



Hi3861V100 / Hi3861LV100 安全模块

使用指南

文档版本 05

发布日期 2020-06-28

版权所有 © 上海海思技术有限公司2020。保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明



HISILICON、海思和其他海思商标均为海思技术有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受海思公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，海思公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

上海海思技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： <https://www.hisilicon.com/cn/>

客户服务邮箱： support@hisilicon.com



前言

概述

本文档详细描述了Hi3861V100、Hi3861LV100安全模块的接口功能及使用方法。

产品版本

与本文档相对应的产品版本如下。

产品名称	产品版本
Hi3861	V100
Hi3861L	V100




读者对象

本文档主要适用于以下工程师：



- 技术支持工程师
- 软件开发工程师

符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

符号	说明
 危险	表示如不避免则将会导致死亡或严重伤害的具有高等级风险的危害。
 警告	表示如不避免则可能导致死亡或严重伤害的具有中等级风险的危害。
 注意	表示如不避免则可能导致轻微或中度伤害的具有低等级风险的危害。



符号	说明
 须知	用于传递设备或环境安全警示信息。如不可避免则可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。 “须知”不涉及人身伤害。
 说明	对正文中重点信息的补充说明。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害信息。

修改记录

文档版本	发布日期	修改说明
05	2020-06-28	<ul style="list-style-type: none">更新“2.3.2 开发流程”中使用场景的内容。更新“2.3.2 开发流程”中功能的表2-3内容。更新“4.3 安全存储设计”的内容。删除“2.3.4 编程示例”中的代码注释。更新“3.2.1 依赖条件”中EFUSE的内容。更新“3.2.1 依赖条件”中工厂区NV的说明。更新“3.2.1 依赖条件”中加密密钥的内容。更新“3.2.3 实现方法”中HiBurn产线命令行版本步骤6的内容。更新“3.3 注意事项”的内容。更新“4.1 概述”的内容。更新“4.2 根密钥”的内容。更新“4.3 安全存储设计”中用户数据安全存储步骤1、步骤3和用户数据安全读取步骤1、步骤4。更新“4.4 注意事项”的内容。更新“4.5 编程示例”的内容。新增“5 TEE HUKS”章节。
04	2020-06-22	<ul style="list-style-type: none">更新“2.3.1 概述”的内容。



文档版本	发布日期	修改说明
03	2020-6-17	<ul style="list-style-type: none">在“3.2.1 依赖条件”的EFUSE中更新HiBurn工具为HiBurn产线命令行版本工具。更新“3.2.3 实现方法”中HiBurn产线命令行版本和HiBurn界面版本的版本名称；更新HiBurn产线命令行版本的步骤6；更新HiBurn界面版本的步骤5。更新“3.3 注意事项”。更新“4.4 注意事项”中关于KDF迭代次数的说明。更新“4.5 编程示例”中关于盐值的注释。
02	2020-06-05	<ul style="list-style-type: none">在“2.4.2 开发流程”的功能中新增关于HASH计算的说明。在“2.5.4 编程示例”中新增AES Config参数说明、AES-XTS加解密参数说明、加解密数据长度。在“2.6.1 概述”中新增密钥对、密钥长度的说明。在“2.6.3 注意事项”中新增关于枚举hi_cipher_rsa_sign_scheme的说明。
01	2020-04-30	<p>第一次正式版本发布。</p> <ul style="list-style-type: none">更新“2.5.4 编程示例”的AES加解密示例。更新“3.2.1 依赖条件”中EFUSE关于根密钥盐值、FLASH_ENCPT_CNT的说明，删除关于DIE_ID的说明；更新工厂区NV的说明；在加密密钥中更新关于开启Flash加密的说明，新增关于Flash加密功能开启的注意说明。更新“3.2.3 实现方法”中HiBurn产线命令行版本的步骤4描述。更新“3.3 注意事项”。在“4.2 根密钥”中更新KDF函数的参数配置控制结构体定义；更新关于根密钥盐值获取来源的说明。在“4.3 安全存储设计”中更新AES算法参数配置控制结构体定义。在“4.4 注意事项”中新增关于KDF迭代次数、暂存用户关键信息的内存在释放之前先进行清零的说明。
00B05	2020-04-20	新增“ 4 用户数据安全存储 ”章节。
00B04	2020-04-15	<ul style="list-style-type: none">在“3.2.1 依赖条件”中新增加密密钥。在“3.2.2 烧录工具”中更新HiBurn界面版本的描述，新增HiBurn产线命令行版本的描述。



文档版本	发布日期	修改说明
00B03	2020-04-07	<ul style="list-style-type: none">在“3.2.1 依赖条件”中新增关于ROOT_KEY、DIE_ID的说明。在“3.2.3 实现方法”中更新HiBurn产线命令行版本的步骤5；新增关于工厂段烧写EFUSE方法请参见《Hi3861V100 / Hi3861LV100 产线工装 用户指南》的说明。在“3.3 注意事项”中新增关于烧写Flash的注意说明。
00B02	2020-03-25	<ul style="list-style-type: none">新增“2.1 模块初始化及时钟模式切换接口”小节。删除“表2-3”中hi_cipher_init接口的描述。在“2.4.3 注意事项”中新增关于hi_cipher_hash_start与hi_cipher_hash_final使用互斥锁机制的注意说明。在“2.5.3 注意事项”中更新hi_cipher_aes_config与hi_cipher_aes_destroy_config使用互斥锁机制的注意说明。新增“3 Flash加解密”章节。
00B01	2020-01-15	第一次临时版本发布。



目录

前言.....	i
1 概述.....	1
2 系统接口.....	2
2.1 模块初始化及时钟模式切换接口.....	2
2.1.1 概述.....	2
2.1.2 开发流程.....	2
2.1.3 注意事项.....	3
2.2 TRNG.....	3
2.2.1 概述.....	3
2.2.2 开发流程.....	3
2.2.3 注意事项.....	4
2.2.4 编程示例.....	4
2.3 KDF.....	4
2.3.1 概述.....	4
2.3.2 开发流程.....	5
2.3.3 注意事项.....	5
2.3.4 编程示例.....	5
2.4 HASH.....	6
2.4.1 概述.....	6
2.4.2 开发流程.....	6
2.4.3 注意事项.....	7
2.4.4 编程示例.....	7
2.5 AES.....	8
2.5.1 概述.....	8
2.5.2 开发流程.....	8
2.5.3 注意事项.....	9
2.5.4 编程示例.....	9
2.6 RSA.....	13
2.6.1 概述.....	13
2.6.2 开发流程.....	13
2.6.3 注意事项.....	13
2.6.4 编程示例.....	14



2.7 ECC.....	15
2.7.1 概述.....	15
2.7.2 开发流程.....	15
2.7.3 注意事项.....	15
2.7.4 编程示例.....	15
3 Flash 加解密.....	18
3.1 概述.....	18
3.2 使用指导.....	18
3.2.1 依赖条件.....	18
3.2.2 烧录工具.....	20
3.2.3 实现方法.....	21
3.3 注意事项.....	22
4 用户数据安全存储.....	24
4.1 概述.....	24
4.2 根密钥.....	24
4.3 安全存储设计.....	25
4.4 注意事项.....	27
4.5 编程示例.....	27
5 TEE HUKS.....	33
5.1 概述.....	33
5.2 开发流程.....	33
5.3 注意事项.....	35
5.4 编程示例.....	35



1 概述

Cipher安全模块包含随机数生成、哈希算法、对称加密算法和非对称加密算法模块，哈希算法保证了数据的完整性，对称加密算法保证了数据本身的保密性，非对称加密算法用于签名和验签，保证了数据传输过程中的安全性。对称加密算法又包含ECB、CBC、CTR、CCM、XTS模式，适配不同的密钥长度和加密需求。

Hi3861V100、Hi3861LV100芯片的数据加解密、签名和验签由Cipher模块负责，Cipher模块保证数据安全传输。

使用约束：

- AES加解密的源数据地址和结果数据地址要求4byte对齐，ECB/CBC/CTR/XTS数据长度要求16byte对齐，CCM可不要求16byte对齐。
- HASH计算输入数据地址要求4byte对齐，结果输出指针指向空间要求 ≥ 32 byte。

说明

Cipher模块初始化接口hi_cipher_init在系统初始化时调用一次即可，本文的加解密、签名和验签均是在此基础上进行。



2 系统接口

- 2.1 模块初始化及时钟模式切换接口
 - 2.2 TRNG
 - 2.3 KDF
 - 2.4 HASH
 - 2.5 AES
 - 2.6 RSA
 - 2.7 ECC

2.1 模块初始化及时钟模式切换接口

2.1.1 概述

Cipher模块初始化包括pke和sym的初始化，时钟切换为模块使用时提供两种时钟模式。

2.1.2 开发流程

使用场景

初始化阶段对Cipher模块进行初始化和时钟模式的选择。

功能

Cipher模块初始化及时钟模式选择提供的接口如表2-1所示。

表 2-1 Cipher 模块初始化及时钟模式选择接口描述

接口名称	描述
hi_cipher_init	Cipher模块初始化接口。



接口名称	描述
hi_cipher_set_clk_switch	Cipher模块时钟控制接口（函数参设为FALSE，时钟常开；设为TRUE，在使用Cipher算法时打开，计算结束后关闭。）

开发流程

直接调用使用即可。

2.1.3 注意事项

- hi_cipher_init接口不支持多任务，程序入口初始化时调用一次即可。
- hi_cipher_set_clk_switch接口的时钟切换在程序初始化时使用，不调用该接口默认为时钟常开模式。

2.2 TRNG

2.2.1 概述

TRNG（真随机数发生器）是一个利用物理方法实现的随机数发生器（本模块随机数的源头）。

2.2.2 开发流程

使用场景

为密钥的生成产生随机数。

功能

TRNG模块提供的接口如表2-2所示。

表 2-2 TRNG 模块接口描述

接口名称	描述
hi_cipher_trng_get_random	TRNG获取随机数（每次只能获取4byte的随机数）。
hi_cipher_trng_get_random_bytes	TRNG获取随机数（每次获取多个byte的随机数）。



开发流程

hi_cipher_trng_get_random和hi_cipher_trng_get_random_bytes接口均是随机数获取接口，已经包含TRNG初始化，无需重新初始化，只需申请空间存放获取的随机数即可。

2.2.3 注意事项

TRNG两个接口已经包含TRNG的初始化和低功耗的相关操作，无需在调用该函数之前进行初始化操作，直接获取随机数即可。

2.2.4 编程示例

hi_cipher_trng_get_random与hi_cipher_trng_get_random_bytes两种方式获取随机数示例：

```
/*获取随机数*/
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 概述

KDF（密钥导出函数，Key derivation function）可根据需要使用，根据根密钥，派生出二级工作密钥。



2.3.2 开发流程

使用场景

用于派生密钥。

功能

KDF模块提供的接口如表2-3所示。

表 2-3 KDF 模块接口描述

接口名称	描述
hi_cipher_kdf_key_derive	KDF算法计算，生成密钥。当kdf_mode为HI_CIPHER_SSS_KDF_KEY_DEVICE时，表示根据用户指定的root_key派生出密钥，派生出的密钥可被读取；当kdf_mode为HI_CIPHER_SSS_KDF_KEY_STORAGE表示根据efuse中的root_key派生出密钥，该密钥不可读取，将直接作用于AES模块，对应的AES算法配置key_from参数为HI_CIPHER_AES_KEY_FROM_KDF。

开发流程

Cipher模块初始化的基础上直接调用即可。

2.3.3 注意事项

hi_cipher_kdf_key_derive内不包含相关初始化，app_main.c中已经进行了Cipher初始化。

密钥派生迭代次数建议不小于10000次，对于有性能要求的场景，至少不小于1000次。

2.3.4 编程示例

KDF模块实现密钥生成示例：

```
/*函数功能实现*/
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;
    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 概述

哈希算法用于检验传输信息是否相同，保证传输数据的完整性。

2.4.2 开发流程

使用场景

发送方与接收方对一段数据进行HASH计算，对计算结果进行验证实现对收发数据的校验。

功能

说明

本HASH计算基于SHA-256算法实现。

HASH模块提供的接口如表2-4所示。

表 2-4 HASH 模块接口描述

接口名称	描述
hi_cipher_hash_start	HASH/HMAC算法参数配置（HASH/HMAC计算前调用）。



接口名称	描述
hi_cipher_hash_update	HASH计算（支持多段计算，HMAC计算只支持单段计算）。
hi_cipher_hash_final	HASH/HMAC计算结束（输出计算结果）。
hi_cipher_hash_sha256	对一段数据进行HASH计算并输出HASH结果。

开发流程

计算并输出一段数据的HASH值步骤如下：

步骤1 调用hi_cipher_hash_start，进行HASH/HMAC算法参数配置。

步骤2 调用hi_cipher_hash_update，进行HASH计算。

步骤3 调用hi_cipher_hash_final，输出计算结果。

----结束

2.4.3 注意事项

- HASH计算输入数据地址要求4byte对齐。
- HASH结果长度为32byte，HASH/HMAC计算结果输出指针指向空间长度，要求输出长度满足≥32byte。
- hi_cipher_hash_sha256接口是对“表2-4”中其他接口的封装，实现了参数的配置、HASH计算并输出计算结果。
- hi_cipher_hash_start与hi_cipher_hash_final使用互斥锁机制，必须配套使用，避免再次使用时出错。

2.4.4 编程示例

```
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 概述

AES（高级对称加密算法）是一种加解密时使用同一密钥对数据进行加解密的计算方法。

2.5.2 开发流程

使用场景

为保证系统运行和信息传输的安全性，需要对一些数据进行加解密操作。

功能

AES模块提供的接口如[表2-5](#)所示。



表 2-5 AES 模块接口描述

接口名称	描述
hi_cipher_aes_config	AES密钥配置。
hi_cipher_aes_crypto	AES加解密。
hi_cipher_aes_get_tag	输出Tag（AES CCM模式加密或解密计算完成后，输出Tag值进行校验）。
hi_cipher_aes_destroy_config	AES销毁配置的参数（与参数配置成对使用）。

开发流程

使用AES模块的典型流程如下：

- 步骤1 调用hi_cipher_aes_config，配置密钥参数（密钥长度、模式（ECB、CBC和CTR）等）。
- 步骤2 调用hi_cipher_aes_crypto，进行加解密操作。
- 步骤3 CCM模式调用hi_cipher_aes_get_tag，输出tag值。
- 步骤4 加密后调用hi_cipher_aes_destroy_config，销毁密钥配置参数。

----结束

2.5.3 注意事项

- 待加密或解密的源数据物理地址要求4byte对齐。
- 加密或解密结果数据物理地址要求4byte对齐。
- ECB/CBC/CTR/XTS模式数据长度要求16byte对齐；CCM模式数据长度可以不要求16byte对齐。
- 加解密配置选项配置HI_TRUE为加密、HI_FALSE为解密。
- hi_cipher_aes_config与hi_cipher_aes_destroy_config使用互斥锁机制，必须配套使用。

2.5.4 编程示例

AES Config参数说明：

```
typedef struct {  
    hi_u32 key[AES_MAX_KEY_IN_WORD]; // 配置密钥，AES_MAX_KEY_IN_WORD = 16  
    hi_u32 iv[AES_IV_LEN_IN_WORD]; // 配置 iv，AES_IV_LEN_IN_WORD = 4  
    hi_bool random_en; // 是否使能随机延时  
    hi_u8 resv[3]; // 3 byte保留字段  
    hi_cipher_aes_key_from key_from; // 配置密钥来源  
    hi_cipher_aes_work_mode work_mode; // 配置工作模式：ECB/CBC/CTR/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;
```

AES-XTS加解密参数说明：



```
typedef struct {  
    hi_char *key1;        // 数据密钥，对明文/密文进行AES-enc/dec过程使用的密钥；  
    hi_char *key2;        // 调整值密钥，对tweak进行AES-enc过程使用的密钥；  
    hi_u32 klen;          // 密钥长度，key1 key2密钥长度都是klen；  
    hi_char *data_unit;   // 128-bit数据块在数据单元中的位置值，配置到API中的 iv；  
    hi_u32 data_unit_len; // 位置值长度，同AES iv长度；  
    hi_char *plaintext;   // 明文数据；  
    hi_char *ciphertext;  // 加密数据；  
    hi_u32 length;        // 数据长度；  
    hi_char *tweak;       // XTS调整值，一般使用加密数据的逻辑位置，同data_unit；  
    hi_u32 tweak_len;     // 调整值长度；  
} aes_xts_test;
```

加解密数据长度：

0 < len < 1M bytes;

AES-XTS 加解密示例：

```
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 概述

RSA（非对称加密算法）采用私钥对信息进行签名、公钥验签的方式可以实现信息传输的完整性验证、发送者的身份认证、防止交易中的抵赖发生。

- 密钥对：公钥（n, e）、私钥（n, d）。RSA所需密钥对由用户自己生成。
- 密钥长度：支持RSA-2048（256byte）和RSA-4096（512byte）。

2.6.2 开发流程

使用场景

用于文件签名。

功能

RSA模块提供的接口如表2-6所示。

表 2-6 RSA 模块接口描述

接口名称	描述
hi_cipher_rsa_sign_hash	RSA签名输出签名结果。
hi_cipher_rsa_verify_hash	RSA签名结果校验。

开发流程

使用RSA签名的流程如下：

- 步骤1** 调用hi_cipher_rsa_sign_hash，进行RSA签名。
- 步骤2** 调用hi_cipher_rsa_verify_hash，对RSA签名结果进行校验。
- 结束

2.6.3 注意事项

- 待校验的HASH数据的长度为32byte有效数据。
- 输出的签名结果长度为klen。
- 枚举hi_cipher_rsa_sign_scheme类型中选择对应的PKCS编码标准，其中枚举包含PKCS编码标准和SHA-256两层含义，用户可以有RSASSA_PKCS1_V15和RSASSA_PKCS1_PSS两种选择，HASH计算必须是基于SHA-256算法实现。



2.6.4 编程示例

使用RSA签名及验签示例：

```
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 概述

ECC（椭圆加密算法）功能与RSA算法相同，相比于基于RSA密码学的数字签名算法，ECC在计算数字签名时所需的公钥长度可以大大缩短。

2.7.2 开发流程

使用场景

用于数据算法签名。

功能

ECC模块提供的接口如表2-7所示。

表 2-7 ECC 模块接口描述

接口名称	描述
hi_cipher_ecc_sign_hash	ECC签名输出签名结果。
hi_cipher_ecc_verify_hash	ECC签名结果校验。

开发流程

使用ECC签名的流程如下：

- 步骤1 调用hi_cipher_ecc_sign_hash，进行ECC签名。
- 步骤2 调用hi_cipher_ecc_verify_hash，对ECC签名结果进行校验。
- 结束

2.7.3 注意事项

ECC椭圆曲线参数长度不足Key的大小时前面补0。

2.7.4 编程示例

ECC签名及验签示例：

```
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 *ecd_h_b[] = { ECDH_SECP256K1_B, ECDH_SECP256R1_B,
ECDH_BRAIN_POOL_R1_B };
const hi_char *ecd_h_gx[] = { ECDH_SECP256K1_GX, ECDH_SECP256R1_GX,
ECDH_BRAIN_POOL_R1_GX };
const hi_char *ecd_h_gy[] = { ECDH_SECP256K1_GY, ECDH_SECP256R1_GY,
ECDH_BRAIN_POOL_R1_GY };
const hi_char *ecd_h_n[] = { ECDH_SECP256K1_N, ECDH_SECP256R1_N,
ECDH_BRAIN_POOL_R1_N };
const hi_u32 ecd_h_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 加解密

3.1 概述

3.2 使用指导

3.3 注意事项

3.1 概述

通过二级密钥加密架构为用户提供关键代码的加密保护，避免用户关键代码被读取、反编译后盗取。

3.2 使用指导

说明

本节指导用户如何实现Flash加密及该功能的调试。

3.2.1 依赖条件

EFUSE

EFUSE中FLASH_ENCPT_CFG用于表示加密锁死是否开启，FLASH_ENCPT_CNT用于表示Flash是否已加密，ROOT_KEY字段为根密钥的root_key值，根密钥盐值由硬件随机生成。

- FLASH_ENCPT_CFG: 1bit，是否开启加密锁死标志。
 - 0: 未开启。
 - 1: 已开启，开启表示使用FLASH_ENCPT_CNT字段来标记Flash是否已加密。
建议：研发使用写0，量产芯片版本写1。
- FLASH_ENCPT_CNT: 共6个FLASH_ENCPT_CNT字段，每个字段2bit（此2bit默认值均为0）。
 - 当字段中bit[0]置1且bit[1]为0: Flash烧写后未加密。



- 当字段中bit[1]置1：Flash已加密。

在量产芯片版本中，如果通过menuconfig开启了Flash加密功能，开启Flash加密的menuconfig配置请参见3.2.3 实现方法，芯片在HiBurn产线命令行版本工具烧写程序时自动烧写FLASH_ENCPT_CNT字段bit[0]，烧写程序完成后对程序固件进行加密，加密完成后自动烧写bit[1]。

- ROOT_KEY：256bit，产线生产时软件写入，同一类产品的root_key值相同。ROOT_KEY字段软件不可读，派生根密钥时直接硬连接读取，ROOT_KEY字段默认为全0，需在烧写Flash之前写入。

说明

EFUSE的使用请参见《Hi3861V100 / Hi3861LV100 EFUSE 使用指南》。

工厂区 NV

由于EFUSE仅有6个FLASH_ENCPT_CNT字段，因此当FLASH_ENCPT_CFG置1且当前待烧写可执行文件开启Flash加密功能时，Flash仅有6次正式烧写的机会，烧写失败不计入次数限制。为了支持用户的产品开发，提供工厂NV项0x8来模拟FLASH_ENCPT_CNT，用于调试加密功能。当FLASH_ENCPT_CFG值为0时，读取该NV项中的flash_encpt_cnt，如果此值为0，说明固件没有加密，此时需要对固件进行加密，加密完成后将flash_encpt_cnt的值写为1；如果flash_encpt_cnt值为1，说明固件已经加密。flash_encpt_cnt默认值为0。

NV项0x8设计如下：

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

说明

NV的使用请参见《Hi3861V100 / Hi3861LV100 NV 使用指南》。

后续可更新版本，通过修改flashboot和loaderboot开源代码，支持不限制烧写次数为6次，且后续该NV项可删除。

加密密钥

如果开启Flash加密，则在编译时同时会对升级文件进行加密，配置升级文件加密使用的密钥，需要在“tools/sign_tool”下增加upg_aes_key.txt，密钥格式举例如下：

```
[E]:15A146973144B9C52FEC...;EFUSE_DATA
[C]:C783FFE88FECD7...;CPU_DATA
[I]:ECD749AB307CC...;IV_DATA
```

其中：

- EFUSE_DATA：EFUSE中存储的root_key，长度为32byte，为保证安全，需要使用真随机数；由于EFUSE存储的数据不能更改，所以EFUSE_DATA确定后不能再更改。
- CPU_DATA：构成盐值的一部分，长度为16byte，为保证安全，需要使用真随机数。
- IV_DATA：初始化向量，长度为16byte，为保证安全，需要使用真随机数。



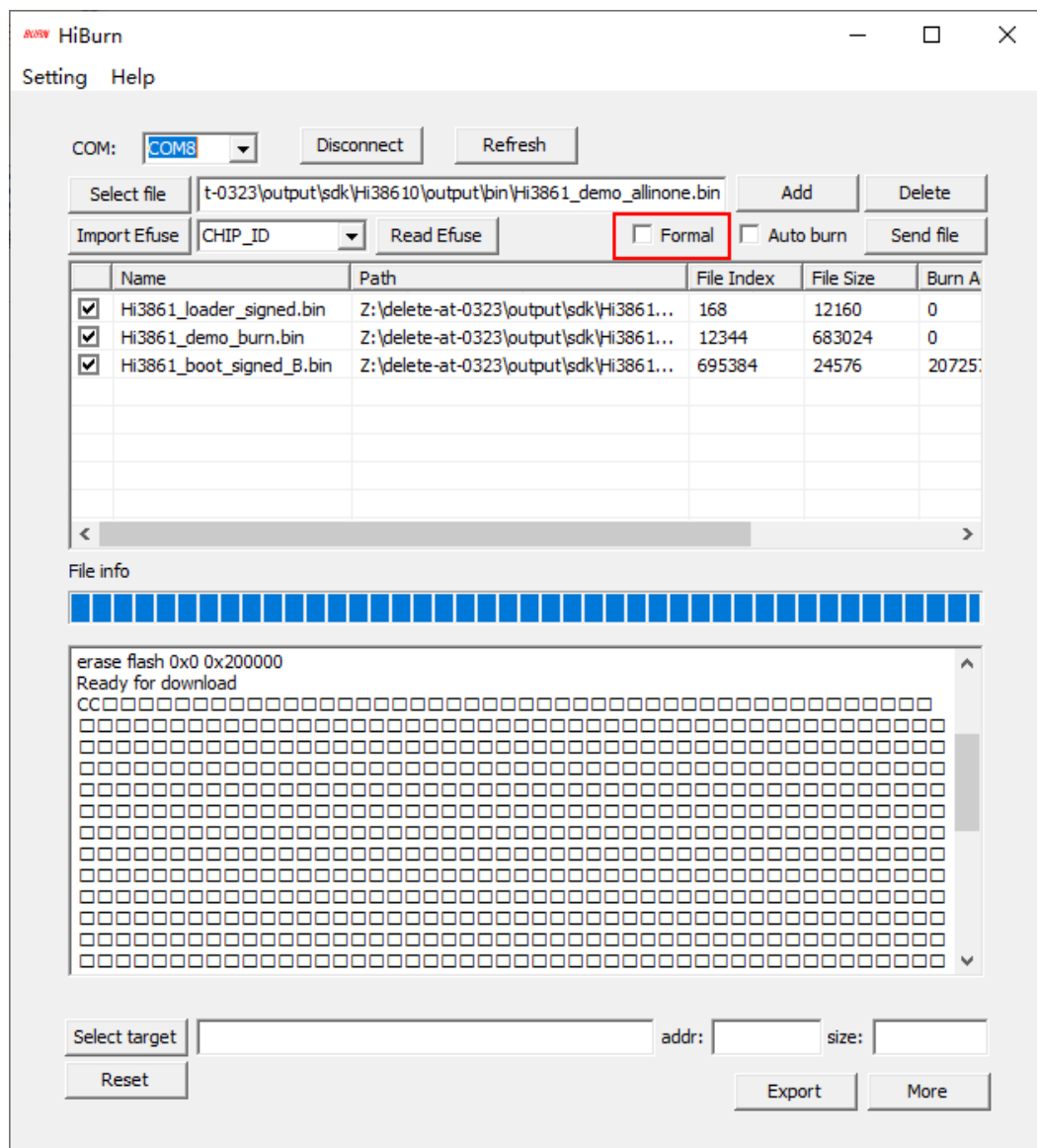
须知

Flash加密功能开启，必须在烧写Flash之前将upg_aes_key.txt中的EFUSE_DATA烧写到EFUSE中的root_key字段中，如果单板EFUSE的root_key与upg_aes_key.txt文件中的EFUSE_DATA不一致，将会导致升级失败。

3.2.2 烧录工具

- HiBurn界面版本：Flash烧写时可以选择是否烧写EFUSE的FLASH_ENCPT_CNT字段，选中“Formal”表示烧写Flash前烧写FLASH_ENCPT_CNT字段bit[0]；否则，不烧写FLASH_ENCPT_CNT字段。
- HiBurn产线命令行版本：默认携带Formal参数，无需额外配置。

图 3-1 HiBurn 界面示意图





3.2.3 实现方法

HiBurn 产线命令行版本

HiBurn产线命令行版本实现Flash加密的步骤如下：

步骤1 在SDK根目录下执行“sh build.sh menuconfig”。

步骤2 选择“Security Settings”菜单：

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

步骤3 在“Security Settings”中选中“FLASH ENCRYPT support”选项（即完成了Flash加解密功能的menuconfig配置）：

```
(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
```

步骤4 将用户的代码加入到加密代码段，需要在代码中使用宏“CRYPTO_RAM_TEXT_SECTION”修饰用户关键函数，使用该宏修饰的函数如果menuconfig中的“FLASH ENCRYPT support”选项关闭，则会正常链接到普通的代码段中。例如：

```
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);
}
```

步骤5 工厂烧写Flash之前烧写EFUSE的FLASH_ENCPT_CFG字段为1。

工厂烧写EFUSE方法请参见《Hi3861V100 / Hi3861LV100 产线工装 用户指南》中的软件烧写步骤。

步骤6 编译镜像完成后通过命令行工具烧写即可，程序能够正常引导启动，加密段中的代码能够实现预期功能。

----结束

HiBurn 界面版本

HiBurn界面版本实现Flash加密的步骤如下：

步骤1 在SDK根目录下执行“sh build.sh menuconfig”。

步骤2 选择“Security Settings”菜单：



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

步骤3 在“Security Settings”中选中“FLASH ENCRYPT support”选项（即完成了Flash加解密功能的menuconfig配置）：

```
(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
```

步骤4 将用户的代码加入到指定的代码段，需要在代码中使用宏“CRYPTO_RAM_TEXT_SECTION”修饰用户关键函数，使用该宏修饰的函数如果menuconfig中的“FLASH ENCRYPT support”选项关闭，则会正常链接到普通的代码段中。例如：

```
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);
}
```

步骤5 确保工厂区NV 0x8项的flash_encpt_cnt值为0，编译后通过HiBurn界面版本工具烧录程序，程序能够正常引导启动，加密段中的代码能够实现预期功能。

----结束

3.3 注意事项

- HiBurn界面版本工具与HiBurn产线命令行版本工具不同，当EFUSE 的FLASH_ENCPT_CFG置1后：
 - HiBurn界面版本工具：用户可以通过是否选中“Formal”选项来决定是否烧写EFUSE的FLASH_ENCPT_CNT字段。
 - HiBurn产线命令行版本工具：默认携带Formal参数，无需额外配置。
- 开启Flash加密功能，且EFUSE的FLASH_ENCPT_CFG字段为1，此时烧写Flash，如果HiBurn界面版本工具不选中“Formal”选项，将会导致引导启动失败。
- OTA升级时，不支持Flash加密开启的flashboot和Flash加密关闭的flashboot交替升级，也不支持Flash加密开启的kernel和Flash加密关闭的kernel交替升级。
- 因为EFUSE的一次可编程特性，EFUSE的烧写是不可逆的，而Hi3861芯片的EFUSE仅有6个FLASH_ENCPT_CNT字段，所以开启Flash加密功能时，HiBurn产线命令行版本工具仅有6次烧写Flash的机会。
- 开启Flash加密功能，烧写Flash之前如果没有烧写EFUSE的ROOT_KEY字段及LOCK_ROOT_KEY锁定位置，将会烧写失败。



- 后续可更新版本，通过修改flashboot和loaderboot开源代码，支持不限制FLASH烧写次数为6次。



4 用户数据安全存储

- 4.1 概述
- 4.2 根密钥
- 4.3 安全存储设计
- 4.4 注意事项
- 4.5 编程示例

4.1 概述

用户记录账号、密码等保密信息时，需要保证这些数据的存储安全，所以要求这些数据在存储之前进行加密保护。Hi3861V100/Hi3861LV100可以提供基于efuse root_key的用户数据加密保护。

该存储方法的核心思想是：采用EFUSE中的root_key，派生出根密钥，使用该派生出的根密钥，对用户数据进行加解密处理。其中EFUSE中的root_key软件不可读取。

4.2 根密钥

根密钥通过KDF函数派生，每次需要时恢复。KDF模块介绍请参见“[2.3 KDF](#)”，本小节主要介绍KDF函数的参数配置控制结构体，该结构体中存放了KDF函数所需入参以及函数执行后的出参。

- KDF模式枚举值定义：

```
typedef enum {  
    HI_CIPHER_SSS_KDF_KEY_DEVICE = 0x0, /* KDF设备密钥派生：root_key由用户指定。*/  
    HI_CIPHER_SSS_KDF_KEY_STORAGE, /* KDF存储密钥派生：root_key由EFUSE指定。*/  
    HI_CIPHER_SSS_KDF_KEY_MAX,  
    HI_CIPHER_SSS_KDF_KEY_INVALID = 0xFFFFFFFF,  
} hi_cipher_kdf_mode;
```
- KDF函数的参数配置控制结构体定义：

```
typedef struct {  
    const hi_u8 *salt; /* 用于派生密钥的盐值，输入参数。*/  
    hi_u32 salt_len; /* 盐值长度，输入参数：  
    * KDF模式为HI_CIPHER_SSS_KDF_KEY_DEVICE时长度是16字节；  
    * KDF模式为HI_CIPHER_SSS_KDF_KEY_STORAGE时长度是32字节。  
    */
```




```
hi_u8 key[32]; /* KDF模式为HI_CIPHER_SSS_KDF_KEY_DEVICE时存放KDF函数所需的
root_key, 输入参数。 */
hi_cipher_kdf_mode kdf_mode; /* KDF模式, 输入参数。 */
hi_u32 kdf_cnt; /* KDF迭代次数, 建议迭代次数不小于10000次,
* 有性能要求的场景不应小于1000次, 并且不大于0xffff次, 输入参数。 */
hi_u8 result[32]; /* KDF模式为HI_CIPHER_SSS_KDF_KEY_DEVICE时, 存储派生得到的密钥,
输出参数。 */
} hi_cipher_kdf_ctrl;
```

参数配置控制结构体关键参数说明:

- 盐值: 盐值必须由用户显式提供, KDF不同模式对应不同长度的盐值。
 - KDF模式为HI_CIPHER_SSS_KDF_KEY_DEVICE时, 需要16byte长度的盐值。
 - KDF模式为HI_CIPHER_SSS_KDF_KEY_STORAGE时, 需要32byte长度的盐值。
- root_key: 根据不同的KDF模式获取方式不同。
 - KDF模式为HI_CIPHER_SSS_KDF_KEY_DEVICE时, 需要用户将root_key显式复制到参数配置控制结构体的key成员中。
 - KDF模式为HI_CIPHER_SSS_KDF_KEY_STORAGE时, KDF模块自动硬连接EFUSE的root_key字段。
- result: 输出参数result对于不同的KDF模式意义不同。
 - KDF模式为HI_CIPHER_SSS_KDF_KEY_DEVICE时, 存储派生得到的密钥。
 - KDF模式为HI_CIPHER_SSS_KDF_KEY_STORAGE, result成员没有意义, 派生得到的密钥直接存储在KDF模块中, 软件不可读。
- kdf_mode: 建议派生根密钥使用HI_CIPHER_SSS_KDF_KEY_STORAGE模式, 派生工作密钥使用HI_CIPHER_SSS_KDF_KEY_DEVICE模式。

根密钥的root_key及盐值获取来源:

- root_key: 256bit, 用户量产时产线阶段写入EFUSE的root_key字段, 同一类产品的root_key值应该相同。
- 盐值: 128bit, 真随机数接口生成, 例如: Flash加密特性根密钥的盐值通过硬件随机生成。

说明

EFUSE root_key字段的详细信息请参见《Hi3861V100 / Hi3861LV100 EFUSE 使用指南》。

须知

根密钥生成后保存在KDF模块中, KDF模块只保存最新派生的一组根密钥, 建议在使用根密钥加密数据之前先使用KDF生成根密钥, 确保KDF模块最新保存的是用户所需的根密钥。

4.3 安全存储设计

根密钥加密用户数据需要使用AES模块 (AES模块的使用方式请参见“[2.5 AES](#)”)。AES通过根密钥加密数据时, 需要将密钥来源设置为HI_CIPHER_AES_KEY_FROM_KDF模式, 此时AES将直接从KDF模块读取根密钥。

AES算法参数配置控制结构体定义:



```
typedef struct {  
    hi_u32 key[16]; /* 密钥 */  
    hi_u32 iv[4]; /* 初始化向量 */  
    hi_bool random_en; /* 随机延时使能 */  
    hi_u8 resv[3]; /* 3byte保留字段 */  
    hi_cipher_aes_key_from key_from; /* 密钥来源，当来源选择HI_CIPHER_AES_KEY_FROM_KDF  
    时，从KDF模块读取密钥，不需要软件提供 */  
    hi_cipher_aes_work_mode work_mode; /* 工作模式，根密钥加密数据建议使用CBC模式 */  
    hi_cipher_aes_key_length key_len; /* 密钥长度，aes-ecb/cbc/ctr支持128/192/256位密钥，ccm仅  
    支持128位密钥，  
        * xts仅支持256/512位密钥。  
        */  
    hi_cipher_aes_ccm *ccm; /* ccm结构体 */  
} hi_cipher_aes_ctrl;
```

根密钥加密数据建议使用安全性较高的CBC模式，此加密模式还需要用户提供一个初始向量值，该向量值可通过硬件随机生成（即通过调用TRNG模块接口生成随机数），同时为了能够校验回读的用户数据的完整性，还需要字段用于存储用户数据的哈希计算结果，数据的哈希计算请参见“[2.4 HASH](#)”。

用户实际需要存储的数据结构如下：

```
typedef struct {  
    hi_u8 iv_nv[16]; /* 根密钥加密用户数据的初始向量值，在Flash中以明文存储 */  
    hi_u8 user_text[32]; /* 用户数据明文，根密钥加密后存储 */  
    hi_u8 content_sh256[32]; /* 以上两组数据256位哈希计算结果，根密钥加密后存储 */  
} hi_flash_crypto_content;
```

为了提升数据存储的安全性，建议对用户加密数据进行备份，当用户加密数据被破坏时，可以从数据备份区中加载用户加密数据。

用户数据安全存储步骤如下：

- 步骤1** 通过KDF派生根密钥，其中派生密钥使用的盐值需要使用真随机数生成，并存储到FLASH（如NV中）。
- 步骤2** 硬件随机生成根密钥加密用户数据的初始向量值。
- 步骤3** 计算初始向量值和用户数据明文的256bit哈希值，结果存放在content_sh256中。
- 步骤4** 通过AES接口加密用户数据和256bit哈希值。
- 步骤5** 加密数据存储到用户指定的Flash区域中。
- 步骤6** 加密数据存储到数据备份区。

----结束

用户数据安全读取步骤如下：

- 步骤1** 通过KDF派生根密钥，其中派生密钥使用的盐值，需要从FLASH中获取。
- 步骤2** 从指定存放用户加密数据的Flash区域读取加密数据。
- 步骤3** 通过AES接口解密用户数据和256bit哈希值。
- 步骤4** 通过HASH接口计算初始向量值和用户数据明文的256bit哈希值，与content_sh256中的数据比较校验数据的完整性。
- 步骤5** [步骤4](#)校验成功，则user_text中的数据即为用户的原始数据，完成数据读取；否则执行[步骤6](#)。



步骤6 从数据备份区读取加密数据，重复**步骤3** ~ **步骤4**。

步骤7 **步骤4**校验成功，则user_text中的数据即为用户的原始数据，完成数据读取；否则数据安全读取失败。

----结束

4.4 注意事项

- 整个加解密流程中根密钥软件不可读。
- KDF派生密钥使用的盐值，也应当随机生成。
- AES模块CBC加密模式中数据长度要求16byte对齐。
- KDF模块只保存最新派生的一组根密钥，使用根密钥之前先使用KDF生成根密钥，确保最新保存的是用户所需的根密钥。
- KDF迭代次数默认推荐≥10000次，如果存在性能要求，迭代次数至少1000次。
- 为了保证安全，暂存用户关键信息的内存在释放之前先进行清零。

4.5 编程示例

根密钥加密存储用户数据示例：

```
/*
 * 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_CRYPT, (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_CRYPT_BLOCK_SIZE 16
/* 此处指定的用户加密数据存储地址仅作为示例，用户需根据实际情况自行指定存储地址 */
#define CRYPTO_KEY_STORE_ADDR 0x001E1000
#define CRYPTO_KEY_BACKUP_ADDR 0x001E2000

typedef struct {
```



```
    hi_u8 iv_nv[IV_BYTE_LENGTH];    /* 根密钥加密用户数据的初始向量值，在Flash中以明文存
    储 */
    hi_u8 work_text[KEY_BYTE_LENGTH]; /* 用户数据明文，根密钥加密后存储 */
    hi_u8 content_sh256[SHA_256_LENGTH]; /* 以上2组明文数据256位哈希计算结果，根密钥加密
    后存储 */
} hi_flash_crypto_content;

typedef enum {
    CRYPTO_WORKKEY_KERNEL = 0x1,
    CRYPTO_WORKKEY_KERNEL_BACKUP = 0x2,
    CRYPTO_WORKKEY_KERNEL_BOTH = 0x3,
} crypto_workkey_partition;
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] = {0};
    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;
    }
    /* 示例代码省去了KDF派生密钥时获取盐值步骤，实际应用时，盐值应当随机生成并存储到
    FLASH,如NV中，再次使用时，从FLASH中读取 */
    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; /* 硬连接Efuse的root_key字段读取HUK
    值，根密钥存储在KDF模块中 */
    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_CRYPT, 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_CRYPT0, content_size);
    if (content_tmp == HI_NULL) {
        return HI_ERR_FLASH_CRYPT0_MALLOC_FAIL;
    }

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

    /* 根密钥加密初始化向量值和用户数据 */
    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_CRYPT0_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_CRYPT0_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 content_size = sizeof(hi_flash_crypto_content);

    hi_flash_crypto_content *new_content = (hi_flash_crypto_content
```



```
*)hi_malloc(HI_MOD_ID_CRYPT, sontent_size);
    if (new_content == HI_NULL) {
        return HI_ERR_FLASH_CRYPT_PREPARE_ERR;
    }

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

    /* 派生根密钥 */
    ret = crypto_prepare();
    if (ret != HI_ERR_SUCCESS) {
        ret = HI_ERR_FLASH_CRYPT_PREPARE_ERR;
    }

    /* 计算初始向量值和用户数据的256位哈希值 */
    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;
    }

    /* 从Flash密钥分区或备份密钥区加载密钥 */
    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;
}
```




5 TEE HUKS

5.1 概述

5.2 开发流程

5.3 注意事项

5.4 编程示例

5.1 概述

TEE (Trusted Execution Environment) HUKS (Huawei KeyStore) 是一个轻量级的可信执行环境 (以下简称huks)，提供了数据加解密、签名验签等接口。相比于上述硬件加解密接口，其优点在于执行环境完全独立，同时用户无需接触明文密钥，密钥的加解密处理，均在HUKS内部完成。

5.2 开发流程

使用场景

基于可信执行环境的数据加解密、密钥对生成、签名验签、HMAC计算等。

功能

HUKS模块提供的接口如下表5-1所示。

表 5-1 HUKS 模块接口描述

接口名称	描述
hks_init	huks初始化，签名和验签需要。
hks_destroy	huks销毁。
hks_refresh_key_info	huks刷新key信息，签名和验签依赖； huks_init失败时，调用该接口。



接口名称	描述
hks_generate_key	huks生成对称密钥(公钥+私钥)，加密存储到内部文件系统中； 仅支持ED25519。
hks_import_public_key	将对端设备的公钥导入到huks文件系统中。
hks_export_public_key	导出公钥明文。
hks_get_pub_key_alias_list	导出公钥别名列表：只能导出通过 hks_import_public_key导入的公钥别名列表。
hks_delete_key	删除密钥。
hks_get_key_param	导出密钥属性。
hks_is_key_exist	判断密钥是否存在。
hks_asymmetric_sign	私钥签名。
hks_asymmetric_verify	公钥验签。
hks_generate_symmetric_key	生成对称密钥； 仅支持AES-128/AES-192/AES-256密钥生成。
hks_symmetric_encrypt	AES加密； 支持GCM CCM CBC模式。
hks_symmetric_decrypt	AES解密； 支持GCM CCM CBC模式。
hks_generate_random	随机数生成，等价于： hi_cipher_trng_get_random_bytes。
hks_hmac	HMAC计算，基于SHA256或SHA512。
hks_hash	SHA256或SHA512计算，SHA256等价于： hi_cipher_hash_sha256。

开发流程

- 通过menuconfig使能TEE HUKS support，并且huks签名/验签相关接口，依赖开启文件系统。
- 通过meunconfig使能TEE HUKS demo support，开启huks相关的DEMO AT命令，详见《Hi3861V100 / Hi3861LV100 AT命令 使用指南》中“安全存储相关AT指令”章节。
- 签名验签相关接口，需要先调用hks_init，当hks_init失败时，需要调用hks_refresh_key_info。



5.3 注意事项

- huks签名/验签相关接口，依赖开启文件系统。
- 产测模式下，如果要使用上述huks相关接口，需要确保platform\system\cfg\saf_cfg.c中，hi_tee_irq_handler执行的是hks_handle_secure_call函数。仅在HILINK PKI预置功能场景下，才执行hks_handle_secure_pki_provision函数。

5.4 编程示例

huks加解密相关编程示例，参见《Hi3861V100 / Hi3861LV100 AT命令 使用指南》中“安全存储相关AT指令”章节。其中：

AT+GCONNKEY、AT+SCONN、AT+SRCONN 实现了通过huks接口，加解密STA连接AP所需要的参数；

AT+GCERTKEY、AT+CERTENC、AT+CERTDEC实现了通过huks接口，加解密FLASH上预置的证书和密钥文件。

huks签名和验签编程示例如下所示。

```
/* 生成密钥对 */
```

```
int app_hks_generate_asymmetric_key(void)
{
    hi_char test_file_name[] = "keyalias1";
    hi_char test_file_name1[] = "key_auth_id1";
    struct hks_blob key_alias;
    hi_u32 status;
    key_alias.type = HKS_BLOB_TYPE_ALIAS;
    key_alias.data = (uint8_t*)test_file_name;
    key_alias.size = sizeof(test_file_name);
    struct hks_key_param key_param;
    key_param.key_auth_id.data = (uint8_t*)test_file_name1;
    key_param.key_auth_id.size = sizeof(test_file_name1);
    key_param.key_auth_id.type = HKS_BLOB_TYPE_AUTH_ID;
    key_param.key_type = HKS_KEY_TYPE_EDDSA_KEYPAIR_ED25519;
    key_param.key_len = 0;
    key_param.key_usage = 0;
    key_param.key_pad = 0;
    status = hks_generate_key(&key_alias, &key_param);
}
```



```
if (status == 0)
printf("\r\n%s SUCCESS:status = %d\r\n", __func__, status);
else
printf("\r\n%s FAIL: status = %d\r\n", __func__, status);
return status;
}
/* 删除密钥 */
hi_void app_hks_delete_asymmetric_key(void)
{
hi_u32 status;
hi_char test_file_name[] = "keyalias1";
struct hks_blob key_alias = {0};
// Init vairables
key_alias.data = (uint8_t *)test_file_name;
key_alias.size = sizeof(test_file_name);
key_alias.type = HKS_BLOB_TYPE_ALIAS;
// delete key
status = hks_delete_key(&key_alias);
if (status == 0)
printf("\r\n%s SUCCESS:status = %d\r\n", __func__, status);
else {
printf("\r\n%s FAIL: status = %d\r\n", __func__, status);
}
}
#define HASH_LEN 32
#define ASYMMETRIC_KEY_LEN 32
/* 签名和 验签 */
hi_u32 app_hks_asymmetric_sign_verify(void)
{
/* hks init */
int ret = hks_init();
if (ret != HI_ERR_SUCCESS) {
ret = hks_refresh_key_info();
```



```
printf("refresh key_info ret:%d\r\n", ret);
}
if (ret != HI_ERR_SUCCESS) {
printf("refresh key_info fail:%d\r\n", ret);
return HI_ERR_FAILURE;
}
int status;
hi_char test_file_name[] = "keyalias1";
struct hks_blob key_alias = {0};
// Init vairables
key_alias.data = (uint8_t *)test_file_name;
key_alias.size = sizeof(test_file_name);
key_alias.type = HKS_BLOB_TYPE_ALIAS;
hi_u8 hash1[HASH_LEN];
struct hks_blob hash;
hash.data = (uint8_t*)hash1;
hash.size = sizeof(hash1);
hi_u32 alg = HKS_ALG_HASH_SHA_256;
struct hks_blob src;
src.data = (uint8_t *) "123456";
src.size = 0x6;
status = hks_hash(alg, &src, &hash);
struct hks_key_param key_param = {0};
key_param.key_type = HKS_KEY_TYPE_EDDSA_KEYPAIR_ED25519;
key_param.key_usage = HKS_KEY_USAGE_SIGN | HKS_KEY_USAGE_VERIFY;
key_param.key_mode = HKS_ALG_GCM;
key_param.key_len = ASYMMETRIC_KEY_LEN;
key_param.key_auth_id.type = HKS_BLOB_TYPE_AUTH_ID;
key_param.key_auth_id.data = (uint8_t *)test_file_name;
key_param.key_auth_id.size = sizeof(test_file_name);
ret = app_hks_generate_asymmetric_key();
if (ret != 0) {
return HI_ERR_FAILURE;
```



```
}  
  
struct hks_blob signature = {0};  
  
if (hks_set_blob(1, 0x40, HI_NULL, &signature) != HI_ERR_SUCCESS) {  
    (void)app_hks_delete_asymmetric_key();  
    return HI_ERR_FAILURE;  
}  
  
status = hks_asymmetric_sign(&key_alias, &key_param, &hash, &signature);  
if (status == 0) {  
    printf("\r\n%s asymmetric_sign_SUCCESS:status = %d\r\n", __func__, status);  
} else {  
    printf("\r\n%s asymmetric_sign_FAIL: status = %d\r\n", __func__, status);  
    free(signature.data);  
    (void)app_hks_delete_asymmetric_key();  
    return HI_ERR_FAILURE;  
}  
  
status = hks_asymmetric_verify(&key_alias, &key_param, &hash, &signature);  
if (status == 0) {  
    printf("\r\n%s asymmetric_verify_SUCCESS:status = %d\r\n", __func__, status);  
} else {  
    printf("\r\n%s asymmetric_verify_FAIL: status = %d\r\n", __func__, status);  
    free(signature.data);  
    (void)app_hks_delete_asymmetric_key();  
    return HI_ERR_FAILURE;  
}  
  
free(signature.data);  
  
(void)app_hks_delete_asymmetric_key();  
  
return HI_ERR_SUCCESS;  
}
```