and Secure Debug are the basic features of CryptoCell (CC) boot services. Secure boot and Secure debug are based on certificate chain mechanisms using the RSA private and public key schemes.

This tutorial introduces the definition, generation and verification of Secure boot and Secure debug certificate chains. It also describes the main differences between the Secure boot certificate chain and the Secure debug certificate chain.

This guide contains the following sections:

- Certificate and certificate chain.
- Certificate chain verification flow.
- Certificate chain generation and verification.
- Differences between the Secure boot certificate chain and the Secure debug certificate chain.

2 Certificate and certificate chain

In cryptography, a certificate is also called a digital certificate, or identity certificate. It is an electronic document used to prove the ownership and integrity of the information included in the certificate.

The following figure shows the general certificate structure:

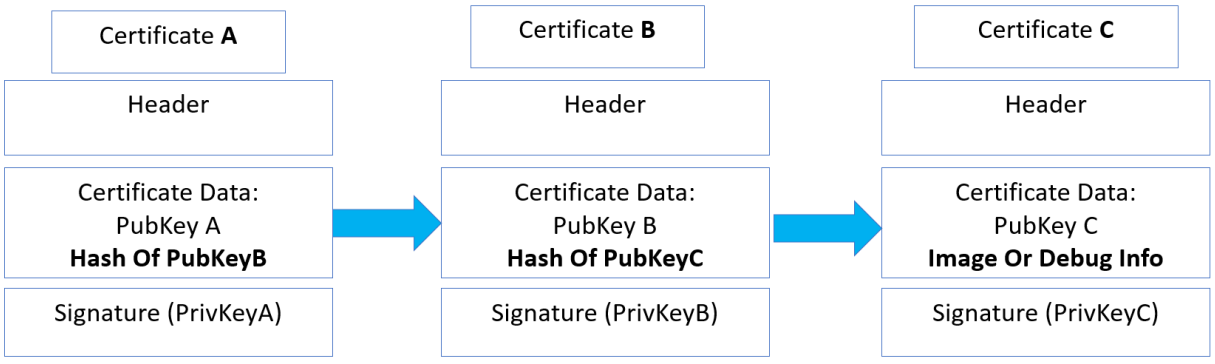
Figure 2-1: General Certificate Structure

The certificate structure consists of the following parts:

- Header, which includes information such as certificate type, version, size, owner, flags, and validity period.
- Certificate data, which includes public key and other information that must be signed.
- Signature, which is calculated over the header and certificate data by using RSA PSS scheme.

A certificate chain is made up of a list of certificates. The following figure shows an example certificate chain structure, which includes three certificates in the chain: Certificate A, Certificate B, and Certificate C.

Figure 2-2: General Certificate Chain Structure



The link between a certificate and its next certificate is that the certificate includes the hash of the public key of the next certificate. The advantage of using a certificate chain over a single certificate is that the certificate chain provides renewability and classification. For example, if Key B or Key C is leaked, it is easy to replace Key B or Key C.

There are four different types of certificates in CC products:

- Key certificate
- Content certificate
- Enabler certificate
- Developer certificate

The certificate chain of CC products is composed of two or three self-signed certificates. Self-signed means that the public key is included in the certificate and the certificate itself is signed with the corresponding private key. As shown in Figure 2-2, Certificate A is signed by PrivKeyA, and Certificate A includes PubKayA in its certificate data.

The CC supports two certificate flavors, Arm proprietary and X.509. Different certificate flavors have different structures. However, both certificate flavors can be abstracted as previous figures. Arm proprietary is designed to reduce the size. Therefore, its size is smaller than the X.509 certificate, and it is suitable for resource-limited devices, such as IoT devices.

Secure boot and Secure debug certificates must use the same certificate flavor, either proprietary or X.509. The certificate flavor must be defined for the entire certificate chain. You can configure the flavor by changing the configuration flag `CC_CONFIG_SB_X509_CERT_SUPPORTED` in the file `proj.ext.cfg` as follows:

Proprietary certificate

```
CC_CONFIG_SB_X509_CERT_SUPPORTED = 0
```

X.509 certificate

```
CC_CONFIG_SB_X509_CERT_SUPPORTED = 1
```

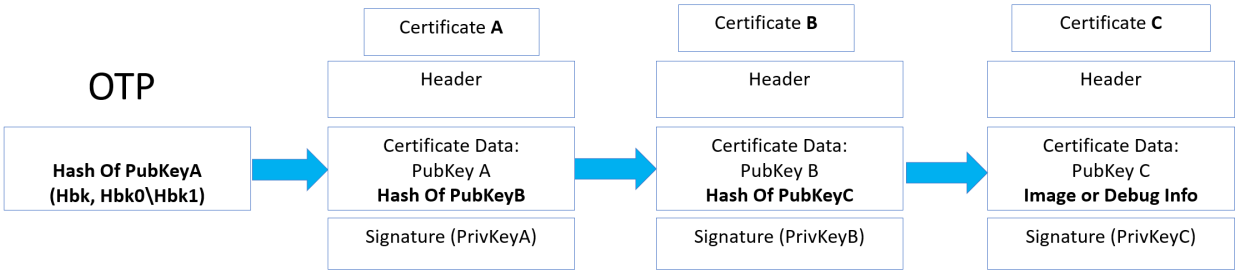
3 Certificate Chain Verification Flow

3.1 Generic certificate chain verification flow

The certificate chain is based on that the previous certificate includes the public key hash of the next certificate. Therefore, Certificate A can use Hash of PubkeyB to authenticate Certificate B by Hash algorithm. Then, PubkeyB is used to verify the integrity of Certificate B by the RSA algorithm. Certificate C can be authenticated and verified by Certificate B through using the same method. If the integrity and authority of certificate A can be guaranteed, the integrity and authority of the whole chain is guaranteed.

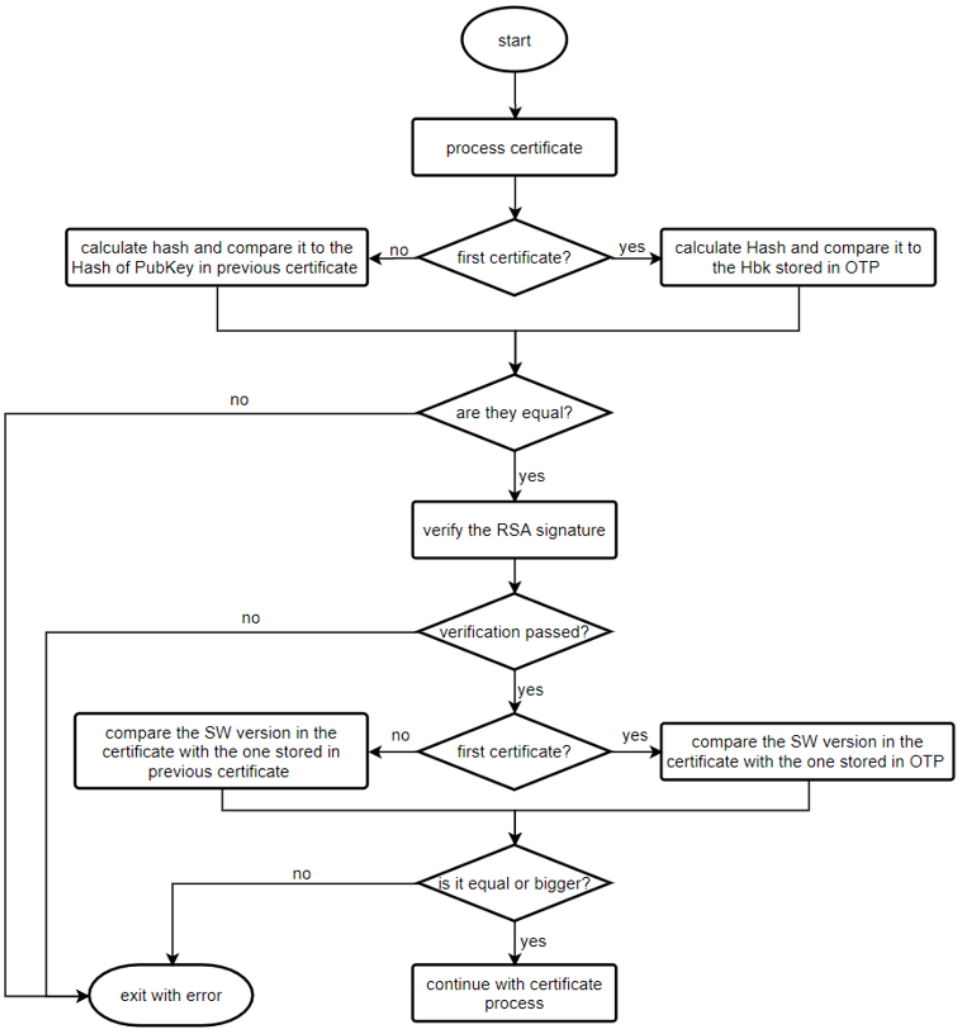
To guarantee the integrity and authority of certificate A in the CC, Arm uses one-time programmable (OTP) memory to save the hash of public key A. It is also called Hbk, Hbk0/1, or Root of Trust (RoT) in the CC. Therefore, the whole certificate chain in the CC is shown as follows:

Figure 3-1: Whole Generate Certificate Chain Structure



Based on the above certificate chain, the following figure shows the general verification procedure of the certificate chain.

Figure 3-2: General Verification Procedure



To verify a certificate, the CC performs the following steps:

1. Retrieves the public key from the certificate and calculate its hash.
2. Verifies the calculated hash:
    - If it is the first certificate in the chain, compares it with the hash value (Hbk, Hbk0\1) that is stored in the OTP.
    - Otherwise, compares it with the saved hash from the previous certificate in the chain.
3. Verifies the RSA signature using the public key of the certificate.
4. Saves the public key hash of the next certificate unless it is the last certificate in the chain.
5. Checks the software version from the certificate:
    - If it is the first certificate in the chain, verifies that it is equal to or larger than the software version (NV counter) stored in the OTP.
    - Otherwise, verifies that it is equal to the software version in the previous certificate in the chain.

Note

All certificates in the chain must have the same software version, which must be later than or the same as the version saved in the OTP.

For details about how to verify other information in each certificate, this is covered later combining with a specific certificate type.

3.2 Secure boot certificate chain and verification flow

The Secure boot certificate chain is composed of the key certificate and the content certificate. The purpose of Secure boot is to guarantee that only authenticated and verified software images are loaded on the target system.

The CC supports two Secure boot certificate schemes: two-level and three-level certificate schemes. You must choose one of them based on system resource.

The two-level Secure boot certificate chain order is as follows:

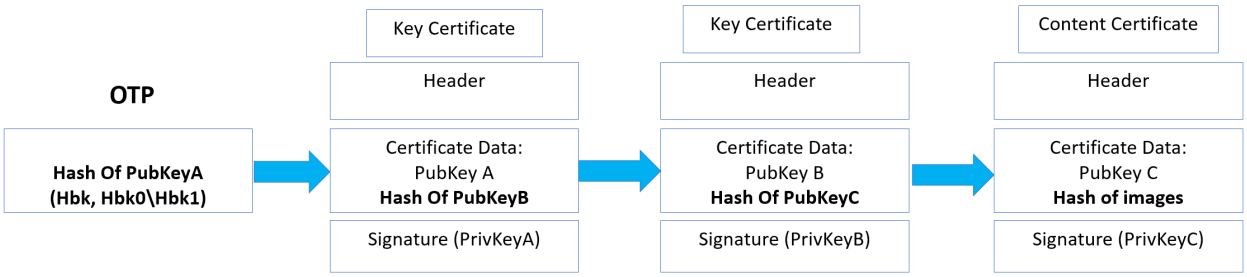
1. Key certificate
2. Content certificate

The two-level Secure boot certificate chain order as follows:

1. Key certificate
2. Key certificate
3. Content certificate

The following figure shows an example of three-level Secure boot certificate chain:

Figure 3-3: Secure Boot Certificate Chain Structure



The key certificate is used to validate the hash of the public key of next certificate in the chain. The detailed information of the key certificate is listed in Table 3-1. The table also shows the structure of Arm proprietary key certificate. The X.509 key certificate includes the corresponding items, but the structure is slightly different. This document uses the Arm proprietary certificate structure as an example.

Table 3-1: Arm Proprietary Key Certificate

Section	Field	Size (32bits word)	Description	Detailed information
Header	Magic Number	1	Certificate name	0x53426b63 SBkc (Secure Boot Key Certificate).
	Version Number	1	Version identifier	This version number is the certificate version number (1.0), not the software version.
	Size	1	Signed content size (in words)	The length from the start of Header to the start of Signature (not including Signature).
	Flags	1	Bit[3:0] Hbk-ID Bit[31:4] not used	Bit[3:0] Hbk-ID: 0=Hbk0, 1=Hbk1, 2=Hbk
Body	Public Key	96	N	Correspond with the private key, which is used to sign this certificate.
		5	Np	
	Software Version	1	Software version	Software version is compared with the Non-Volatile (NV) counter in the OTP.
	HASH	8	Hash value	Hash of the public key of the next certificate.
Signature		96	RSA signature	The RSA signature of all preceding items. Computed by using certificate private key with RSA-PSS SHA256.

For key certificate verification, except the steps in previous general verification procedure shown in Figure 3-2, the further process includes saving hash of public key of the next certificate.

The content certificate is used to load and validate software components. The detailed information of content certificate is listed in the following table:

Table 3-2: Arm Proprietary Content Certificate

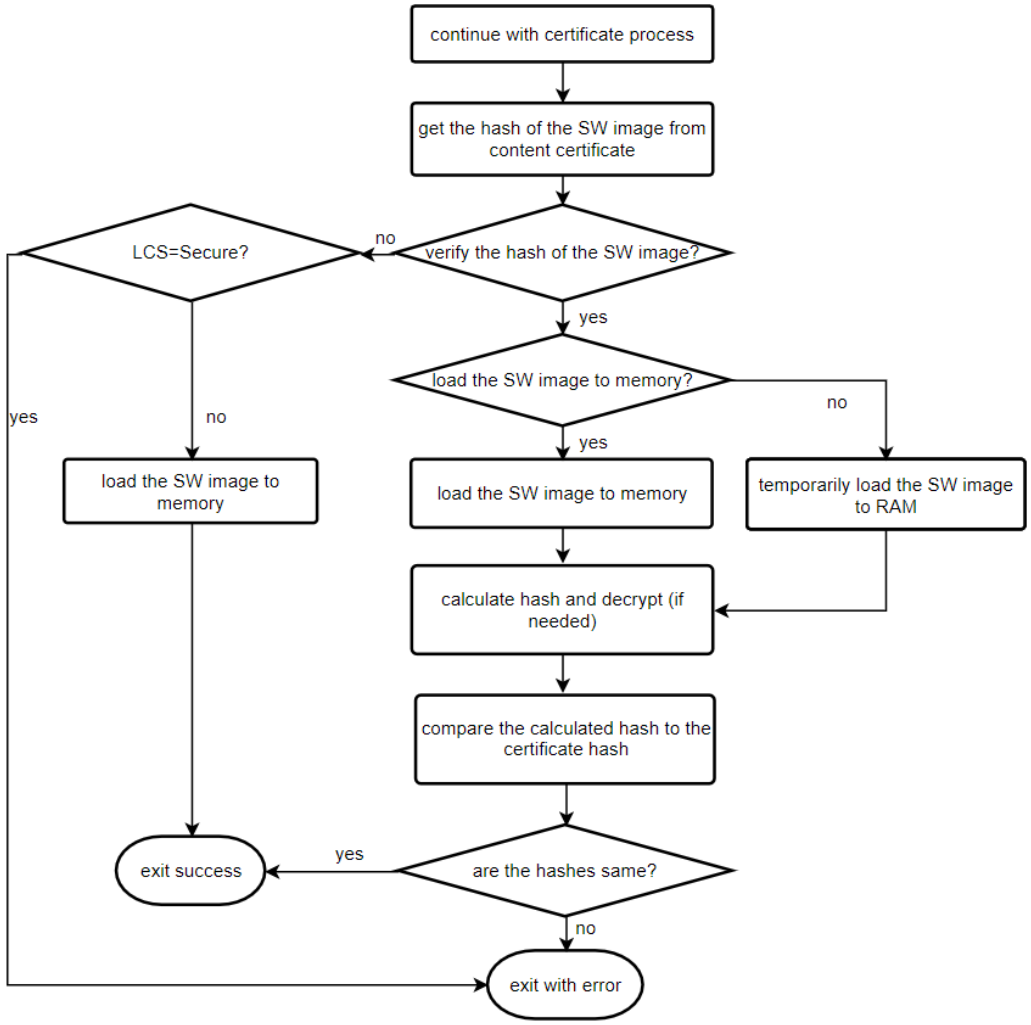
Section	Field	Size (32bits word)	Description	Detailed information
Header	Magic Number	1	Certificate name	0x53426363 SBcc (Secure Boot Content Certificate).
	Version Number	1	Version identifier	This version number is the certificate version number, not software version number.
	Size	1	Signed content size (in words)	The length from the start of Header to the start of Signature (not including Signature).

	Flags	1	• Bit[3:0] not used • Bit[7:4] aes-ce-id • Bit[11:8]load-verify-scheme • Bit[15:12]crypto-type • Bit[31:16] counter of images	• Bit[7:4] image encrypt key: 0=not encrypt, 1=Kceicv, 2=Kce • Bit[11:8]: load and verify, 1= verify only in flash, 2= verify only in memory, 3=load only • Bit[15:12]: 0=AES and HASH, 1=AES then HASH bit[31:16]: how many images are signed, valid value=[1, 16]
Body	Public Key	96	N	Correspond with the private key which is used to sign this certificate.
		5	Np	
	SW Version	1	Software version	It is compared with the software version in previous key certificate.
	NONCE	2		It is used to generate AES IV.
	Software Records	N*(8+3) N= counter of images	8+3 = 1 Hash value + 1 load address + 1 max size + 1 Flag	Flag: 0=image is not encrypted, 1=image is encrypted.
Signature		96	RSA signature	The RSA signature of all preceding items. Computed by using certificate private key with RSA-PSS SHA256.
None Signed Info		N*2 N= counter of images	2= 1 store address + 1 image actual size	The sequence here must be the same as software records in the body part.

For content certificate verification, except the steps in previous general verification procedure shown in Figure 3-2, the further process includes loading and verifying the software images.

The following figure shows the specific remaining verification processing of the content certificate.

Figure 3-4: Content Certificate Specific Verification Procedure



3.3 Secure debug certificate chain and verification flow

The Secure debug certificate chain is composed of the key certificate, enabler certificate, and developer certificate. The purpose of the Secure debug is to guarantee that only authenticated and verified authorities can enable or disable the device-specific debug functions or features, or shift the device to the RMA Lifecycle.

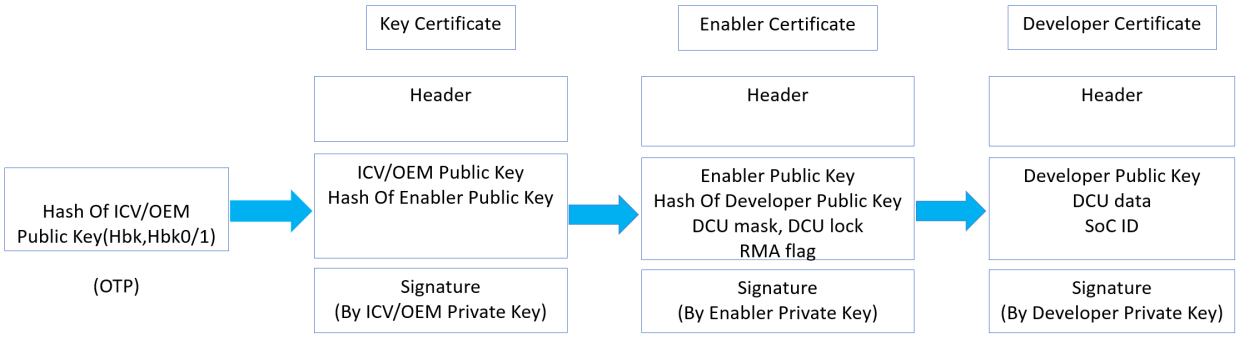
The CC supports two Secure debug certificate schemes: 2-level and 3-level certificate schemes. The customer must choose one of them based on the system resource.

The 2-level Secure debug certificate chain order is enabler certificate→developer certificate.

The 3-level Secure debug certificate chain order is key certificate→ enabler certificate→developer certificate.

The following figure shows an example of 3-level Secure debug certificate chain.

Figure 3-5: Secure Debug Certificate Chain Structure



The key certificate in the Secure debug certificate chain is the same as in the Secure boot certificate chain. Its purpose is to validate the hash of the public key of the next certificate in the chain. For detailed information, see Table 3-1.

The enabler certificate defines the allowed DCU value and DCU lock value or RMA flag. The DCU value determines which DCU bits are open for editing by the developer. The DCU lock value determines which of the DCU bits are locked after successful Secure debug. The DCU value and DCU lock value are used to enable or disable specific debug functions. The RMA flag is used to shift the device LCS to RMA. The detailed information of enabler certificate is shown in the following table:

Table 3-3: Arm Proprietary Enabler Certificate

Section	Field	Size (32bits word)	Description	Detailed information
Header	Magic Number	1	Certificate name	0x5364656E Sden (Secure Debug Enabler Certificate)
	Version Number	1	Version identifier	This version number is the certificate version number, not the software version.
	Size	1	Signed content size (in words)	The length from the start of Header to the start of Signature (not including Signature).
	Flags	1	• Bit[3:0] Hbk-ID • Bit[7:4] LCS • Bit[11:8] RMA • Bit[31:12] reserved	• Bit[3:0] Hbk-ID: 0=Hbk0, 1=Hbk1, 2=Hbk • Bit[7:4] LCS: 0=CM, 1=DM, 5=SE, 7=RMA • Bit[11:8] RMA: 0=not enter RMA, !0=enter RMA

Body	Public Key	96	N	Correspond with the private key, which is used to sign this certificate.
		5	Np	
	DCU Mask	4	DCU mask values	The DCU value that the enabler authorizes to use.
	DCU Lock	4	DCU lock values	The DCU lock value that the enabler authorizes to use.
	Hash	8	Hash value	Hash of public key of developer certificate.
Signature		96	RSA signature	• RSA signature of all preceding items. • Computed by using certificate private key with RSA-PSS SHA256.

The developer certificate is generated by embedding the device SoC-ID, and the DCU value is based on the enabler certificate. The DCU value and DCU lock value in the enabler certificate represent which debug functions are authorized by the ICV and OEM. The DCU value in the developer certificate represents which debug functions are enabled or disabled by the developer. The developer can only enable the functions authorized by the ICV and OEM, but cannot enable the functions that are not authorized by the ICV and OEM.

The developer is the person who debugs the device. The enabler is the ICV, OEM, or the third party, which is trusted and authorized by the ICV and OEM. The enabler has the authority to open specific debug capabilities to developers. Developers enable debug capabilities on a device based on the enabler certificate.

The detailed information about the developer certificate is listed in the following table:

Table 3-4: Arm Proprietary Developer Certificate

Section	Field	Size (32bits word)	Description	Detailed information
Header	Magic Number	1	Certificate name	0x53646465 Sdde (Secure Debug Developer Certificate).
	Version Number	1	Version identifier	This version number is the certificate version number, not the software version. Version 1.0 is used in this document.
	Size	1	Signed content size (in words)	The length from the start of Header to the start of Signature (not including Signature).

	Flags	1	Bit[31:0] reserved	All fields are reserved.
Body	Public Key	96	N	Correspond with the private key, which is used to sign this certificate.
		5	Np	
	DCU Mask	4	DCU mask values	The DCU value that the developer decides to use.
	SoC ID	8	SoC ID	The SoC ID is derived from Huk by using Kdf in Secure boot.
Signature		96	RSA signature	The RSA signature of all preceding items. Computed by using certificate private key with RSA-PSS SHA256

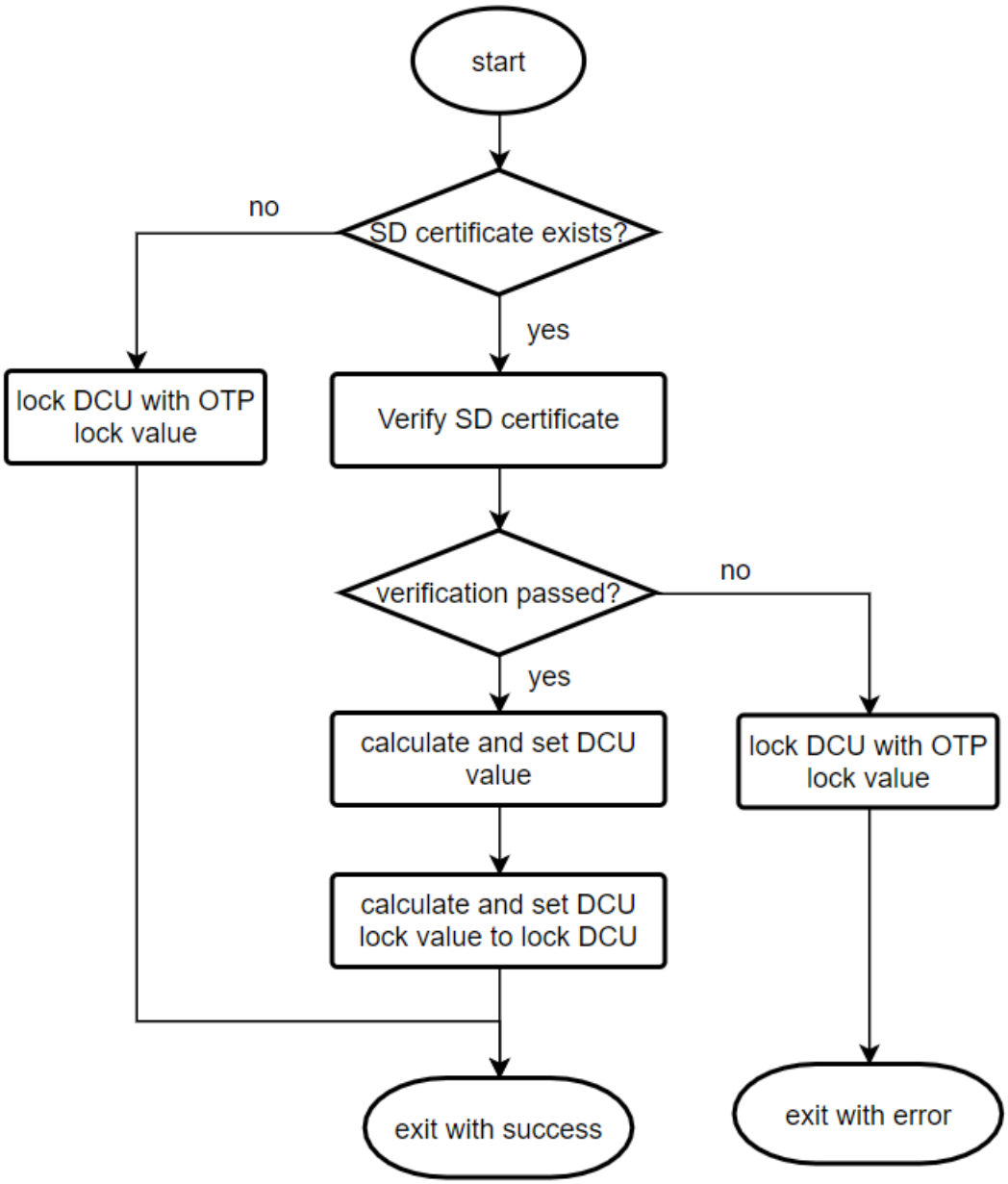
The main items in Secure debug certificates are DCU Mask and DCU Lock. The Secure debug uses a dedicated hardware interface to output control words (DCU value and DCU lock value) based on:

- The DCU Mask and DCU Lock in Secure debug certificates
- The device internal DCU-related items, such as:
    - The DCU lock value in the OTP
    - The DCU default value
    - The DCU permanent disable value
    - The DCU restriction mask in the RTL

For Secure debug, except certificate authentication and verification, the remining process is to extract and calculate DCU value and DCU lock value.

The following figure shows the flow of Secure debug processing:

Figure 3-6: Secure Debug Certificate Processing



4 Certificate Chain Creation and Verification

4.1 Generating and verifying Secure boot certificate chain

CC products provide offline tools to generate certificates for the Secure boot and Secure debug. The offline tools are Python scripts, which receive a configuration file as a parameter to create certificates.

The offline tools are in `utils/src/cc3x_boot_cert/cert_utils/` or `utils/src/cc3x_boot_cert/x509cert_utils/` to generate Arm proprietary certificates or X.509 certificates respectively.

Before using the offline tools, you must build utilities first. You can consult the release note to build the utilities. After the build is complete, the corresponding tools are in `utils/bin/`. To run the tools, open a command line under the directory `utils/bin/` and execute corresponding scripts.

For the Secure boot, the offline tools include key certificate generation tool (`cert_key_util.py`) and content certificate generation tool (`cert_sb_content_util.py`). To run the tools, execute the following commands:

```
python3 cert_key_util.py <cfg_file>

python3 cert_sb_content_util.py <cfg_file>
```

Therefore, it is important to generate the cfg_file, which includes all required parameters for generating the certificate.

The following table shows the required parameters to generate key certificate. Of all the parameters, `cert-keypair-pwd` is optional and other parameters are mandatory. According to the table, you must prepare `RSA keypair`, `hbk-id`, NV counter, and the public key of the next certificate before generating the key certificate.

Table 4-1: `cert_key_util.py` Configuration File Parameters

Parameter	Input/Output (I/O)	Description
`[KEY-CFG]`	I	A mandatory Header.
`cert-keypair`	I	A mandatory PEM format file including RSA keypair for signing this certificate.
`cert-keypari-pwd`	I	Optional password for the RSA keypair file
`hbk-id`	I	Mandatory and the `Hbk-id` indicates which field of OTP memory is used to verify this certificate. The Hbk-id can be: • 0: Hbk0 (ICV)

		• 1: Hbk1 (OEM) • 2: Hbk
`nvcounter-val`	I	It is a mandatory counter, corresponding to the software version in the certificate.
`next-cert-pubkey`	I	A mandatory PEM format file, including RSA public key for signing the next certificate in the chain.
`cert-pkg`	O	A mandatory filename of the final output key certificate package in binary format.

To generate key certificate, perform the following steps:

1. Prepare the RSA key pair
   A. Use the following openssl command to generate RSA key pair.
   ```
   openssl genrsa -aes256 -out firstkey.pem 3072
   ```
   The output file `firstkey.pem` includes both public key and private key. When you execute the command, you are requested to input a password. This password is used to encrypt the private key of the key pair and the encrypt algorithm is aes256.

   B. Use the following command to extract the public key from the key pair of `firstkey.pem`.

   ```
   openssl rsa -in firstkey.pem -out firstpublic.pem -pubout
   ```

   If this key pair is used to generate the first certificate in the chain, its public key is used to generate RoT.

   If this key pair is used to generate other certificates in the chain, its public key is used as a parameter to generate the previous certificate.

2. Decide the Hbk-id based on the RoT scheme.
   Hbk-id has the following values:
   0 = Hbk0 (128bit)
   1 = Hbk1 (128bit)
   2 = Hbk (256bit)

   If you want to generate the first key certificate in the chain, the public key hash of step 1 is RoT. You must generate Hbk or Hbk0/1 based on the first step. Arm provides a tool (`hbk_gen_util.py`) to help you generate Hbk or Hbk0/1. You can use the following command to generate a 256bit Hbk in little endianness：
   ```
   python3 ./hbk_gen_util.py -key firstpublic.pem
   ```

   If you want to generate Hbk0 based on `firstpublic.pem`, use the following command:
   ```
   python3 ./hbk_gen_util.py -key firstpublic.pem -hash_format
SHA256_TRUNC
   ```

   You can use the following command to show more information about `hbk_gen_util.py`.
   ```
   ./hbk_gen_util.py --help
   ```

3. Determine the NV counter value. For example, the NV counter value is 5.
4. Get the public key of the next certificate. For example, the public key is secondpubkey.pem
5. Generate the configuration file.
   The following example is a key certificate configuration file, named as `sb_first_key_cert.cfg`

```
[KEY-CFG]
cert-keypair = ./firstkey.pem
cert-keypair-pwd = ./pwd.txt
hbk-id = 2
nvcounter-val = 5
next-cert-pubkey = ./secondpubkey.pem
cert-pkg = firstkey_cert.bin
```

6. Generate the key certificate by using the following command:

```
python3 ./cert_key_util.py sb_first_key_cert.cfg
```

After the previous steps are complete, the first key certificate `firstkey_cert.bin` is generated, according to the `sb_first_key_cert.cfg` configuration file.

For the Secure boot, the steps to generate the second key certificate are similar to the steps to generate the first key certificate in the chain. The differences include cert-keypair and its own `cert-keypair-pwd` for signing the second key certificate and the `next-cert-pubkey` of the keypair for signing the content certificate.

The configuration file for generating the content certificate is different from the configuration file for generating the key certificate. The following table shows the required parameters for generating the content certificate.

Table 4-2: cert_sb_content_util.py Configuration File Parameters

Parameter	Input/Output (I/O)	Description
`[CNT-CFG]`	I	A mandatory header.
`cert-keypair`	I	A mandatory PEM format file, including RSA keypair for signing this certificate.
`cert-keypari-pwd`	I	An optional password for the RSA keypair file.
`load-verify-scheme`	I	A mandatory software image verification scheme. • 0: load and verify from flash to memory • 1: full hash verification in flash without loading to memory • 2: verify in memory • 3: load from flash
`crypto-type`	I	Mandatory cryptographic verification and decryption mode. • 0: both AES and hash are calculated on plain image • 1: AES is calculated on plain image, but hash is calculated on the encrypted image.

aes-ce-id	I	Mandatory ID of the key that is used for encryption. • 0: None (no encryption for the SW image) • 1: Kceicv (ICV) • 2: Kce (OEM)
aes-enc-key	I	An optional text file, which includes the key used to encrypt the image. If aes-ce-id=0, aes-enc-key is not included.
images-table	I	A mandatory text file containing the list of SW image files to be processed. Each line refers to a single image with following parameters: <image file name> <mem load addr> <flash store addr> < image max size> <encryption flag: 0=not encrypted, 1=encrypted>
nvcounter-val	I	A mandatory counter and it is corresponding to the software version in the certificate.
cert-pkg	O	A mandatory filename of the final output content certificate package in binary format.

Compared to the key certificate parameter table, the following parameters are specific to the content certificate: `load-verify-scheme`, `crypto-type`, `ase-ce-id`, `aes-enc-key` and `images-table`. For more information about each parameter, see the Software Integrators Manual (SIM).

The steps to generate the content certificate are as follows:

1. Prepare the RSA keypair
2. Decide the `lode-verify-scheme`.
3. Decide `crypto-type`.
4. Decide `aes-ce-id` and `aes-enc-key`. If `aes-ce-id=0`, you do not need to list `aes-enc-key`. If `ase-ce-id=1`, the `ase-enc-key` should indicate the file that includes Kceicv. If `ase-ce-id=2`, the `ase-enc-key` should indicate the file that includes Kce.
5. Prepare images-table.
   `Images-table` is a text file, which contains the information of the list of authenticated software image files. Each line refers to a single image, with the following parameters:
   `<image file name> <mem load addr> <flash store addr> <image max size> <encryption flag>`
   The following is an example of image-table named as `images.tbl`.
   ```
   image3.bin 0x30008000 0x0000cef0 0x00004000 0x0
   image2.bin 0x30006000 0x00003458 0x00003000 0x0
   ```
6. Decide `nvcounter-val`. It should be same as the `nvconter-val` in previous key certificates in the chain.
7. Generate a configuration file based on above steps. The following is an example named as `content_config_file.cfg`.
   ```
   [CNT-CFG]
   cert-keypair = ./content_keypair.pem
   cert-keypair-pwd = ./pwd.txt
   images-table = ./images.tbl
   nvcounter-val = 5
   load-verify-scheme = 0
   ```

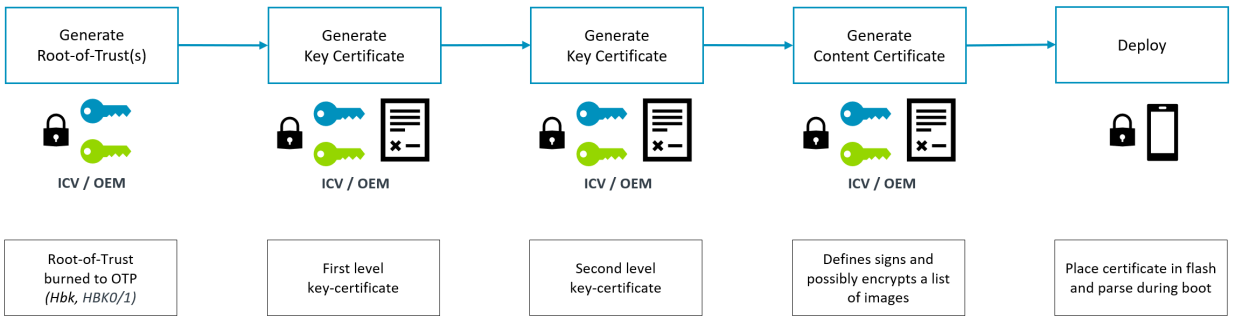
```
aes-ce-id = 2
crypto-type = 0
aes-enc-key = ./aes_key.txt
cert-pkg = content_pkg.bin
```

8. Generate content certificate by using the following command:
   `python3 ./cert_sb_content_util.py content_config_file.cfg`

After above steps, the content certificate of binary format `content_pkg.bin` is generated.

Now you can generate a Secure boot certificate chain. The following figure shows the overall flow of generating a 3-level Secure boot certificate chain. After you finish this process, you can generate three separate certificates.

Figure 4-1: Secure Boot Certificate Chain Generation Flow



After the Secure boot certificate chain is deployed to device, the ROM uses the following APIs to authenticate and verify the certificate chain.

```
CC_SbCertChainVerificationInit()

CC_SbCertVerifySingle()
```

A piece of sample code for how to use the APIs to verify the certificate chain is listed as follows:

```
CCError_t BootROM_SecureBootSequence()
{
        ret = CC_SbCertChainVerificationInit(keyA_CertPkgInf);
        if (ret != OK)
                return ret;
        ret = CC_SbCertVerifySingle(… keyA_CertPkgInf …);
        if (ret != OK)
                return ret;
#ifdef THREE_LEVEL_SCHEME
        ret = CC_SbCertVerifySingle(… keyB_CertPkgInf …);
        if (ret != OK)
                return ret;
#endif /* THREE_LEVEL_SCHEME */
        ret = CC_SbCertVerifySingle(… content_CertPkgInf …);
        return ret;
}
```

For details about using Secure boot APIs to verify the Secure boot certificate chain, refer to the integration test function `bsvIt_secureBoot()` in `bsv_integration_test.c`.

4.2 Generating and verifying Secure debug certificate chain

As for Secure debug, the 3-level certificate chain is key certificate → enabler certificate → developer certificate. The generation of key certificate for Secure debug is the same as for Secure boot. For the enabler and developer certificate, the CC provides the following python tools to generate them:

```
cert_dbg_enabler_util.py

cert_dbg_developer_util.py
```

To run the tools, use the following commands:

```
python3 cert_dbg_enabler_util.py <cfg_file>

python3 cert_dbg_developer_util.py <cfg_file>
```

The parameters to generate the enabler certificate are listed in the following table:

Table 4-3: cert_dbg_enabler_util.py Configuration File Parameters

Parameter	I/O (Input/Output)	Description
[ENABLER-DBG-CFG]	I	Mandatory Header
cert-keypair	I	Mandatory PEM format file including RSA keypair for signing this certificate
cert-keypari-pwd	I	Optional password for the RSA keypair file
lcs	I	Mandatory LCS indicates this certificate is intended for: 0: CM, 1: DM, 5: SE, 7: RMA
rma-mode	I	Mandatory flag when you want to use this certificate for entry into RMA LCS by setting it to nonzero value.
debug-mask	I	Mandatory if you want to use this certificate to authorize debug function but not entry into RMA LCS.
debug-lock	I	Mandatory if you want to use this certificate to authorize debug function but not entry into RMA LCS.
hbk-id	I	Mandatory if rma-mode is defined or if the Secure debug certificate chain is 2 level.
key-cert-pkg	I	Mandatory if the certificate chain is 3-level.
next-cert-pubkey	I	Mandatory PEM format file including RSA public key for signing developer certificate in the chain.

cert-pkg	O	Mandatory filename of the final output enabler certificate package in binary format.

There are some points need to be noted. Firstly, the Secure debug certificate is used to enable or disable debug function or transfer LCS. Therefore, rma-mode, debug-mask, and debug-lock are mutually exclusive. Secondly, the key-cert-pkg is one of the input parameters. That means the key certificate is embedded in the enabler certificate. Similarly, the enabler certificate package is also embedded in the developer certificate. For the Secure debug certificate chain, there is only one binary package, which includes all the certificates in the chain.

To generate enabler certificate, the general steps are as follows:

1. Prepare RSA keypair.
2. Extract the public key of the RSA keypair.
3. Embed the public key to the key certificate if you use a 3-level certificate chain. Or use it to generate Hbk, Hbk0/1 if you use a 2-level certificate chain.
4. Decide the lcs.
5. Decide whether to use rma-mode or debug-mask, debug-lock, and prepare corresponding parameters.
6. Decide the hbk-id if rma-mode is set or a 2-level certificate chain is used.
7. Prepare Key certificate if a 3-level certificate chain is used.
8. Generate a configuration file based on above steps. The following example uses the certificate to enable or disable debug functions. The example file is named as `enabler_config_file.cfg`.

```
[ENABLER-DBG-CFG]
cert-keypair =  oem_keypair1.pem
cert-keypair-pwd = pwd.txt
lcs =  1
#rma-mode =
debug-mask[0-31]   = 0x00112233
debug-mask[32-63]  = 0x44556677
debug-mask[64-95]  = 0x8899AABB
debug-mask[96-127] = 0xCCDDEEFF
debug-lock[0-31]   = 0x00112233
debug-lock[32-63]  = 0x44556677
debug-lock[64-95]  = 0x8899AABB
debug-lock[96-127] = 0xCCDDEEFF
hbk-id = 2
#key-cert-pkg =
next-cert-pubkey = dev_pubkey1.pem
cert-pkg = cert_enabler_pkg.bin
```

9. Generate the enabler certificate by using the following command:

```
python3 ./cert_dbg_enabler_util.py enabler_config_file.cfg
```

After above steps, the enabler certificate of binary format `cert_enabler_pkg.bin` is generated.

To generate developer certificate, consult the following table to generate the configuration file:

Table 4-4: cert_dbg_developer_util.py Configuration File Parameters

Parameter	I/O	Description
`[ENABLER-DBG-CFG]`	I	A mandatory Header
`cert-keypair`	I	A mandatory PEM format file including RSA keypair for signing this certificate
`cert-keypari-pwd`	I	An optional password for the RSA keypair file
`soc-id`	I	A mandatory binary file holding 256-bit SoC_ID of the device.
`debug-mask`	I	A mandatory DCU mask that is set by the developer.
`enabler-cert-pkg`	I	A mandatory enabler debug certificate package.
`cert-pkg`	O	A mandatory filename of the final output enabler certificate package in binary format.

To generate developer certificate, perform the following steps:

1. Prepare the RSA keypair.
2. Extract the public key of the RSA keypair.
3. Send the public key to ICV and OEM to ask for an enabler certificate.
4. Extract `soc-id` from the target device.
5. Determine the debug-mask to be used.
6. Generate a configuration file based on previous steps.
   For example, the following configuration file, `developer_config_file.cfg`, uses the certificate to enable or disable debug functions:

```
[DEVELOPER-DBG-CFG]
cert-keypair =  dev_keypair1.pem
cert-keypair-pwd = pwd.txt
soc-id = soc_id1.bin
debug-mask[0-31]   = 0x00112233
debug-mask[32-63]  = 0x44556677
debug-mask[64-95]  = 0x8899AABB
debug-mask[96-127] = 0xCCDDEEFF
enabler-cert-pkg = cert_enabler_pkg.bin
cert-pkg = cert_developer_pkg.bin
```

7. Generate the enabler certificate by using the following command.

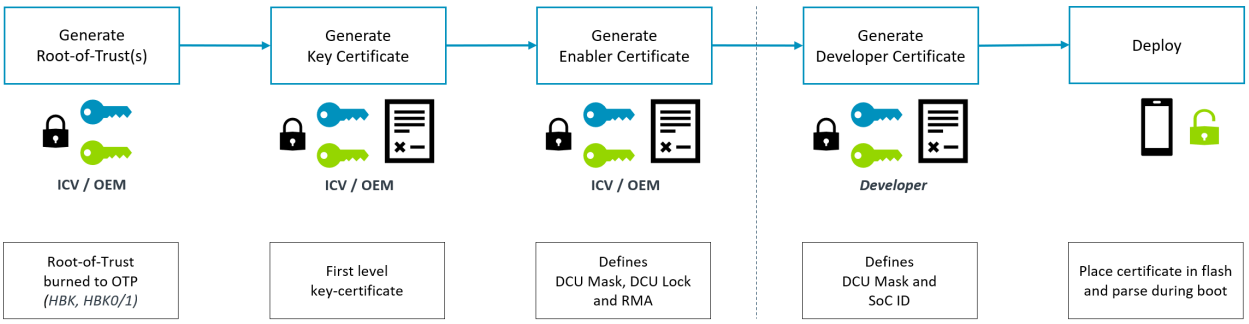
```
python3 ./cert_dbg_developer_util.py developer_config_file.cfg
```

After the previous steps are complete, the developer certificate of binary format `cert_developer_pkg.bin` is generated. This means that the whole Secure debug certificate chain is

generated. The `cert_developer_pkg` includes the key certificate, enabler certificate, and developer certificate in one binary file. The chain for the Secure debug is logic.

The following figure shows the flow of generating the Secure debug certificate chain. The vertical dot-line means the left part is in a trusted environment, while the right part is in an untrusted environment.

Figure 4-2: Secure Debug Certificate Chain Generation Flow



To verify the Secure debug certificate chain in ROM code, only the API `CC_BsvSecureDebugSet()` is needed.

For details about using Secure debug APIs to verify the Secure debug certificate chain, see the integration test function `bsvIt_secureDebug ()` in the `bsv_integration_test.c` file.

5 Differences between Secure boot certificate chain and Secure debug certificate chain

Logically speaking, both the Secure boot certificate chain and Secure debug certificate chain have two or three certificates. Physically speaking, the Secure boot certificate chain consists of three sperate binary packages, while the Secure debug certificate chain has only one binary package. For Secure debug, the key certificate is embedded in enabler certificate and then the enabler certificate is embedded in developer certificate.

Another difference is that the Secure boot certificate chain and Secure debug certificate chain can apply to a different number of devices. The Secure boot certificate chain can work on a serial of devices. However, the Secure debug certificate chain can apply to only one device, because the device SoC-ID is embedded in the developer certificate in the chain.

For the Secure boot certificate chain, all key certificates and content certificates are owned by ICV or OEM that produces the device. For the Secure debug certificate chain, the key certificate is owned by ICV or OEM. The enabler certificate is owned by ICV or OEM or the authorized third party. However, the developer certificate is owned by another party, which might be the seller. Therefore, the Secure boot certificate is initiated by ICV or OEM, while the Secure debug certificate is usually initiated by the developer.

The following scenario describes an example use case for the Secure debug certificate:

1. The device in field is broken.
2. The user asks for the developer, who sells the device or the maintenance station, to fix the issue.
3. The developer generates his or her own RSA key pair, submits the public key to ICV or OEM, and asks for an enabler certificate.
4. The ICV or OEM generates the enabler certificate that includes HASH of the developer public key, and authorized debug functions that are represented by the DCU value and DCU mask.
5. The ICV or OEM sends the enabler certificate to the developer.
6. The developer extracts the SoC-ID from the broken device.
7. The developer generates the developer certificate based on SoC-ID, his or her own DCU value and enabler certificate.
8. The developer transfers the developer certificate to the broken device to enable debug functions, and tries to fix the issues in the broken device.
9. The developer fixes the issue and removes the developer certificate from the device.

Another use case is that the broken device cannot be fixed by the developer. Then, the device is returned to the ICV or OEM. The ICV or OEM generates the enabler certificate and developer certificate directly and sets the RMA flags, but not the DCU value and DCU musk in the enabler certificate. This certificate chain transfers the LCS of the device to RMA. Then the broken device can be analyzed deeply.