Hi3861 V100 / Hi3861L V100 Cipher Module

User Guide

Issue 03

Date 2020-06-17

HiSilicon (Shanghai) Technologies Co., Ltd.

Address:	New R&D Center, 49 Wuhe Road, Bantian, Longgang District, Shenzhen 518129 P. R. China
Website:	https://www.hisilicon.com/en/
Email:	support@hisilicon.com

About This Document

Purpose

This document describes the API functions and usage of the Hi3861 V100/Hi3861L V100 cipher module.

Related Versions

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

Product Name	Version
Hi3861	V100
Hi3861L	V100

Intended Audience

The document is intended for:

- Technical support engineers
- Software development engineers

Symbol Conventions

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

Symbol	Description
⚠ DANGER	Indicates a hazard with a high level of risk which, if not avoided, will result in death or serious injury.
⚠ WARNING	Indicates a hazard with a medium level of risk which, if not avoided, could result in death or serious injury.

Symbol	Description
⚠ CAUTION	Indicates a hazard with a low level of risk which, if not avoided, could result in minor or moderate injury.
NOTICE	Indicates a potentially hazardous situation which, if not avoided, could result in equipment damage, data loss, performance deterioration, or unanticipated results. NOTICE is used to address practices not related to personal injury.
🕮 NOTE	Supplements the important information in the main text. NOTE is used to address information not related to personal injury, equipment damage, and environment deterioration.

Change History

Issue	Date	Change Description
03	2020-6-17	• In **eFUSE** of **3.2.1 Dependencies**, the HiBurn tool is renamed the HiBurn CLI version tool for the production line. • In **3.2.3 Implementation Method**, the HiBurn CLI version for the production line and HiBurn GUI version are used. **Step 6** of **Mass Production Chip Version** is updated. **Step 5** of **R&D Version** is updated. • **3.3 Precautions** is updated. • In **4.4 Precautions**, the number of KDF iterations is updated. • In **4.5 Code Sample**, the salt value description is updated.
02	2020-06-05	• In **Function** of **2.4.2 Development Procedure**, the description of hash calculation is added. • In **2.5.4 Code Sample**, the descriptions of the AES config parameters, AES-XTS encryption and decryption parameters, and encryption and decryption data length are added. • In **2.6.1 Overview**, the descriptions of the key pair and key length are added. • In **2.6.3 Precautions**, the description of the enumeration **hi_cipher_rsa_sign_scheme** is added.

Issue	Date	Change Description
01	2020-04-30	This issue is the first official release. ● In **2.5.4 Code Sample**, the AES encryption and decryption examples are updated. ● In **eFUSE** of **3.2.1 Dependencies**, the descriptions of the root key salt value and **FLASH_ENCPT_CNT** are updated. The description of **DIE_ID** is deleted. **NV Factory Partition** is updated. In **Key Encryption**, the description of enabling the flash encryption function is updated, and the related precaution is added. ● In **Mass Production Chip Version** of **3.2.3 Implementation Method**, **Step 4** is updated. ● **3.3 Precautions** is updated. ● In **4.2 Root Key**, the definition of the parameter configuration control structure of the KDF is updated. The description about the source of the root key salt value is updated. ● In **4.3 Secure Storage Design**, the definition of the AES algorithm parameter configuration control structure is updated. ● In **4.4 Precautions**, the descriptions of the number of KDF iterations and clearing the memory for temporarily storing key user information before the release are added.
00B05	2020-04-20	**4 Secure Storage of User Data** is added.
00B04	2020-04-15	● In **3.2.1 Dependencies**, **Key Encryption** is added. ● In **3.2.2 Burning Tool**, the description of the HiBurn GUI version is updated. The description of the HiBurn CLI version for the production line is added.
00B03	2020-04-07	● In **3.2.1 Dependencies**, the descriptions of **ROOT_KEY** and **DIE_ID** are added. ● In **3.2.3 Implementation Method**, **Step 5** of **Mass Production Chip Version** is updated. For details about how to burn the eFUSE in the factory, see the *Hi3861 V100/Hi3861L V100 Production Line Equipment Test User Guide*. ● In **3.3 Precautions**, the precaution for burning the flash memory is added.

Issue	Date	Change Description
00B02	2020-03-25	• **2.1 APIs for Module Initialization and Clock Mode Switching** is added. • In **Table 2-3**, the description of hi_cipher_init is deleted. • In **2.4.3 Precautions**, the description about the mutex mechanism used by **hi_cipher_hash_start** and **hi_cipher_hash_final** is added. • In **2.5.3 Precautions**, the description about the mutex mechanism used by **hi_cipher_aes_config** and **hi_cipher_taes_destroy_config** is updated. • **3 Flash Encryption and Decryption** is added.
00B01	2020-01-15	This issue is the first draft release.

Contents

1 Introduction

A cipher module consists of the random number generation module, hash algorithm module for data integrity, symmetric encryption algorithm module for data confidentiality, and asymmetric encryption algorithm module for data transmission security by signing and signature verification. The symmetric encryption algorithm includes ECB, CBC, CTR, CCM, and XTS modes to adapt to different key lengths and encryption requirements.

The cipher module is used for data encryption, decryption, signing, and signature verification of Hi3861 V100 and Hi3861L V100, ensuring secure data transmission.

Restrictions:

- The source and result data addresses for AES encryption and decryption must be 4-byte aligned. The data length in ECB, CBC, CTR, or XTS mode must be 16-byte aligned. This requirement is not mandatory in CCM mode.
- The input data address for hashing must be 4-byte aligned, and the memory space to which the output pointer points must be greater than or equal to 32 bytes.

NOTE

The **hi_cipher_init** API of the cipher module needs to be called only once during system initialization. This is the basis of the encryption, decryption, signing, and signature verification in this document.

2 System APIs

2.1 APIs for Module Initialization and Clock Mode Switching

2.1.1 Overview

The cipher module can be initialized in pke or sym mode. When being used, the module can switch between two clock modes.

2.1.2 Development Procedure

Application Scenario

Initializing the cipher module and selecting the clock mode during initialization

Function

Table 2-1 describes the APIs for initialization and clock mode switching of the cipher module.

Table 2-1 APIs for initialization and clock mode switching of the cipher module

API Name	Description
hi_cipher_init	Initializes the cipher module.
hi_cipher_set_clk_switch	Controls the clock of the cipher module. (If the function parameter is set to **False**, the clock is always enabled. If the function parameter is set to **True**, the clock is enabled when the cipher algorithm is used and is disabled after the computing is complete.

Development Procedure

Directly call the APIs.

2.1.3 Precautions

- The **hi_cipher_init** API does not support multiple tasks. It needs to be called only once when the program is initialized.
- The clock mode switching of **hi_cipher_set_clk_switch** API is used during program initialization. If the API is not called, the clock is always enabled by default.

2.2 TRNG

2.2.1 Overview

A true random number generator (TRNG) provides source random numbers for the cipher module using a physical method.

2.2.2 Development Procedure

Application Scenario

Generating random numbers for keys

Function

Table 2-2 shows the APIs of the TRNG module

Table 2-2 APIs of the TRNG module

API Name	Description
hi_cipher_trng_get_random	Obtains random numbers (4 bytes each time).

API Name	Description
hi_cipher_trng_get_random_bytes	Obtains random numbers (multiple bytes each time).

Development Procedure

The **hi_cipher_trng_get_random** and **hi_cipher_trng_get_random_bytes** APIs are used to obtain random numbers. The TRNG has already been initialized. You only need to apply for space to store the obtained random numbers.

2.2.3 Precautions

The TRNG initialization and low-power operations have already been performed for the two TRNG APIs. Therefore, you do not need to initialize the TRNG before calling the APIs. Instead, you can directly obtain the random numbers.

2.2.4 Code Sample

Obtaining random numbers in **hi_cipher_trng_get_random** and **hi_cipher_trng_get_random_bytes** modes:

```
/*Obtaining random numbers*/
hi_s32 sample_trng_test(hi_void)
{
    hi_s32 ret;
    hi_u32 i;
    hi_u8 trng_bytes[32] = {0};  /* 32 */
    hi_u32 trng_word[16] = {0};  /* 16 */

    ret = hi_cipher_trng_get_random_bytes(trng_bytes, sizeof(trng_bytes));
    if (ret != HI_SUCCESS) {
        hi_err_cipher("hi_cipher_trng_get_random_bytes failed.\n");
        (hi_void)hi_cipher_init();
        return ret;
    }

    hi_print_u32("trng_bytes", (hi_u32 *)trng_bytes, sizeof(trng_bytes) / 4);

    for (i = 0; i < sizeof(trng_word) / 4; i++) {  /* 4 */
        ret = hi_cipher_trng_get_random(&trng_word[i]);
        if (ret != HI_SUCCESS) {
            hi_err_cipher("hi_cipher_trng_get_random failed.\n");
            (hi_void)hi_cipher_init();
            return ret;
        }
    }

    hi_print_u32("trng_word", (hi_u32 *)trng_word, sizeof(trng_word) / 4);
    return HI_SUCCESS;
}
```

2.3 KDF

2.3.1 Overview

A key derivation function (KDF) may generate one or more keys based on the random numbers generated by the TRNG.

2.3.2 Development Procedure

Application Scenario

Generating keys

Function

Table 2-3 shows the APIs of the KDF module.

Table 2-3 APIs of the KDF module

API Name	Description
hi_cipher_kdf_key_derive	Generates keys using the KDF algorithm.

Development Procedure

Call this API directly after the cipher module is initialized.

2.3.3 Precautions

Initialization is not involved in **hi_cipher_kdf_key_derive**, because it has been completed in **app_main.c**.

2.3.4 Code Sample

Key generation implemented by the KDF module:

```
/*Function implementation*/
hi_u32 sample_func(HI_CONST hi_u8 *key, hi_u32 key_len, HI_CONST hi_u8 *iv, hi_u32 iv_len,
hi_u32 cnt,
                   HI_CONST hi_u8 *dest, hi_u32 dest_len)
{
    hi_u32 ret;
    hi_cipher_kdf_ctrl ctrl;
    ret = memset_s(&ctrl, sizeof(ctrl), 0, sizeof(ctrl));
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(memset_s, ret);
        return ret;
    }

    ret = memcpy_s(ctrl.key, sizeof(ctrl.key), key, key_len);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(memcpy_s, ret);
        return ret;
    }

    ctrl.salt = iv;
    ctrl.salt_len = iv_len;
    ctrl.kdf_cnt = cnt; /* 1 times iteration */
    ctrl.kdf_mode = HI_CIPHER_SSS_KDF_KEY_DEVICE;

    hi_print_hex("dest", dest, dest_len);
    ret = (hi_s32)hi_cipher_kdf_key_derive(&ctrl);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(hi_cipher_deinit, ret);
        return ret;
    }
    if (memcmp(dest, ctrl.result, sizeof(ctrl.result)) != 0) {
        hi_log_error("Invalid kdf result:\n");
        hi_print_hex("dest", dest, dest_len);
        hi_print_hex("ctrl.result", ctrl.result, sizeof(ctrl.result));
        return HI_FAILURE;
    }

    hi_print_hex("result", ctrl.result, sizeof(ctrl.result));
    return HI_SUCCESS;
}
```

2.4 HASH

2.4.1 Overview

The hash algorithm ensures data integrity by checking whether the transmitted data is the same as that received.

2.4.2 Development Procedure

Application Scenario

Performing hashing on a segment of data and verifying the result by the sender and receiver

Function

NOTE

This hash calculation is implemented based on the SHA-256 algorithm.

Table 2-4 shows the APIs of the HASH module.

Table 2-4 APIs of the HASH module

API Name	Description
hi_cipher_hash_start	Configures HASH or HMAC algorithm parameters (called before HASH/HMAC computing).
hi_cipher_hash_update	Performs hashing (multi-segment computing is supported. HMAC supports only single-segment computing.)
hi_cipher_hash_final	Ends the HASH or HMAC computing (outputs the computing result).

API Name	Description
hi_cipher_hash_sha256	Performs hashing on a segment of data and outputs the result.

Development Procedure

To compute and output the HASH value of a segment of data, perform the following steps:

Step 1 Call **hi_cipher_hash_start** to configure the HASH or HMAC algorithm parameters.

Step 2 Call **hi_cipher_hash_update** to perform hashing.

Step 3 Call **hi_cipher_hash_final** to output the computing result.

----End

2.4.3 Precautions

- The input data address for hashing must be 4-byte aligned.
- The length of the HASH result is 32 bytes. The output pointer of the HASH or HMAC computing result points to the space length. The output length must be greater than or equal to 32 bytes.
- The **hi_cipher_hash_sha256** API is used to encapsulate other APIs in **Table 2-4**, implement parameter configuration and hashing, and output the computing result.
- **hi_cipher_hash_start** and **hi_cipher_hash_final** use the mutual exclusion mechanism and must be used together to avoid errors.

2.4.4 Code Sample

```
hi_s32 hash_sha256_test(hi_void)
{
    hi_s32 ret;
    hi_u8 input[3] = { 0x61, 0x62, 0x63 }; /* abc array size 3 */
    hi_u8 output[32] = { 0 };              /* array size 32 */
    hi_u8 dest[32] = {                     /* array size 32 */
        0xba, 0x78, 0x16, 0xbf, 0x8f, 0x01, 0xcf, 0xea, 0x41, 0x41, 0x40, 0xde, 0x5d,
        0xae, 0x22, 0x23, 0xb0, 0x03, 0x61, 0xa3, 0x96, 0x17, 0x7a, 0x9c, 0xb4, 0x10,
        0xff, 0x61, 0xf2, 0x00, 0x15, 0xad
    };
    uintptr_t src_addr;
    hi_u32 data_length;
    hi_cipher_hash_atts atts;

    ret = memset_s(&atts, sizeof(atts), 0, sizeof(atts));
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(memset_s, ret);
        return ret;
    }

    atts.sha_type = HI_CIPHER_HASH_TYPE_SHA256;
    data_length = sizeof(input);
    src_addr = (uintptr_t)input;
    hi_print_hex("input", (hi_u8 *)src_addr, data_length);
    ret = (hi_s32)hi_cipher_hash_start(&atts);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(hi_cipher_hash_start, ret);

        return ret;
    }

    ret = (hi_s32)hi_cipher_hash_update(src_addr, data_length);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(hi_cipher_hash_update, ret);

        return ret;
    }

    ret = (hi_s32)hi_cipher_hash_final(output, sizeof(output));
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(hi_cipher_hash_final, ret);

        return ret;
    }
    if (memcmp(dest, output, sizeof(dest)) != 0) {
        hi_log_error("Invalid hash result:\n");
        hi_print_hex("dest", dest, sizeof(dest));
        hi_print_hex("output", output, sizeof(output));
        return HI_FAILURE;
    }
    hi_print_hex("output", output, sizeof(output));

    return HI_SUCCESS;
}
```

2.5 AES

2.5.1 Overview

The advanced encryption standard (AES) algorithm uses the same key to encrypt and decrypt data.

2.5.2 Development Procedure

Application Scenario

Encrypting and decrypting data to ensure system running and information transmission security

Function

Table 2-5 shows the APIs of the AES module.

Table 2-5 APIs of the AES module

API Name	Description
hi_cipher_aes_config	Configures AES keys.
hi_cipher_aes_crypto	Encrypts or decrypts AES.
hi_cipher_aes_get_tag	Outputs the tag value (After the AES CCM encryption or decryption is complete, the tag value is output for verification.)
hi_cipher_aes_destroy_config	Destroys the AES configuration (used in pairs with **hi_cipher_aes_config**).

Development Procedure

The typical process of using the AES module is as follows:

Step 1 Call **hi_cipher_aes_config** to configure the key parameters, including the key length and mode (ECB, CBC, or CTR).

Step 2 Encrypts or decrypts data by calling **hi_cipher_aes_crypto**.

Step 3 Call **hi_cipher_aes_get_tag** in CCM mode to output the tag value.

Step 4 Destroy the key configuration parameters by calling **hi_cipher_aes_destroy_config** after encryption.

----End

2.5.3 Precautions

- The physical address of the source data to be encrypted or decrypted must be 4-byte aligned.
- The physical address of the encrypted or decrypted data must be 4-byte aligned.
- In ECB, CBC, CTR, or XTS mode, the data length must be 16-byte aligned. This requirement is not mandatory in CCM mode.
- **HI_TRUE** represents encryption and **HI_FALSE** represents decryption.
- **hi_cipher_aes_config** and **hi_cipher_aes_destroy_config** use the mutual exclusion mechanism and must be used together.

2.5.4 Code Sample

The AES configuration parameters are described as follows:

```
typedef struct {
    hi_u32 key[AES_MAX_KEY_IN_WORD];  // Configuration key, AES_MAX_KEY_IN_WORD = 16
    hi_u32 iv[AES_IV_LEN_IN_WORD];    // Configuration IV, AES_IV_LEN_IN_WORD = 4
    hi_bool random_en;                // Whether to enable random delay
    hi_u8 resv[3];                    // Reserved field (3 bytes)
    hi_cipher_aes_key_from key_from;  // Key source
    hi_cipher_aes_work_mode work_mode; // Working mode, which can be ECB, CBC, CTR, or XTS
    hi_cipher_aes_key_length key_len;  // Key length. aes-ecb/cbc/ctr support 128/192/256 bits
key, ccm just support
    // 128 bits key, xts just support 256/512 bits key.
    hi_cipher_aes_ccm *ccm;             // Struct for ccm.
}hi_cipher_aes_ctrl;
```

The AES-XTS encryption and decryption parameters are described as follows:

```
typedef struct {
    hi_char *key1;        // Data key, which is used during AES-encryption/decryption for plaintext/
ciphertext
    hi_char *key2;        // Adjustment key, which is used for AES-encryption on tweak
    hi_u32 klen;          // Key length. The key lengths of key1 and key2 are both klen.
    hi_char *data_unit;   // Position value of the 128-bit data block in the data unit, configured in
the IV of the API
    hi_u32 data_unit_len; // Length of the position value, which is the same as the IV length of
the AES
    hi_char *plaintext;   // Plaintext data
    hi_char *ciphertext;  // Encrypted data
    hi_u32 length;        // Data length
    hi_char *tweak;       // XTS adjustment value. Generally, the logical position of the encrypted
data is used. The value is the same as that of data_unit.
    hi_u32 tweak_len;       // Adjustment value length
} aes_xts_test;
```

Length of the encryption/decryption data:

```
0 < len < 1M bytes
```

AES-XTS encryption and decryption:

```
static hi_s32 set_aes_config(HI_CONST hi_u8 *key, hi_u32 klen,
                             HI_CONST hi_u8 *iv, hi_u32 ivlen,
                             hi_bool random_en,
                             hi_cipher_aes_key_from key_from,
                             hi_cipher_aes_work_mode work_mode,
                             hi_cipher_aes_key_length key_len,
                             hi_cipher_aes_ccm *ccm,
                             hi_cipher_aes_ctrl *aes_ctrl)
{
    hi_s32 ret;

    ret = memcpy_s(aes_ctrl->key, sizeof(aes_ctrl->key), key, klen);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(memcpy_s, ret);
        return ret;
    }

    ret = memcpy_s(aes_ctrl->iv, sizeof(aes_ctrl->iv), iv, ivlen);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(memcpy_s, ret);
        return ret;
    }

    aes_ctrl->random_en = random_en;
    aes_ctrl->key_from = key_from;
    aes_ctrl->work_mode = work_mode;
    aes_ctrl->key_len = key_len;
    aes_ctrl->ccm = ccm;

    return HI_SUCCESS;
}
hi_s32 sample_sym_aes_xts_test(HI_CONST aes_xts_test *test)
{
    hi_s32 ret;
    uintptr_t src_phy_addr;
    uintptr_t dest_phy_addr;
    hi_u8 *src_vir = HI_NULL;
    hi_u8 *dest_vir = HI_NULL;
    hi_u8 *buf = HI_NULL;
    hi_s32 data_length = (hi_s32)test->length;
    hi_u32 buf_size = (hi_u32)data_length * 2; /* 2 */
    hi_u8 xts_key[64] = { 0 };                  /* 64 */
    hi_u32 xts_key_len = test->klen * 2;        /* 2 */
    hi_u8 *plaintext = (hi_u8 *)test->plaintext;
    hi_u8 *ciphertext = (hi_u8 *)test->ciphertext;
    hi_u8 *data_unit = (hi_u8 *)test->data_unit;
    hi_cipher_aes_key_length key_len = HI_CIPHER_AES_KEY_LENGTH_256BIT;
    hi_cipher_aes_ctrl aes_ctrl;

    ret = memset_s(&aes_ctrl, sizeof(aes_ctrl), 0, sizeof(aes_ctrl));
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(memset_s, ret);
        return ret;
    }

    if (test->klen == 32) { /* 32 */
        key_len = HI_CIPHER_AES_KEY_LENGTH_512BIT;
    }

    ret = memcpy_s(xts_key, sizeof(xts_key), (hi_u8 *)test->key1, test->klen);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(memcpy_s, ret);
        return ret;
    }

    ret = memcpy_s(xts_key + test->klen, sizeof(xts_key) - test->klen, (hi_u8 *)test->key2, test-
>klen);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(memcpy_s, ret);
        return ret;
    }

    buf = (hi_u8 *)malloc(buf_size);
    if (buf == HI_NULL) {
        hi_log_error("malloc for buf failed.\n");
        return HI_FAILURE;
    }

    ret = memset_s(buf, buf_size, 0, buf_size);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(memset_s, ret);
        free(buf);
        buf = HI_NULL;
        return ret;
    }

    src_vir = buf;
    dest_vir = buf + data_length;

    ret = memcpy_s(src_vir, (hi_u32)data_length, plaintext, (hi_u32)data_length);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(memcpy_s, ret);
        free(buf);
        buf = HI_NULL;
        return ret;
    }
    ret = set_aes_config((HI_CONST hi_u8 *)xts_key, xts_key_len,
                         (HI_CONST hi_u8 *)data_unit, test->data_unit_len,
                         HI_FALSE,
                         HI_CIPHER_AES_KEY_FROM_CPU,
                         HI_CIPHER_AES_WORK_MODE_XTS,
                         key_len,
                         HI_NULL, &aes_ctrl);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(set_aes_config, ret);

        free(buf);
        buf = HI_NULL;
        return ret;
    }
    ret = (hi_s32)hi_cipher_aes_config(&aes_ctrl);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(hi_cipher_aes_config, ret);
        free(buf);
        buf = HI_NULL;
        return ret;
    }

    src_phy_addr = (uintptr_t)src_vir;
    dest_phy_addr = (uintptr_t)dest_vir;

    /* encrypt */
    ret = (hi_s32)hi_cipher_aes_crypto(src_phy_addr, dest_phy_addr, (hi_u32)data_length,
HI_TRUE);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(hi_cipher_aes_crypto, ret);
        (hi_void) hi_cipher_aes_destroy_config();
        free(buf);
        buf = HI_NULL;
        return ret;
    }

    crypto_print_buffer("dest_vir", dest_vir, (hi_u32)data_length);
    if (memcmp(ciphertext, dest_vir, (hi_u32)data_length) != 0) {
        hi_log_error("aes xts encrypt gold test failed.\n");
        crypto_print_buffer("dest_vir", dest_vir, (hi_u32)data_length);
        crypto_print_buffer("ciphertext", ciphertext, (hi_u32)data_length);
        (hi_void) hi_cipher_aes_destroy_config();
        free(buf);
        buf = HI_NULL;
        return ret;
    }

    /* decrypt */
    ret = (hi_s32)hi_cipher_aes_crypto(dest_phy_addr, dest_phy_addr, (hi_u32)data_length,
HI_FALSE);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(hi_cipher_aes_crypto, ret);
        (hi_void) hi_cipher_aes_destroy_config();
         free(buf);
        buf = HI_NULL;
        return ret;
    }

    crypto_print_buffer("dest_vir", dest_vir, (hi_u32)data_length);
    if (memcmp(plaintext, dest_vir, (hi_u32)data_length) != 0) {
        hi_log_error("aes xts decrypt gold test failed.\n");
        crypto_print_buffer("dest_vir", dest_vir, (hi_u32)data_length);
        crypto_print_buffer("plaintext", plaintext, (hi_u32)data_length);
        (hi_void) hi_cipher_aes_destroy_config();
        free(buf);
        buf = HI_NULL;
        return ret;
    }

    ret = (hi_s32)hi_cipher_aes_destroy_config();
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(hi_cipher_aes_destroy_config, ret);
        free(buf);
        buf = HI_NULL;
        return ret;
    }
    return HI_SUCCESS;
}
```

2.6 RSA

2.6.1 Overview

The Rivest-Shamir-Adleman (RSA) algorithm uses a private key to sign information and uses a public key to verify the signature, which implements integrity verification of information transmission, identity authentication of the sender, and prevention of repudiation in transactions.

- Key pairs: public keys (n, e) and private keys (n, d) The key pair required by RSA is generated by the user.
- Key length: RSA-2048 (256 bytes) and RSA-4096 (512 bytes)

2.6.2 Development Procedure

Application Scenario

Signing files

Function

Table 2-6 shows the APIs of the RSA module.

Table 2-6 APIs of the RSA module

API Name	Description
hi_cipher_rsa_sign_hash	Outputs the RSA signature result.
hi_cipher_rsa_verify_hash	Verifies the RSA signature result.

Development Procedure

The RSA signature process is as follows:

Step 1 Call **hi_cipher_rsa_sign_hash** to sign the RSA.

Step 2 Call **hi_cipher_rsa_verify_hash** to verify the RSA signature result.

----End

2.6.3 Precautions

- The length of the HASH data to be verified is 32 bytes.
- The length of the output signature result is **klen**.
- Select the corresponding PKCS encoding standard from the enumerated type **hi_cipher_rsa_sign_scheme**. The enumerated values include PKCS encoding standard and SHA-256. You can select **RSASSA_PKCS1_V15** or **RSASSA_PKCS1_PSS**. The hash calculation must be implemented based on the SHA-256 algorithm.

2.6.4 Code Sample

Using RSA for signing and signature verification:

```
static hi_s32 rsa_sign_verify_test(HI_CONST rsa_testvec *testvec, hi_cipher_rsa_sign_scheme
scheme)
{
    hi_s32 ret;
    hi_u8 sign[512] = { 0 }; /* 512 */
    hi_u32 sign_len = 0;
    hi_cipher_rsa_sign rsa_sign;
    hi_cipher_output sign_out;
    hi_cipher_rsa_verify rsa_verify;

    hi_info_cipher("rsa sign vefiry test start.\n");

    check_ret(memset_s(&rsa_sign, sizeof(rsa_sign), 0, sizeof(rsa_sign)));
    check_ret(memset_s(&rsa_verify, sizeof(rsa_verify), 0, sizeof(rsa_verify)));
    check_ret(memset_s(&sign_out, sizeof(sign_out), 0, sizeof(sign_out)));
    rsa_sign.n = (hi_u8 *)(testvec->n);
    rsa_sign.d = (hi_u8 *)(testvec->d);
    rsa_sign.klen = testvec->key_len;
    rsa_sign.scheme = scheme;

    sign_out.out = sign;
    sign_out.out_buf_len = sizeof(sign);
    sign_out.out_len = &sign_len;

    ret = (hi_s32)hi_cipher_rsa_sign_hash(&rsa_sign, g_sha256_sum, sizeof(g_sha256_sum),
&sign_out);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(hi_cipher_rsa_sign_hash, ret);
        return ret;
    }

    switch (scheme) {
        case HI_CIPHER_RSA_SIGN_SCHEME_RSASSA_PKCS1_V15_SHA256:
            if (memcmp(sign, testvec->m, testvec->key_len) != 0) {
                hi_log_error("rsa sign failed\n");
                crypto_print_buffer("sign", sign, sign_len);
                crypto_print_buffer ("golden", (hi_u8 *)(testvec->m), testvec->key_len);
                return HI_FAILURE;
            }
            break;
        default:
            break;
    }
    crypto_print_buffer("sign", sign, sign_len);

    rsa_verify.n = (hi_u8 *)(testvec->n);
    rsa_verify.e = (hi_u8 *)(testvec->e);
    rsa_verify.klen = testvec->key_len;
    rsa_verify.scheme = scheme;

    ret = (hi_s32)hi_cipher_rsa_verify_hash(&rsa_verify, g_sha256_sum, sizeof(g_sha256_sum),
sign, sign_len);
    if (ret != HI_SUCCESS) {
        hi_log_print_func_err(hi_cipher_rsa_verify_hash, ret);
        return ret;
    }
    return HI_SUCCESS;
}
```

2.7 ECC

2.7.1 Overview

The elliptic curve cryptography (ECC) algorithm has the same function as the RSA algorithm. Compared with the RSA-based digital signature algorithm, the ECC algorithm requires a shorter public key for digital signature computing.

2.7.2 Development Procedure

Application Scenario

Signing data

Function

Table 2-7 shows the APIs of the ECC module.

Table 2-7 APIs of the ECC module

API Name	Description
hi_cipher_ecc_sign_hash	Outputs the ECC signature result.
hi_cipher_ecc_verify_hash	Verifies the ECC signature result.

Development Procedure

The ECC signature process is as follows:

Step 1 Call **hi_cipher_ecc_sign_hash** to sign the ECC.

Step 2 Call **hi_cipher_ecc_verify_hash** to verify the ECC signature result.

----End

2.7.3 Precautions

If the length of the ECC parameter is less than the key size, prefix the value with 0s.

2.7.4 Code Sample

Using ECC for signing and signature verification:

```
hi_s32 sample_ecc_sign_verify(hi_void)
{
    hi_s32 ret;
    hi_u32 i;
    hi_u32 ecdh_sizes[] = { 32,                 32,                 32 };
    const hi_char *ecdh_p[] = { ECDH_SECP256K1_P,  ECDH_SECP256R1_P,
ECDH_BRAIN_POOL_R1_P };
    const hi_char *ecdh_a[] = { ECDH_SECP256K1_A,  ECDH_SECP256R1_A,
ECDH_BRAIN_POOL_R1_A };
    const hi_char *ecdh_b[] = { ECDH_SECP256K1_B,  ECDH_SECP256R1_B,
ECDH_BRAIN_POOL_R1_B };
    const hi_char *ecdh_gx[] = { ECDH_SECP256K1_GX, ECDH_SECP256R1_GX,
ECDH_BRAIN_POOL_R1_GX };
    const hi_char *ecdh_gy[] = { ECDH_SECP256K1_GY, ECDH_SECP256R1_GY,
ECDH_BRAIN_POOL_R1_GY };
    const hi_char *ecdh_n[] = { ECDH_SECP256K1_N,  ECDH_SECP256R1_N,
ECDH_BRAIN_POOL_R1_N };
    const hi_u32 ecdh_h[] = { ECDH_SECP256K1_H,  ECDH_SECP256R1_H,
ECDH_BRAIN_POOL_R1_H };

    hi_u8 hash_test[] = "\x20\x4a\x8f\xc6\xdd\xa8\x2f\x0a\x0c\xed\x7b\xeb\x8e\x08\xa4\x16"
                        "\x57\xc1\x6e\xf4\x68\xb2\x28\xa8\x27\x9b\xe3\x31\xa7\x03\xc3\x35"
                        "\x96\xfd\x15\xc1\x3b\x1b\x07\xf9\xaa\x1d\x3b\xea\x57\x78\x9c\xa0"
                        "\x31\xad\x85\xc7\xa7\x1d\xd7\x03\x54\xec\x63\x12\x38\xca\x34\x45";
    hi_u32 hash_len = 32;
    hi_char *pri_key_gold[] = {
        "4052C1A69DED0B4BA06F1207EC9A9719A22157A6D393763348AE59D87A69F79A",
        "0020BD7A93DA507A71420F7A5407BB3935583979AD0EE778311D8778E786EC77",
        "59dc7e5c69c4a92d8dbd17753390f607d42da1322e74c5cae72b8ae418264b76"
    };
    hi_char *pub_key_x_gold[] = {
        "6ECA4A7BD56225220FD1C82D154C94CB6DEFECB01EE97207F12947F3F837148D",
        "BD7B844B50CB7D636EB9714FE847919314F6CC80BD76F6A4CD869DC527207610",
        "283d8a5633c0926e00da7de6d257eb0fc4f920e1baf21b6a1b2c439ac0623470"
    };
    hi_char *pub_key_y_gold[] = {
        "8AF438A45131ABA72116443E6E5A5970AE87CD6047FF2221F1275A904E8CE63D",
        "D6C4419320EE60D507F84958CE63C675507B768786814D0592D476C492FD3674",
        "1b99e8f828c10ca43a8d1dcb1e000cd1f21b9410f1853bd72769eefe50b3fda9"
    };

    hi_u8 *d = HI_NULL;
    hi_u8 *px = HI_NULL;
    hi_u8 *py = HI_NULL;
    hi_u8 *r = HI_NULL;
    hi_u8 *s = HI_NULL;
    hi_u8 *buf = HI_NULL;
    hi_cipher_ecc_param ecc;
    hi_cipher_ecc_sign sign;
    hi_cipher_ecc_verify verify;
    (hi_void) memset_s(&ecc, sizeof(ecc), 0, sizeof(ecc));
    (hi_void) memset_s(&sign, sizeof(sign), 0, sizeof(sign));
    (hi_void) memset_s(&verify, sizeof(verify), 0, sizeof(verify));

    buf = (hi_u8 *)malloc(ECC_KEY_SIZE * 11); /* mem size ECC_KEY_SIZE * 11 */
    if (buf == HI_NULL) {
        hi_err_cipher("malloc for buf failed\n");
        return HI_FAILURE;
    }
    (hi_void) memset_s(buf, ECC_KEY_SIZE * 11, 0, ECC_KEY_SIZE * 11); /* mem size
ECC_KEY_SIZE * 11 */

    ecc.p = buf;
    ecc.a = ecc.p + ECC_KEY_SIZE;
    ecc.b = ecc.a + ECC_KEY_SIZE;
    ecc.gx = ecc.b + ECC_KEY_SIZE;
    ecc.gy = ecc.gx + ECC_KEY_SIZE;
    ecc.n = ecc.gy + ECC_KEY_SIZE;
    r = (hi_u8 *)(ecc.n + ECC_KEY_SIZE);
    s = r + ECC_KEY_SIZE;
    d = s + ECC_KEY_SIZE;
    px = d + ECC_KEY_SIZE;
    py = px + ECC_KEY_SIZE;
    for (i = 0; i < 3; i++) { /* loop 3 time */
        hi_info_cipher("\n**************************** E C D S A - T E S T %d ***********************\n", i);

        memcpy_s((void *)ecc.p, ecdh_sizes[i], str2hex(ecdh_p[i]), ecdh_sizes[i]);
        memcpy_s((void *)ecc.a, ecdh_sizes[i], str2hex(ecdh_a[i]), ecdh_sizes[i]);
        memcpy_s((void *)ecc.b, ecdh_sizes[i], str2hex(ecdh_b[i]), ecdh_sizes[i]);
        memcpy_s((void *)ecc.gx, ecdh_sizes[i], str2hex(ecdh_gx[i]), ecdh_sizes[i]);
        memcpy_s((void *)ecc.gy, ecdh_sizes[i], str2hex(ecdh_gy[i]), ecdh_sizes[i]);
        memcpy_s((void *)ecc.n, ecdh_sizes[i], str2hex(ecdh_n[i]), ecdh_sizes[i]);
        ecc.h = ecdh_h[i];
        ecc.ksize = ecdh_sizes[i];

        printf("ecdh_sizes: %d\n", ecdh_sizes[i]);
        memcpy_s((void *)d, ecdh_sizes[i], str2hex(pri_key_gold[i]), ecdh_sizes[i]);
        memcpy_s((void *)px, ecdh_sizes[i], str2hex(pub_key_x_gold[i]), ecdh_sizes[i]);
        memcpy_s((void *)py, ecdh_sizes[i], str2hex(pub_key_y_gold[i]), ecdh_sizes[i]);

        sign.d = d;
        sign.hash = hash_test;
        sign.hash_len = hash_len;
        sign.r = r;
        sign.s = s;
        ret = (hi_s32)hi_cipher_ecc_sign_hash(&ecc, &sign);
        if (ret != HI_SUCCESS) {
            hi_err_cipher("hi_cipher_ecc_sign_hash failed, line:0x%x.\n", __LINE__);
            free(buf);
            buf = HI_NULL;
            return ret;
        }

        crypto_print_buffer("r", r, ecdh_sizes[i]);
        crypto_print_buffer("s", s, ecdh_sizes[i]);
        verify.px = px;
        verify.py = py;
        verify.hash = hash_test;
        verify.hash_len = hash_len;
        verify.r = r;
        verify.s = s;
        ret = (hi_s32)hi_cipher_ecc_verify_hash(&ecc, &verify);
        if (ret != HI_SUCCESS) {
            hi_err_cipher("hi_cipher_ecc_verify_hash failed.\n");
            free(buf);
            buf = HI_NULL;
            return ret;
        }
    }

    free(buf);
    buf = HI_NULL;
    return HI_SUCCESS;
}
```

3 Flash Encryption and Decryption

3.1 Overview

The two-level key encryption architecture provides encryption protection to prevent key code from being read and decompiled.

3.2 Instructions

NOTE

This section describes how to encrypt the flash memory and how to debug this function.

3.2.1 Dependencies

eFUSE

In the eFUSE, **FLASH_ENCPT_CFG** indicates whether the encryption lock is enabled. **FLASH_ENCPT_CNT** indicates whether the flash memory is encrypted. The **ROOT_KEY** field indicates the hardware unique key (HUK) value of the root key. The salt value of the root key is randomly generated by hardware.

- **FLASH_ENCPT_CFG**: 1 bit, indicating whether to enable the encryption lock.
  - 0: disabled
  - 1: enabled. The **FLASH_ENCPT_CNT** field is used to indicate whether the flash memory is encrypted.

  Suggestion: Write **0** for R&D and **1** for chip mass production.
- **FLASH_ENCPT_CNT**: There are six **FLASH_ENCPT_CNT** fields. Each field has 2 bits. The default values of the 2 bits are **0**.
  - If bit[0] is set to **1** and bit[1] is set to **0**, the data is not encrypted after being burnt to the flash memory.

  – If bit[1] is set to **1**, the flash memory is encrypted.

  In the mass production chip version, if the flash encryption function is enabled by using menuconfig (see **3.2.3 Implementation Method**), bit[0] of the **FLASH_ENCPT_CNT** field is automatically burnt when programs are burnt by using the HiBurn tool. After the burning is complete, the program firmware is encrypted. Then bit[1] is automatically burnt.

- **ROOT_KEY**: 256 bits. The key is written by software during production. The same type of products have the same HUK value. The **ROOT_KEY** field cannot be read by the software. It is directly read through hard connection when the root key is derived. The **ROOT_KEY** field must be written before the flash memory is burnt. The **ROOT_KEY** field is all 0s by default.

NOTE

For details about how to use the eFUSE, see the *Hi3861 V100/Hi3861L V100 eFUSE User Guide*.

NV Factory Partition

The eFUSE has only six **FLASH_ENCPT_CNT** fields. Therefore, when **FLASH_ENCPT_CFG** is set to **1** and the flash encryption function is enabled for the executable file to be burnt, the flash memory can be burnt only for six times. Burning failures are not included in the count. To support product development, the factory NV item 0x8 is provided to simulate **FLASH_ENCPT_CNT** for debugging the encryption function. When the value of **FLASH_ENCPT_CFG** is **0**, read **flash_encpt_cnt** in this NV item. If the value is **0**, the firmware is not encrypted. In this case, you need to encrypt the firmware. After the encryption is complete, write **1** to **flash_encpt_cnt**. If the value of **flash_encpt_cnt** is **1**, the firmware has been encrypted. The default value of **flash_encpt_cnt** is **0**.

The design of NV item 0x8 is as follows:

```
typedef struct {
   hi_u8 flash_encpt_cnt;
} hi_flash_crypto_cnt;
```

NOTE

For details about how to use the NV, see the *Hi3861 V100/Hi3861L V100 NV User Guide*.

Key Encryption

If flash encryption is enabled, the upgrade file is encrypted during compilation. To configure the key for encrypting the upgrade file, add *upg_aes_key.txt* to **tools/sign_tool**. The key format is as follows:

```
[E]:xxxxxx ;EFUSE_DATA
[C]:xxxxxxx;CPU_DATA
[I]:xxxxxx ;IV_DATA
```

Specifically,

- **EFUSE_DATA**: root_key stored in the eFUSE. The length is 32 bytes. To ensure security, a true random number is required. The data stored in the eFUSE cannot be modified. Therefore, **EFUSE_DATA** cannot be modified after being confirmed.

- **CPU_DATA**: a part of the salt value. The length is 16 bytes. To ensure security, a true random number is required.
- **IV_DATA**: initialization vector. The length is 16 bytes. To ensure security, a true random number is required.

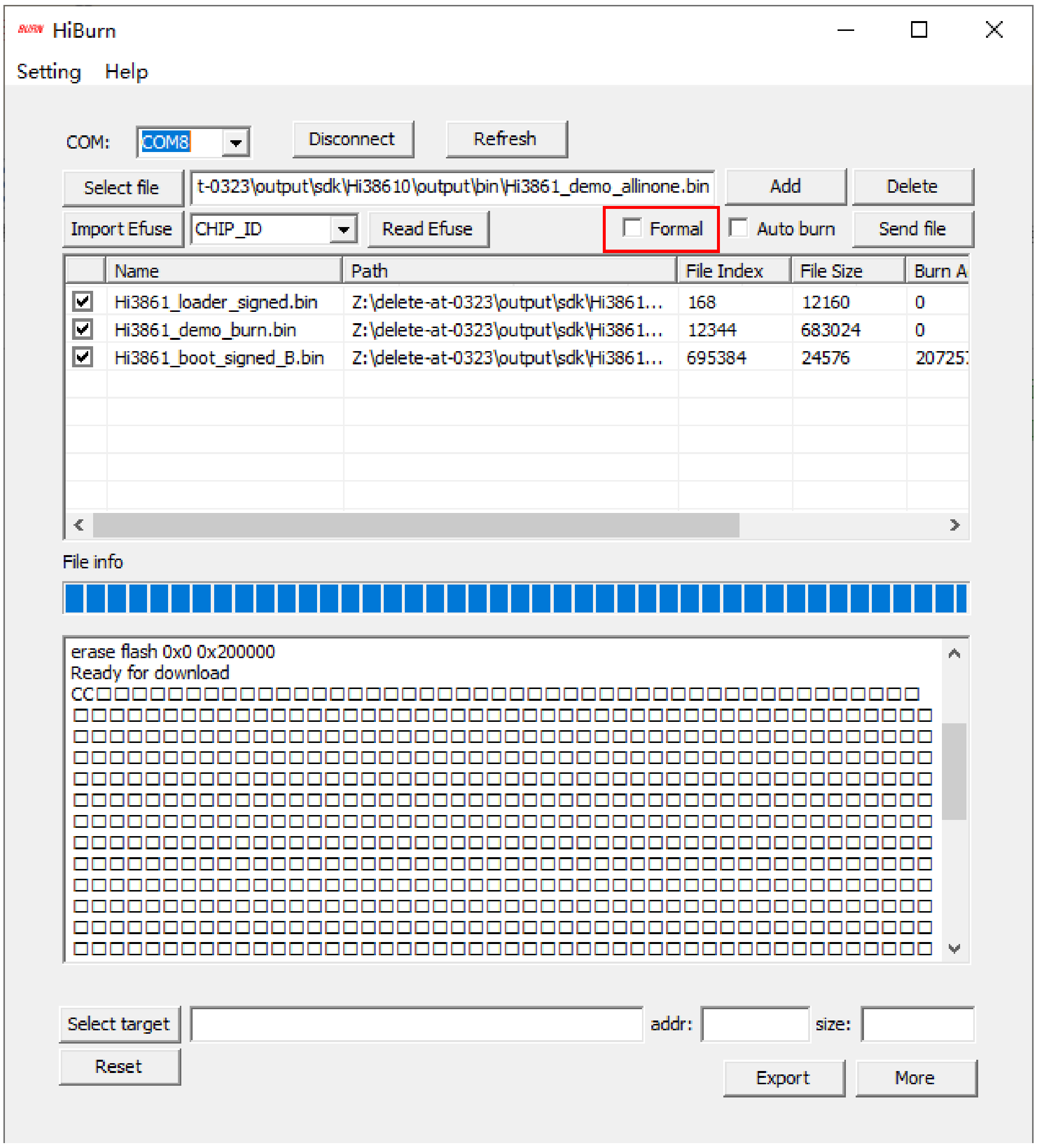
NOTICE

If the flash encryption function is enabled, **EFUSE_DATA** in *upg_aes_key.txt* must be burnt into the **root_key** field in the eFUSE before flash burning. If the value of **root_key** for the eFUSE is different from that of **EFUSE_DATA** in the *upg_aes_key.txt* file, the upgrade will fail.

3.2.2 Burning Tool

- HiBurn GUI version: You can choose whether to burn the **FLASH_ENCPT_CNT** field of the eFUSE during flash burning. If **Formal** is selected, bit[0] of the **FLASH_ENCPT_CNT** field is burnt before the flash memory is burnt. Otherwise, the **FLASH_ENCPT_CNT** field is not burnt.
- HiBurn CLI version for the production line: The **Formal** parameter is available by default, and no additional configuration is required.

Figure 3-1 HiBurn GUI



t-0323\output\sdk\Hi38610\output\bin\Hi3861_demo_allinone.bin

	Name	Path	File Index	File Size	Burn A
✓	Hi3861_loader_signed.bin	Z:\delete-at-0323\output\sdk\Hi3861...	168	12160	0
✓	Hi3861_demo_burn.bin	Z:\delete-at-0323\output\sdk\Hi3861...	12344	683024	0
✓	Hi3861_boot_signed_B.bin	Z:\delete-at-0323\output\sdk\Hi3861...	695384	24576	20725

```
erase flash 0x0 0x200000
Ready for download
CC
```

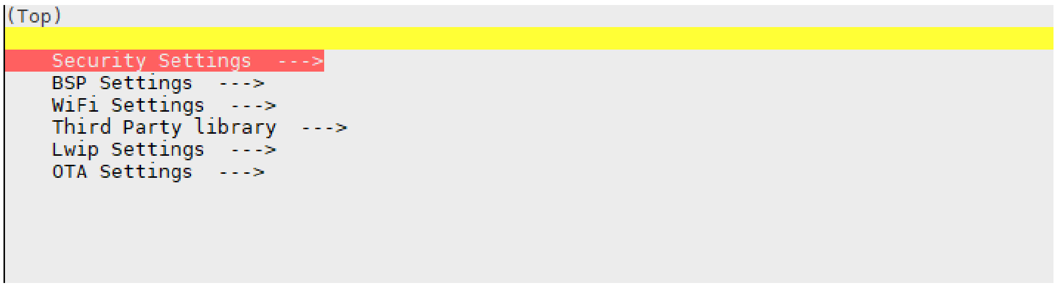
3.2.3 Implementation Method

Mass Production Chip Version

To encrypt the flash memory for the mass production chip version, perform the following steps:

Step 1 Run the **sh build.sh menuconfig** command in the SDK root directory.

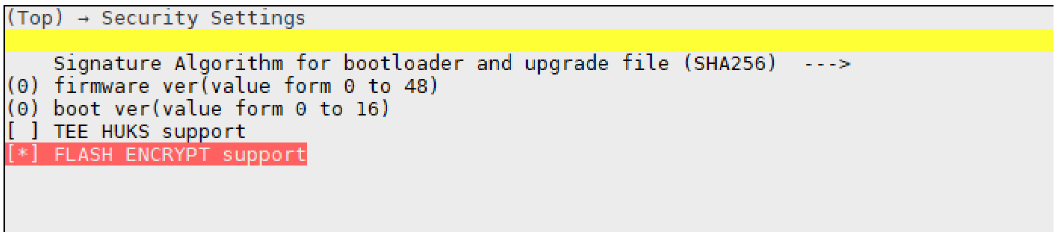
Step 2 Choose **Security Settings**.



```
(Top)

    Security Settings  --->
    BSP Settings  --->
    WiFi Settings  --->
    Third Party library  --->
    Lwip Settings  --->
    OTA Settings  --->
```

Step 3 In **Security Settings**, select **FLASH ENCRYPT support** (that is, **menuconfig** for flash encryption and decryption).



```
(Top) → Security Settings

    Signature Algorithm for bootloader and upgrade file (SHA256)  --->
(0) firmware ver(value form 0 to 48)
(0) boot ver(value form 0 to 16)
[ ] TEE HUKS support
[*] FLASH ENCRYPT support
```

Step 4 To add user code to an encrypted code segment, use the macro **CRYPTO_RAM_TEXT_SECTION** in the code to modify key user functions. If the **FLASH ENCRYPT support** option in **menuconfig** is disabled, the common code segment is linked. For example:

```
CRYPTO_RAM_TEXT_SECTION hi_void flash_crypto_sum_func(hi_u32 a, hi_u32 b)
{
   printf("sum of %d and %d = %d\r\n", a, b, a + b);
}
```

Step 5 Set the **FLASH_ENCPT_CFG** field of the eFUSE to **1** before the flash memory is burnt in factory mode.

For details about how to burn the eFUSE in factory, see the *Hi3861 V100/Hi3861L V100 Production Line Equipment Test User Guide*.

Step 6 After the compilation and burning, the program boots properly, and the code in the encrypted segment can implement the expected function.

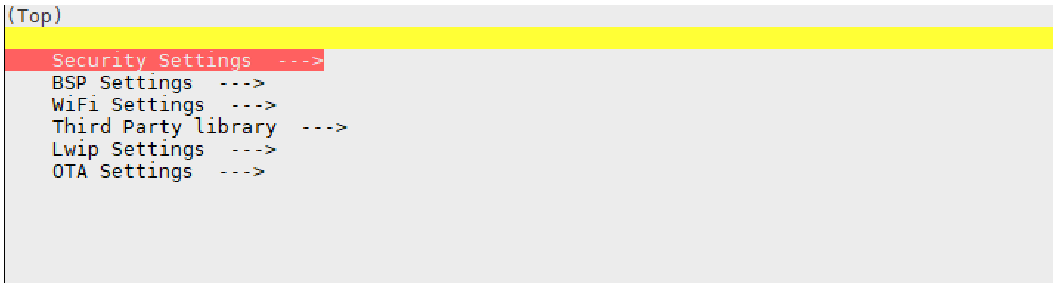
----End

R&D Version

To encrypt the flash memory in the R&D version, perform the following steps:

Step 1 Run the **sh build.sh menuconfig** command in the SDK root directory.

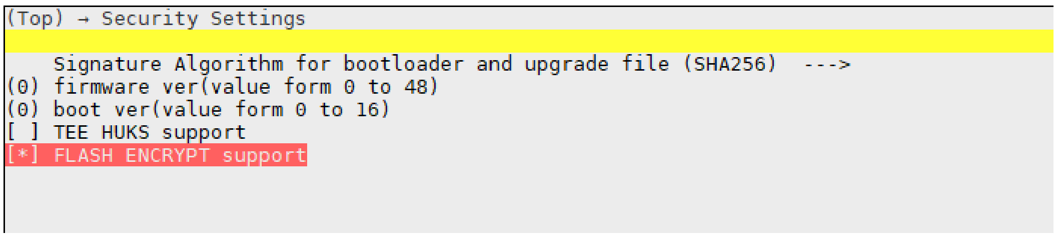
Step 2 Choose **Security Settings**.



```
(Top)

    Security Settings  --->
    BSP Settings  --->
    WiFi Settings  --->
    Third Party library  --->
    Lwip Settings  --->
    OTA Settings  --->
```

Step 3 In **Security Settings**, select **FLASH ENCRYPT support** (that is, **menuconfig** for flash encryption and decryption).



```
(Top) → Security Settings

    Signature Algorithm for bootloader and upgrade file (SHA256)  --->
(0) firmware ver(value form 0 to 48)
(0) boot ver(value form 0 to 16)
[ ] TEE HUKS support
[*] FLASH ENCRYPT support
```

Step 4 To add user code to a specified code segment, use the macro **CRYPTO_RAM_TEXT_SECTION** in the code to modify key user functions. If the **FLASH ENCRYPT support** option in **menuconfig** is disabled, the common code segment is linked. For example:

```
CRYPTO_RAM_TEXT_SECTION hi_void flash_crypto_sum_func(hi_u32 a, hi_u32 b)
{
    printf("sum of %d and %d = %d\r\n", a, b, a + b);
}
```

Step 5 Ensure that **flash_encpt_cnt** of NV item 0x8 in the factory partition is **0**. After compilation and burning, the program boots properly, and the code in the encryption segment can implement the expected function.

----End

3.3 Precautions

- The HiBurn GUI version is different from the HiBurn CLI version for the production line. When **FLASH_ENCPT_CFG** of the eFUSE is set to **1**:
  - HiBurn GUI version: You can determine whether to burn the **FLASH_ENCPT_CNT** field of the eFUSE by selecting the **Formal** check box.
  - HiBurn CLI version for the production line: The **Formal** parameter is available by default, and no additional configuration is required.
- The flash encryption function is enabled in the HiBurn GUI version, and the **FLASH_ENCPT_CFG** field of the eFUSE is **1**. In this case, if the **Formal** check box is not selected during flash burning, the boot fails.
- During OTA update, FlashBoot with flash encryption enabled and that with flash encryption disabled cannot be updated alternately, neither can the kernel with flash encryption enabled and that with flash encryption disabled.
- The eFUSE burning is irreversible due to the one-time programmable feature of the eFUSE. The eFUSE of Hi3861 has only six **FLASH_ENCPT_CNT** fields. Therefore, when flash encryption is enabled in the HiBurn CLI version for the production line, the flash memory can be burnt for only six times.
- If the flash encryption function is enabled and the **ROOT_KEY** field and **LOCK_ROOT_KEY** lock bit of the eFUSE are not burnt before the flash is burnt, the burning fails.

4 Secure Storage of User Data

4.1 Overview

When confidential information such as the account and password is recorded, the storage security of the data must be ensured. Therefore, the data must be encrypted before being stored. Hi3861 V100/Hi3861L V100 uses the 2-layer encryption architecture to implement key security management and provides the root key for encrypting and protecting user data.

4.2 Root Key

The root key is derived from the key derivation function (KDF) and is restored when required. For details about the KDF module, see **2.3 KDF**. This section describes the parameter configuration control structure of the KDF. The structure stores the input parameters required by the KDF and the output parameters after the function is executed.

- The KDF enumerations are as follows:
```
typedef enum {
HI_CIPHER_SSS_KDF_KEY_DEVICE  = 0x0,   /* KDF device key derivation*/
HI_CIPHER_SSS_KDF_KEY_STORAGE,         /* KDF storage key derivation */
HI_CIPHER_SSS_KDF_KEY_MAX,
HI_CIPHER_SSS_KDF_KEY_INVALID = 0xFFFFFFFF,
} hi_cipher_kdf_mode;
```
- The parameter configuration control structure of the KDF is defined as follows:
```
typedef struct {
const hi_u8 *salt;                    /* Salt value used to derive a key. This is an input
parameter. */
hi_u32 salt_len;                    /* Salt length. This is an input parameter:
* When the KDF mode is HI_CIPHER_SSS_KDF_KEY_DEVICE, the length is 16 bytes.
* When the KDF mode is HI_CIPHER_SSS_KDF_KEY_STORAGE, the length is 32 bytes.
*/
hi_u8 key[32];    /* When the KDF mode is HI_CIPHER_SSS_KDF_KEY_DEVICE, the HUK
required by the KDF is stored. This is an input parameter. */
hi_cipher_kdf_mode kdf_mode;        /* KDF mode. This is an input parameter. */
hi_u32 kdf_cnt;                     /* Number of KDF iterations. It is recommended that the
number be greater than or equal to 1000 and less than or equal to 0xffff. This is an input
parameter. */
hi_u8 result[32]; /* When the KDF mode is HI_CIPHER_SSS_KDF_KEY_DEVICE, the derived
key is stored. This is an output parameter. */
} hi_cipher_kdf_ctrl;
```

Key parameters of the parameter configuration control structure are described as follows:

- Salt value: The salt value must be explicitly provided by the user. Different KDF modes correspond to salt values of different lengths.
  - When the KDF mode is **HI_CIPHER_SSS_KDF_KEY_DEVICE**, a 16-byte salt value is required.
  - When the KDF mode is **HI_CIPHER_SSS_KDF_KEY_STORAGE**, a 32-byte salt value is required.
- HUK: The method for obtaining the HUK varies according to the KDF mode.
  - When the **HI_CIPHER_SSS_KDF_KEY_DEVICE** mode is used, you need to explicitly copy the HUK to the **key** member of the parameter configuration control structure.
  - When the KDF mode is **HI_CIPHER_SSS_KDF_KEY_STORAGE**, the KDF module automatically hard-connects to the **root_key** field of the eFUSE to read the HUK value.
- **result**: This output parameter has different meanings for different KDF modes.
  - When the **HI_CIPHER_SSS_KDF_KEY_DEVICE** mode is used, the derived key is stored.
  - When the KDF mode is **HI_CIPHER_SSS_KDF_KEY_STORAGE**, the **result** member is meaningless. The derived key is stored in the KDF module and cannot be read by software.
- **kdf_mode**: It is recommended that the **HI_CIPHER_SSS_KDF_KEY_STORAGE** mode be used to derive the root key and the **HI_CIPHER_SSS_KDF_KEY_DEVICE** mode be used to derive the working key.

HUK and salt value sources of the root key:

- HUK: 256 bits. The value is written to the **root_key** field of the eFUSE in the mass production line phase. The products of the same type have the same HUK value.
- Salt value: 128 bits, which is generated by the true random number API. For example, the salt value of the root key with the flash encryption feature is randomly generated by hardware.

NOTE

For details about the **root_key** field of the eFUSE, see the *Hi3861 V100/Hi3861L V100 eFUSE User Guide*.

NOTICE

The generated root key is stored in the KDF module, which stores only the latest group of root keys. You are advised to use the KDF module to generate a root key before using the root key to encrypt data, ensuring that the latest root key stored in the KDF module is the one required by the user.

4.3 Secure Storage Design

When encrypting user data with the root key, the AES module is required. For details about how to use the AES module, see 2.5 AES. The key source needs to be set to the **HI_CIPHER_AES_KEY_FROM_KDF** mode. In this case, the AES directly reads the root key from the KDF module.

The parameter configuration control structure of the AES algorithm is defined as follows:

```
typedef struct {
hi_u32 key[16];    /* Key*/
hi_u32 iv[4];      /* Initialization vector*/
hi_bool random_en;                  /* Enable random delay.*/
hi_u8 resv[3];                      /* 3-byte reserved field */
hi_cipher_aes_key_from key_from;    /* Key source. If the source is
HI_CIPHER_AES_KEY_FROM_KDF, the key is read from the KDF module and does not need to
be provided by software.*/
hi_cipher_aes_work_mode work_mode;  /* Working mode. The CBC mode is recommended for
encrypting data using the root key.*/
hi_cipher_aes_key_length key_len;   /* Key length. AES-ECB, CBC, and CTR support 128-, 192-,
and 256-bit keys, while CCM supports only 128-bit keys,
                                     * and XTS supports only 256-bit or 512-bit keys.
                                     */
hi_cipher_aes_ccm *ccm;             /* CCM structure*/
} hi_cipher_aes_ctrl;
```

You are advised to use the CBC mode with high security to encrypt data by using the root key. In this encryption mode, you need to provide an initial vector (IV). The vector can be randomly generated by hardware (that is, a random number is generated by calling a TRNG module API). In addition, to verify the integrity of the read-back user data, fields are required to store the hash calculation result of user data. For details about the hash calculation of data, see 2.4 HASH.

The data structure to be stored is as follows:

```
typedef struct {
hi_u8 iv_nv[16];                /* IV value for encrypting user data by using the root key. The value
is stored in plaintext in the flash memory.*/
hi_u8 user_text[32];            /* User data in plaintext, which is stored after being encrypted
using the root key. */
hi_u8 content_sh256[32];        /* The preceding two groups of data are calculated using the
256-bit hash algorithm and stored after being encrypted using the root key. */
} hi_flash_crypto_content;
```

To improve data storage security, you are advised to back up encrypted user data. If the encrypted user data is damaged, you can load the encrypted user data from the data backup partition.

The procedure for secure storage of user data is as follows:

Step 1 The root key is derived by using the KDF.

Step 2 The hardware randomly generates a root key to encrypt the IV value of user data.

Step 3 The IV value and the 256-bit hash value of the user data are calculated, and the result is stored in **content_sh256**.

Step 4 The user data and 256-bit hash value are encrypted through the AES interface.

Step 5 Encrypted data is stored in the specified flash memory.

Step 6 Encrypted data is stored in the data backup partition.

----End

The procedure for secure reading of user data is as follows:

Step 1 The root key is derived by using the KDF.

Step 2 The encrypted data is read from the specified flash memory that stores the encrypted data.

Step 3 The user data and 256-bit hash value are decrypted through the AES interface.

Step 4 The hash interface is used to calculate the IV value and the 256-bit hash value of the user data, which are compared with the data in **content_sh256** to check the data integrity.

Step 5 If the verification in **Step 4** is successful, the data in **user_text** is the original data of the user and the data reading is complete. Otherwise, go to **Step 6**.

Step 6 The encrypted data is read from the data backup partition. Repeat **Step 3** to **Step 4**.

Step 7 If the verification in **Step 4** is successful, the data in **user_text** is the original data of the user and the data reading is complete. Otherwise, data cannot be read securely.

----End

4.4 Precautions

- The root key cannot be read by the software during the entire encryption and decryption process.
- The data length in CBC encryption mode of the AES module must be 16-byte aligned.
- The KDF module stores only the latest root key. Before using a root key, use the KDF to generate the root key to ensure that the latest root key is the one required by the user.
- It is recommended that the number of KDF iterations be greater than or equal to 10000 by default. If high performance is required, the number of iterations must be greater than or equal to 1000.
- To ensure security, the memory that temporarily stores key user information is cleared before being freed.

4.5 Code Sample

Sample of encrypting and storing user data by using the root key:

```
/*
 * Copyright (c) Hisilicon Technologies Co., Ltd. 2012-2019. All rights reserved.
 * Description: Encrypt and store user data.
 * Author: hisilicon
 * Create: 2020-03-16
 */

#include <hi_stdlib.h>
#include <hi_mem.h>
#include <hi_config.h>
#include <hi_cipher.h>
#include <hi_efuse.h>
#include <hi_flash.h>
#include <hi_partition_table.h>

#define crypto_mem_free(sz)                    \
    do {                                       \
        if ((sz) != HI_NULL) {                 \
            hi_free(HI_MOD_ID_CRYPTO, (sz)); \
        }                                      \
        (sz) = HI_NULL;                        \
    } while (0)

#define IV_BYTE_LENGTH              16
#define ROOTKEY_IV_BYTE_LENGTH      32
#define DIE_ID_BYTE_LENGTH          24
#define KEY_BYTE_LENGTH             32
#define SHA_256_LENGTH              32

#define ENCRYPT_KDF_ITERATION_CNT   1024
#define MIN_CRYPTO_BLOCK_SIZE       16

/* The storage address of the user-encrypted data is only an example. You need to specify the
storage address as required.*/
#define CRYPTO_KEY_STORE_ADDR       0x001E1000
#define CRYPTO_KEY_BACKUP_ADDR      0x001E2000

typedef struct {
    hi_u8 iv_nv[IV_BYTE_LENGTH];         /* IV value for encrypting user data by using the root
key. The value is stored in plaintext in the flash memory.*/
    hi_u8 work_text[KEY_BYTE_LENGTH];    /* User data in plaintext, which is stored after being
encrypted using the root key. */
    hi_u8 content_sh256[SHA_256_LENGTH]; /* The preceding two groups of data are calculated
using the 256-bit hash algorithm and stored after being encrypted using the root key. */
} hi_flash_crypto_content;

typedef enum {
    CRYPTO_WORKKEY_KERNEL = 0x1,
    CRYPTO_WORKKEY_KERNEL_BACKUP = 0x2,
    CRYPTO_WORKKEY_KERNEL_BOTH = 0x3,
} crypto_workkey_partition;

/*Hardware-coded magic number, which is a part of the salt value of the root key*/
static hi_u8 g_rootkey_magic_num[ROOTKEY_IV_BYTE_LENGTH] = {
    0x97, 0x0B, 0x79, 0x13, 0x26, 0x79, 0x47, 0xEE,
    0xBD, 0x9C, 0x9D, 0xD3, 0x96, 0xEF, 0xE7, 0xDD,
    0xE4, 0xEE, 0x10, 0x0E, 0x43, 0x4D, 0x94, 0x24,
    0xC7, 0x54, 0x6D, 0xFB, 0x15, 0xA1, 0x46, 0x97,
};

static hi_u32 crypto_prepare(hi_void)
{
    hi_u32 ret;
    hi_u8 hash[SHA_256_LENGTH];
    hi_u8 die_id[DIE_ID_BYTE_LENGTH];
    hi_u8 rootkey_iv[ROOTKEY_IV_BYTE_LENGTH];
    hi_cipher_kdf_ctrl ctrl;
    hi_u32 i;

    ret = hi_efuse_read(HI_EFUSE_DIE_RW_ID, die_id, DIE_ID_BYTE_LENGTH);
    if (ret != HI_ERR_SUCCESS) {
        return ret;
    }
    ret = hi_cipher_hash_sha256((uintptr_t)die_id, DIE_ID_BYTE_LENGTH, hash,
SHA_256_LENGTH);
    if (ret != HI_ERR_SUCCESS) {
        return ret;
    }

   /* The salt value must be generated using a true random number. */
    for (i = 0; i < ROOTKEY_IV_BYTE_LENGTH; i++) {
        rootkey_iv[i] = g_rootkey_magic_num[i] ^ hash[i];
    }
    ctrl.salt = rootkey_iv;
    ctrl.salt_len = sizeof(rootkey_iv);
    ctrl.kdf_cnt = ENCRYPT_KDF_ITERATION_CNT;
   ctrl.kdf_mode = HI_CIPHER_SSS_KDF_KEY_STORAGE; /* Hard-connect to the root_key field in
the eFUSE to read the HUK value. The root key is stored in the KDF module.
    return hi_cipher_kdf_key_derive(&ctrl);
}

static hi_void crpto_set_aes_ctrl_default_value(hi_cipher_aes_ctrl *aes_ctrl)
{
    if (aes_ctrl == HI_NULL) {
        return;
    }
    aes_ctrl->random_en = HI_FALSE;
    aes_ctrl->key_from = HI_CIPHER_AES_KEY_FROM_CPU;
    aes_ctrl->work_mode = HI_CIPHER_AES_WORK_MODE_CBC;
    aes_ctrl->key_len = HI_CIPHER_AES_KEY_LENGTH_256BIT;
}

static hi_u32 crypto_decrypt_hash(hi_flash_crypto_content *key_content)
{
    hi_u32 ret;
    hi_u32 content_size = (hi_u32)sizeof(hi_flash_crypto_content);

    hi_flash_crypto_content *content_tmp = (hi_flash_crypto_content
*)hi_malloc(HI_MOD_ID_CRYPTO, content_size);
    if (content_tmp == HI_NULL) {
        return HI_ERR_FLASH_CRYPTO_MALLOC_FAIL;
    }

    ret = (hi_u32)memcpy_s(content_tmp, content_size, key_content, content_size);
    if (ret != EOK) {
        goto fail;
    }
    hi_cipher_aes_ctrl aes_ctrl = {
        .random_en = HI_FALSE,
        .key_from = HI_CIPHER_AES_KEY_FROM_KDF,
        .work_mode = HI_CIPHER_AES_WORK_MODE_CBC,
        .key_len = HI_CIPHER_AES_KEY_LENGTH_256BIT,
    };

    ret = (hi_u32)memcpy_s(aes_ctrl.iv, sizeof(aes_ctrl.iv), content_tmp->iv_nv, IV_BYTE_LENGTH);
    if (ret != EOK) {
        goto fail;
    }
    ret = hi_cipher_aes_config(&aes_ctrl);
    if (ret != HI_ERR_SUCCESS) {
        goto crypto_fail;
    }
    ret = hi_cipher_aes_crypto((uintptr_t)content_tmp->iv_content, (uintptr_t)key_content-
>iv_content,
        content_size - IV_BYTE_LENGTH, HI_FALSE);
    if (ret != HI_ERR_SUCCESS) {
        goto crypto_fail;
    }

crypto_fail:
    (hi_void) hi_cipher_aes_destroy_config();
fail:
    crypto_mem_free(content_tmp);
    return ret;
}

static hi_u32 crypto_encrypt_hash(hi_flash_crypto_content *key_content)
{
    hi_cipher_kdf_ctrl ctrl;
    hi_cipher_aes_ctrl aes_ctrl;
    hi_u32 content_size = (hi_u32)sizeof(hi_flash_crypto_content);

    hi_flash_crypto_content *data_tmp = (hi_flash_crypto_content
*)hi_malloc(HI_MOD_ID_CRYPTO, content_size);
    if (data_tmp == HI_NULL) {
        return HI_ERR_FLASH_CRYPTO_MALLOC_FAIL;
    }

    ret = (hi_u32)memcpy_s(aes_ctrl.iv, sizeof(aes_ctrl.iv), key_content->iv_nv, IV_BYTE_LENGTH);
    if (ret != EOK) {
        goto fail;
    }

    aes_ctrl.random_en = HI_FALSE;
    aes_ctrl.key_from = HI_CIPHER_AES_KEY_FROM_KDF;
    aes_ctrl.work_mode = HI_CIPHER_AES_WORK_MODE_CBC;
    aes_ctrl.key_len = HI_CIPHER_AES_KEY_LENGTH_256BIT;
    ret = hi_cipher_aes_config(&aes_ctrl);
    if (ret != HI_ERR_SUCCESS) {
        goto crypto_fail;
    }
    ret = hi_cipher_aes_crypto((hi_u32)(uintptr_t)key_content->iv_content, (hi_u32)(uintptr_t)
(data_tmp->iv_content),
        content_size - IV_BYTE_LENGTH, HI_TRUE);
    if (ret != HI_ERR_SUCCESS) {
        goto crypto_fail;
    }

    ret = (hi_u32)memcpy_s(key_content->iv_content, content_size - IV_BYTE_LENGTH,
data_tmp->iv_content,
        content_size - IV_BYTE_LENGTH);

crypto_fail:
    (hi_void) hi_cipher_aes_destroy_config();
fail:
    crypto_mem_free(data_tmp);
    return ret;
}

static hi_u32 crypto_load_user_data(crypto_workkey_partition part, hi_flash_crypto_content
*key_content)
{
    hi_u32 ret = HI_ERR_SUCCESS;
    hi_u8 hash[SHA_256_LENGTH];
    hi_u8 key_e[KEY_BYTE_LENGTH] = { 0 };

    memset_s(key_e, sizeof(key_e), 0x0, KEY_BYTE_LENGTH);
    if (part == CRYPTO_WORKKEY_KERNEL) {
        ret = hi_flash_read(CRYPTO_KEY_STORE_ADDR, sizeof(hi_flash_crypto_content), (hi_u8
*)key_content);
        if (ret != HI_ERR_SUCCESS) {
            goto fail;
        }
    } else if (part == CRYPTO_WORKKEY_KERNEL_BACKUP) {
        ret = hi_flash_read(CRYPTO_KEY_BACKUP_ADDR, sizeof(hi_flash_crypto_content), (hi_u8
*)key_content);
        if (ret != HI_ERR_SUCCESS) {
            goto fail;
        }
    }

    if (memcmp(key_content->work_text, key_e, KEY_BYTE_LENGTH) == HI_ERR_SUCCESS) {
        ret = HI_ERR_FLASH_CRYPTO_KEY_EMPTY_ERR;
        goto fail;
    }

    ret = crypto_decrypt_hash(key_content);
    if (ret != HI_ERR_SUCCESS) {
        goto fail;
    }

    ret = hi_cipher_hash_sha256((uintptr_t)(key_content->iv_nv), sizeof(hi_flash_crypto_content)
- SHA_256_LENGTH,
                                hash, SHA_256_LENGTH);
    if (ret != HI_ERR_SUCCESS) {
        goto fail;
    }
    if (memcmp(key_content->content_sh256, hash, SHA_256_LENGTH) != HI_ERR_SUCCESS) {
        ret = HI_ERR_FLASH_CRYPTO_KEY_INVALID_ERR;
        goto fail;
    }
fail:
    return ret;
}

static hi_u32 crypto_save_user_data(crypto_workkey_partition part, hi_flash_crypto_content
*key_content)
{
    hi_u32 ret;
    hi_u32 content_size = (hi_u32)sizeof(hi_flash_crypto_content);
    hi_flash_crypto_content *content_tmp = (hi_flash_crypto_content
*)hi_malloc(HI_MOD_ID_CRYPTO, content_size);
    if (content_tmp == HI_NULL) {
        return HI_ERR_FLASH_CRYPTO_MALLOC_FAIL;
    }

    ret = (hi_u32)memcpy_s(content_tmp, content_size, key_content, content_size);
    if (ret != EOK) {
        goto fail;
    }

   /* Encrypt the IV value and user data by using the root key. */
    ret = crypto_encrypt_hash(content_tmp);
    if (ret != HI_ERR_SUCCESS) {
        goto fail;
    }

    if ((hi_u32)part & CRYPTO_WORKKEY_KERNEL) {
        ret = hi_flash_write(CRYPTO_KEY_STORE_ADDR, content_size, (hi_u8 *)content_tmp,
HI_TRUE);
        if (ret != HI_ERR_SUCCESS) {
            return HI_PRINT_ERRNO_CRYPTO_KEY_SAVE_ERR;
        }
    }
    if ((hi_u32)part & CRYPTO_WORKKEY_KERNEL_BACKUP) {
        ret = hi_flash_write(CRYPTO_KEY_BACKUP_ADDR, content_size, (hi_u8 *)content_tmp,
HI_TRUE);
        if (ret != HI_ERR_SUCCESS) {
            return HI_PRINT_ERRNO_CRYPTO_KEY_SAVE_ERR;
        }
    }
    return HI_ERR_SUCCESS;

fail:
    crypto_mem_free(content_tmp);
    return ret;
}

static hi_u32 crypto_calculate_hash(hi_flash_crypto_content *key_content)
{
    (hi_void)hi_cipher_trng_get_random_bytes(key_content->iv_nv, IV_BYTE_LENGTH);

    if (hi_cipher_hash_sha256((uintptr_t)(key_content->iv_nv), sizeof(hi_flash_crypto_content) -
SHA_256_LENGTH,
                              key_content->content_sh256, SHA_256_LENGTH) != HI_ERR_SUCCESS) {
        return HI_ERR_FAILURE;
    }

    return HI_ERR_SUCCESS;
}

hi_u32 user_key_management_sample(hi_void)
{
    hi_u32 ret;
    hi_u8 need_gen_key = 0;
    hi_u32 sontent_size = sizeof(hi_flash_crypto_content);

    hi_flash_crypto_content *new_content = (hi_flash_crypto_content
*)hi_malloc(HI_MOD_ID_CRYPTO, sontent_size);
    if (new_content == HI_NULL) {
        return HI_ERR_FLASH_CRYPTO_PREPARE_ERR;
    }
    hi_flash_crypto_content *load_content = (hi_flash_crypto_content
*)hi_malloc(HI_MOD_ID_CRYPTO, sontent_size);
    if (new_content == HI_NULL) {
        crypto_mem_free(new_content);
        return HI_ERR_FLASH_CRYPTO_PREPARE_ERR;
    }

   /* Derive the root key. */
    ret = crypto_prepare();
    if (ret != HI_ERR_SUCCESS) {
        ret = HI_ERR_FLASH_CRYPTO_PREPARE_ERR;
    }

   /*Calculate the IV value and the 256-bit hash value of the user data.*/
    ret = crypto_calculate_hash(new_content);
    if (ret != HI_ERR_SUCCESS) {
        goto fail;
    }

    ret = crypto_save_user_data(CRYPTO_WORKKEY_KERNEL_BOTH, new_content);
    if(ret != HI_ERR_SUCCESS) {
        goto fail;
    }

   /*Load the key from the flash key partition or backup key partition.*/
    ret = crypto_load_user_data(CRYPTO_WORKKEY_KERNEL, load_content);
    if (ret != HI_ERR_SUCCESS) {
        ret = crypto_load_user_data(CRYPTO_WORKKEY_KERNEL_BACKUP, load_content);
        if (ret != HI_ERR_SUCCESS) {
            goto fail;
        }
    }

fail:
    crypto_mem_free(new_content);
    crypto_mem_free(load_content);
    return ret;
}
```