



Advanced CA

Development Guide

Issue	13
Date	2016-03-04

Copyright © HiSilicon Technologies Co., Ltd. 2016. All rights reserved.

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of HiSilicon Technologies Co., Ltd.

Trademarks and Permissions



HISILICON, and other HiSilicon icons are trademarks of HiSilicon Technologies Co., Ltd.

All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between HiSilicon and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

HiSilicon Technologies Co., Ltd.

Address: Huawei Industrial Base
Bantian, Longgang
Shenzhen 518129
People's Republic of China

Website: <http://www.hisilicon.com>

Email: support@hisilicon.com



About This Document

Purpose

This document describes the security solutions and production configuration process of advanced conditional access (CA) chips.

Related Versions

The following table lists the product versions related to this document.

Product Name	Version
Hi3716C	V200
Hi3716M	V300/V400/V310/V420/V410
Hi3719C	V100
Hi3719M	V100
Hi3796C	V100
Hi3798C	V100/V200
Hi3798M	V100
Hi3796M	V100
Hi3110E	V500

The advanced CA solutions for Hi3716M V300, Hi3716M V400, Hi3716M V310, Hi3110E V500, Hi3716C V200, Hi3719C V100, Hi3719M V100, Hi3796C V100, Hi3798C V100, Hi3798C V200, Hi3798M V100, Hi3796M V100, Hi3716M V420, and Hi3716M V410 are the same.

Intended Audience

This document is intended for:






- Technical support personnel



- Software development engineers

Symbol Conventions

The symbols that may be found in this document are defined as follows.

Symbol	Description
 DANGER	Alerts you to a high risk hazard that could, if not avoided, result in serious injury or death.
 WARNING	Alerts you to a medium or low risk hazard that could, if not avoided, result in moderate or minor injury.
 CAUTION	Alerts you to a potentially hazardous situation that could, if not avoided, result in equipment damage, data loss, performance deterioration, or unanticipated results.
 TIP	Provides a tip that may help you solve a problem or save time.
 NOTE	Provides additional information to emphasize or supplement important points in the main text.

Change History

Changes between document issues are cumulative. Therefore, the latest document issue contains all changes made in previous issues.

Issue 13 (2016-03-04)

This issue is the thirteenth official release, which incorporates the following changes:

The position of "HiSilicon Common CA Chip" is moved from section 3.3.8 to section 3.3.10 and section 3.3.9 "HiSilicon Non-CA Chip" is added.

Issue 12 (2016-02-26)

This issue is the twelfth official release, which incorporates the following change:

Hi3798C V200 is supported.

Issue 11 (2015-06-02)

This issue is the eleventh official release, which incorporates the following changes:

Hi3716M V420 and Hi3716M V410 are supported.

Issue 10 (2015-03-10)

This issue is the tenth official release, which incorporates the following changes:



Hi3110E V500 is supported. A HiSilicon common CA chip with the flag **H** is added in Table 1-1.

Chapter 2 Advanced CA Solutions

Section 2.4.2.2 is modified.

Chapter 3 Mass Production of Advanced CA STBs

Section 3.2.1 and section 3.3.4.2 are modified, and sections 3.3.8 and 3.3.9 are added.

Issue 09 (2014-12-10)

This issue is the ninth official release, which incorporates the following changes:

Some descriptions are deleted because Hi3110E V300, Hi3716C V100, Hi3716M V200, and Hi3716H V100 are not supported.

Descriptions are added to support Hi3716M V310/Hi3798M V100.

Issue 08 (2014-06-05)

This issue is the eighth official release, which incorporates the following changes:

Chapter 1 Requirements for Advanced Secure CA Chips

Section 1.3.2 is added.

Chapter 3 Development

Section 3.6.8 is modified.

Chapter 4 Production

Section 4.5 is modified.

Issue 07 (2014-04-02)

This issue is the seventh official release, which incorporates the following changes:

Chapter 4 Production

Section 4.3 is modified, and sections 4.4, 4.5, 4.6, and 4.7 are added.

Issue 06 (2013-09-16)

This issue is the sixth official release, which incorporates the following changes:

Hi3716C V200, Hi3719C V100, Hi3719M V100, and Hi3716M V400 are supported.

Issue 05 (2013-04-12)

This issue is the fifth official release, which incorporates the following changes:

Chapter 3 Development

Concept of the chip ID is added.

Descriptions of system software security verification are modified.



Chapter 4 Production

Descriptions of setting PVs of Verimatrix advanced secure chips are added.

Issue 04 (2012-12-06)

This issue is the fourth official release. The entire document is updated.



Contents

About This Document.....	i
1 Advanced CA Chips	1
1.1 Introduction.....	1
1.2 Features	2
2 Advanced CA Solutions.....	3
2.1 Overview	3
2.2 Secure Boot Process	3
2.3 Booting by Using the Secure BOOT Image	5
2.3.1 Classification of BOOT Images	5
2.3.2 Storage Structure of the Secure BOOT Image in a Flash Memory	6
2.3.3 MSID Configuration	8
2.3.4 Development Process for the Secure BOOT Image	8
2.4 Verifying Non-BOOT Software	10
2.4.1 Security Verification	10
2.4.2 Signature Classification	11
2.4.3 Reference Code	18
2.5 Upgrading Secure Software	19
2.6 Decrypting TSs.....	20
2.6.1 Using the Ciphertext CW	21
2.6.2 Using the Plaintext CW.....	23
2.7 Encrypting/Decrypting Contents.....	24
2.7.1 Solution Description	24
2.7.2 Encrypting/Decrypting Memory Data by Using a Random Number Key	24
2.7.3 Encrypting/Decrypting Memory Data by Using a Fixed Key	25
2.8 Recording/Playing Streams by Using the CA PVR.....	27
2.8.1 Recording Streams by Using the Advanced CA PVR.....	27
2.8.2 Playing Streams by Using the Advanced CA PVR	30
2.9 Setting JTAG Access Protection.....	32
3 Mass Production of Advanced CA STBs	33
3.1 Overview	33
3.2 Factory Production Procedures for Advanced CA STBs.....	33



3.2.1 Basic Configurations of Advanced CA Chips	35
3.3 Usage of Various Advanced CA Chips	37
3.3.1 Novel Advanced CA Chips	37
3.3.2 Suma Advanced CA Chips	39
3.3.3 CTI Advanced CA Chips	40
3.3.4 Verimatrix Advanced CA Chips	42
3.3.5 Nagra Advanced CA Chips	47
3.3.6 Conax Advanced CA Chips	47
3.3.7 Irdeto Advanced CA Chips	47
3.3.8 Panaccess Advanced CA Chips	47
3.3.9 HiSilicon Non-CA Chips	51
3.3.10 HiSilicon Common CA Chips	57



Figures

Figure 2-1 STB boot process.....	4
Figure 2-2 Storage structure of the secure BOOT image in a flash memory.....	6
Figure 2-3 Storage structure of the new BOOT image in a flash memory	7
Figure 2-4 Structure of non-BOOT images with special signatures.....	11
Figure 2-5 Format of a signed image for a single file	12
Figure 2-6 Format of a signed image for combined files	13
Figure 2-7 Structure of non-BOOT images with common signatures	16
Figure 2-8 Format of the signed image	17
Figure 2-9 Secure software upgrade.....	19
Figure 2-10 DVB TS decryption	21
Figure 2-11 3-level ciphertext CW decryption	22
Figure 2-12 Data encryption/decryption solution.....	24
Figure 2-13 Encrypting/Decrypting memory data by using a random number key	25
Figure 2-14 Encrypting/Decrypting memory data by using a fixed key.....	27
Figure 2-15 Recording streams by using the advanced CA PVR.....	28
Figure 2-16 File structures	29
Figure 2-17 Playing streams by using the advanced CA PVR	31
Figure 3-1 Factory production process for advanced CA STBs	34
Figure 3-2 Setting basic PVs for an advanced CA chip.....	36
Figure 3-3 Setting PVs for Verimatrix standard STB chips (Hi3716M V300/Hi3716C V200)	45
Figure 3-4 Setting PVs for Panaccess advanced CA chip Hi3716M V310	50
Figure 3-5 Typical lock process.....	52
Figure 3-6 Typical mass production process	54
Figure 3-7 Typical mass production process	55
Figure 3-8 Typical mass production process	56
Figure 3-9 Typical mass production process	57



Figure 3-10 Process for setting PVs for Hi3716C V200	59
Figure 3-11 Configuration of the CAS private data.....	62
Figure 3-12 Mass production process that includes burning the CAS private data	63



Tables

Table 1-1 HiSilicon advanced CA chip models	1
Table 1-2 Differences of security features between common chips and advanced CA chips.	2
Table 2-1 Application scopes of three BOOT images.....	6
Table 2-2 Test commands for verifying special signatures for non-BOOT images	14
Table 2-3 Test commands for verifying common signatures for non-BOOT images	18
Table 3-1 Keys for advanced CA chips of Novel.....	38
Table 3-2 Keys for advanced CA chips of Suma	39
Table 3-3 Keys for advanced CA chips of CTI.....	40
Table 3-4 Keys for standard STB chips of Verimatrix.....	42
Table 3-5 Keys for advanced CA chips of Panaccess	48
Table 3-6 Keys that need to be configured by the STB vendor	58
Table 3-7 CAS vendor private data.....	61



1 Advanced CA Chips

1.1 Introduction

The set-top box (STB) chips are used for digital television broadcasting services. When providing digital television services by using STBs, operators require that the STBs jointly developed by chip vendors, CA vendors, and STB vendors provide some security services to ensure operators' legitimate earnings.

However, the current common STB chips can hardly protect against pirate attacks of hackers. CA vendors and chip vendors need to work together to develop more secure solutions, which brings out the advanced CA chips.

HiSilicon designs various advanced CA chips based on the requirements of a specific CA company and embeds various keys provided by the CA company into the chips. Therefore, STB vendors must be explicit about the CA vendor of the chip and then purchase chips of the corresponding model.

Table 1-1 describes the advanced CA chip models provided by HiSilicon by taking Hi3716M V300 as an example.

Table 1-1 HiSilicon advanced CA chip models

Chip	Description	Remarks
Hi3716M RBCV300 0 x0	Non-advanced CA chip, with the flag 0	Mass-produced
Hi3716M RBCV300 C x0	Conax advanced CA chip, with the flag C	Mass-produced
Hi3716M RBCV300 R x0	Nagra advanced CA chip, with the flag R	Mass-produced
Hi3716M RBCV300 Y x0	Novel advanced CA chip, with the flag Y	Mass-produced
Hi3716M RBCV300 S x0	Suma advanced CA chip, with the flag S	Mass-produced
Hi3716M RBCV300 T x0	CTI advanced CA chip, with the flag T	Mass-produced
Hi3716M RBCV300 M x0	Verimatrix advanced CA chip, with the flag M	Mass-produced
Hi3716M RBCV300 I x0	Irdeto advanced CA chip, with the flag I	Mass-produced
Hi3716M RBCV300 H x0	HiSilicon common CA chip, with the flag H	Mass-produced
Hi3716MRBCV300 K x0	DCAS chip, with the flag K	Mass-produced



Chip	Description	Remarks
Hi3716MRBCV310P _x 0	PANACCESS chip, with the flag P	Mass-produced (Hi3716M V300 does not support the advanced CA feature, while Hi3716M V310 and the later versions support this feature.)

1.2 Features

An advanced CA chip has the following features:

- Provides a unique identifier, that is, the chip ID.
- Supports secure boot using the BOOT image by providing boot signature verification and boot encryption mechanism.
- Supports descrambling by using the ciphertext control word (CW).
- Provides content protection and advanced encryption standard (AES) and triple data encryption standard (3DES) algorithms.
- Supports the advanced CA personal video recorder (PVR).
- Provides an embedded secure CPU.
- Provides JTAG protection.

[Table 1-2](#) lists the differences of security features between common chips and advanced CA chips.

Table 1-2 Differences of security features between common chips and advanced CA chips.

Function	Common Chip	Advanced CA Chip
Descrambling of encrypted streams by using plaintext CWs	Provided	Provided
Descrambling of encrypted streams by using ciphertext CWs	Not provided	Provided
Secure boot. The STB allows only verified programs.	Not provided	Provided
Secure PVR service	Not provided	Provided
JTAG port protection	Not provided	Provided



2 Advanced CA Solutions

2.1 Overview

STBs that use advanced CA chips can provide the following security solutions:

- Secure boot
- TS decryption by using the ciphertext CW
- Content encryption/decryption
- Secure PVR
- JTAG protection

2.2 Secure Boot Process

An STB runs various programs when it works properly, including the services provided by the operator. During the running process, if an unauthorized program is executed, the running environment is under threat, and the operator and customer information may be disclosed.

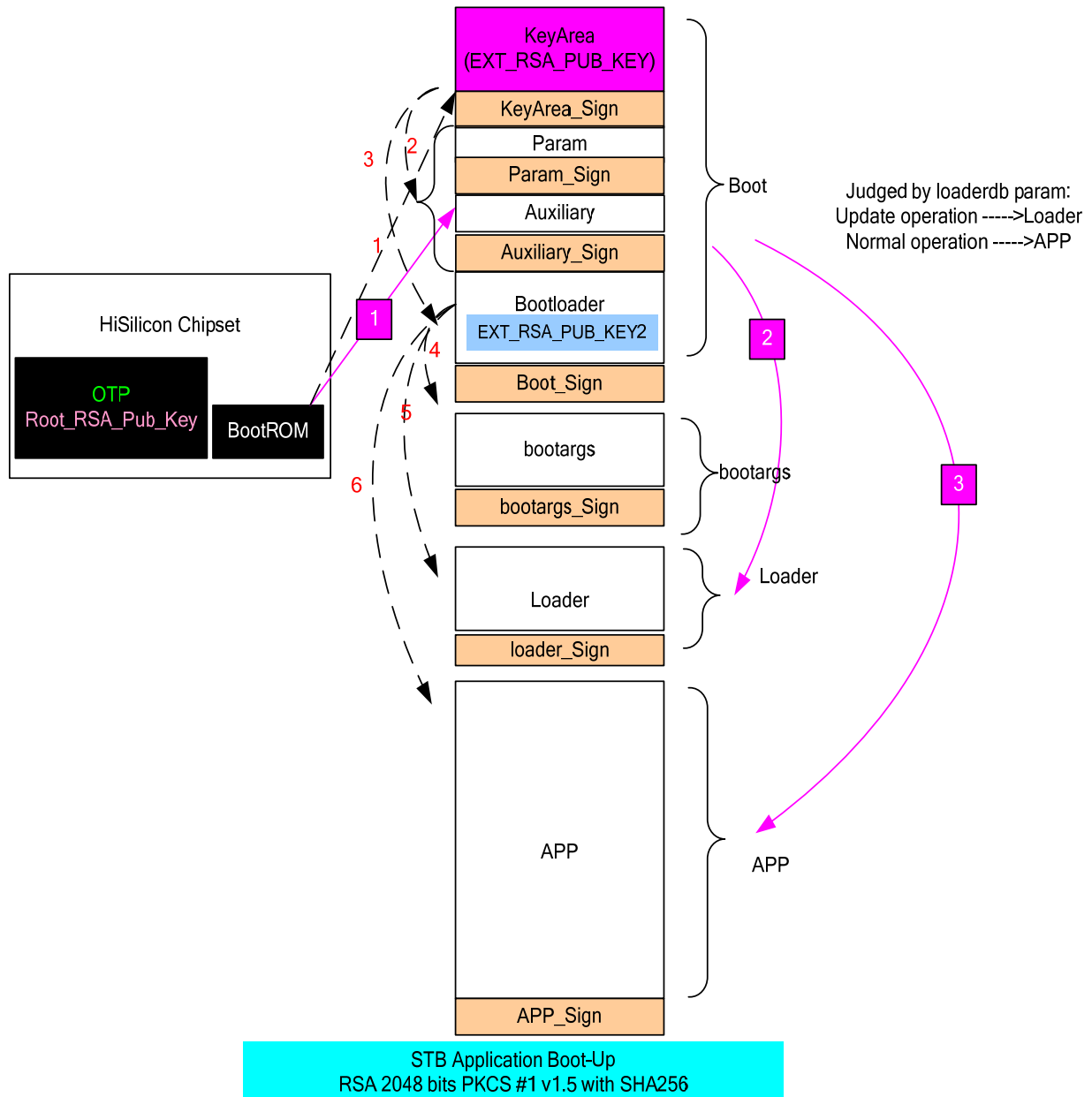
To avoid the security vulnerability, the STB needs to provide an effective solution to prevent the system from being tampered with, that is, the secure boot solution.

HiSilicon advanced CA chips provide a complete secure boot solution. The secure boot solution enables the STB to run only verified programs. Programs that are not signed by the operator or STB vendor cannot be executed in the STB.

The signature verification algorithm is used for verifying programs in the industry. The RSA 2048 bits PKCS #1 v1.5 with SHA256 algorithm is recommended for the digital signature and verification.

[Figure 2-1](#) shows the STB boot process.

Figure 2-1 STB boot process



The one-time programming (OTP) area stores a **ROOT_RSA_PUB_KEY** key provided by the CA company when advanced CA chips are delivered. The **BOOT** image in the flash memory stores the **EXT_RSA_PUB_KEY**, boot code, and their signature data.

After the advanced CA chip is started, the off-chip **BOOT** program, loader program, and **APP** program may be executed. The following describes the program execution process:

- Step 1** After the STB is powered on, the master chip runs the **BootROM** code hardened in the chip. The **BootROM** code obtains the **ROOT_RSA_PUB_KEY** from the **OTP** and verifies the signature in the **KeyArea** of the boot code. After the signature is verified, the **EXT_RSA_PUB_KEY** in the **KeyArea** is obtained. The **BootROM** verifies data in the **Param** area and auxiliary code area (**Aux-Code**) by using the **EXT_RSA_PUB_KEY**, initializes the **DDR** by using the **Param** data and codes in the auxiliary code area, loads the data in the **Boot**



area to the DDR, and then verifies the data by using the EXT_RSA_PUB_KEY. After the data is verified, the Boot program is executed.

Step 2 The BOOT program checks the upgrade flag in the loaderdb partition (customized by the STB vendor) of the flash memory. If upgrade is required, it verifies the loader signature. If verification succeeds, it redirects to the loader to upgrade the software. The key used for verification can be the EXT_RSA_PUB_KEY in the KeyArea of the BOOT image or EXT_RSA_PUB_KEY2 in the BOOT code, depending on the CA vendor or operator.

If the upgrade flag of the loaderdb partition cannot be found, or the upgrade flag is not enabled, the BOOT program verifies the APP signature. After the signature is verified, the system redirects to the APP. In a HiSilicon advanced CA solution, the APP image contains the kernel and file system. The key used for verification can be the EXT_RSA_PUB_KEY in the KeyArea of the BOOT image or EXT_RSA_PUB_KEY2 in the BOOT code, depending on the CA vendor or operator.

----End

The secure boot solution consists of three parts:

- Booting by using the secure BOOT image
- Verifying non-BOOT software
- Upgrading secure software

2.3 Booting by Using the Secure BOOT Image

This section describes the structure of the BOOT image for advanced CA chips, the secure boot process, and how to enable the secure boot function.

2.3.1 Classification of BOOT Images

The BOOT image is the first program image in the off-chip flash memory that the master chip executes directly after power-on. It is also called the BootLoader.

The BOOT image for advanced CA chips differs from that for common chips in the structure. The advanced CA chips use the secure BOOT image, while common chips use the common BOOT image.



NOTE

Unless otherwise specified, the BOOT image in this document indicates the secure BOOT image for advanced CA chips.

HiSilicon chips are classified into three types based on the requirements on the BOOT image:

- Common chips, which use the common BOOT image generated in the SDK
- Advanced CA chips (with secure boot function disabled), which use the unsigned secure BOOT image
- Advanced CA chips (with secure boot function enabled), which use the signed secure BOOT image (typically signed by the CA vendor)



Table 2-1 Application scopes of three BOOT images

BOOT Image Type	Common Chips	Advanced CA Chips with the Secure Boot Function Disabled	Advanced CA Chips with the Secure Boot Function Enabled
Common BOOT image	√	×	×
Unsigned secure BOOT image	×	√	×
Signed secure BOOT image	×	√	√

2.3.2 Storage Structure of the Secure BOOT Image in a Flash Memory

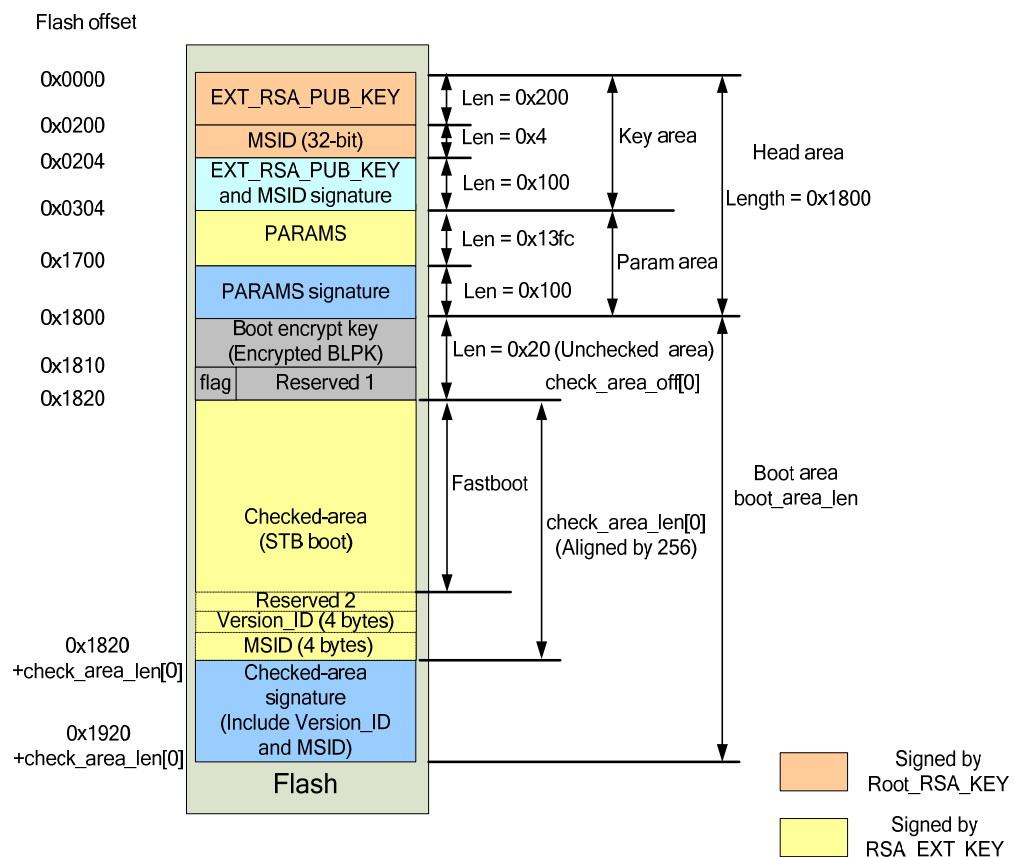


NOTE

Hi3716M V300, Hi3716M V400, Hi3716M V310, Hi3110E V500, Hi3716C V200, Hi3719C V100, Hi3719M V100, Hi3796C V100, Hi3798C V100, Hi3798M V100, Hi3796M V100, Hi3716M V420, Hi3716M V410, and Hi3716M V330 use the same structure.

Hi3716M V310 is taken as an example. Figure 2-2 shows the internal storage structure of the secure BOOT image in the flash memory.

Figure 2-2 Storage structure of the secure BOOT image in a flash memory

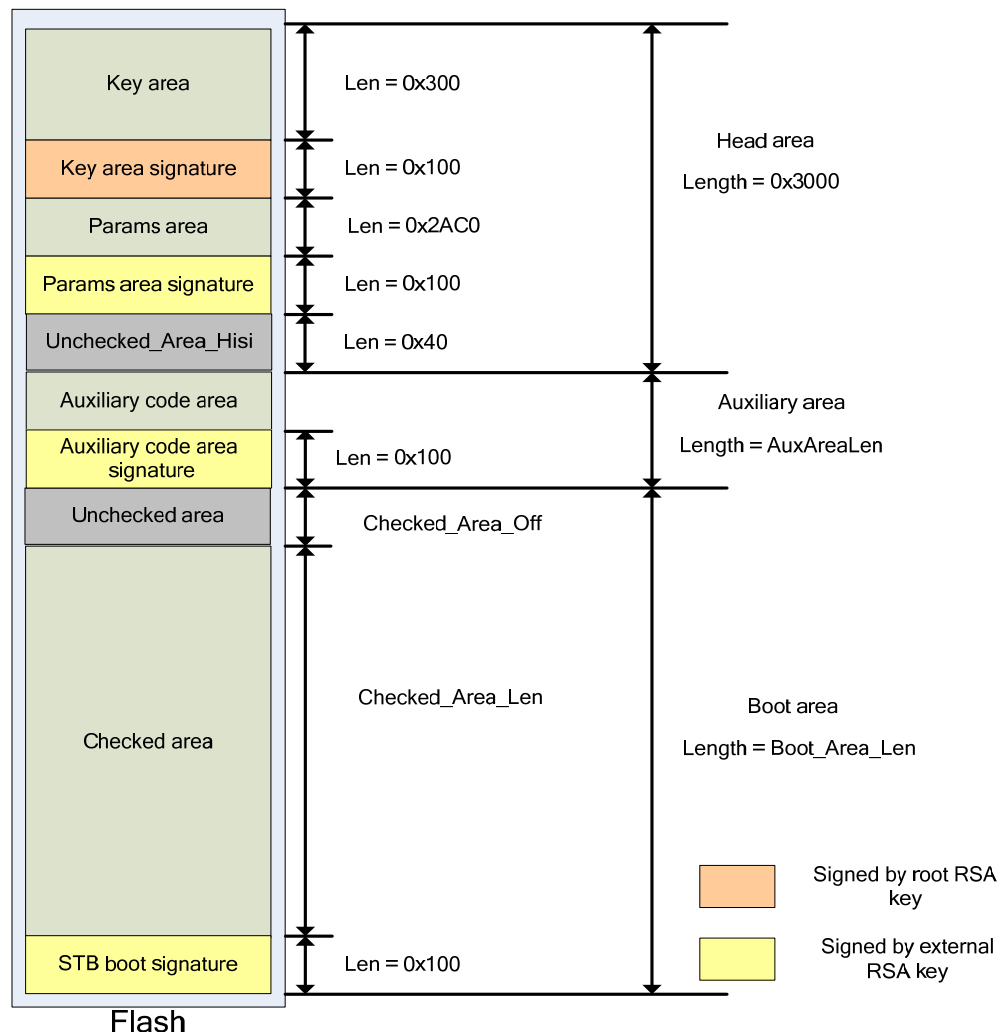


The secure BOOT image is divided into the following two parts, as shown in [Figure 2-2](#):

- Head area
 - Key area: This area stores an external RSA public key, a market segment ID (MSID), and their signatures.
 - Param area: This area stores certain DDR configuration data and their signatures.
- Boot area
 - Checked area: This area stores the boot code and its signature.
 - Unchecked area: This area stores some private data of the CA vendor. The flash memory partition where this area is located does not need signature verification.

[Figure 2-3](#) shows the internal storage structure of the new BOOT image used by Hi3798C V200 and later versions in the flash memory.

Figure 2-3 Storage structure of the new BOOT image in a flash memory



The new BOOT image is divided into the following three parts, as shown in [Figure 2-3](#):



- Head area
 - Key area: This area stores an external RSA public key, a market segment ID (MSID), the address and length of the auxiliary code area, and their signatures.
 - Param area: This area stores certain DDR configuration data and their signatures.
- Auxiliary code area: This area stores the configuration and information of the auxiliary code area.
- Boot area
 - Checked area: This area stores the boot code and its signature.
 - Unchecked area: This area stores some private data of the CA vendor. The flash memory partition where this area is located does not need signature verification.

2.3.3 MSID Configuration

The MSID is the boot security control bit provided by some CA vendors. It has four bytes and is used to distinguish between markets.

MSIDs are stored in the OTP and flash memory. The MSID stored in the OTP is called (MSID)_{OTP}, and the MSID stored in the flash memory is called (MSID)_{Flash}. (MSID)_{OTP} and (MSID)_{Flash} must be consistent to ensure secure boot.

- (MSID)_{OTP} is written to the OTP by calling the corresponding UNF interface during production.
- (MSID)_{Flash} is configured and written into the secure BOOT image when the secure BOOT image is generated by using the CASignTool.

The (MSID)_{OTP} is configured as follows:

Step 1 Initialize the advanced CA module by calling HI_UNF_ADVCA_Init.

Step 2 Set the MSID by calling HI_S32 HI_UNF_ADVCA_SetMarketId().

Step 3 Deinitialize the advanced CA module by calling HI_UNF_ADVCA_DeInit.

----End

(MSID)_{Flash} is configured when the secure BOOT image is generated by using the CASignTool.



NOTE

Pay attention to the byte sequence during the configuration of (MSID)_{Flash}. For details, see section 2.1.3.1 "Setting the Market Segment ID (MSID)" in the *CASignTool Application Notes*.

2.3.4 Development Process for the Secure BOOT Image

To develop, sign, and use the secure BOOT image, perform the following steps:

Step 1 Generate an unsigned secure boot.

- Run **make menuconfig** during compilation of the BOOT image of the SDK to configure the following options (**cfg.mak** in the SDK will be modified):
- Set **CFG_HI_ADVCA_SUPPORT** to **y**.
- Specify the CA vendor. For example, for Conax CA chips, set **CFG_ADVCA_CONAX** to **y**.



- Run the command for compiling the BOOT image in the SDK. **fastboot-burn.bin** is generated.
- Sign **fastboot-burn.bin** by using the CASignTool to obtain the secure BOOT image.

If the secure boot function is disabled during development, the STB vendor can use the self-signed advanced CA boot for development and debugging.



NOTE

- The compilation options **CFG_FUNCTION_DEBUG**, **CFG_FUNCTION_RELEASE**, and **CFG_FUNCTION_FINAL** are required by some CA vendors. Currently only Conax and Nagra require the preceding options.
- During SDK compilation, if any of the preceding three options is enabled, the generated **fastboot-burn.bin** does not provide the network and bootstrapping functions. The STB vendor cannot directly download the signed BOOT image by using the HiTool. In this case, run **make advca_programmer** in the SDK directory to generate **advca_programmer.bin**. After this image is signed, it can be burned to the flash memory by using the HiTool as the programmer file for bootstrapping.
- For Hi3798C V200, the structure of the secure BOOT image complies with that of the non-secure BOOT image. The images generated during SDK compilation are secure images without signatures, which can be burnt directly.

Step 2 Download the secure BOOT image to the flash memory by using the HiTool. The board can be started after the secure BOOT image is downloaded properly to the flash memory.



NOTE

- Select the chip models with a postfix **_CA** when using the HiTool. The secure boot flag bit is still disabled, and the chip does not verify the boot signature. The secure boot flag bit is enabled after the boot development test is complete.
- For Hi3798C V200, the structure of the secure BOOT image complies with that of the non-secure BOOT image. Therefore, the chip type is Hi3789C V200 no matter whether it is a CA chip.

Step 3 Submit the boot-related files (such as **KeyArea.bin**, **ParamArea.bin**, and **BootArea.bin**) to the CA vendor for signature. After obtaining the signatures, merge the preceding images by using the **Merge Signed-BootImage** function of the CASignTool to obtain the official signed BOOT image (**FinalBoot.bin**).

Download the generated secure boot to the flash memory and restart the board. The board can be started properly.



NOTE

- In this step, the secure boot function of the chip is disabled, but the chip can start properly according to step 2.
- Before performing the next step, the STB vendor can send the officially signed BOOT image to HiSilicon to check whether the image is signed properly. If the image is signed properly, go to step 4; otherwise, repeat step 3.

Step 4 Enable the secure boot function of the chip. Restart the chip to check whether it can start properly.

----End



2.4 Verifying Non-BOOT Software

2.4.1 Security Verification

The BOOT image has been verified when the chip is started. Therefore, in the secure boot, the signature for the non-BOOT software needs to be verified before the software is executed. This ensures that all programs run on the STB are valid.

Non-BOOT software is also called system software, and the corresponding image signature is called a non-BOOT image signature or system software signature.

Non-BOOT software is classified into three types:

- **Key data**
Take the bootargs as an example. Bootargs is a set of commands for setting the boot parameters and parameters for starting the kernel on a Linux OS. Although it is non-code data, bootargs determines the system boot process. The bootargs data must be signed if there is any. The bootargs parameters can be configured only after the bootargs signature is verified. You can use the bootargs in either of the following ways:
 - Write the bootargs parameters directly into the boot code. The code lacks flexibility, but data verification is not required.
 - Use the mkbootargs tool provided by HiSilicon to generate the **bootargs.bin** file by using the preconfigured bootargs parameters, and then sign the image.
- **Common system software**
This type of software includes the kernel, loader, and application images.
- **File system software**
The file system images are usually large. Based on the secure boot requirements, file systems are mounted only after the signature is verified. Read-only file systems, such as the Squashfs, are preferred.

File systems with write and read properties can be used during development because signature verification is not required during development. In this case, the parameter **ro** must be added when the kernel boot parameters are configured. For example:

```
setenv bootargs 'mem=96M console=ttyAMA0,115200 root=/dev/mtdblock2
rootfstype=yaffs2 ro
mtdparts=hi_sfc:1M(Boot);hinand:5M(Boot),60M(rootfs)ro, -(others)
mmz=ddr,0,0x86000000,160M'
```



CAUTION

If a file system with read and write properties is used, after the file system is started, it cannot be modified by running commands such as **remount**.

The non-BOOT software is verified as follows:

- Step 1** The secure BOOT program reads the signed non-BOOT images to the memory. If a signed non-BOOT image is encrypted, the secure BOOT program decrypts it first.

Step 2 The secure BOOT program verifies the signed non-BOOT images one by one by using the EXT_RSA_PUB_KEY or EXT_RSA_PUB_KEY2 in the BOOT code. After verification, the program is redirected to the system software.

----End

2.4.2 Signature Classification

HiSilicon provides two signature modes for non-BOOT images:

- Special signature for non-BOOT images. For details, see section 2.4.1 "Special CA Signature for Non-BOOT Images" in the *CASignTool Application Notes*.
- Common signature for non-BOOT images. For details, see section 2.4.2 "Common CA Signature for Non-BOOT Images" in the *CASignTool Application Notes*.

2.4.2.1 Special Signature for Non-BOOT Images

Scenario

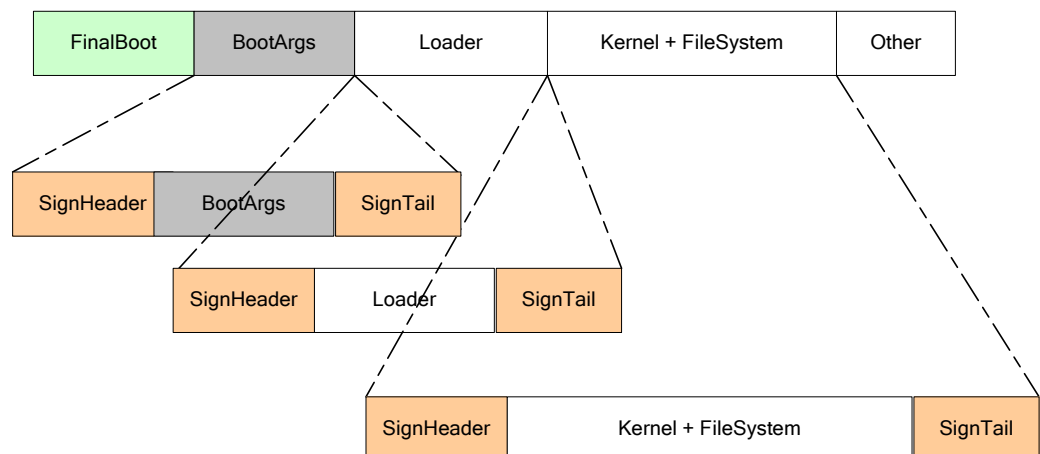
The special signature for non-BOOT images applies to images that need to be encrypted in the flash memory.

- Advantage: The signature can be found quickly, and the system can quickly determine whether the non-BOOT software is encrypted.
- Disadvantage: A data header needs to be added before the non-BOOT software, which may affect the boot parameter segment, kernel, and file system. The offset due to the signature header data needs to be considered.

For example, if the non-BOOT image is copied to a specific address (for example, ADD) in the memory, the actual address of the image is ADD+0x2000.

Figure 2-4 shows the structure of non-BOOT images with special signatures.

Figure 2-4 Structure of non-BOOT images with special signatures



Take Hi3716M V310 as an example. The system secure boot code is stored in `$(SDK_Linux)\source\boot\product\driver\advca`. Modify `ca_config.c` under `boot\product\driver\advca\common\auth`.

- **g_EnvFlashAddr**: specifies the position of boot parameters such as bootargs in the flash memory.
- **g_EnvBakFlashAddr**: specifies the position of the bootargs backup area in the flash memory. If there is no need to verify the bootargs, **g_EnvFlashAddr** and **g_EnvBakFlashAddr** do not need to be specified.
- **g_customer_rsa_public_key_N[256]**: specifies the *N* value of the RSA public key used by the customer.
- **g_customer_rsa_public_key_E**: specifies the *E* value of the RSA public key. The following values are recommended:
 - **g_customer_rsa_public_key_E** = 0x10001
 - **g_customer_rsa_public_key_E** = 0x3
- **isCipherkeyByCA**: specifies whether to use the hardware cipher (R2R key ladder) for encryption.
- **g_CipherAlg**: specifies a cipher algorithm.
- **g_AuthAlg**: specifies the HASH algorithm used for signature. Ensure that the algorithm is the same as the one used for creating a signed image.

Operation Procedures

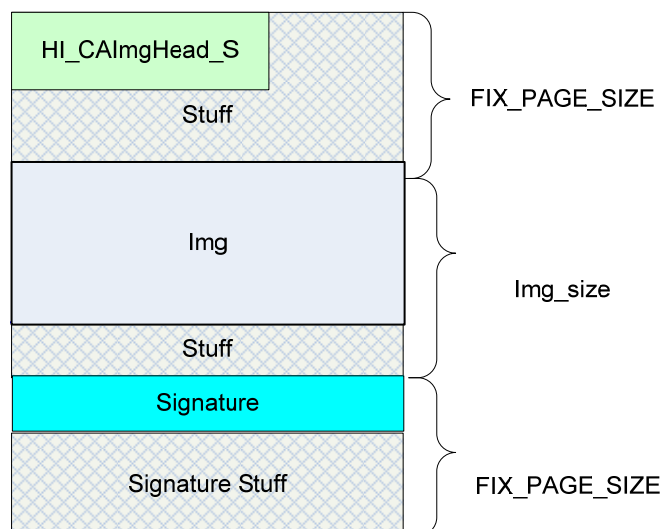
Perform the following steps:

- Step 1** Recompile the boot by running **make hiboot_prepare**, **make hiboot_clean**, and **make hiboot_install**.
- Step 2** Create a signed image by using the **CASignTool** and burn the image to a specific flash partition. Then the software in these partitions can be verified.

For details about how to create a signed image, see the *CASignTool Application Notes*. The **CASignTool** can be used to either generate a signed image for a single file or combine several files and generate a final signed image.

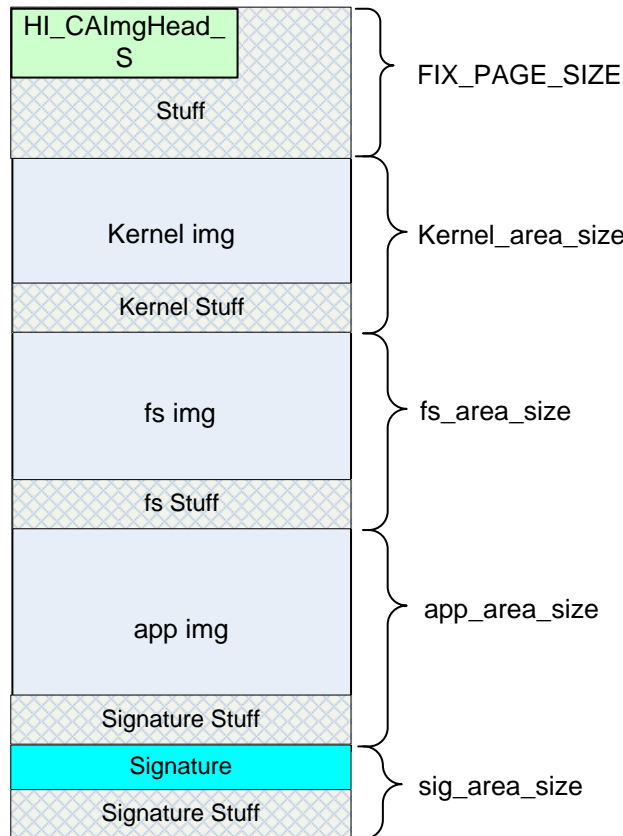
- [Figure 2-5](#) shows the format of a signed image for a single file.

Figure 2-5 Format of a signed image for a single file



- Figure 2-6 shows the format of a signed image created after several files such as the kernel, file system, and application images are combined.

Figure 2-6 Format of a signed image for combined files



Step 3 Mount the memory file system, such as the Squashfs. Some advanced CA systems require that the file system be a read-only file system, more specifically, a memory file system.

- Compile the kernel that supports mounting of a memory file system. Set the kernel configuration **CONFIG_BLK_DEV_RAM** to **y** and recompile the kernel.
- Modify the bootargs to support the memory file system, and change **ramdisk_size** to 40 MB. A modification example is as follows:

```
set bootargs 'mem=96M console=ttyAMA0,115200
initrd=0x82400000,0xc00000 root=/dev/ram ramdisk_size=40960
mtdparts=hinand:1M(Boot), 1M(bootpara),
1M(bootparabak), 40M(sys), 1M(baseparam), 1M(logo), -(others)
mmz=ddr,0,0x86000000,
160M DmxPoolBufSize=0x200000';save
```




CAUTION

Note that the information in red is modified based on image burning, and **initrd** specifies the start address and length of a file system in the memory. Assume that the system image sys created is a combination of the kernel and file system. If the offset address of the file system is 0x400000 and its size is 0xc0000, the system image will be loaded to the memory position with the address 0x82000000. To boot the memory file system, configure the bootargs as follows:

initrd=0x82400000,0xc00000

FIX_PAGE_SIZE is fixed at **0x2000**.

----End

Test Commands

The secure BOOT image provides the test commands described in [Table 2-2](#) for verifying special signatures for non-BOOT images on the board.

Table 2-2 Test commands for verifying special signatures for non-BOOT images

Test Command	Description	Usage
CX_EncryptBurnImage	<p>This command uses the R2R key ladder to encrypt an executable code image in the memory. The encrypted image is written to the specified flash partition.</p> <p>The format of the executable code image must be the format of a non-boot image with special signatures.</p> <p>This command is the same as ca_special_burnflashname in the earlier SDK version.</p>	<p>CX_EncryptBurnImage DDRAddress FlashPartition</p> <p>DDRAddress indicates the start address for the executable code image in the memory.</p> <p>FlashPartition indicates the flash partition to which the encrypted image is written.</p>
CX_EncryptBurnData	<p>This command uses the R2R key ladder to encrypt a data image in the memory. The encrypted image is written to the flash memory.</p> <p>The data image can be a data segment in the memory.</p> <p>This command is the same as ca_special_burnflashnamebylen in the earlier SDK version.</p>	<p>CX_EncryptBurnData DDRAddress ImageLen FlashPartition</p> <p>DDRAddress indicates the start address for the data image in the memory.</p> <p>ImageLen indicates the length of the data image in the memory.</p> <p>FlashPartition indicates the flash partition to which the encrypted image is written.</p>



Test Command	Description	Usage
cx_decrypt_partition	This command is used to decrypt the image in a specified flash partition to a specific memory address. This command is the same as ca_decryptflashpartition in the earlier SDK version.	cx_decrypt_partition FlashPartition FlashPartition indicates the flash partition name.
cx_verify	This command is used to verify the image in a specific flash partition. This command is the same as ca_special_verify in the earlier SDK version.	cx_verify FlashPartition FlashPartition indicates the flash partition name.
verify_test_ex	This command is used to verify the image with a specific flash offset address. This command is the same as ca_special_verifyaddr in the earlier SDK version.	verify_test_ex FlashAddOffset FlashAddOffset indicates the specific flash offset address.
cx_verify_bootargs	This command is used to verify the bootargs partition. This command is the same as ca_special_verifybootargs in the earlier SDK version.	cx_verify_bootargs No parameter is involved.

2.4.2.2 Common Signature for Non-BOOT Images

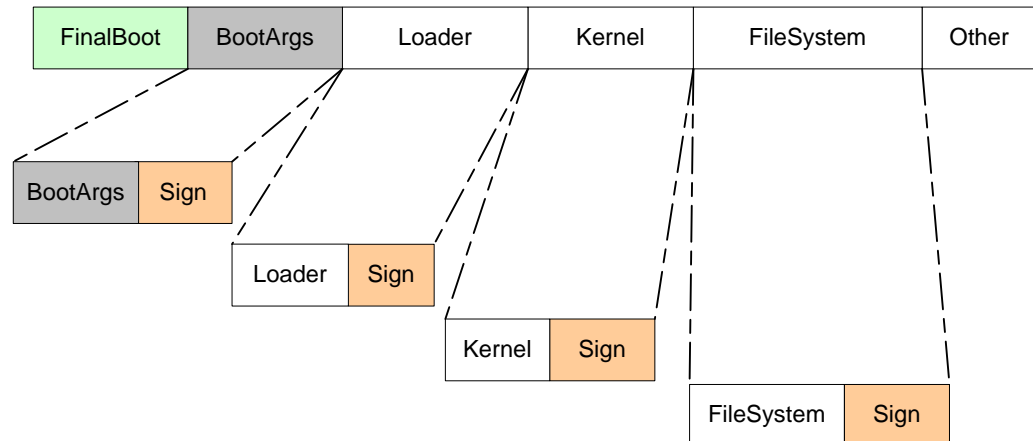
Scenario

The common signature for non-BOOT images applies to the operator market that does not impose high requirements on system security. It can be used if non-BOOT software can be loaded and executed directly from the flash memory or the plaintext non-BOOT software image is allowed.

- Advantage: The storage formats of the images in the flash memory can be the same in debugging and actual application. Signatures can be placed flexibly.
- Disadvantage: The signature is not easily found.

Figure 2-7 shows the structure of non-BOOT images with common signatures.

Figure 2-7 Structure of non-BOOT images with common signatures



Take Hi3716M V310 as an example. The system secure boot code is stored in **boot\product\driver\advca**. Modify **ca_config.c** under **boot\product\driver\advca\common\auth**.

- **g_EnvFlashAddr**: specifies the position of boot parameters such as bootargs in the flash memory.
- **g_EnvBakFlashAddr**: specifies the position of the bootargs backup area in the flash memory.
If there is no need to verify the bootargs, **g_EnvFlashAddr** and **g_EnvBakFlashAddr** do not need to be specified.
- **g_customer_rsa_public_key_N[256]**: specifies the *N* value of the RSA public key used by the customer.
- **g_customer_rsa_public_key_E**: specifies the *E* value of the RSA public key used by the customer. The value can only be either of the following:
 - **g_customer_rsa_public_key_E** = 0x10001
 - **g_customer_rsa_public_key_E** = 0x3
- **isCipherkeyByCA**: specifies whether to use the hardware cipher (R2R) for encryption. If the STB vendor needs to use the encryption function, it is recommended that the encrypted images and signatures be stored in two different partitions.
- **g_CipherAlg**: specifies a cipher algorithm.
- **g_AuthAlg**: specifies the HASH algorithm used for signature. Ensure that the algorithm is the same as the one used for creating a signed image.

Operation Procedures

Perform the following steps:

- Step 1** Recompile the boot by running **make hiboot_prepare**, **make hiboot_clean**, and **make hiboot_install**.
- Step 2** Create a signed image by using the CASignTool and burn the image to a specific flash partition. Then the software in these partitions can be verified.

For details about how to create a signed image, see the *CASignTool Application Notes*. The CASignTool can generate an independent signature image and a signed image at a time.

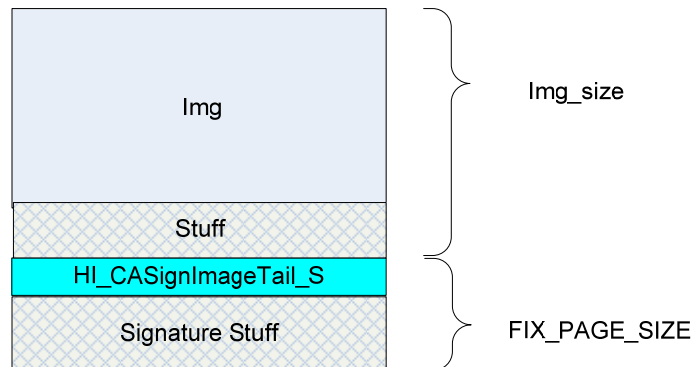


CAUTION

For the Yaffs file system image, only a signature image is generated when it is signed by using the CASignTool.

Figure 2-8 shows the format of the signed image.

Figure 2-8 Format of the signed image



----End

The BOOT code provides a variable **signature_check** for obtaining the signature of an image quickly. The format is as follows:

```
sign:Image,Image_Signature, Image_SignatureOffset
```

- If the values of the variables *Image* and *Image_Signature* are the same, the signed image and the signature are stored in the same partition.
- If the values of the variables *Image* and *Image_Signature* are different, the signed image and the signature are stored in different partitions.

SignatureOffset is used to find the signature data with the offset **SignatureOffset** in the Image_Signature partition. The data is searched by 0x2000. For example:

```
setenv signature_check 'sign:kernel,kernel_sign,0x0 sign:fs,fs_sign,0x00
sign:subfs,subfs_sign,0x00'
setenv signature_check 'sign:kernel,kernel,0x28000 sign:fs,fs,0x600000
sign:subfs,subfs,0x400000'
```

For details, see **ca_common_verify_signature_check** in [Table 2-3](#).

Test Commands

The secure BOOT image provides the test commands described in [Table 2-3](#) for verifying common signatures for non-BOOT images on the board.



Table 2-3 Test commands for verifying common signatures for non-BOOT images

Test Command	Description	Usage
ca_verify	<p>This command is used to verify the signature of a specific flash partition image and copy the image data to a fixed memory address.</p> <p>This command is the same as ca_common_verify_image_signature in the earlier SDK version.</p>	<p>ca_verify image_partition_name sign_partition_name sign_offset_partition</p> <p>image_partition_name indicates the flash partition image to be verified.</p> <p>sign_partition_name indicates the flash partition where signature data is stored.</p> <p>sign_offset_partition indicates the address offset of the signature data in the flash partition.</p>
ca_verify_by_env	<p>This command is used to verify signatures of multiple images in batches.</p> <p>Before running this command, you need to use the signature_check variable to set the parameters for batch signature verification.</p> <p>This command is the same as ca_common_verify_signature_check in the earlier SDK version.</p>	<p>ca_verify_by_env</p> <p>No parameter is involved.</p>
ca_verify_bootargs_by_env	<p>This command is used to verify the signature of the bootargs partition.</p> <p>This command is the same as ca_common_verify_bootargs in the earlier SDK version.</p>	<p>ca_verify_bootargs_by_env</p> <p>No parameter is involved.</p>
advca_verify	<p>This command is used to verify the signature of the specified partition.</p>	<p>advca_verify partition_name</p> <p>partition_name indicates the name of the partition to be verified.</p>

2.4.3 Reference Code

The reference code for non-BOOT software signature is stored in the **Boot** directory of the SDK. The directory may vary according to the SDK version. For example, for Hi3716M V310, the security verification interface under the boot is defined in the **ca_verify.h** file.



If security verification of the non-BOOT software is required, modify `boot\product\main.c` to add the security verification interface in the `fastapp_entry()` function.

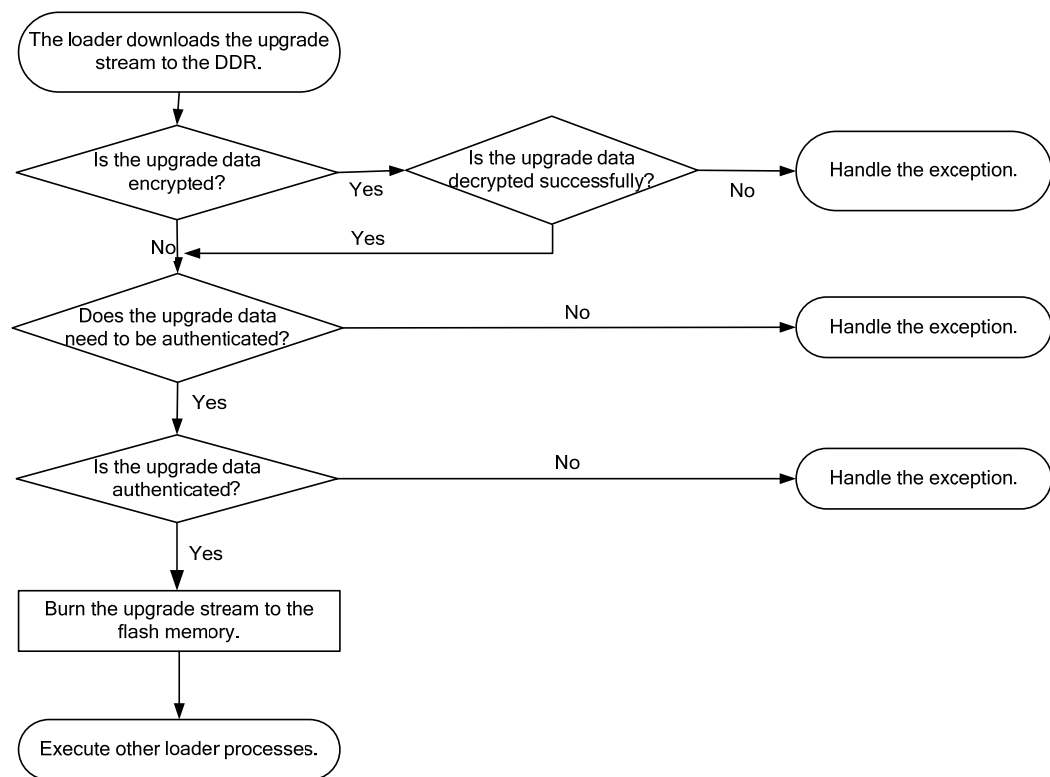
2.5 Upgrading Secure Software

Secure software upgrade is a part of the secure boot solution.

Software upgrade data needs to be verified to ensure security for advanced CA chips. The upgrade image may also need to be encrypted for transmission as required by the operator or CA vendor.

Figure 2-9 shows a secure software upgrade.

Figure 2-9 Secure software upgrade



Take Hi3716M V310 as an example. The secure software upgrade process is as follows:

Step 1 Configure a security verification environment.

Modify `cx_config.c` in the loader code.



CAUTION

The verification algorithm and encryption/decryption algorithms used in secure software upgrade and non-BOOT software security verification described in section 2.4 "[Verifying Non-BOOT Software](#)" must be the same.

- **g_customer_rsa_public_key_N[256]**: specifies the *N* value of the RSA public key used by the customer.
- **g_customer_rsa_public_key_E**: specifies the *E* value of the RSA public key used by the customer. The value can only be either of the following:
 - g_customer_rsa_public_key_E = 0x10001
 - g_customer_rsa_public_key_E = 0x3
- **isCipherkeyByCA**: specifies whether to use the hardware cipher (R2R) for encryption.
- **g_CipherAlg**: specifies a cipher algorithm.
- **g_AuthAlg**: specifies the HASH algorithm used for signature. Ensure that the algorithm is the same as the one used for creating a signed image.

Step 2 Compile the loader with the advanced CA function.

- Modify **cfg.mk**, and ensure that **CFG_ADVCA_TYPE** is defined (not Nagra).
- Run **make hiloader_build** in the SDK top directory.

Step 3 Create signed images by using the CASignTool and create an upgrade stream by using the loader. For details about how to create a signed image, see the *CASignTool Application Notes*.

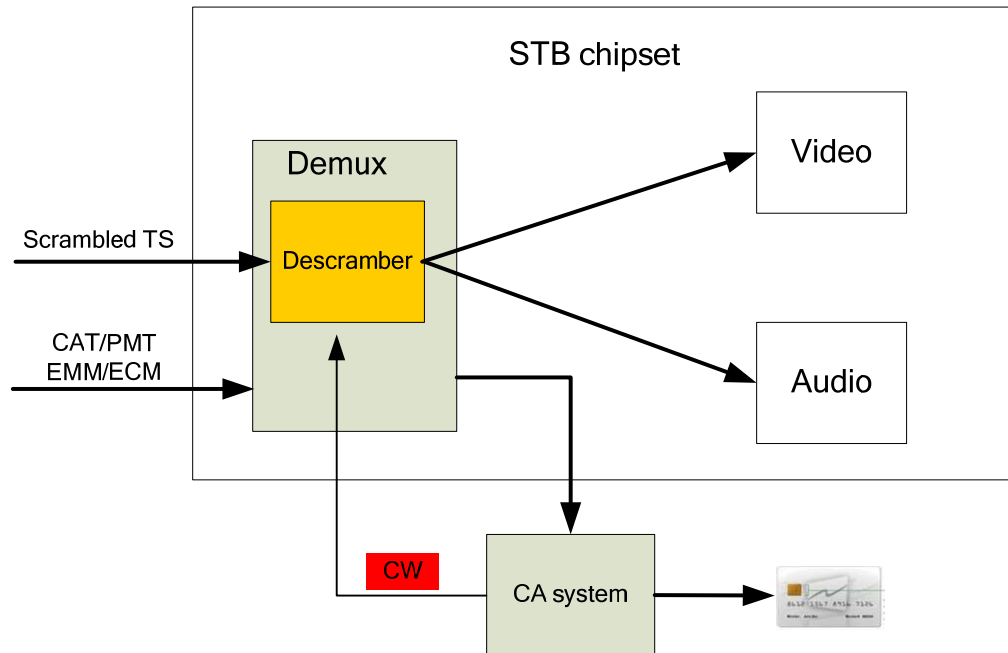
----End

2.6 Decrypting TSs

TS decryption is the core service of DVB. After TSs pass the tuner, analog signals are modulated as digital signals. Then the Demux filters the signals to obtain valid audio/video TS data and transmits the data to the audio/video decoder and then the television for output.

If the TS data transmitted to the Demux is encrypted, it needs to be descrambled as plaintext TSs by using the descrambler. The process is controlled by the CA system. The CA system receives the conditional access table (CAT), entitlement control message (ECM), and entitlement management message (EMM) private data streams and transmits these data streams to the smart card for parsing to obtain the CW of the scrambled TSs. The CA system then sets the CW to the descrambler to descramble the data streams as plaintext TSs.

Figure 2-10 DVB TS decryption



The main security threat of the current STB services is that the CW for decrypting streams may be stolen by hackers and leaked to illegal markets by various means. Therefore, more effective measures need to be taken to protect the CW. The core service of advanced CA system is to encrypt and protect the CW by using the dedicated encryption method of the CA vendor.

The essential difference between the advanced CA system and non-advanced CA system is as follows:

- In the normal CA system, the CW for decrypting TSs is transmitted to the descrambler of the chip in plaintext mode.
Both common chips and advanced CA chips can implement plaintext TS decryption.
- In the advanced CA system, the CW for decrypting TSs is transmitted to the descrambler of the chip in ciphertext mode.
Only advanced CA chips can implement ciphertext TS decryption.

2.6.1 Using the Ciphertext CW

Scenario

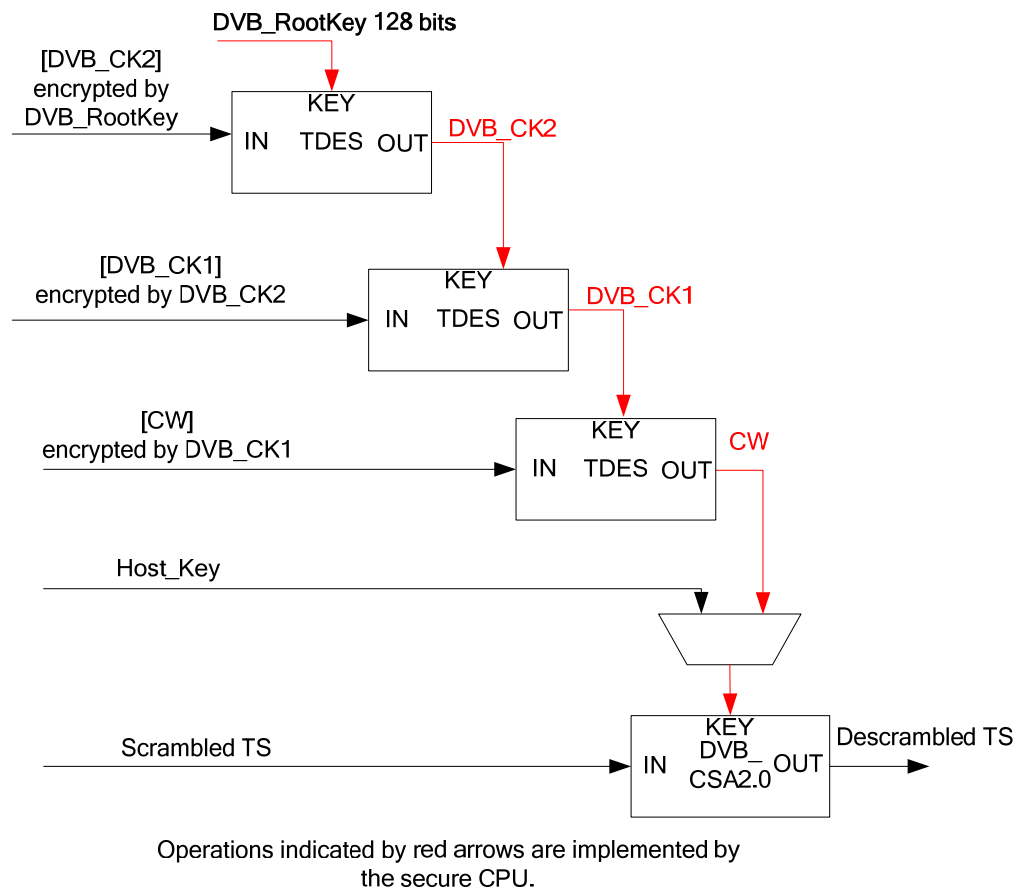
Each advanced CA chip has an embedded root key for decrypting the CW (for example, DVB_RootKey). The CW is encrypted by using multi-level keys for transmission. The number of encryption levels may be 2 or 3, and the encryption algorithm is typically the 3DES.

The following uses the 3-level encryption as an example to describe how to decrypt the ciphertext CW.

The ciphertext DVB_CK1, ciphertext DVB_CK2, and ciphertext CW are called session keys, which are set to the master chip by the application by calling a specific API.

Figure 2-11 shows the decryption process.

Figure 2-11 3-level ciphertext CW decryption



The CW is set to the advanced CA chip as follows:

- Step 1** The STB vendor sets the ciphertext DVB_CK2 to the master chip, and the master chip decrypts the DVB_CK2 by using the DVB_RootKey to obtain the plaintext DVB_CK2.
 - Step 2** The STB vendor sets the ciphertext DVB_CK1 to the master chip, and the master chip decrypts the DVB_CK1 by using the plaintext DVB_CK2 to obtain the plaintext DVB_CK1.
 - Step 3** The STB vendor sets the ciphertext CW to the master chip, and the master chip decrypts the CW by using the plaintext DVB_CK1 to obtain the plaintext CW.
 - Step 4** After obtaining the plaintext CW, the master chip sets it to the descrambler for descrambling.
- End

Operation Procedures

For details about how to create a player to play TSs, see the *HMS Development Guide*. This section focuses on how to configure an advanced CA descrambler for a video channel.

- Step 1** Initialize the advanced CA module by calling `HI_UNF_ADVCA_Init`.
- Step 2** Obtain a video Demux channel by calling `HI_UNF_AVPLAY_GetDmxVidChnHandle`.



- Step 3** Create an advanced CA descrambler by calling `HI_UNF_DMX_CreateDescramblerExt`. Set the parameter **enCaType** to **HI_UNF_DMX_CA_ADVANCE**.
- Step 4** Bind the descrambler to the video channel by calling `HI_UNF_DMX_AttachDescrambler`.
- Step 5** Set the CW encryption algorithm by calling `HI_UNF_ADVCA_SetDVBAIlg`. The default algorithm is 3DES.
- Step 6** If the 3-level keys are used, set the first 2-level encrypted session keys by calling `HI_UNF_ADVCA_SetDVBSessionKey` twice. If the 2-level keys are used, set the level-1 encrypted session keys by calling `HI_UNF_ADVCA_SetDVBSessionKey`.
- Step 7** Set the ciphertext odd CW by calling `HI_UNF_DMX_SetDescramblerOddKey`.
Set the ciphertext even CW by calling `HI_UNF_DMX_SetDescramblerEvenKey`.
- End

Reference Code

See `sample_ca_dvbplay.c`.

2.6.2 Using the Plaintext CW

Scenario

Advanced CA chips support stream decryption by using the plaintext CW by default. For details about the decryption process, see the process for using the HostKey in [Figure 2-11](#).

Operation Procedures

For details about how to create a player to play TSs, see the *HMS Development Guide*. This section focuses on how to configure an advanced CA descrambler for a video channel.

- Step 1** Obtain a video Demux channel by calling `HI_UNF_AVPLAY_GetDmxVidChnHandle`.
- Step 2** Create an advanced CA descrambler by calling `HI_UNF_DMX_CreateDescramblerExt`. Set the parameter **enCaType** to **HI_UNF_DMX_CA_NORMAL**.
- Step 3** Bind the descrambler to the video channel by calling `HI_UNF_DMX_AttachDescrambler`.
- Step 4** Set the plaintext odd CW by calling `HI_UNF_DMX_SetDescramblerOddKey`.
Set the plaintext even CW by calling `HI_UNF_DMX_SetDescramblerEvenKey`.
- End

Reference Code

See `sample_ca_dvbplay.c`.

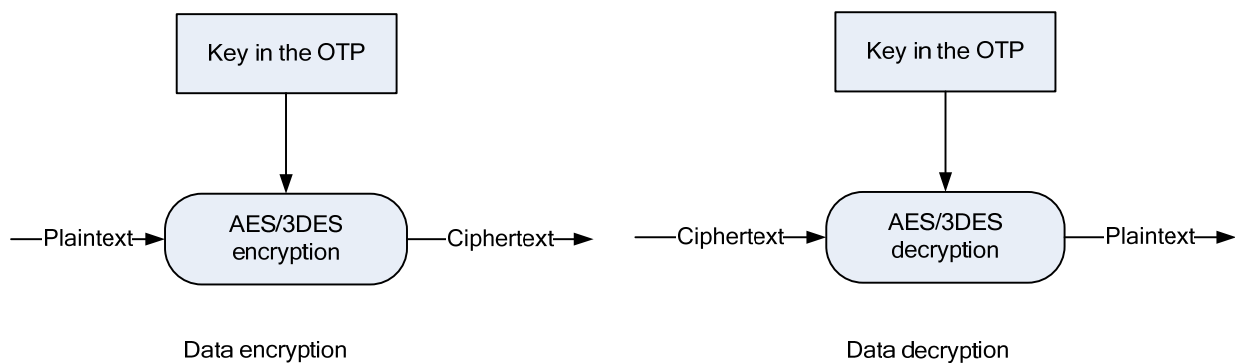
2.7 Encrypting/Decrypting Contents

2.7.1 Solution Description

STBs may store operator or customer private data. Leakage of the private data could jeopardize interests of the operator or customer. To prevent hackers from obtaining the STB information by illegal means, data stored in the flash memory must be encrypted.

Figure 2-12 shows a secure data encryption/decryption solution.

Figure 2-12 Data encryption/decryption solution



The STB vendor can store the key for encrypting data in the OTP area of the master chip. The key stored in the OTP for encrypting or decrypting data can be either accessible or inaccessible to the main CPU.

HiSilicon provides two secure modes for encrypting/decrypting memory data:

- Encrypting/Decrypting memory data by using a random number key
- Encrypting/Decrypting memory data by using a fixed key

2.7.2 Encrypting/Decrypting Memory Data by Using a Random Number Key

Scenario

If the STB vendor requires that the key for encrypting data in each STB be unique, the R2R_Rootkey or STB_RootKey can be used.

Operation Procedures

To encrypt/decrypt memory data by using a random number key, perform the following steps:

- Step 1** Initialize the advanced CA module by calling `HI_UNF_ADVCA_Init`. Start the cipher module by calling `HI_UNF_CIPHER_Open`. Request a physical memory with consecutive addresses for the input and output buffers by calling `HI_MMZ_Malloc`. The input buffer stores the unencrypted data. The output buffer area stores the encrypted data.
- Step 2** Create a cipher handle by calling `HI_UNF_CIPHER_CreateHandle`.
- Step 3** Set the R2R encryption/decryption algorithm by calling `HI_UNF_ADVCA_SetR2Ralg`.



Step 4 Set the session random number key by calling HI_UNF_ADVCA_SetR2RsessionKey.

The HiSilicon advanced CA chips use the 2-level session key by default. Therefore, HI_UNF_ADVCA_SetR2RsessionKey needs to be called only once. However, for some CA chips, the session key level may be set to 3. In that case, HI_UNF_ADVCA_SetR2RsessionKey needs to be called twice.

Step 5 Set the cipher encryption parameters by calling HI_UNF_CIPHER_ConfigHandle. Set **bKeyByCA** to **HI_TRUE**.

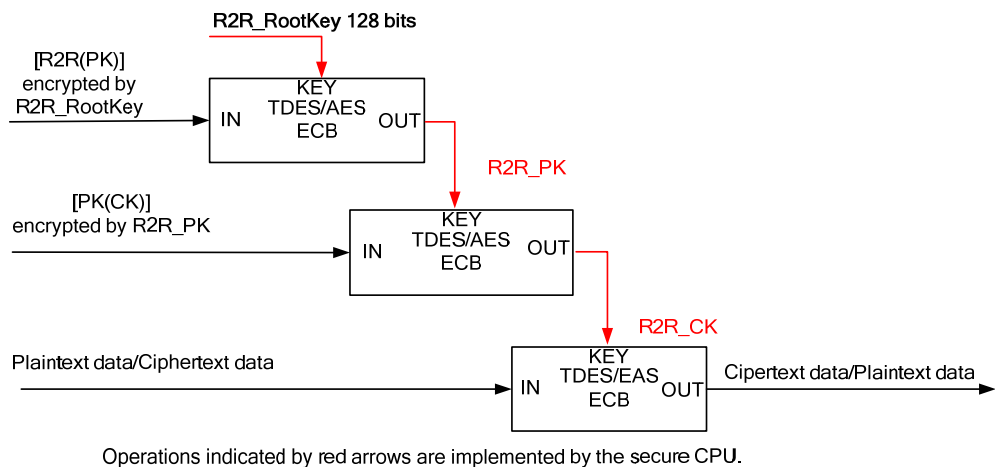
Step 6 Encrypt data by calling HI_UNF_CIPHER_Encrypt.

Step 7 Deinitialize the advanced CA module by calling HI_UNF_CIPHER_Close/HI_UNF_ADVCA_DeInit.

----End

Figure 2-13 shows the process for encrypting/decrypting memory data by using a random number key.

Figure 2-13 Encrypting/Decrypting memory data by using a random number key



Reference Code

See `sample_ca_crypto.c`.

2.7.3 Encrypting/Decrypting Memory Data by Using a Fixed Key

Scenario

The STB vendor needs a key for encrypting data, and the key cannot be stored in the flash memory in plaintext mode. That is, the key needs to be encrypted by using the `R2R_RootKey` before being stored in the flash memory.

Assume that the encrypted key is `R2R_RootKey(Key)`. Then the key can be used to encrypt data, and the encrypted data can be transferred to an STB through a USB bus or network.



The STB uses the R2R_RootKey(Key) to decrypt the data with the SWPK key ladder. The key is confidential and cannot be read by the CPU during decryption, thereby ensuring key security.

This method applies to the scenario in which data needs to be encrypted, but the key for encrypting data cannot be stored in plaintext mode.



NOTE

The implementation of this feature varies according to the model of HiSilicon CA chips. If this feature is required, contact HiSilicon.

Operation Procedures

The memory data is encrypted before the STB is mass-produced and decrypted after the STB is delivered.

To encrypt the specified memory data, perform the following steps:

- Step 1** Define a global key (GlobalKey) (by the STB vendor).
- Step 2** Encrypt the memory data using the AES/TDES algorithm by using the GlobalKey (by the STB vendor) and store the encrypted data in the flash memory of the STB.
- Step 3** Initialize the advanced CA module by calling HI_UNF_ADVCA_Init.
- Step 4** Encrypt the GlobalKey by calling HI_UNF_ADVCA_EncryptSWPK and store the encrypted GlobalKey in the flash memory of the STB.
- Step 5** Deinitialize the advanced CA module by calling HI_UNF_ADVCA_DeInit.

----End

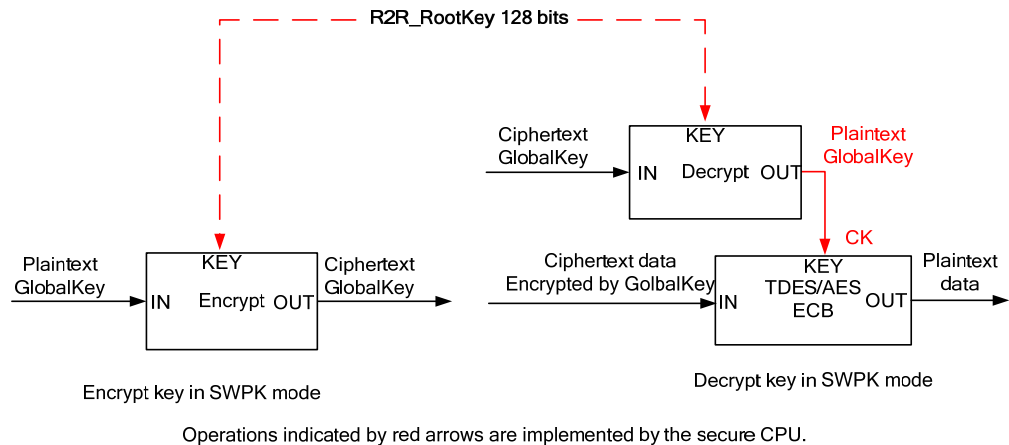
To decrypt the specified memory data, perform the following steps:

- Step 1** Initialize the advanced CA module and cipher module by calling HI_UNF_ADVCA_Init and HI_UNF_CIPHER_Open.
- Step 2** Create a cipher handle by calling HI_UNF_CIPHER_CreateHandle.
- Step 3** Enable the SWPK feature by calling HI_UNF_ADVCA_SWPKKeyLadderOpen.
- Step 4** Set the cipher encryption parameters by calling HI_UNF_CIPHER_ConfigHandle. Set **bKeyByCA** to **HI_TRUE**.
- Step 5** Decrypt the data by calling HI_UNF_CIPHER_Decrypt.
- Step 6** Disable the SWPK decryption by calling HI_UNF_ADVCA_SWPKKeyLadderClose.
- Step 7** Deinitialize the cipher module and advanced CA module by calling HI_UNF_CIPHER_Close and HI_UNF_ADVCA_DeInit.

----End

Figure 2-14 shows the process for encrypting/decrypting memory data by using a fixed key.

Figure 2-14 Encrypting/Decrypting memory data by using a fixed key



Reference Code

See `sample_ca_swpk_keyladder.c`.

2.8 Recording/Playing Streams by Using the CA PVR

In the current PVR solution, streams are decrypted by the descrambler and then recorded to a hard disk. During recording, a known key is used to reencrypt the streams. This key is easily cracked by hackers, and the recorded content may be disclosed. Operators and users require a more secure recording solution, in which the recorded data is accessible only to authorized users.

Under this circumstance, the CA PVR solution is developed. The CA PVR solution uses the unique key of the STB to encrypt decrypted streams. The recorded streams can be decrypted only by the same STB. The unique key of the STB is stored in the OTP area of the master chip.

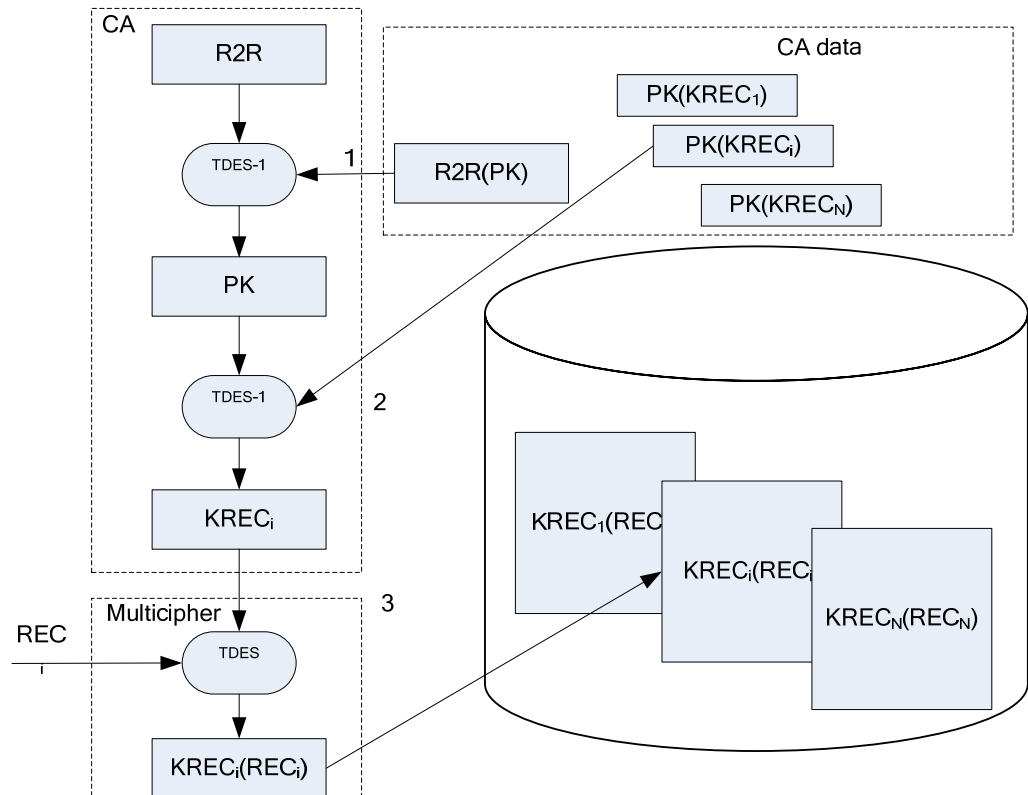
2.8.1 Recording Streams by Using the Advanced CA PVR

Scenario

During recording by using the advanced CA PVR, the unique key embedded in the chip is used as the root key to calculate the key (KPECi) for encrypting streams by using the key ladder. The STB then encrypts streams by using the calculated key (KPECi).

Figure 2-15 shows the process for recording streams by using the advanced CA PVR.

Figure 2-15 Recording streams by using the advanced CA PVR



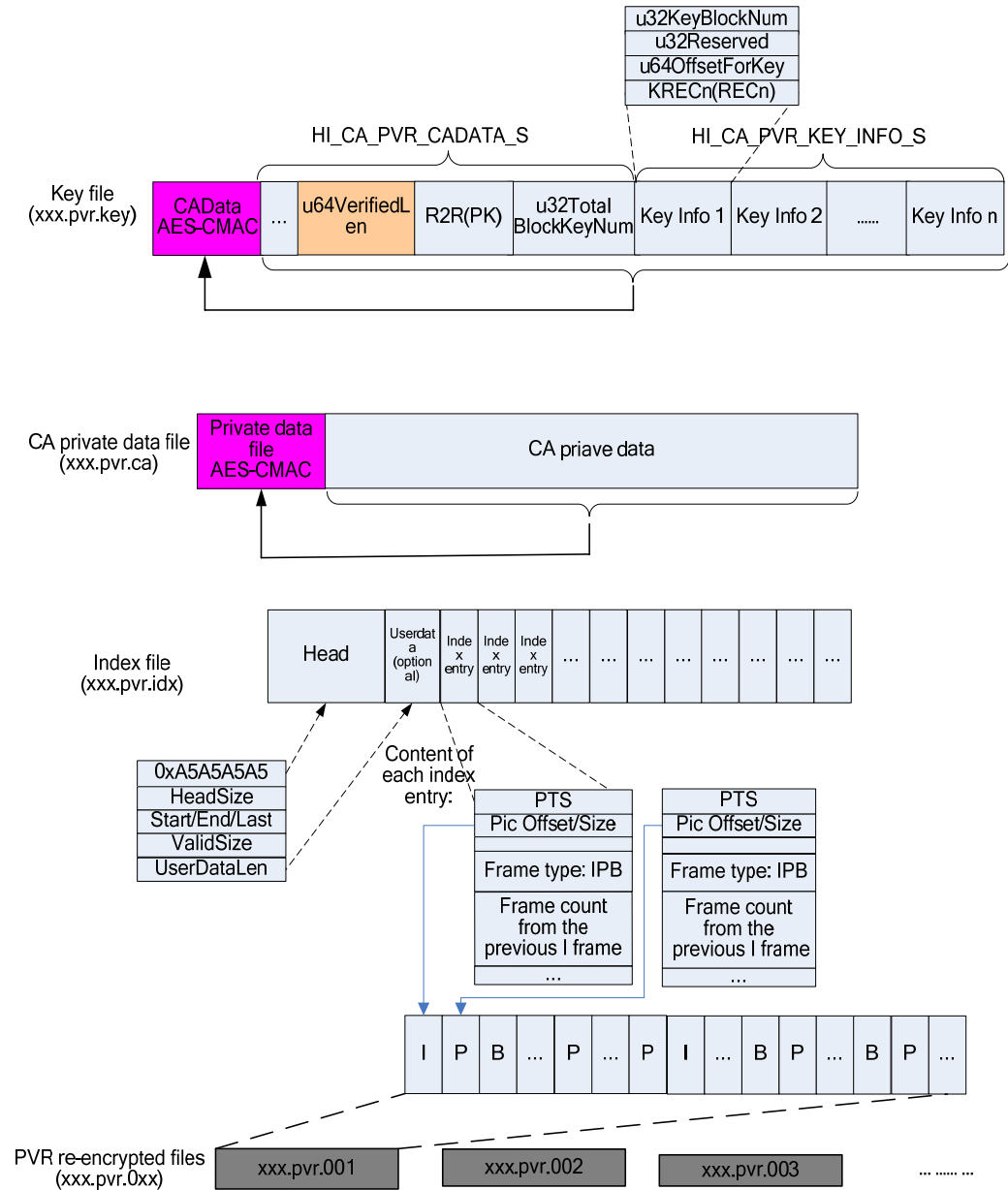
The process is described as follows:

R2R(PK) is the random data generated by the STB program after startup. The STB decrypts the **R2R(PK)** by using the **R2R_RootKey** to obtain the **PK**, which is stored in the internal register of the CA module.

During program recording, the STB program generates a data segment **PK(KRECI)** randomly. After being decrypted by using the **PK**, **KRECI** is obtained and transmitted to the multicipher. **KRECI** is used to encrypt programs during recording. **PK(KRECI)** is stored in the PVR CA private data. Each time the content recorded exceeds a specific size (1 GB by default), another **PK(KRECI)** is generated randomly to record new content.

During recording of the advanced CA PVR, some random keys, such as **R2R(PK)** and **PK(KRECI)**, are generated and stored in the CA private key file (**xxx.pvr.key**) for PVR playback. Because requirements on the PVR solution vary according to the CA vendor, a private data file (**xxx.pvr.ca**) is provided during advanced CA PVR recording for storing private data of the CA vendor. See [Figure 2-16](#).

Figure 2-16 File structures



Operation Procedures

To record streams by using the advanced CA PVR, perform the following steps:

- Step 1** Enable the recording channel of the advanced CA PVR by calling HI_S32 HI_UNF_ADVCA_PVR_RecOpen().
- Step 2** Create a private data file by calling HI_UNF_ADVCA_PVR_GetCAPrivateFileName() and HI_UNF_ADVCA_PVR_CreateCAPrivateFile().
- Step 3** Register the advanced CA PVR recording callback function with the recording channel by calling HI_S32 HI_UNF_PVR_RegisterExtraCallback().



During PVR recording, the parameter **enExtraCallbackType** is **HI_UNF_PVR_EXTRA_WRITE_CALLBACK**. For details about the callback function, see the WriteCallBack function.

- Step 4** Encrypt the recorded streams by calling HI_UNF_ADVCA_PVR_WriteCallBack in the WriteCallBack function.
 - Step 5** (Optional) Save information about the recorded file by calling HI_UNF_ADVCA_PVR_SetStreamInfo() when recording starts.
 - Step 6** Deregister the callback function that is registered with the recording channel by calling HI_UNF_PVR_UnRegisterExtraCallBack() before calling HI_UNF_PVR_RecDestroyChn when the recording is complete.
 - Step 7** Disable the advanced CA recording channel by calling HI_S32 HI_UNF_ADVCA_PVR_RecClose() when the recording is complete.
 - Step 8** Deinitialize the advanced CA module by calling HI_UNF_ADVCA_DeInit.
- End

Reference Code

See **ca_pvr_rec.c** and **sample_ca_adp_pvr.c**.

2.8.2 Playing Streams by Using the Advanced CA PVR

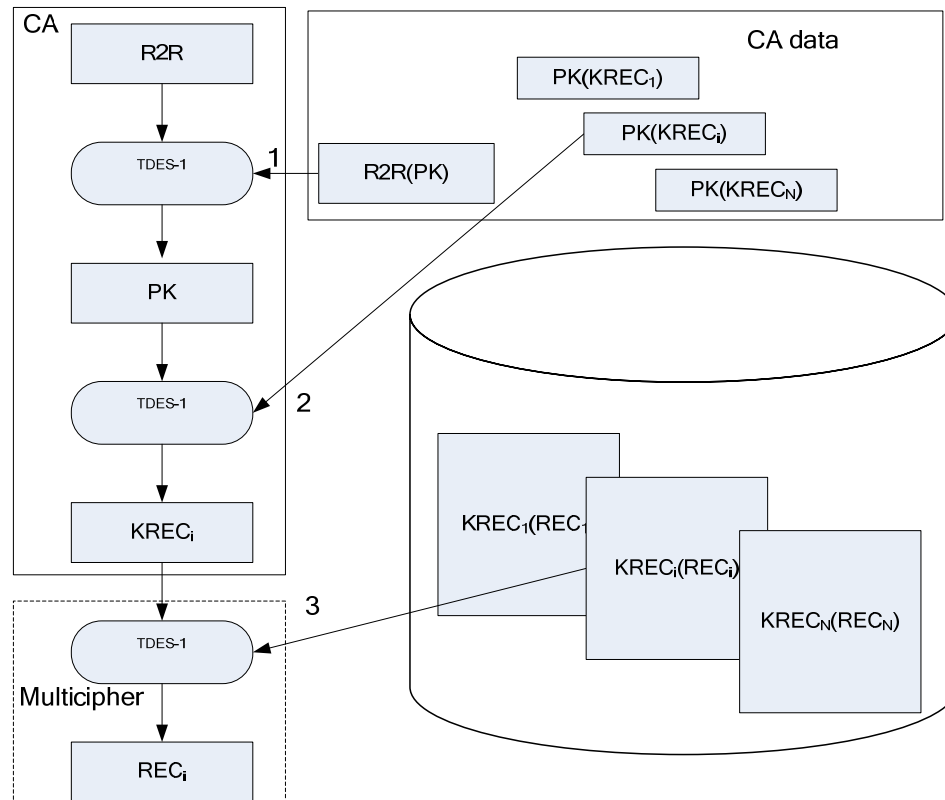
Scenario

The advanced CA PVR can also play the streams it has recorded. The R2R(PK) is read from the recorded private file. The STB decrypts the R2R(PK) by using the R2R_RootKey to obtain the PK, which is stored in the internal register of the CA module.

When a program recorded by the advanced CA PVR is played, the PK(KRECi) is transmitted to the CA module and then decrypted by using the PK to obtain the key (KRECi) for encrypting the streams. KRECi is then transmitted to the multicipher to decrypt the program.

[Figure 2-17](#) shows the process for playing streams by using the advanced CA PVR.

Figure 2-17 Playing streams by using the advanced CA PVR



Operation Procedures

To play streams by using the advanced CA PVR, perform the following steps:

- Step 1** Enable the playback channel of the advanced CA PVR by calling HI_S32 HI_UNF_ADVCA_PVR_PlayOpen(HI_U32 u32PlayChnID).
- Step 2** Verify the PVR private data by calling HI_UNF_ADVCA_PVR_GetCAPrivateFileName() and HI_UNF_ADVCA_PVR_CheckCAPrivateFileMAC().
- Step 3** Register the advanced CA PVR playback callback function with the playback channel by calling HI_S32 HI_UNF_PVR_RegisterExtraCallback().
During PVR recording, the parameter **enExtraCallbackType** is **HI_UNF_PVR_EXTRA_READ_CALLBACK**. For details about the callback function, see the ReadCallBack function.
- Step 4** Decrypt the recorded streams by calling HI_UNF_ADVCA_PVR_ReadCallBack in the ReadCallBack function.
- Step 5** (Optional) Obtain private information about the played file when the playback starts by calling HI_UNF_ADVCA_PVR_GetStreamInfo).
- Step 6** Deregister the callback function that is registered with the playback channel by calling HI_UNF_PVR_UnRegisterExtraCallBack() before calling HI_UNF_PVR_PlayDestroyChn when the playback is complete.



- Step 7** Disable the advanced CA playback channel by calling HI_S32 HI_UNF_ADVCA_PVR_PlayClose() when the playback is complete.
- Step 8** Deinitialize the advanced CA module by calling HI_UNF_ADVCA_DeInit.
- End

Reference Code

See `ca_pvr_play.c` and `sample_ca_adp_pvr.c`.

2.9 Setting JTAG Access Protection

Scenario

The JTAG interface is used for development and test verification of the STB.

The HiSilicon SDK provides the HiWorkbench. STB vendors can connect the HiWorkbench to the STB for debugging by using a JTAG connector over the JTAG interface.

However, the use of the JTAG interface should be restricted for security after mass production of the STB.

The working mode of the JTAG interface for advanced CA chips can be set to password mode or closed mode.

Before the advanced CA chips are delivered, the unique JTAG password provided by the CA company is configured in each chip. During CA certification or STB production, the STB vendor should set the working mode of the JTAG interface to the password mode or closed mode as required by CA companies.

Operation Procedures

To set the working mode of the JTAG interface, perform the following steps:

- Step 1** Initialize the CA module.
- Step 2** Set the access control mode of the JTAG interface by calling HI_UNF_ADVCA_SetJtagMode().
- If the operator requires that the JTAG interface be set to the password mode, the STB vendor needs to set the parameter to HI_UNF_ADVCA_JTAG_MODE_PROTECT.
 - If the operator requires that the JTAG interface be set to the closed mode, the STB vendor needs to set the parameter to HI_UNF_ADVCA_JTAG_MODE_CLOSED.
- End



3

Mass Production of Advanced CA STBs

3.1 Overview

STB vendors purchase HiSilicon advanced CA chips, integrate the CA middleware, and request for STB certification. After being certified by the CA vendor, the STB is mass-produced.

This chapter describes a typical process for producing advanced CA STBs. Producing advanced CA STBs is different from producing common STBs in the permanent value (PV) settings and mass production report. This chapter describes how to configure the PVs according to the requirements of different CA vendors.

The recommended STB production process is as follows:

- Step 1** Purchase advanced CA chips from HiSilicon (determine the CA vendor and chip model).
- Step 2** Conduct production tests to verify hardware and software functions in the STB production line.
- Step 3** Set the PVs for the advanced CA chips.
- Step 4** Apply to the CA vendor for the streams and smart cards for testing whether the advanced CA STBs can descramble streams properly.
- Step 5** Read the chip ID, generate chip ID reports based on the CA vendor's requirements, and send them to the operator.

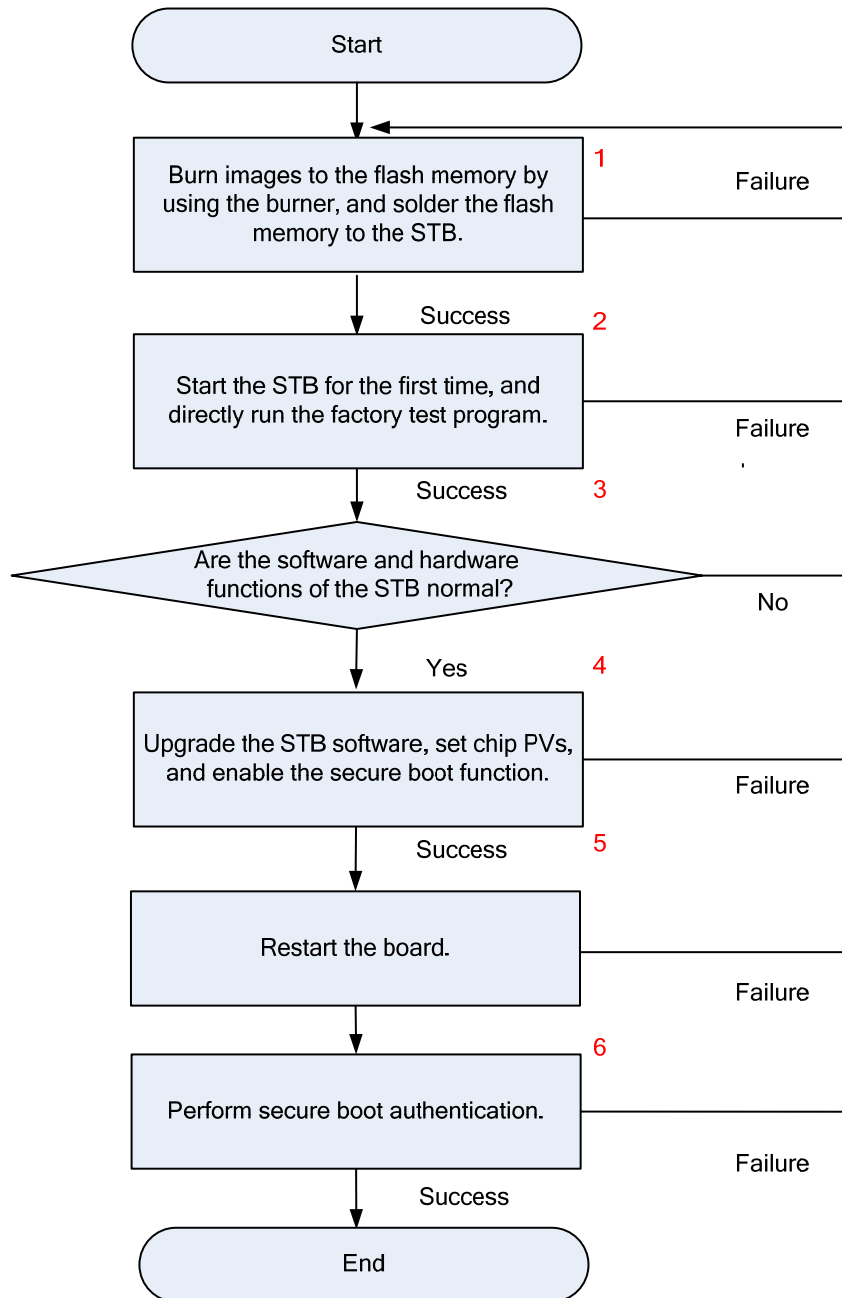
----End

3.2 Factory Production Procedures for Advanced CA STBs

The security control bit PV of the chip needs to be configured during the production of advanced CA chips. [Figure 3-1](#) shows the factory production process for advanced CA STBs.



Figure 3-1 Factory production process for advanced CA STBs



Operation Procedures

Perform the following steps:

- Step 1** Burn the main images (tboot, bootargs, loader, kernel, and file system images) to the flash memory by using the flash burner. Solder the flash memory to the STB board.
- Step 2** Start the board for the first time. The secure boot flag (see the **getSecureBootEn** command) is not configured, which can be identified by the BOOT program. The flag bit is **0**, and the BOOT program is redirected to the STB production test program.



NOTE

- The BOOT program executed in step 2 is not the one for mass production. It contains various production debugging methods to facilitate mass production. The official BOOT program is obtained after step 4 is performed.
- The STB production test program is a small independent program customized by the STB vendor. It can be a part of the loader program or a subprogram (attached to the main program) stored in the file system.
- The STB production test program should provide functions such as checking the software and hardware and downloading data from the network.
- This document does not describe the implementation of the STB production test program.

Step 3 Verify that the software and hardware functions of the STB are normal.

Step 4 Download the upgrade images (including the official BOOT image and non-BOOT images) and set the STB information (including the SN), advanced CA PVs, and secure boot.

Step 5 Restart the STB after the preceding operations are complete.

Step 6 Verify that the STB performs secure boot verification properly according to the advanced CA requirements.

----End

Notes

The high-bandwidth digital content protection (HDCP) key is used to protect the HDMI interface and applies to all HD STB chips. For details, see the related HDCP documents. If you need to protect the HDMI for an advanced CA chip by using HDCP, burn the PVs of the chip, and then burn the HDCP key:

- Burn the HDCP_Key for Hi3716C V100/Hi3716M V200/Hi3716M V300.
- Burn the HDCP_ROOT_KEY for Hi3716C V200/Hi3798M V100/Hi3716M V310/Hi3716M V420/Hi3716M V410 and later versions.

The operation of burning the PV is irreversible. Once the data fails to be written, the chip is damaged and needs to be replaced. The STB vendor can perform the next operation after verifying that the current operation is successful.

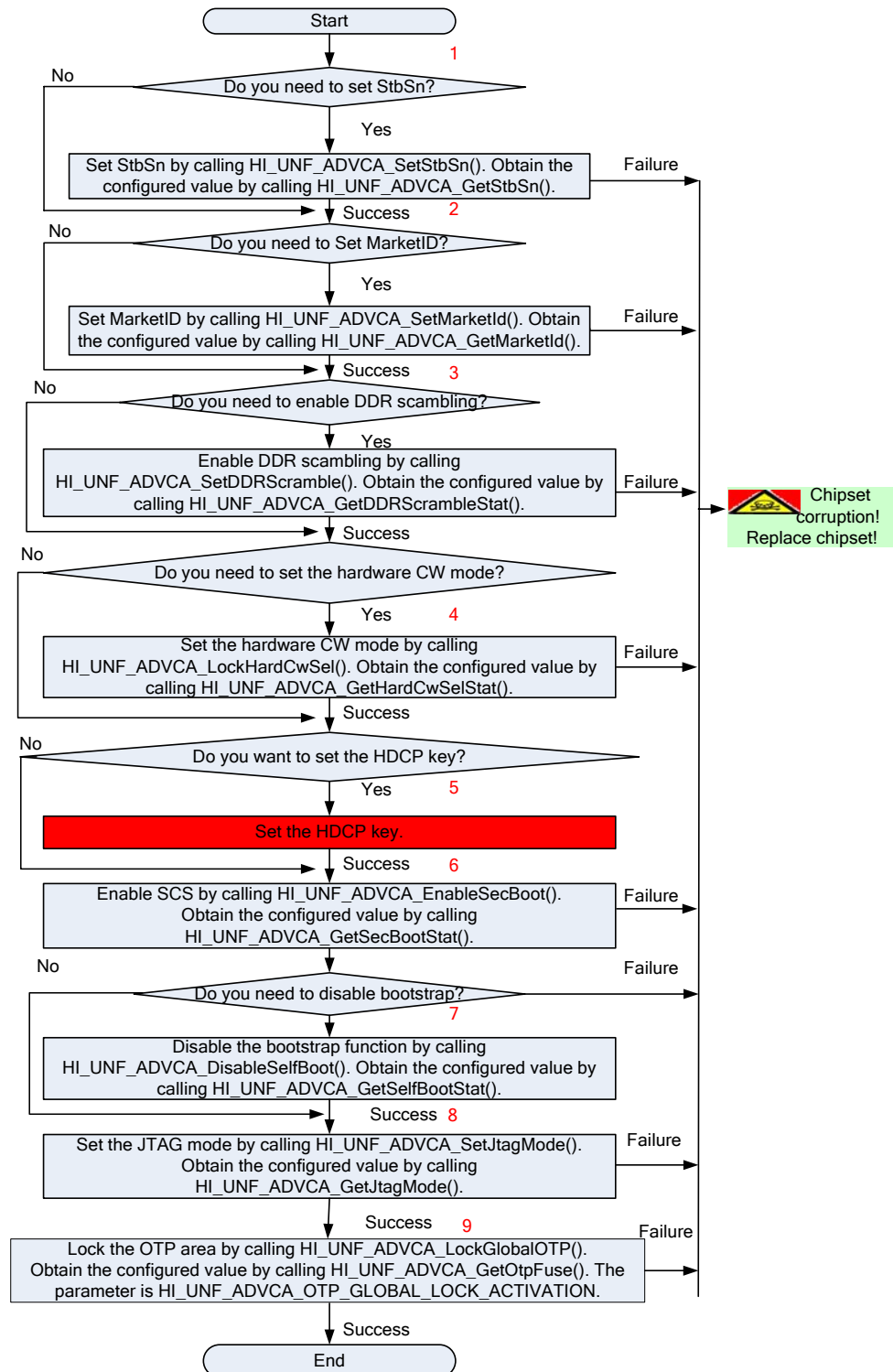
3.2.1 Basic Configurations of Advanced CA Chips

Requirements on the STB security vary according to the CA vendor. Unless otherwise specified, CA vendors require that various embedded security control functions be enabled when the advanced CA chips are used.

This process is described as setting the PVs for an advanced CA chip in this document.

Figure 3-2 shows the process for setting the PVs for an advanced CA chip. The secure boot and JTAG protection mode must be specified. Other options, such as **StbSn**, **MarketID**, **Enable DDRScramble**, **DisableSelfboot**, and **LockHardCw**, can be configured as required by the CA vendor.

Figure 3-2 Setting basic PVs for an advanced CA chip



PVs are stored in the chip OTP area. However, the configuration on most OTP data takes effect only after the board is restarted. During production, if **HI_SUCCESS** is returned by the PV configuration function, the configuration is successful.



Operation Procedures

Perform the following steps:

- Step 1** (Optional) Set the STB SN as required by the CA vendor by calling `HI_UNF_ADVCA_SetStbSn()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetStbSn()`.
- Step 2** (Optional) Set the market ID as required by the CA vendor by calling `HI_UNF_ADVCA_SetMarketId()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetMarketId()`. See section 2.1.3.1 "Setting the Market Segment ID (MSID)" in the *CASignTool Application Notes*.
- Step 3** (Optional) Set the DDR scrambling flag bit as required by the CA vendor by calling `HI_UNF_ADVCA_SetDDRScramble()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetDDRScrambleStat()`.
- Step 4** (Optional) Set the hardware CW mode as required by the CA vendor by calling `HI_UNF_ADVCA_LockHardCwSel()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetHardCwSelStat()`.
- Step 5** (Optional) Burn the HDCP key as required. This operation varies for Hi3716C V100, Hi3716M V300, and Hi3716C V200. For details, see the *HDCP Key User Guide*.
- Step 6** Enable the secure boot SCS mode by calling `HI_UNF_ADVCA_EnableSecBoot()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetSecBootStat()`. This function can be verified only after the STB is restarted.
- Step 7** (Optional) Disable the bootstrapping function as required by the CA vendor by calling `HI_UNF_ADVCA_DisableSelfBoot()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetSelfBootStat()`.
- Step 8** Set the JTAG mode to password mode by calling `HI_UNF_ADVCA_SetJtagMode()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetJtagMode()`.
- Step 9** (Optional) Set the OTP global lock control bit by calling `HI_UNF_ADVCA_LockGlobalOTP()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetOtpFuse()`. The parameter is **HI_UNF_ADVCA_OTP_GLOBAL_LOCK_ACTIVATION**.



WARNING

Once the OTP global lock control bit is configured, the chip OTP area cannot be changed any more. Therefore, this operation is performed only after all other OTP operations (such as burning the PVs, HDCP root key, and STB root key) are performed.

----End

3.3 Usage of Various Advanced CA Chips

3.3.1 Novel Advanced CA Chips

The advanced CA bit field is **Y** for advanced CA chips of Novel.



3.3.1.1 Keys

STB vendors can use the keys described in [Table 3-1](#) as required by Novel.

Table 3-1 Keys for advanced CA chips of Novel

Key	Key Owner	Location	Description
CHIPID	Novel	OTP	Chipset unique ID This value is defined by Novel. It has already been burned into the chip.
CSA2_RootKey	Novel	OTP	Novel secret key This value has already been burned into the chip.
R2R_RootKey	Novel	OTP	Novel secret key This value has already been burned into the chip.
MISC_RootKey	Novel	OTP	Novel secret key This value has already been burned into the chip.
JTAG_Key	Novel	OTP	Novel secret key This value has already been burned into the chip.
ROOT_RSA_PUB_KEY	Novel	OTP	Secure SCS root key This value has already been burned into the chip.
EXT_RSA_PUB_KEY	STB vendor	Flash	Secure SCS external key It is used by BootROM to verify param and check-area (BOOT) of BOOT in the flash memory.
EXT_RSA_PUB_KEY2	STB vendor	Flash	Secure SCS second external key Its public key is stored in the BOOT code. It is used to verify the signature of software images (bootargs, system, loader, and stbid). Its private key is managed by the STB vendor. It can be the same value as that of EXT_RSA_PUB_KEY.
MarketID	Novel	OTP/Flash	This value is defined by Verimatrix.
HDCP_Key	STB vendor	OTP	It is used to protect the HDMI.

3.3.1.2 Stream Decryption

Novel requires that STB vendors use the DVB_RootKey (that is, CSA2.0_RootKey) as the root key of DVB services and use the 3-level key ladder to descramble TSs.

For Hi3716C V200 and later versions, Novel requires that STB vendors use the MISC_RootKey as the backup root key of DVB services and use the 3-level key ladder to descramble TSs.

HiSilicon provides dedicated R2R encryption and decryption interfaces for Novel.



See `sample_ca_novel_dvbplay.c`, `sample_ca_novel_dvbplay_misc.c`, and `ca_novel_crypto.c`.

3.3.1.3 Image Signature

There are two types of image signatures:

- For the BOOT image
The STB vendor needs to obtain the market ID from Novel.
The STB vendor submits the boot source code to Novel for approval, and Novel compiles the mass production boot and signs the BOOT image.
The STB vendor then obtains the signed BOOT image.
- For non-BOOT images, such as the bootargs, kernel, and FS images
Novel has issued the signature solutions for non-BOOT images. The STB vendor needs to negotiate with Novel for the final solution.
This document has provided a reference solution for STB vendors about signing non-BOOT images. For details, see section 2.4 "[Verifying Non-BOOT Software](#)."

3.3.2 Suma Advanced CA Chips

The advanced CA bit field is **S** for advanced CA chips of Suma.

3.3.2.1 Keys

STB vendors can use the keys described in [Table 3-2](#) as required by Suma.

Table 3-2 Keys for advanced CA chips of Suma

Key	Key Owner	Location	Description
CHIPID	Suma	OTP	Chipset unique ID This value is defined by Suma. It has already been burned into the chip.
CSA2_RootKey	Suma	OTP	Suma secret key This value has already been burned into the chip.
R2R_RootKey	Suma	OTP	Suma secret key This value has already been burned into the chip.
JTAG_Key	Suma	OTP	Suma secret key This value has already been burned into the chip.
ROOT_RSA_PUB_KEY	Suma	OTP	Secure SCS root key This value has already been burned into the chip.
EXT_RSA_PUB_KEY	STB vendor	Flash	Secure SCS external key It is used by BootROM to verify param and check-area (BOOT) of BOOT in the flash memory.



Key	Key Owner	Location	Description
EXT_RSA_PUB_KEY2	STB vendor	Flash	Secure SCS second external key Its public key is stored in the BOOT code. It is used to verify the signature of software images (bootargs, system, loader, and stbid). Its private key is managed by the STB vendor. It can be the same value as that of EXT_RSA_PUB_KEY.
HDCP_Key	STB vendor	OTP	It is used to protect the HDMI.

3.3.2.2 Stream Decryption

Suma requires that STB vendors use the DVB_RootKey (that is, CSA2.0_RootKey) as the root key of DVB services. STB vendors must use the 2-level or 3-level key ladder to descramble TSs as required by Suma.

See `sample_ca_suma_tsplay.c` and `sample_ca_crypto.c`.

3.3.2.3 Image Signature

There are two types of image signatures:

- For the BOOT image
The STB vendor can compile and generate images related to secure boot (including **cfg.bin** and **fastboot-burn.bin**) by using the SDK.
The STB vendor then submits the preceding images to Suma for signature to obtain the signed BOOT image.
- For non-BOOT images, such as the bootargs, kernel, and FS images
This document has provided a reference solution for STB vendors about signing non-BOOT images. For details, see section 2.4 "[Verifying Non-BOOT Software](#)."

3.3.3 CTI Advanced CA Chips

The advanced CA bit field is **T** for advanced CA chips of CTI.

3.3.3.1 Keys

STB vendors can use the keys described in [Table 3-3](#) as required by CTI.

Table 3-3 Keys for advanced CA chips of CTI

Key	Key Owner	Location	Description
CHIPID	CTI	OTP	Chipset unique ID This value is defined by CTI. It has already been burned into the chip.



Key	Key Owner	Location	Description
CSA2_RootKey	CTI	OTP	CTI secret key This value has already been burned into the chip.
R2R_RootKey	CTI	OTP	CTI secret key This value has already been burned into the chip.
JTAG_Key	CTI	OTP	CTI secret key This value has already been burned into the chip.
ROOT_RSA_PUB_KEY	CTI	OTP	Secure SCS root key This value has already been burned into the chip.
STB_ROOT_KEY	CTI	OTP	CTI static secret key This value has already been burned into the chip. It is used to encrypt the boot image.
EXT_RSA_PUB_KEY	STB vendor	Flash	Secure SCS external key It is used by BootROM to verify param and check-area (BOOT) of BOOT in the flash memory.
EXT_RSA_PUB_KEY2	STB vendor	Flash	Secure SCS second external key Its public key is stored in the BOOT code. It is used to verify the signature of software images (bootargs, system, loader, and stbid). Its private key is managed by the STB vendor. It can be the same value as that of EXT_RSA_PUB_KEY.
HDCP_Key	STB vendor	OTP	It is used to protect the HDMI.

3.3.3.2 Stream Decryption

CTI requires that STB vendors use the DVB_RootKey (that is, CSA2.0_RootKey) as the root key of DVB services and use the 2-level key ladder to descramble TSs.

See `sample_ca_cti_tsplay.c` and `sample_ca_crypto.c`.

3.3.3.3 Image Signature

There are two types of image signatures:

- For the BOOT image
The STB vendor can compile and generate images related to secure boot (including **cfg.bin** and **fastboot-burn.bin**) by using the SDK.
The STB vendor then submits the preceding images to CTI for signature to obtain the signed BOOT image.
- For non-BOOT images, such as the bootargs, kernel, and FS images
This document has provided a reference solution for STB vendors about signing non-BOOT images. For details, see section 2.4 "[Verifying Non-BOOT Software](#)."



3.3.4 Verimatrix Advanced CA Chips

The advanced CA bit field is **M** for advanced CA chips of Verimatrix.

3.3.4.1 Verimatrix Security Level Classification

Verimatrix supports two security certification levels, which are described as follows:

- Standard STB security

This level is used in the current market. The STB must meet the following requirements:

- Provides secure boot solutions implemented by using the hardware. The algorithm is RSA2048-HASH256.
- Supports secure upgrade solutions, in which programs are upgraded only after the upgrade data is verified.
- Supports a unique chip ID, which is at least 32 bits.
- Supports memory encryption and DDR scrambling.
- Supports the secure JTAG interface.

STB vendors can implement the preceding features by using Verimatrix chips with standard security. For this type of chips, STB vendors need to set the keys (including the chip ID and keys such as the Root_RSA_Public_Key) and set the chip PVs during mass production.

- Advanced STB security

This security level is supported by Hi3716C V200 and later versions. The STB must meet the following requirements:

- Supports all features in the standard STB security level.
- Supports the DVB key ladder decryption solutions for ciphertext CWs.

STB vendors can implement the preceding features by using Verimatrix chips with advanced security. For this type of chips, the Verimatrix keys (including the chip ID and keys such as the Root_RSA_Public_Key) have been written to the chip before delivery. STB vendors need only to set the other PVs.

3.3.4.2 Verimatrix Standard STB Chips

Keys

STB vendors can use the keys described in [Table 3-4](#) as required by Verimatrix.

Table 3-4 Keys for standard STB chips of Verimatrix

Key	Key Owner	Location	Description
CHIPID	STB vendor	OTP	Chipset unique ID, burnt by the STB vendor
R2R_RootKey	STB vendor	OTP	R2Rroot key This key has 16 bytes. The first 8 bytes cannot be the same as the last 8 bytes of the key.
JTAG_Key	STB vendor	OTP	JTAG password, burnt by the STB vendor



Key	Key Owner	Location	Description
ROOT_RSA_PUB_KEY	Verimatrix	OTP	Secure SCS root key, requested from Verimatrix, burnt by the STB vendor It is used only by BootROM to verify the key area of boot in the flash memory.
EXT_RSA_PUB_KEY	STB vendor	Flash	Secure SCS external key It is used by BootROM to verify param and check-area (BOOT) of BOOT in the flash memory.
EXT_RSA_PUB_KEY2	STB vendor	Flash	Secure SCS second external key Its public key is stored in the BOOT code. It is used to verify the signature of software images (bootargs, system, loader, and stbid). Its private key is managed by the STB vendor. It can be the same value as that of EXT_RSA_PUB_KEY.
HDCP_Key	STB vendor	OTP	It is used to protect the HDMI.

Process for Signing the BOOT Image

The ROOT_RSA_PUB_KEY required for secure boot is not written to the HiSilicon Verimatrix advanced CA chips by default before delivery. Therefore, it must be written to the chips before the secure boot function is enabled.

The Finalboot for secure boot contains a 2-level signature solution. Verimatrix maintains only one group of RSA keys (which can be ROOT_RSA_PUB_KEY or EXT_RSA_PUB_KEY) and provides the image signature.

The following takes the ROOT_RSA_PUB_KEY as an example. In this case, customers request ROOT_RSA_PUB_KEY directly from Verimatrix, and the external RSA public key stored in the flash memory can be generated and maintained by the STB vendor.

The BOOT image is signed and used as follows:

- Step 1** Generate a pair of external RSA keys by using the CASignTool. For details, see the *CASignTool Application Notes*.
- Step 2** Sign the boot by using the CASignTool. For details, see section 2.3 in the *CASignTool Application Notes*. Submit the generated **KeyArea.bin** to Verimatrix for signature.
- Step 3** Verimatrix returns the signature file for **KeyArea.bin** (assume that the image is **KeyArea.bin.sign**). The signature contains a ROOT_RSA_PUB_KEY.
- Step 4** Merge multiple BOOT images into **FinalBoot.bin** by using the **Merge Signed-BootImage** function of the CASignTool.
- Step 5** Write **FinalBoot.bin** to the flash memory by using the download tool. Encrypt the boot because Verimatrix requires that the boot be encrypted before being stored to the flash memory. For details about how to encrypt the boot, see **sample_product_encrypt_boot.c**.
- Step 6** Write the ROOT_RSA_PUB_KEY to the chip OTP, and set the key by calling HI_UNF_ADVCA_SetRSAKey(). For details, see **sample_ca_writeRSAkey.c**.



Step 7 Enable the secure boot mode. For details, see **sample_ca_opensecboot.c**.

----End

Decryption Requirements

Verimatrix DRM is also called Verimatrix IPTV.

The program map table (PMT) in encrypted Verimatrix streams contains the CA system descriptor, which is defined as follows:

- If the CA system descriptor does not contain private data segments, the streams are scrambled by using the CSA 2.0 algorithm.
- If the CA system descriptor contains private data segments, the first byte indicates the encryption algorithm. STB vendors can obtain more information from the CA vendor.

Besides CSA 2.0, Verimatrix DRM also uses the AES-ECB and AES-CBC algorithms to encrypt data streams. All TSs are decrypted by the cipher module before being transferred to the chip Demux.

To decrypt the specific data in the memory by using the AES algorithm, perform the following steps (for details, see **sample_ca_cipher.c**):

Step 1 Initialize the cipher module by calling **HI_UNF_CIPHER_Open**.

Step 2 Obtain a cipher handle by calling **HI_UNF_CIPHER_CreateHandle**.

Step 3 Set encryption/decryption options by calling **HI_UNF_CIPHER_ConfigHandle**.

- If you want to use AES-ECB, set **CipherCtrl.enAlg** to **HI_UNF_CIPHER_ALG_AES** and **CipherCtrl.enWorkMode** to **HI_UNF_CIPHER_WORK_MODE_ECB**.
- If you want to use AES-CBC, set **CipherCtrl.enAlg** to **HI_UNF_CIPHER_ALG_AES** and **CipherCtrl.enWorkMode** to **HI_UNF_CIPHER_WORK_MODE_CBC**.
- If the IV vector needs to be updated, set **stChangeFlags.bit1IV** to **1**. If **stChangeFlags.bit1IV** is set to **0**, the IV is the previous calculation result when the cipher module is used.

Step 4 Allocate the DDR physical memory by calling **HI_MMZ_New()**, and copy the data to be decrypted to the physical memory.

Step 5 Decrypt data in the physical memory by calling **HI_UNF_CIPHER_Decrypt**. The return value is the decrypted data.

Step 6 Release the cipher handle by calling **HI_UNF_CIPHER_DestroyHandle**.

Step 7 Stop the cipher device by calling **HI_UNF_CIPHER_Close**.

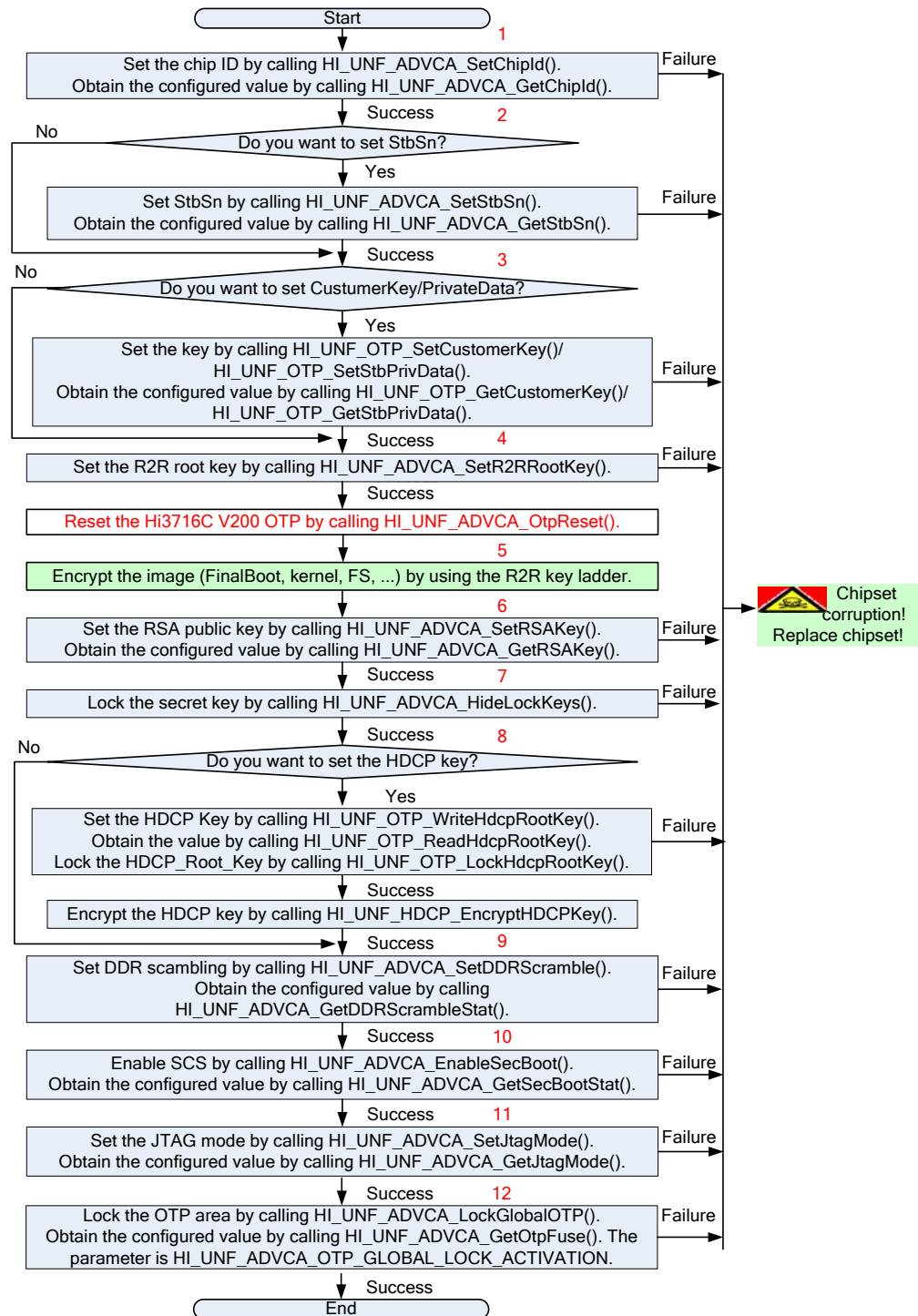
----End

PV Settings

The HiSilicon chips that support the Verimatrix standard STB security feature include the following: Hi3716M V300, Hi3716C V200, Hi3716M V400, Hi3716M V310, Hi3719C V100, Hi3719M V100, Hi3796C V100, Hi3798C V100, Hi3798M V100, Hi3796M V100, Hi3716M V420, Hi3716M V410, and Hi3798C V200.

As required by Verimatrix, if STBs with Verimatrix standard STB CA chips are to be certified or produced, PVs must be correctly set according to the process shown in [Figure 3-3](#).

Figure 3-3 Setting PVs for Verimatrix standard STB chips (Hi3716M V300/Hi3716C V200)



The steps in [Figure 3-3](#) must be performed in sequence, but do not need to be performed at a time.

PVs are stored in the chip OTP. However, the configuration on some OTP data takes effect only after the board is restarted. If **HI_SUCCESS** is returned by the PV configuration function, the configuration is successful.



To set PVs for Verimatrix standard STB CA chips, perform the following steps:

Step 1 Set the chip ID by calling `HI_UNF_ADVCA_SetChipId()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetChipId()`.



NOTE

- The chip ID is the unique identifier of a chip.
- The ID has four bytes. It is not burnt by default before the delivery of Verimatrix standard STB CA chips.
- If the chip ID is set to **0x12345678**, the chip ID read by calling `HI_UNF_ADVCA_GetChipId()` is **0x78563412**.

Step 2 (Optional) Set the STB SN by calling `HI_UNF_ADVCA_SetStbSn()` (if the STB vendor needs to save private data by using it).

StbSn is the extension of the chip ID. If you want to set a chip ID with more than 32 bits, store the lower 32-bit data to the chip ID and the rest data to **StbSn**. You can obtain the configured value by calling `HI_UNF_ADVCA_GetStbSn()`.

Step 3 (Optional) Set **CustomerKey/StbPrivateData** by calling `HI_UNF_OTP_SetCustomerKey()/HI_UNF_OTP_SetStbPrivData()` (if the STB vendor needs to save private data by using **CustomerKey/StbPrivateData**).

CustomerKey and **StbPrivateData** each can store 16-byte private user data. You can obtain the configured value by calling `HI_UNF_OTP_GetCustomerKey()` or `HI_UNF_OTP_GetStbPrivData()`.

Step 4 Set the `R2R_Root_Key`.

Verimatrix recommends that STB images stored in the flash memory be encrypted. STB vendors can encrypt non-BOOT software by using the R2R key ladder (`R2R_Root_Key` as the root key) and then store the encrypted software to the flash memory.



NOTE

- After the `R2R_Root_Key` is configured, Hi3716C V200 and later versions cannot read the written key. In this case, you can restart the system or call `HI_UNF_ADVCA_OtpReset()` for the key to take effect.
- The `R2R_Root_Key` consists of 16 bytes. Data in the first eight bytes cannot be equal to that in the last eight bytes.

Step 5 Encrypt the boot, kernel, and file system images after the `R2R_Root_Key` takes effect.

For Hi3716C V200, the BOOT image must be signed before being encrypted. Create the signed BOOT image (**FinalBoot.bin**) by using the SignTool, load **FinalBoot.bin** to the flash memory, and encrypt the code in the Finalboot area by using the AES algorithm and the cipher. Finally, encrypt the key for encrypting **Finalboot.bin** by calling `HI_UNF_ADVCA_EncryptSWPK()`. For details, see `ca_blpk.c`.

To verify encryption for the kernel or file system image, run **common_verify_encryptimage** under boot.

Step 6 Set the `ROOT_RSA_PUB_KEY` by calling `HI_UNF_ADVCA_SetRSAKey()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetRSAKey()`.

Step 7 Lock the keys such as the `R2R_Root_Key` and `ROOT_RSA_PUB_KEY` by calling `HI_UNF_ADVCA_HideLockKeys()`.

Step 8 (Optional) Burn the HDCP key.



- Step 9** Set DDR scrambling by calling `HI_UNF_ADVCA_SetDDRScramble()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetDDRScrambleStat()`.
- Step 10** Enable the secure boot SCS mode by calling `HI_UNF_ADVCA_EnableSecBoot()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetSecBootStat()`. This function can be verified only after the STB is restarted.
- Step 11** Set the JTAG mode to password mode by calling `HI_UNF_ADVCA_SetJtagMode()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetJtagMode()`.
- Step 12** (Optional) Set the OTP global lock control bit by calling `HI_UNF_ADVCA_LockGlobalOTP()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetOtpFuse()`. The parameter is **HI_UNF_ADVCA_OTP_GLOBAL_LOCK_ACTIVATION**.



WARNING

Once the OTP global lock control bit is configured, the chip OTP area cannot be changed any more. Therefore, this operation is performed only after all other OTP operations (such as burning the PVs, HDCP root key, and STB root key) are performed.

----End

3.3.4.3 Verimatrix Advanced STB Chips

For details about the development of Verimatrix advanced CA STBs, see the *Supplement to the Advanced CA Development Guide for Verimatrix*.

3.3.5 Nagra Advanced CA Chips

The advanced CA bit field is **R** for advanced CA chips of Nagra.

For details about the development of Nagra advanced CA STBs, see the *CASignTool Application Notes* and *Nagra Advanced Security CA BootLoader Development Guide*.

3.3.6 Conax Advanced CA Chips

The advanced CA bit field is **C** for advanced CA chips of Conax.

For details about the development of Conax advanced CA STBs, see the *Supplement to Advanced Secure CA Development Guide for Conax*.

3.3.7 Irdeto Advanced CA Chips

The advanced CA bit field is **I** for advanced CA chips of Irdeto.

HiSilicon and Irdeto have worked together to provide the *Supplement to Advanced Secure CA Development Guide for Irdeto*, which describes how to request the signature of advanced CA boot from Irdeto. STB vendors can obtain the document from Irdeto.

3.3.8 Panaccess Advanced CA Chips

HiSilicon is cooperating with Panaccess and supports the Panaccess advanced CA chip on the Hi3716M V310 platform.



3.3.8.1 Keys for the Panaccess Advanced CA Chips

STB vendors can use the keys described in [Table 3-5](#) as required by Panaccess.

Table 3-5 Keys for advanced CA chips of Panaccess

Key	Key Owner	Location	Description
CHIPID	Panaccess	OTP	Chip unique ID. This value is defined by Panaccess. It has already been burned into the chip. It is an 8-byte unique value.
CSA2_RootKey	Panaccess	OTP	Panaccess secret key This value has already been burned into the chip.
R2R_RootKey	Panaccess	OTP	Panaccess secret key This value has already been burned into the chip.
JTAG_Key	Panaccess	OTP	Panaccess secret key This value has already been burned into the chip.
STB_ROOT_KEY	Panaccess	OTP	Panaccess static secret key This value has already been burned into the chip. It is used to encrypt the boot image.
ROOT_RSA_PUB_KEY	Panaccess	OTP	Security SCS root key This value has already been burned into the chip.
EXT_RSA_PUB_KEY	STB vendor	Flash	Secure SCS external key It is used by BootROM to verify param and check-area (BOOT) of BOOT in the flash memory.
EXT_RSA_PUB_KEY2	STB vendor	Flash	Secure SCS second external key Its public key is stored in the BOOT code. It is used to verify the signature of software images (bootargs, system, loader, and stbid). Its private key is managed by the STB vendor. It can be the same value as that of EXT_RSA_PUB_KEY.
HDCP_Key	STB vendor	OTP	It is used to protect HDMI.

3.3.8.2 Decrypting Streams

Panaccess requires that STB vendors use the DVB_RootKey (that is, CSA2.0_RootKey) as the root key of DVB services and use the 3-level key ladder to decrypt TSs.

For details, see `sample_ca_panaccess_tsplay.c` and `sample_ca_crypto.c`.



3.3.8.3 Signing Images

As required by Panaccess, the executable images stored in the flash memory of the STB must be encrypted. According to this requirement, HiSilicon provides the following signature solution:

There are two types of image signatures:

- For the BOOT image

The STB vendor can compile and generate images related to secure boot (including **cfg.bin** and **fastboot-burn.bin**) by using the SDK.

The STB vendor then submits the preceding images to Panaccess for signature to obtain the signed BOOT image.

Note that the signed boot image returned by Panaccess is the encrypted binary image with signature.

- For non-BOOT images, such as the bootargs, kernel, and FS images

This document has provided a reference solution for STB vendors about signing non-BOOT images. For details, see section 2.4 "[Verifying Non-BOOT Software](#)." STB vendors can sign non-BOOT images by using the special CA signature mode.

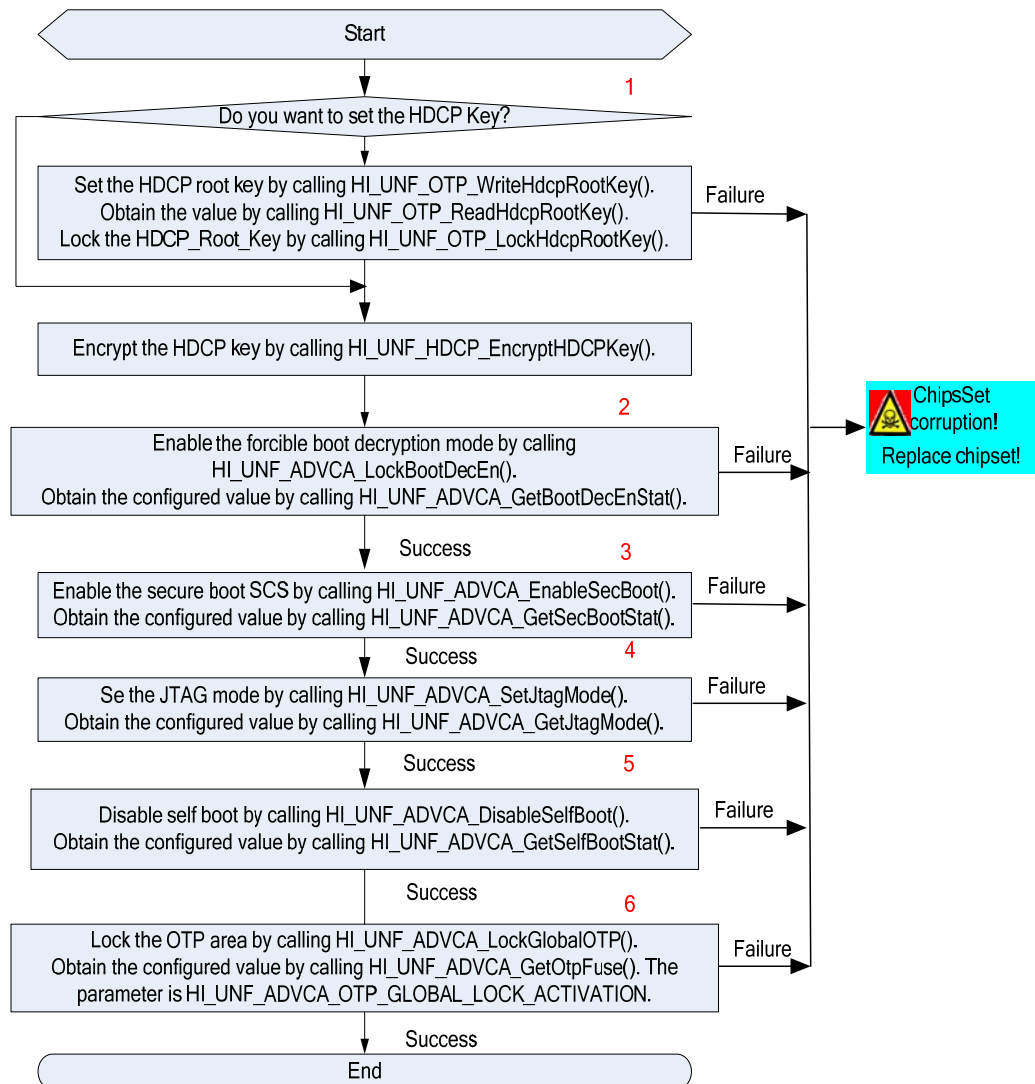
3.3.8.4 Setting PVs for Panaccess Advanced CA Chips

Hi3716M V310 supports the Panaccess advanced CA features.

As required by Panaccess, the Panaccess keys (including the chip ID and ROOT_RSA_PUB_KEY) have been embedded in the advanced CA chips before delivery.

If STBs with Panaccess advanced CA chips are to be certified or produced, PVs must be correctly set according to the process shown in [Figure 3-4](#).

Figure 3-4 Setting PVs for Panaccess advanced CA chip Hi3716M V310



The steps in [Figure 3-4](#) must be performed in sequence, but do not need to be performed at a time.

PVs are stored in the chip OTP. However, the configuration on some OTP data takes effect only after the board is restarted. If **HI_SUCCESS** is returned by the PV configuration function, the configuration is successful.

To set PVs for Panaccess advanced CA chips, perform the following steps:

- Step 1** (Optional) Burn the HDCP key as required. For details, see the *HDCP Key User Guide*.
- Step 2** Set the forcible boot decryption function by calling `HI_UNF_ADVCA_LockBootDecEn()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetBootDecEnStat()`.
- Step 3** Enable the secure boot SCS mode by calling `HI_UNF_ADVCA_EnableSecBoot()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetSecBootStat()`. This function can be verified only after the STB is restarted.



- Step 4** Disable the bootstrapping upgrade function by calling `HI_UNF_ADVCA_DisableSelfBoot()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetSelfBootStat()`.
- Step 5** Set the JTAG mode to password mode by calling `HI_UNF_ADVCA_SetJtagMode()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetJtagMode()`.
- Step 6** (Optional) Set the OTP global lock control bit by calling `HI_UNF_ADVCA_LockGlobalOTP()`. You can obtain the configured value by calling `HI_UNF_ADVCA_GetOtpFuse()`. The parameter is **HI_UNF_ADVCA_OTP_GLOBAL_LOCK_ACTIVATION**.



WARNING

Once the OTP global lock control bit is configured, the chip OTP area cannot be changed any more. Therefore, this operation is performed only after all other OTP operations (such as burning the PVs and HDCP root key) are performed.

-----End

3.3.9 HiSilicon Non-CA Chips

In addition to the preceding multiple advanced CA chips, HiSilicon chips also include non-CA chips and HiSilicon common CA chips (chips with the H mark in the past). Starting from 2015, these two types of HiSilicon chips are not distinguished upon delivery to provide unified products to STB vendors and allow STB vendors to burn them into non-CA chips or HiSilicon common CA chips based on market demand.

The following chips can be burnt after delivery: Hi3796M V100, Hi3798M V100, Hi3716M V410, Hi3716M V420, and Hi3798C V200.

This section describes how to burn the delivered chips into non-CA chips or HiSilicon common CA chips and the typical process for mass production.

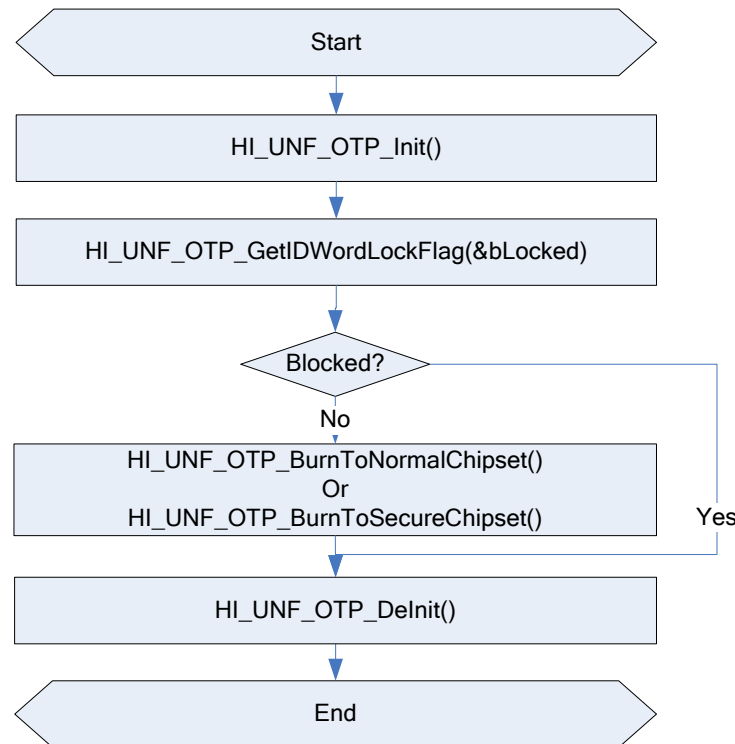
3.3.9.1 Burning Chips

The SDK provides the following UNF interfaces:

- `HI_S32 HI_UNF_OTP_BurnToNormalChipset (HI_VOID)`: Burns a common chip into a non-CA chip.
- `HI_S32 HI_UNF_OTP_BurnToSecureChipset (HI_VOID)`: Burns a common chip into a HiSilicon common CA chip.
- `HI_S32 HI_UNF_OTP_GetIDWordLockFlag (HI_BOOL *pbLockFlag)`: Obtains the chip burning lock status.

Figure 3-5 shows the typical lock process.

Figure 3-5 Typical lock process



NOTE

At the development and debugging phase, it is recommended that the board be restarted after burning to avoid conflicts with other burning operations. At the mass production phase, in general, the board does not need to be restarted.

The preceding interfaces are provided under the boot and the system. You can flexibly choose whether to burn the chips under the boot or in the factory test software as required. Burning the chips in the factory test software is recommended for the following reasons:

- At the development and debugging phase, if the chip is burnt automatically under the boot, the chip is burnt as soon as the board starts, which is inconvenient for managing the CA and non-CA boards.
- When the board starts at the first time, the chip is a non-CA chip. Therefore, the unsigned advanced CA boot needs to be burnt to the flash memory and after the factory test is completed, the signed advanced CA boot image needs to be burnt. If the chip is automatically burnt into a HiSilicon common CA chip when the boot starts, the boot will not match with the chip when the board restarts before the Finalboot is burnt. The board cannot start and the maintenance is complicated.

The preceding issues do not occur when the burning is performed in the factory test software.



NOTE

Hi3798C V200 uses the unified boot solution, and therefore the case that the boot does not match with the chip will not occur. For details, see section [3.3.9.7 "Mass Production of HiSilicon Common CA Chips for Hi3798C V200."](#)

Hi3796M V100 and Hi3798M V100 are automatically burnt under the boot by default (if the non-advanced CA boot is compiled, the chip is burnt into a non-CA chip; if the advanced CA boot is compiled, the chip is burnt into a HiSilicon common CA chip). To disable the



automatic burning function, comment out "HI_OTP_LockIdWord()" in **Code/source/boot/product/main.c**.

Hi3716M V410, Hi3716M V420, and Hi3798C V200 are not burnt under the boot by default. You can burn them in the factory test software.

After the burning is successful, the following message about the CPU type is displayed when the non-CA chip starts:

```
CPU: Hi3796Mv100
```

After the chip is burnt into a HiSilicon common advanced CA chip and then starts, the following message about the CPU type is displayed:

```
CPU: Hi3796Mv100(CA)
```

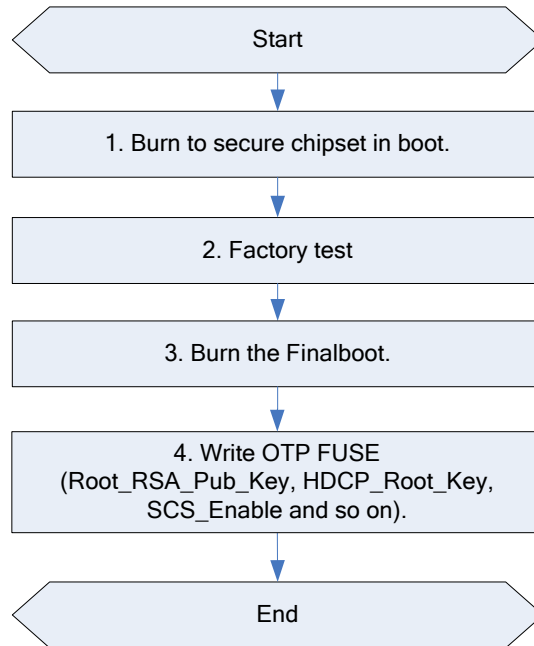
3.3.9.2 Mass Production of Non-CA Chips for Hi3796M V100 and Hi3798M V100

Hi3796M V100 and Hi3798M V100 are automatically burnt by default when the boot is enabled. If the non-advanced CA boot is compiled, the chips are burnt into non-CA chips, which does not affect the original manufacture process.

3.3.9.3 Mass Production of HiSilicon Common CA Chips for Hi3796M V100 and Hi3798M V100

Hi3796M V100 and Hi3798M V100 are automatically burnt by default when the boot is enabled. If the advanced CA boot is compiled, the chips are burnt into HiSilicon common CA chips. When the board starts at the first time, the chips are non-CA chips. Therefore, the unsigned advanced CA boot needs to be burnt to the flash memory. After the boot starts, chips are automatically burnt into HiSilicon common CA chips. When the factory test is finished, burn Finalboot and then burn OTP FUSE that contains Root_RSA_Pub_Key enable and secure boot enable. After the board restarts, the environment is a secure environment.

[Figure 3-6](#) shows the typical mass production process.

Figure 3-6 Typical mass production process

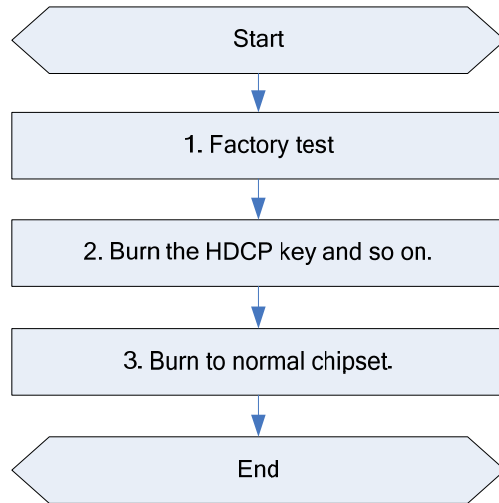
The typical mass production process is as follows:

- Step 1** Enable the unsigned advanced CA boot. The chips are automatically burnt into HiSilicon common CA chips.
- Step 2** Perform the factory test, including testing basic functions and input/output interfaces.
- Step 3** Burn the Finalboot.
- Step 4** Burn the root key, HDCP key, and PV values. For details, see section [3.3.10 "HiSilicon Common CA Chips."](#)

----End

3.3.9.4 Mass Production of Non-CA Chips for Hi3716M V410 and Hi3716M V420

Different from Hi3796M V100 and Hi3798M V100, Hi3716M V410 and Hi3716M V420 are not automatically burnt under the boot and need to be burnt in the factory test software by vendors. [Figure 3-7](#) shows the typical mass production process.

Figure 3-7 Typical mass production process

The typical mass production process is as follows:

Step 1 Perform the factory test, including testing basic functions and input/output interfaces.

Step 2 Burn the HDCP key as required.

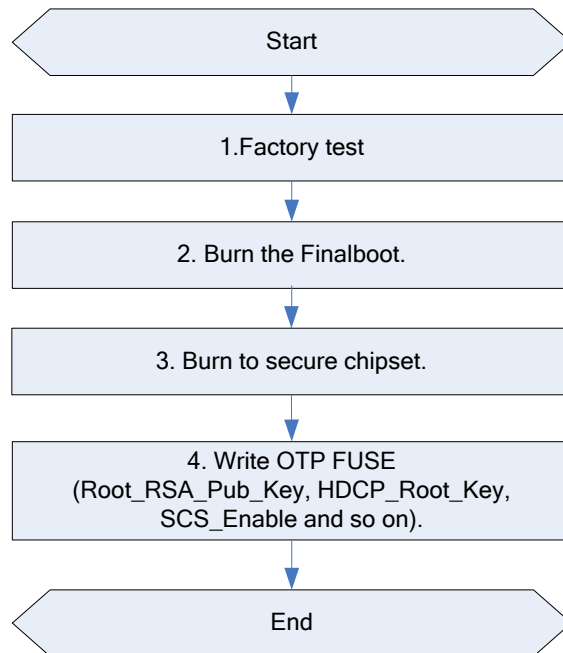
Step 3 Burn the chips into non-CA chips.

----End

3.3.9.5 Mass Production of HiSilicon Common CA Chips for Hi3716M V410 and Hi3716M V420

Different from Hi3796M V100 and Hi3798M V100, Hi3716M V410 and Hi3716M V420 are not automatically burnt under the boot and need to be burnt in the factory test software by vendors. When the board starts at the first time, the chips are non-CA chips. Therefore, the unsigned advanced CA boot needs to be burnt to the flash memory. When the factory test is finished, burn Finalboot and then burn OTP FUSE that contains Root_RSA_Pub_Key enable and secure boot enable. After the board restarts, the environment is a secure environment.

Figure 3-8 shows the typical mass production process.

Figure 3-8 Typical mass production process

The typical mass production process is as follows:

- Step 1** Perform the factory test, including testing basic functions and input/output interfaces.
- Step 2** Burn the Finalboot.
- Step 3** Burn the chip into a HiSilicon common CA chip.
- Step 4** Burn the root key, HDCP key, and PV values. For details, see section [3.3.10 "HiSilicon Common CA Chips."](#)

----End

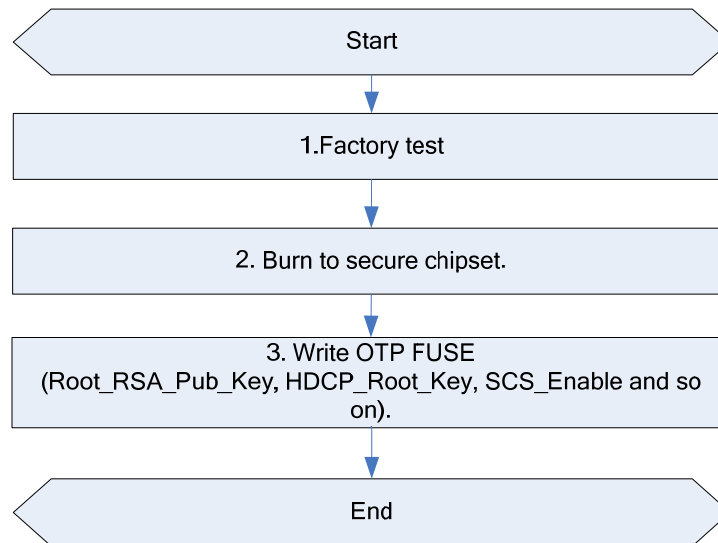
3.3.9.6 Mass Production of Non-CA Chips for Hi3798C V200

The mass production process of non-CA chips for Hi3798C V200 is the same as that for Hi3716M V410 and Hi3716M V420.

3.3.9.7 Mass Production of HiSilicon Common CA Chips for Hi3798C V200

The Hi3798C V200 chips use the unified boot design, that is, the non-advanced CA boot, unsigned advanced CA boot, and the signed advanced CA Finalboot are compatible. When the unsigned advanced CA boot and the signed advanced CA Finalboot are burnt to the non-advanced CA chips, the board can restart. Therefore, during the flash burning, the signed advanced CA Finalboot can be directly burnt and other boots do not need to be burnt in the mass production process.

[Figure 3-9](#) shows the typical mass production process.

Figure 3-9 Typical mass production process

The typical mass production process is as follows:

- Step 1** Perform the factory test, including testing basic functions and input/output interfaces.
- Step 2** Burn the chip into a HiSilicon common CA chip.
- Step 3** Burn the root key, HDCP key, and PV values. For details, see section [3.3.10 "HiSilicon Common CA Chips."](#)

----End

3.3.10 HiSilicon Common CA Chips

HiSilicon common CA chips refer to the previous chips with the H mark. STB vendors can use this type of chips to realize some security functions, such as secure boot and copy protection.

The H-mark chips for Hi3716C V200, Hi3719C V100, Hi3716M V310, and Hi3110E V500 are available.

The H-mark chips for Hi3796M V100, Hi3798M V100, Hi3716M V410, Hi3716M V420, and Hi3798C V200 are not provided. Vendors can burn the delivered chips into HiSilicon common CA chips as required. For details, see section [3.3.9 "HiSilicon Non-CA Chips."](#) The PVs for the burnt HiSilicon common CA chips also need to be set according to the description in this section.

3.3.10.1 STB Vendor Secret Keys for HiSilicon Common CA Chips

To reach the requirement of secure boot, STB vendors need to configure the keys as shown in [Table 3-6](#).



Table 3-6 Keys that need to be configured by the STB vendor

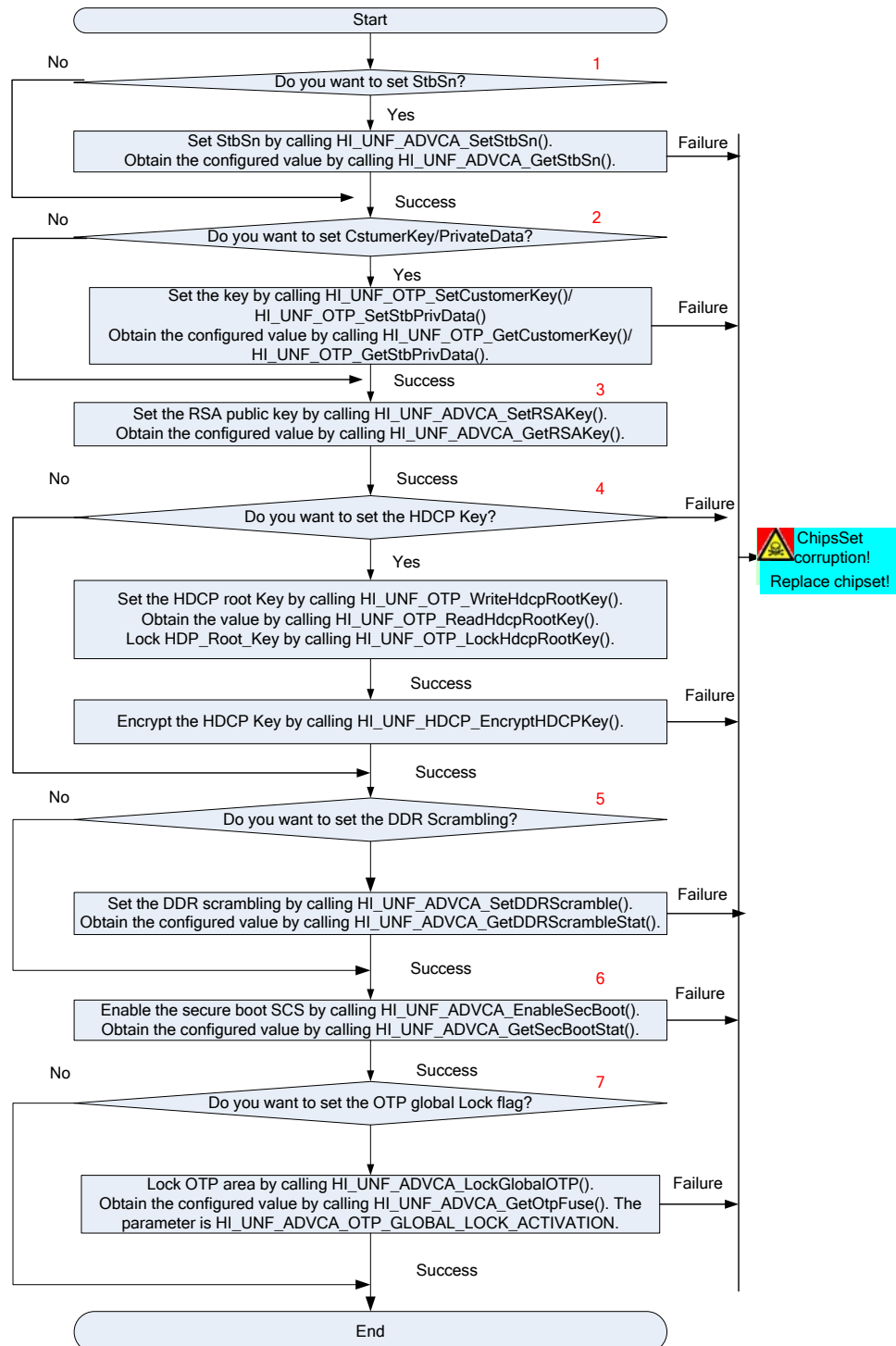
Key	Key Owner	Location	Description
Root_RSA_Pub_Key	STB vendor	OTP	Secure SCS root key, burnt by the STB vendor Used only by BootRom to verify Key-Area of the Bootloader in the flash
Ext_RSA_Pub_Key	STB vendor	Flash	Secure SCS external key Use only by BootRom to verify param and check-area (boot) of the Bootloader in the flash
Ext_RSA_Pub_Key2	STB vendor	Flash	Secure SCS second external key Its public key is stored in Finalboot. It is used to verify the signature of software images (bootargs, system, loader, and stbid) Its private key is managed by the STB vendor. The value can be the same as that of Ext_RSA_Pub_Key.
HDCP_Root_Key	STB vendor	OTP	It is used to protect HDCP key of HDMI.
CustomerKey	STB vendor	OTP	STB private data, burnt by the STB vendor (Optional)
StbPrivateData	STB vendor	OTP	STB private data, burnt by the STB vendor (Optional)

3.3.10.2 PV Configuration

Figure 3-10 shows the process for setting PVs for HiSilicon common CA chips to enable the security features.



Figure 3-10 Process for setting PVs for Hi3716C V200





The process in [Figure 3-10](#) must be performed in sequence, but do not need to be performed at a time.

PVs are stored in the chip OTP. However, the configuration on some OTP data takes effect only after the board is restarted. If **HI_SUCCESS** is returned by the PV configuration function, the configuration is successful.

To set PVs for HiSilicon common CA chips, perform the following steps:

Step 1 Set **StbSn**.

- Set this parameter by calling `HI_UNF_ADVCA_SetStbSn()`.
- Obtain the configured value by calling `HI_UNF_ADVCA_GetStbSn()`.

Step 2 Set **CustomerKey** or **StbPrivateData**. The following interfaces are used to store private user data. **CustomerKey** and **StbPrivateData** each can store 16-byte private user data.

- Set the two parameters by calling `HI_UNF_OTP_SetCustomerKey()` and `HI_UNF_OTP_SetStbPrivData()` respectively.
- Obtain the configured values by calling `HI_UNF_OTP_GetCustomerKey()` and `HI_UNF_OTP_GetStbPrivData()`.

Step 3 Set the root RSA public key.

- Set the key by calling `HI_UNF_ADVCA_SetRSAKey()`.
- Lock the RSA key by calling `HI_UNF_ADVCA_ConfigLockFlag()`.
- Obtain the configured value by calling `HI_UNF_ADVCA_GetRSAKey()`.

Step 4 (Optional) Burn the HDCP root key. For Hi3716C V200 and later chips, usage of HDCP has been changed. For details, see the *HDCP Key User Guide*.

- Set the HDCP root key by calling `HI_UNF_OTP_WriteHdcpRootKey ()`.
- Obtain the configured value by calling `HI_UNF_OTP_ReadHdcpRootKey ()`.
- Lock the root key by calling `HI_UNF_OTP_LockHdcpRootKey()`.
- Encrypt the HDCP key using the HDCP root key by calling `HI_UNF_HDCP_EncryptHDCPKey()`.

Step 5 Set the DDR scrambling function.

- Set the scrambling function by calling `HI_UNF_ADVCA_SetDDRScramble()`.
- Obtain the configured value by calling `HI_UNF_ADVCA_GetDDRScrambleStat()`.

Step 6 Enable the secure boot SCS mode.

- Enable this mode by calling `HI_UNF_ADVCA_EnableSecBootEx()`.
- Obtain the configured value by calling `HI_UNF_ADVCA_GetSecBootStat()`.

After this control bit is configured, restart the STB to verify this function before performing the following operations.

Step 7 (Optional) Set the OTP global lock control bit by calling `HI_UNF_ADVCA_LockGlobalOTP ()`. Obtain the configured value by calling `HI_UNF_ADVCA_GetOtpFuse()`. The parameter is **HI_UNF_ADVCA_OTP_GLOBAL_LOCK_ACTIVATION**.



CAUTION

Once the OTP global lock control bit is configured, the chip OTP area cannot be changed any more. Therefore, this operation is performed only after all other OTP operations (such as burning the PVs, HDCP root key, and STB root key) are performed.

----End

3.3.10.3 CAS Vendor Secret Keys for HiSilicon Common CA Chips

To enable the CAS function of the HiSilicon common CA chips, configure the CAS private data as shown in [Table 3-7](#).

Table 3-7 CAS vendor private data

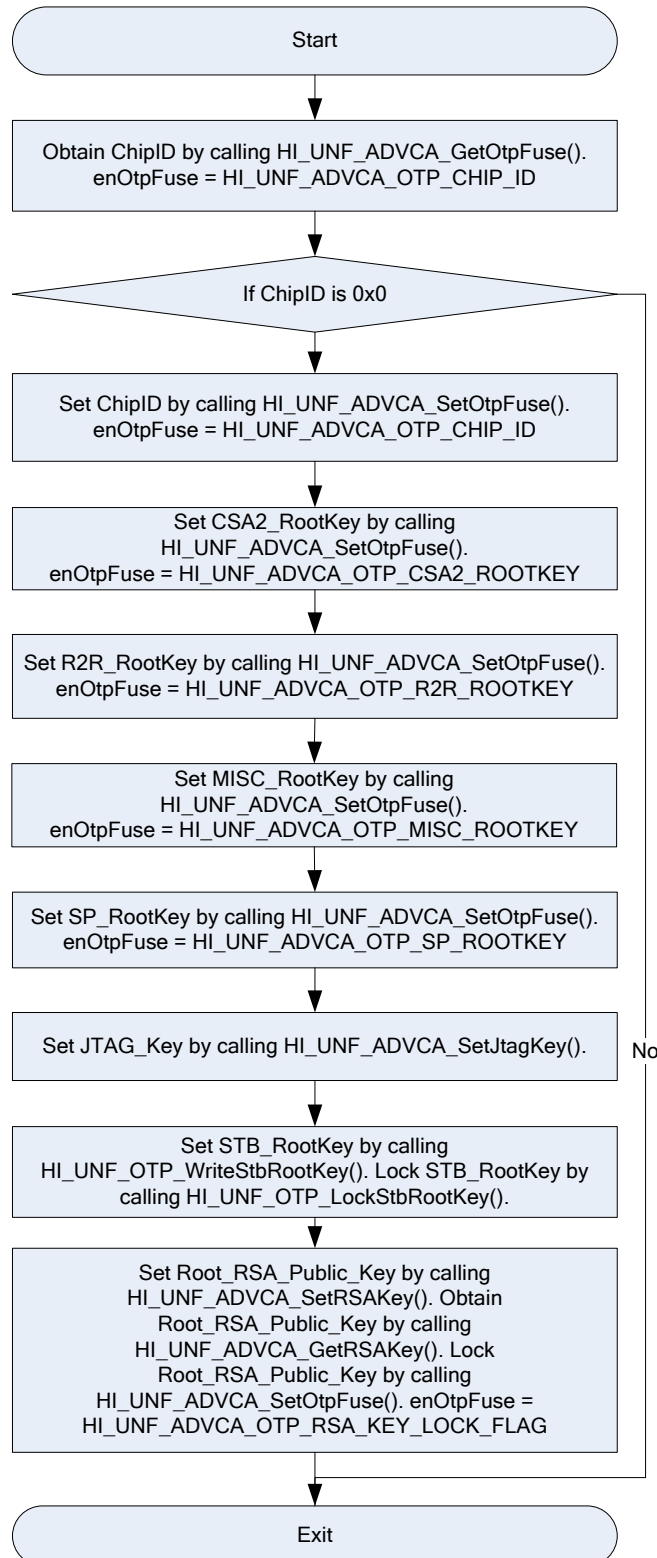
Key	Key Owner	Location	Description
CHIPID	CA	OTP	Chipset unique ID
Root_RSA_Pub_Key	CA	OTP	Secure SCS root key, burnt by the STB vendor Used only by BootRom to verify Key-Area of the Bootloader in the flash
CSA2_RootKey	CA	OTP	Root key for CSA2 key ladder
CSA3_RootKey	CA	OTP	Root key for CSA3 key ladder
R2R_RootKey	CA	OTP	Root key for R2R key ladder
SP_RootKey	CA	OTP	Root key for SP key ladder
MISC_RootKey	CA	OTP	Root key for MISC key ladder
JTAG_Key	CA	OTP	JTAG password
STB_RootKey	CA	OTP	Root key for encrypt boot

3.3.10.4 Configuration of the CAS Vendor Private Data

[Figure 3-11](#) shows the configuration of the CAS private data.



Figure 3-11 Configuration of the CAS private data



To configure the CAS private data, perform the following steps:

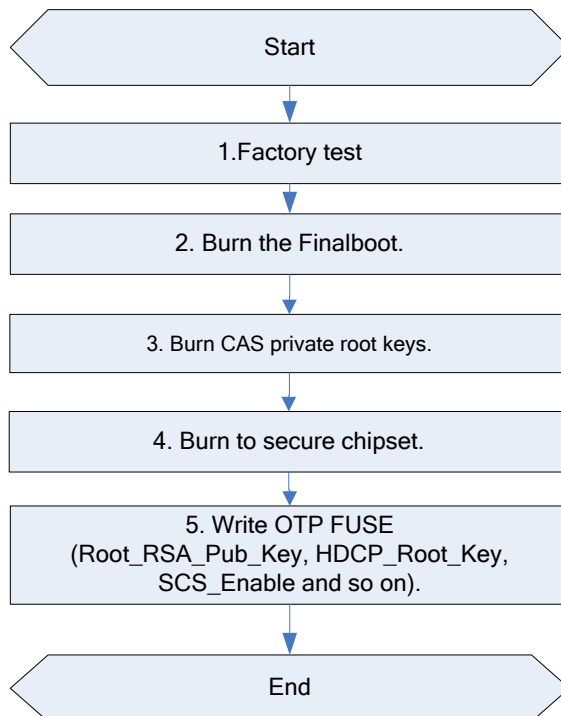


- Step 1** Obtain the ChipID. If the ChipID is 0x0, it needs to be configured. Otherwise, exit from this step.
- Step 2** Set the CSA2_RootKey.
- Step 3** Set the R2R_Root_Key.
- Step 4** Set the MISC_RootKey.
- Step 5** Set the SP RootKey.
- Step 6** Set the Jtag RootKey.
- Step 7** Set the STB RootKey.
- Step 8** Set the root RSA public key.

----End

The CAS private data must be burnt before the chips are burnt into HiSilicon common CA chips. Take Hi3716M V410 as an example. [Figure 3-12](#) shows the mass production process that includes burning the CAS private data.

Figure 3-12 Mass production process that includes burning the CAS private data



The mass production process is as follows:

- Step 1** Perform the factory test, including testing basic functions and input/output interfaces.
- Step 2** Burn the Finalboot.
- Step 3** Burn the CAS private data.
- Step 4** Burn the chip into a HiSilicon common CA chip.



Step 5 Burn the root key, HDCP key, and PV values. For details, see section [3.3.10 "HiSilicon Common CA Chips."](#)

----End