# Sirius Security系统设计文档

v4.0

xiaohui.zhu

目录

[Sirius Security系统设计文档 1](#_Toc482207815)

[1. 竞争分析（占坑，后续再补） 3](#_Toc482207816)

[2. Sirius Security总体需求 3](#_Toc482207817)

[3. optee结构介绍 4](#_Toc482207818)

[4. Sirius Secure Boot 设计方案 6](#_Toc482207819)

[1. Uboot格式及编译过程 6](#_Toc482207820)

[2. Optee分区格式 7](#_Toc482207821)

[3. Kernel分区格式 8](#_Toc482207822)

[4. Emmc分区格式 9](#_Toc482207823)

[5. key管理方案 10](#_Toc482207824)

[6. Secure boot过程 13](#_Toc482207825)

[7. 量产/升级方案 15](#_Toc482207826)

[8. Sirius Secure Storage设计 16](#_Toc482207827)

[9. Sirius Version Control设计 19](#_Toc482207828)

[10. Sirius Secure Debug设计 20](#_Toc482207829)

[11. 其他 20](#_Toc482207830)

[5. 参考文档： 20](#_Toc482207831)

## 竞争分析（占坑，后续再补）

联芯 ?

高通 ?

海思 ?

## Sirius Security总体需求

1. 需要保护bootloader（u-boot）、linux kernel、TrustZone image、DSP firmware以及其他用户数据（若有，比如音视频录像文件等）。
2. 对应的文件形式：uboot.bin、vmlinux、TrustZone Image以binary形式存放在SPI NOR或者eMMC或USB中，DSP firmware及用户数据以文件形式放在文件系统中。
3. 用户可以使用自己的key对上述文件进行加密签名，Artosyn提供tool来修改OTP，从而支持用户单独保护。
4. OTP中Key的保存及管理。

Root key每个客户自管理，生产时烧录自己的root key，同时考虑以下几种方案：

* 方案A，OTP中保存一组root key，用于验签TrustZone OS，后续u-boot/Kernel/files等以chain的方式在header中保存下一级的Key
* 方案B，OTP中保存一组root key，其他key采用derive方式，由各级image通过算法生成，这要求在生成image时使用相同的生成算法；
* 方案C，OTP中保存各级image所需要的Key组合，所有key在manufacture时由产测软件随机生成后按照规定好的index写入OTP，并以此Key组合对所有image重新进行加密和签名。

Sirius集成了tRoot/SPAcc/TRng IP以及ARM TrustZone，用于安全方面的应用。

tRoot: 安全启动模块

SPAcc: AES加密算法硬件加速模块

TRNG: 随机数发生器

PKA: RSA加密算法硬件加速

TrustZone: ARM安全IP

## optee结构介绍

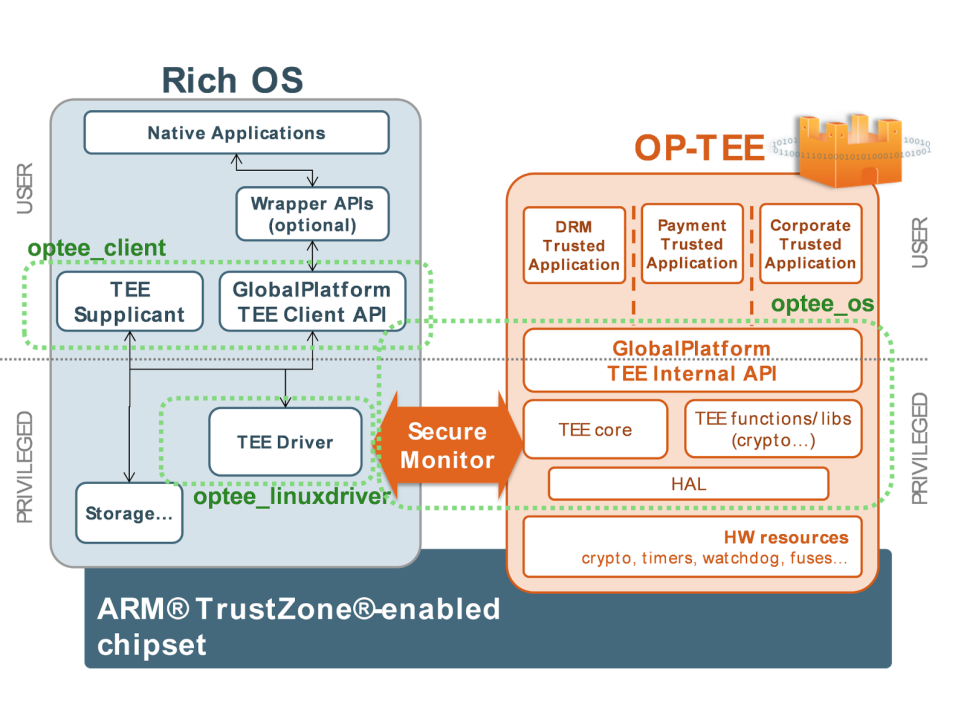


图 optee工作原理示意

如上图安全环境下系统由Rich Os(如Linux，非安全的系统)和optee（安全环境）组成，两个环境独立运行并通过Secure monitor进行切换, Secure monitor保存了两组环境的contex，并根据要求选择一组切换。

optee由optee os和TA（Trusted Application）组成，Optee实现了基本的OS功能（如内存管理，文件系统， 加解密算法等），并提供系统调用给TA用，不同TA实现不同功能，RichOS发送给optee的请求都是由TA完成。

Armv7和ARMv8的Secure monitor实现方式不同，如下图：

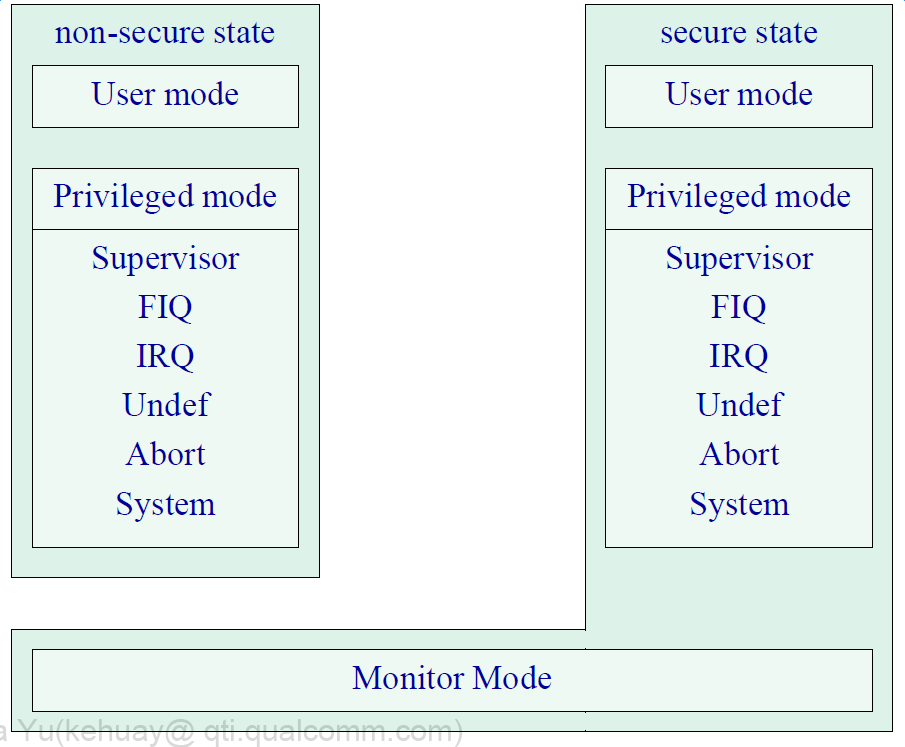
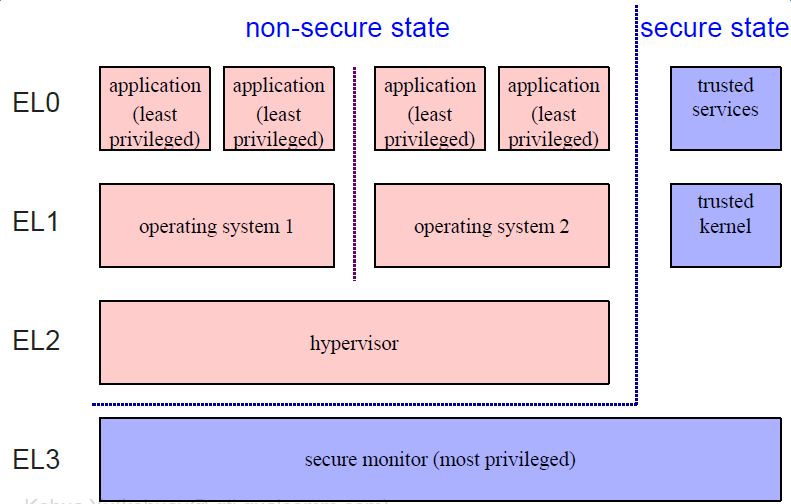
 

图 ARMv7/v8的权限结构

根据ARM官方文档，ARM v7下，Secure Monitor模式用于运行Secure Monitor软件。ARMv7中有一个寄存器MVBAR（在cp15中： 操作方法如下：

adr r1, \_monitor\_vectors

mcr p15, 0, r1, c12, c0, 1 @ set MVBAR to secure vectors

），

MVBAR用于存储Monitor Vector，Monitor Vector类似于中断向量表，smc命令或者安全中断发生后，系统转到Secure monitor模式，并从Monitor Vector选择对应的处理向量并运行，Monitor Vector把运行权限转交给optee，optee操作完成后通过smc再次进入Secure monitor然后返回非安全模式。

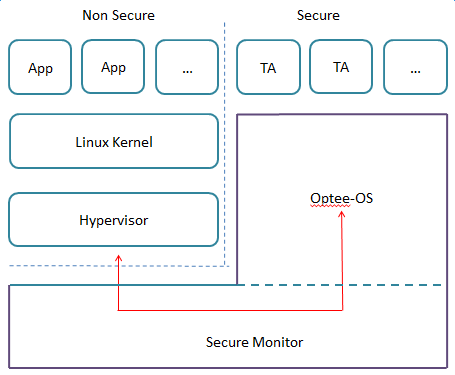
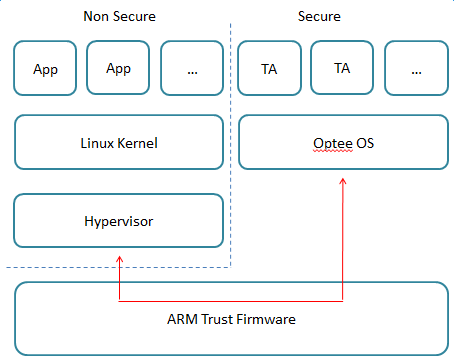
 

图 基于ARMv7/ARMv8的Secure/Non Secure切换方式

因此基于ARMv7的Optee没有arm trusted firmware，Secure Monitor包含在Optee-Os里面：

UNWIND( .cantunwind)

b . /\* Reset \*/

b . /\* Undefined instruction \*/

b sm\_smc\_entry /\* Secure monitor call \*/

b . /\* Prefetch abort \*/

b . /\* Data abort \*/

b . /\* Reserved \*/

b . /\* IRQ \*/

b sm\_fiq\_entry /\* FIQ \*/

UNWIND( .fnend)

END\_FUNC sm\_vect\_table

/\* Set monitor vector (MVBAR) \*/

ldr r0, =sm\_vect\_table

write\_mvbar r0

optee内存主要有三部分组成：

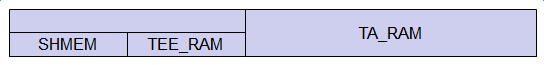


图 optee内存使用划分

SHMEM：用于安全和非安全环境之间数据传输的内存，SHMEM不是安全内存，只能用来传数据，不能用于安全操作（如存放key等）。

TEE\_RAM：Op-tee OS的运行内存，应设置为安全内存。

TA\_RAM: TA的运行内存，应设置为安全内存。

## Sirius Secure Boot 设计方案

### Uboot格式及编译过程

uboot中有spl作为bootloader，Sirius Secure方案采用扩展spl作为bootloader。编译方法如下图：

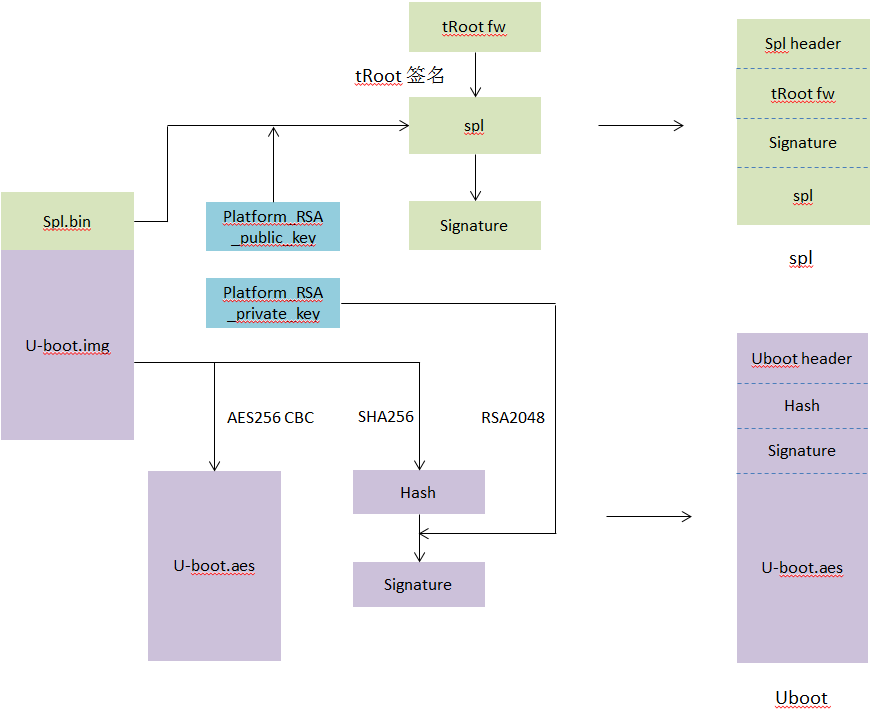


图 Secure boot编译Uboot过程

spl编译过程:

1 编译生成spl, spl包含RSA公钥

2 使用tRoot AIC tool对新的spl进行签名

3 把tRoot firmware，spl签名，spl image拼起来，并加header

uboot编译过程：

1 编译生成uboot.img

2 用SHA256计算 Uboot.img的Hash

3 使用Platform RSA private key对Hash签名

4 加密uboot.img(AES256 CBC)，得到uboot.aes

5 把Uboot.aes, hash和签名拼起来，并加header

spl header结构定义如下：

struct spl\_header

{

uint32\_t magic;

uint32\_t img\_type;

uint32\_t spl\_len;

uint32\_t troot\_fw\_len;

uint32\_t signature\_len;

uint32\_t reserved[3];

};

uboot header结构定义如下：

struct arto\_img\_header

{

uint32\_t magic;

uint32\_t img\_type;

uint32\_t img\_size;

uint32\_t algo;

uint16\_t hash\_size

uint16\_t sig\_len;

uint32\_t reserved[3];

};

### Optee分区格式

下图是optee编译后的bin档文件格式，虚线框内的是原始flow

optee后续可能会加配置文件（比如使用的OTP的key idx等），所以在tee后面附加一个配置文件，tee.bin加密后为tee.aes，并且把tee.bin的hash和签名也附在前面。最后加一个head2，标明tee中各个部分的长度，以便处理。

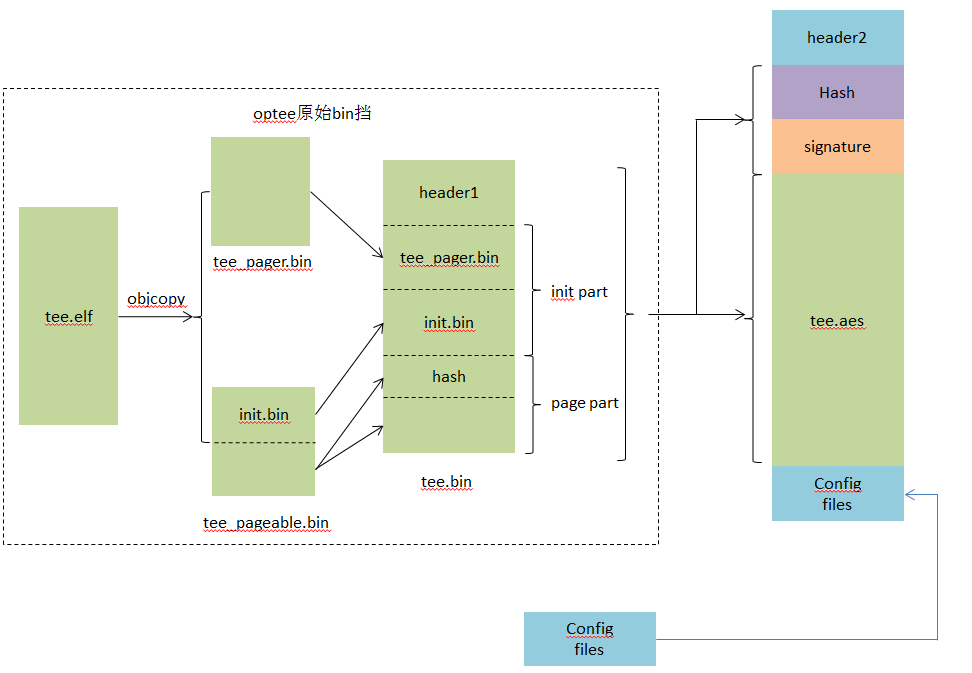


图 optee bin档格式(虚线框内是Optee原生编译流程)

其中head1结构如下：

struct optee\_header {

uint32\_t magic;

uint8\_t version;

uint8\_t arch;

uint16\_t flags;

uint32\_t init\_size;

uint32\_t init\_load\_addr\_hi;

uint32\_t init\_load\_addr\_lo;

uint32\_t init\_mem\_usage;

uint32\_t paged\_size;

};

header2的定义同uboot中arto\_img\_header的定义

### Kernel分区格式

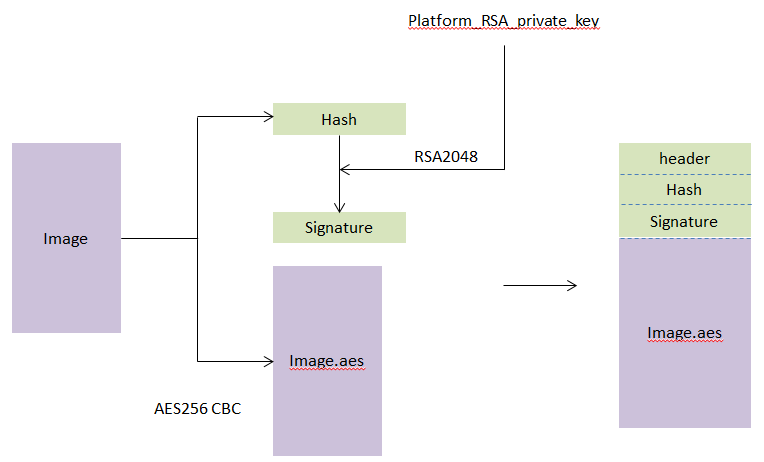


图 其他分区的格式

其他分区格式以及编译方式见上图：

1 生成对应分区的raw image

2 使用AES对raw image进行加密

3 对加密后的image计算Hash

4 对Hash签名

5 将加密后的image和Hash、签名拼接成新的image，并加header存放image和签名的长度， header的定义参考arto\_img\_header

6 把AES key保存在Emmc特定位置（keybank分区）或者AES key通过OTP root key派生而来，不需要保存，参考“量产/升级方案”。

### Emmc分区格式

#### 分区表格式

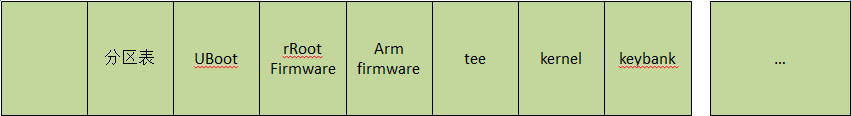


图 Emmc分区示意

如上图，EMMC需要设置分区表，分区表格式可灵活设置，简单的分区表如下：

Struct emmc\_partition\_tbl\_entry {

Unsigned char name[32];

Unsigned long long offset;

Unsigned long long len;

Unsigned int dx;

}

或者Emmc可以采用标准GPT分区格式：

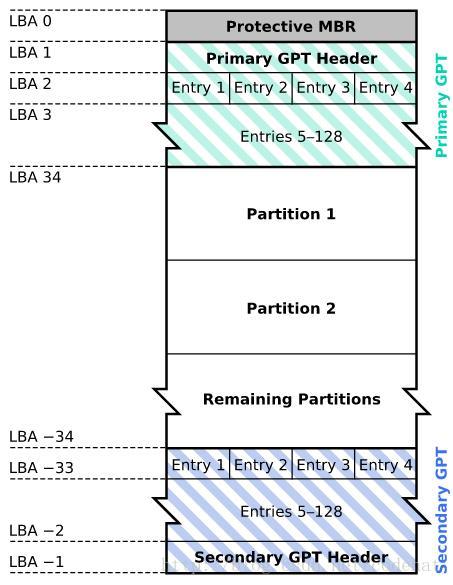


图 GPT分区格式

表格 GPT分区表项数据格式

|  |  |  |
| --- | --- | --- |
| 起始字节 | 长度 | 内容 |
| 0 | 16字节 | 分区类型GUID |
| 16 | 16字节 | 分区GUID |
| 32 | 8字节 | [起始LBA（小端序）](http://zh.wikipedia.org/wiki/%E5%B0%8F%E7%AB%AF%E5%BA%8F) |
| 40 | 8字节 | 末尾LBA |
| 48 | 8字节 | 属性标签（如：bit60表示“只读”） |
| 56 | 72字节 | [分区名（可以包括36个UTF-16（小端序）字符）](http://zh.wikipedia.org/wiki/UTF-16) |

### key管理方案

#### keybank分区方案

是否需要key bank分区根据不同方案而定，参考“量产/升级方案一节”。本节仅作参考。如果客户需要在Emmc存放一些自己的key，可以采用keybank方案。

Keybank分区存放两组TEE\_key和REE\_KEY或者更多Customer定制的key：

* TEE\_key用于校验和解密optee和arm trusted firmware分区。
* REE\_key用于校验和解密kernel以及其他需要安全保护的分区。
* CUS\_key用于存放客户定制的key。

Keybank分区存放格式如下：

struct keybank

{

struct keybank\_entry REE\_keybank;

struct keybank\_entry TEE\_keybank;

struct keybank\_entry CUS\_keyban[10];

}

每组key bank可以用如下结构体表示：

Struct keybank\_entry {

Unsigned char rsa\_public\_key[256]; //RSA2048（这一项也可以不要，全部都用bootloader里面的public key）

Unsigned char aes\_key[32] //AES 256

}

#### OTP中的key管理

OTP中存放的key包括：

1. tRoot的firmware的key(校验和解密)

2. AES root key(AES 256 bit)

3. RSA public key(2048 bit)

AES root key功能是用于系统派生key，在OTP root key多于一把的情况下， 具体用哪一把root key可以通过配置文件来表明，配置文件可以编译到Optee或者bootloader中。

RSA private key用于升级时对文件解密。

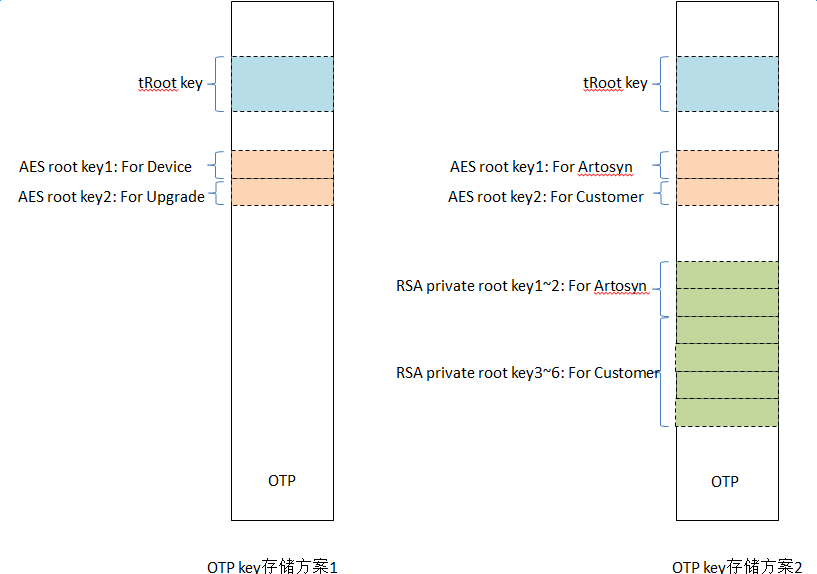


图 OTP key存储方案

根据“量产/升级方案”一节的描述，OTP保存key的方案有以上两种，详见“量产/升级方案”的设计方案。

#### 基于Secure Store的key存放方案

略

#### 派生key方法

方法1：

以下图为例，步骤如下：

1. 采用AES cbc作为加密方法，OTP root key作为key，platform id作为IV，对字符串String1加密，得到intermediate key。

2. 采用AES cbc作为加密方法， Intermediate key作为key，Emmc id作为IV，对字符串String2加密，得到derived key。

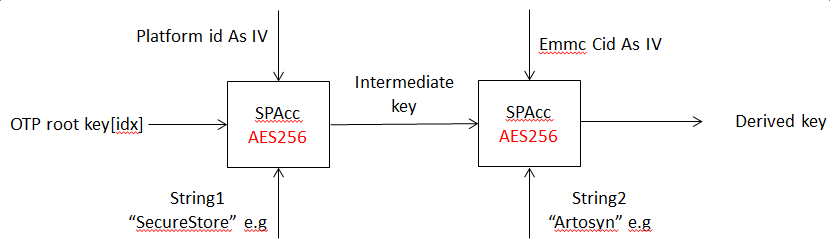


图 派生key方法1

方法2

1. 采用AES cbc作为加密方法，OTP root key(32字节)作为key，Emmc cid(16字节)作为IV，对字符串String1(32字节)加密，得到intermediate key(32字节)。

2. intermediate key前16字节与后16字节做异或，得到16字节新IV

2. 采用AES cbc作为加密方法，OTP root key(32字节)作为key，新IV，对字符串String2加密，得到derived key(32字节)。

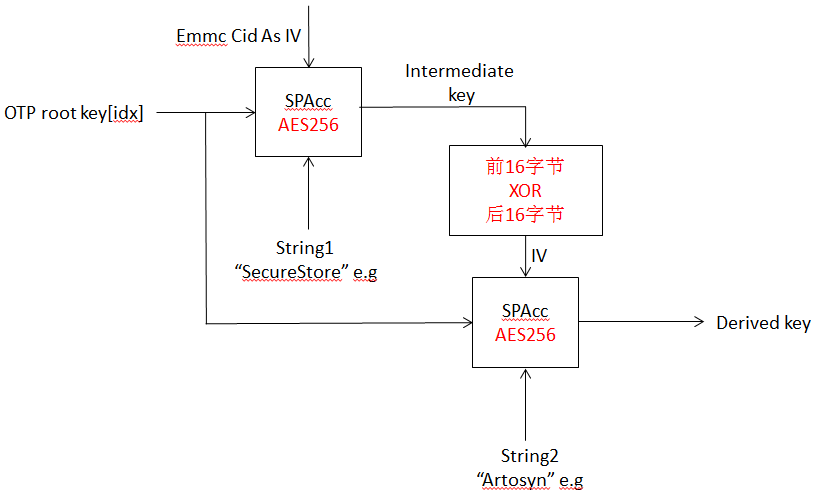


图 派生key方法2

用派生的方式得到key, 不需要在Emmc存key，每次开机可以重新gen key。

需要保存在Emmc上的key，在第一次开机时使用derived key加密，之后开机之后使用derived key解密后取得。

### Secure boot过程

* cpu0启动过程

1. ARMv7上电ROM启动在secure world的SVC模式下。
2. 初始化SPI/eMMC/DDR/secure memory等。
3. A7 load tRoot firmware和bootloader及其签名到SRAM，并通GPIO启动tRoot。
4. tRoot使用OTP中的PUK和PKf验证bootloader。
5. A7到固定地址运行bootloader。
6. bootloader初始化硬件。
7. bootloader使用OTP内的root key派生key。
8. parse Emmc/SPI上面的分区表，得到optee与uboot在Emmc/SPI上的位置
9. 从Emmc/SPI load Uboot、 optee到RAM。
10. 使用bootloader中的Platform RSA public key验证Uboot签名。
11. 使用TEE\_key中的rsa public key（或者Platform RSA public key）验证optee签名，使用TEE\_key中的aes\_key 解密optee。参考“量产升级方案”，本步骤可以用派生key来解密。
12. 使用OTP root key的派生key
13. 设置安全内存。
14. 将Uboot和OpteeOS搬到对应的entry point位置处。
15. bootloader转到optee的entry point运行optee，并传入uboot的entry point作为参数
16. optee初始化基本初始化（内存管理，页表，MMU，cache，线程等）
17. 初始化Secure Monitor设置monitor vector。
18. optee初始化完成，Optee发送IPI中断，等待其Secondary CPU启动并进入Optee
19. Secondary CPU进入Optee初始化完成，Primary CPU退出Secure模式，并进入Uboot运行。
20. Uboot加载、校验、解密kernel，并进入kernel entry point运行。
21. Linux设置Secondary CPU的Non-Secure入口地址，发送IPI中断

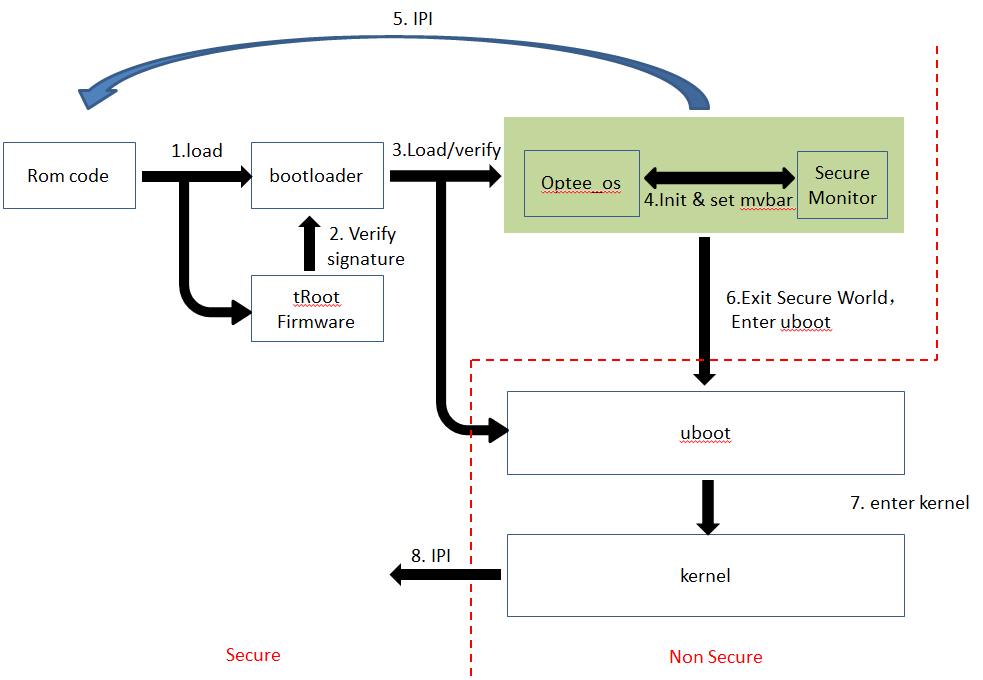


图 Secure boot（cpu0）流程

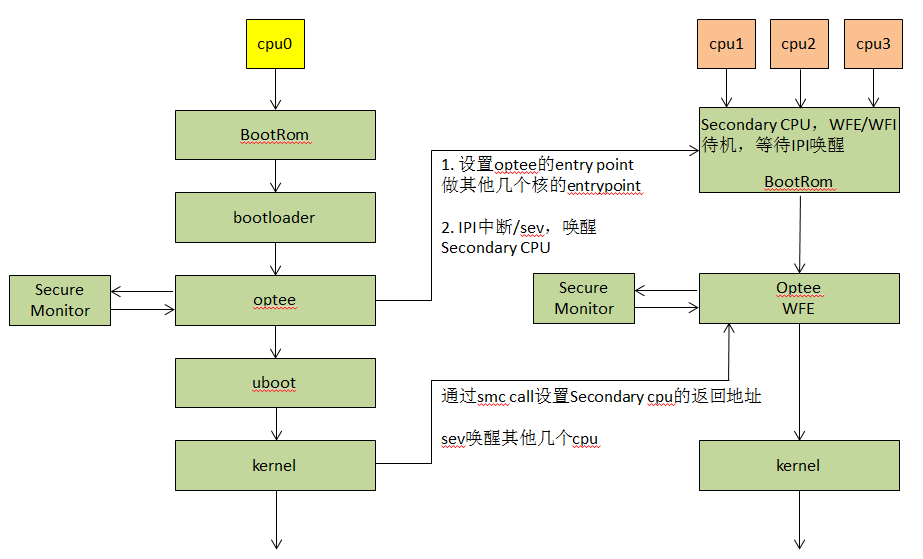


图 ARM多核启动过程

* 多核启动流程如下：

1. Romcode判断CPU id，如果是CPU0，则run bootloader，如果是其他CPU，执行WFE/WFI待机。
2. CPU0进入optee初始化并运行。
3. CPU0发送IPI中断。
4. 其他cpu收到IPI中断，进入optee，初始化完成后在optee中等待。
5. CPU0进入uboot并运行
6. CPU0 load kernel，校验并跳到kernel运行
7. CPU0初始化kernel，设置Secondary CPU的entry point（通过smc与optee交互）。然后通过IPI唤醒其他几个CPU
8. Secondary CPU唤醒后退出Optee到kernel运行

### 地址空间使用划分

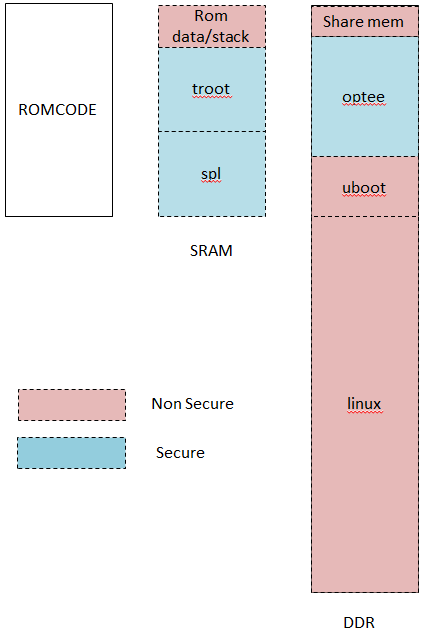


图 地址空间使用划分

|  |  |  |  |
| --- | --- | --- | --- |
| Table-1 Overview |  |  |  |
| Sirius memory allocation for Veloce Emulation - From Software Perspactive | | | |
|  | Start\_Address | End\_Address | Size |
| ROM data and stack | 0x30140000 | 0x301BFFFF | 512KB |
| tRoot FW and signature | 0x301C0000 | 0x301DFFFF | 128KB |
| U-boot-spl image and signature | 0x301E0000 | 0x301FFFFF | 128KB |
| OpTee OS and TA memory | 0x80000000 | 0x83FFFFFF | 64MB |
| U-boot image and signature | 0x84000000 | 0x841FFFFF | 2MB |
| Reserved | 0x84200000 | 0x84FFFFFF | 14MB |
| Kernel image and signature | 0x85000000 | 0x85FFFFFF | 16MB |
| DTB | 0x86000000 | 0x860FFFFF | 1MB |
|  |  |  |  |
| Note: 1. This only for Veloce Emulation use, should be used on all Veloce Buildx. When we bring up our chip, this allocation may need to be fine tuned. 2. Some temparory data can be put in "Reserved" region, such as SMMU page table, shared memory between A7 and CEVA/M7, etc. This shared memory will be automatically malloc by kernel in real case. 3. For Non-Security-Extension emulation, the tRoot/OpTee space not used should be reserved. 4. There are many sub-stack needed in ROM: IRQ stack, FIQ stack, CPU 0/1/2/3 stacks, these are defined in the following Table-2. 5. tRoot FW MUST be located at 0x301C0000, this is hard coded by hardware design. 6. OpTee space includes shared memory, OpTee OS and TA space, see Table-3. | | | |
|
|
|
|  |  |  |  |
| Table-2 ROM stack |  |  |  |
|  | Start\_Address | Size |  |
| CPU 0 stack (main CPU) | 0x301BFFF0 | 8KB |  |
| CPU 0 IRQ stack | 0x301BDFF0 | 8KB |  |
| CPU 0 FIQ stack | 0x301BBFF0 | 8KB |  |
| CPU 1 stack | 0x301B9FF0 | 8KB |  |
| CPU 1 IRQ stack | 0x301B7FF0 | 8KB |  |
| CPU 1 FIQ stack | 0x301B5FF0 | 8KB |  |
| CPU 2 stack | 0x301B3FF0 | 8KB |  |
| CPU 2 IRQ stack | 0x301B1FF0 | 8KB |  |
| CPU 2 FIQ stack | 0x301AFFF0 | 8KB |  |
| CPU 3 stack | 0x301ADFF0 | 8KB |  |
| CPU 3 IRQ stack | 0x301ABFF0 | 8KB |  |
| CPU 3 FIQ stack | 0x301A9FF0 | 8KB |  |
| data+bss+heap | 0x30140000 | 416KB |  |
|  |  |  |  |
| Table-3 OpTee memory |  |  |  |
|  | Start\_Address | End\_Address | Size |
| Shared memory with NS world | 0x80000000 | 0x807FFFFF | 8MB |
| OpTee OS | 0x80800000 | 0x80FFFFFF | 8MB |
| TA memory | 0x81000000 | 0x83FFFFFF | 48MB |

表 Sirsius内存地址配置

### 量产/升级方案

OTP容量可以放200把左右AES key（256bit），方案多样性比较多。

#### 方案1

* 量产：OTP中存放两把AES root key：key1每个平台不同，key2每个平台相同。用OTP root key1派生几把key，用来加密不同文件，然后把加密后的Image和OTP root key1、OTP root key2烧到板子上，开机后用key1出派生key，解密Image。
* 升级：发布的Image用OTP root key2加密，升级时用OTP root key2（或其派生key）解密然后用OTP root key1派生key，直接替换板子上的文件。

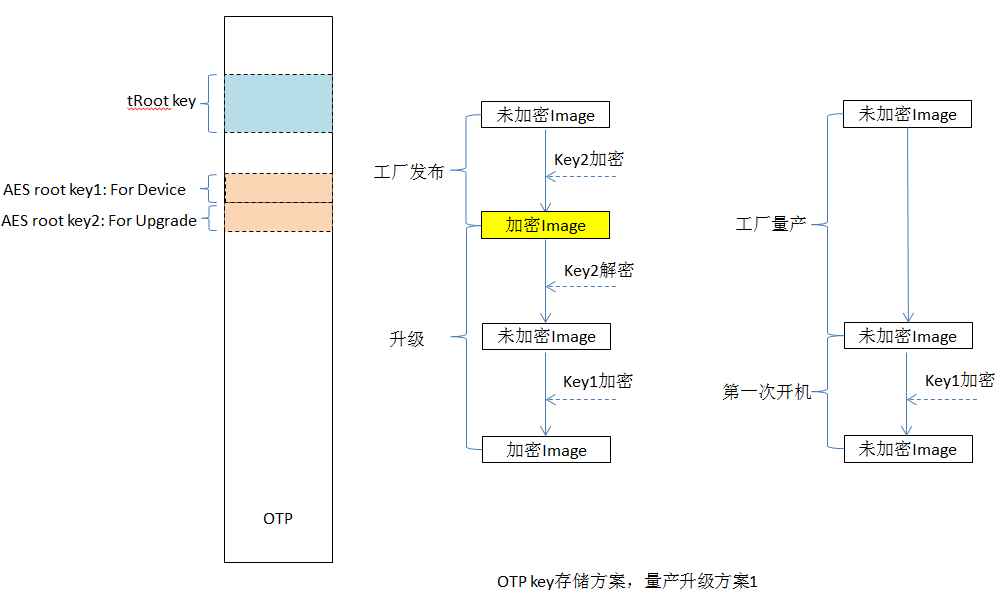


图 量产升级方案1

#### 方案2

在OTP存放两把AES root key，rootkey1用于Artosync的加解密操作， rootkey2用于Customer的加解密操作。

在OTP存放六把RSA root public key, RSA root public key1，RSA root public key2用于Artosync的加解密操作，另外4把用于Customer的加解密操作。

* 量产：

用OTP AES root key1的派生key加密Artosyn的Image，用OTP root key2的派生key加密Customer的Image。烧录加密Image、两把OTP root key、两把RSA root private key。

* 升级：

1. 厂家随机生成AES key，加密Image，使用RSA root private key对AES key加密
2. 发布加密后的Image、加密后的AESkey、RSA root key index。
3. 板子升级的时候首先根据RSA root key index选定RSA root private key对AES key解密，然后用AES key解密Image，接下来用对应的OTP AES root key加密Image并写到板子上。

Artosyn与Customer的Image相同，两者所用的key隔离，保证Artosyn与Customer的信息都不会泄露。

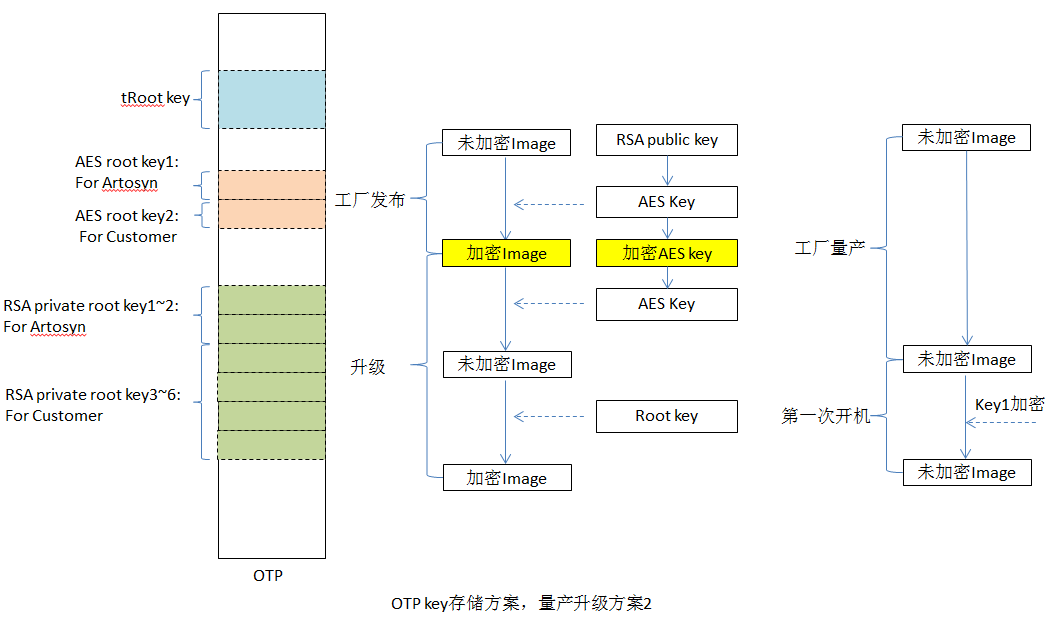


图 量产升级方案2

注：方案2中OTP中的rsa private key可以被CPU读出来，有安全隐患，因此考虑采用先用Root key加密，在往OTP中烧写密文的方案。

### Sirius Secure Storage设计

Optee有两套Secure Store方案：

* 一个是把文件存放在Linux文件系统上，比如在/data/tee下，以每个TA（Trusted application的UUID建立一个目录，把自己的文件放在这个目录下）。
* 另一个是Optee实现的一个简单的文件系统 (类似于FAT32，但是没有簇链，文件数据只能连续存放)，存放在Emmc的RPMB分区上（RPMB是Emmc中一块独立的存储空间，RPMB在第一次访问之前要写一把key进去， 并且每块Emmc只能写一次key， 此后写到RPMB的data pack都这把key通过HMAC-SHA256计算对应的HMAC，并一起写到RPMB中，RPMB会校验这个HMAC来验证写入的合法性，key可以通过OTP root key派生而来）。

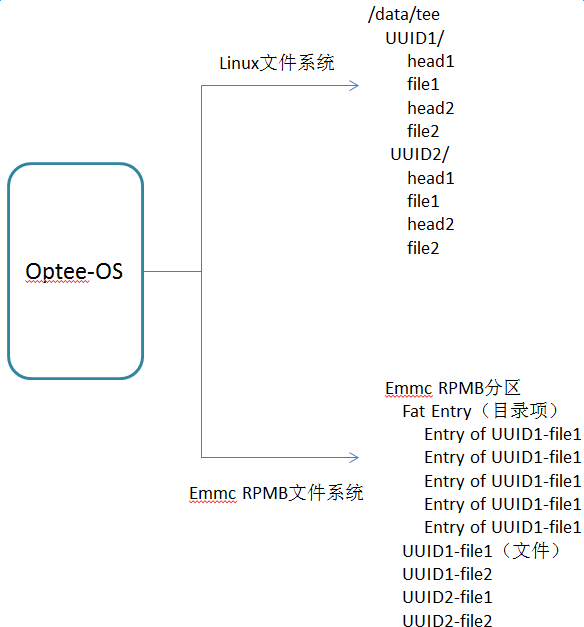


图 optee的secure store实现

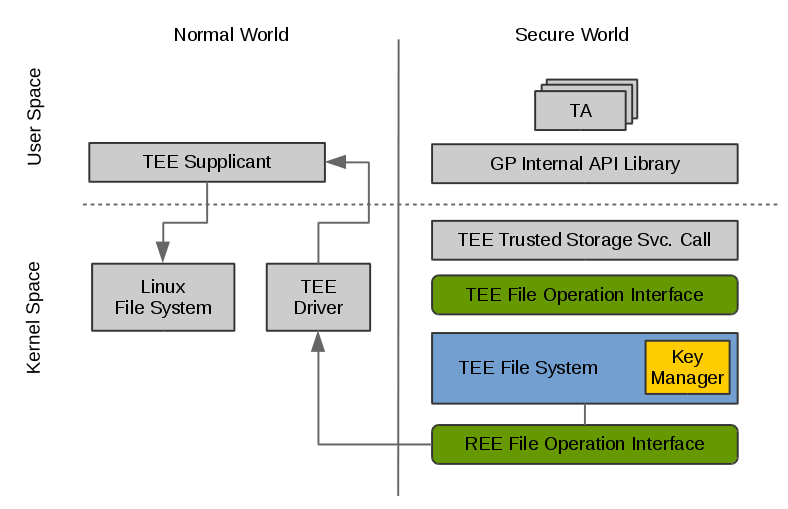


图 Secure Store文件操作流程

两套文件系统安全方案类似，如下是RPMB FS格式，Fat entry是每个文件的目录项，file是文件数据，fat entry中有一项FEk，FEK是由TRNG生成的随机数。FEK的作用对文件数据加密，然后使用派生key再把FEK加密。加密后的FEK写到FAT entry中。

RPMB FS的fat entry格式如下：

/\*\*

\* File entry for a single file in a RPMB\_FS partition.

\*/

struct rpmb\_fat\_entry {

uint32\_t start\_address;

uint32\_t data\_size;

uint32\_t flags;

uint32\_t write\_counter;

**uint8\_t fek[TEE\_FS\_KM\_FEK\_SIZE];**

char filename[TEE\_RPMB\_FS\_FILENAME\_LENGTH];

};

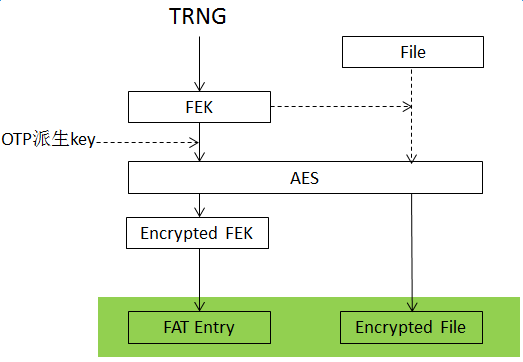
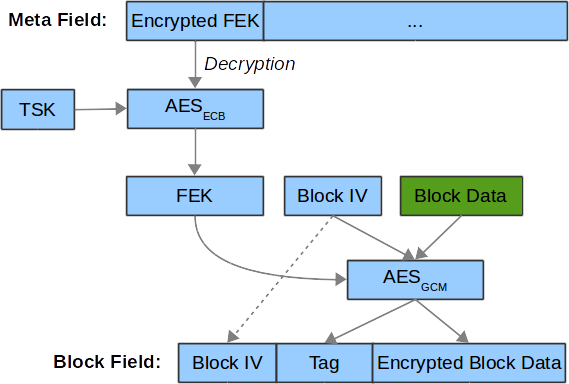


图 RPMB FS写文件流程

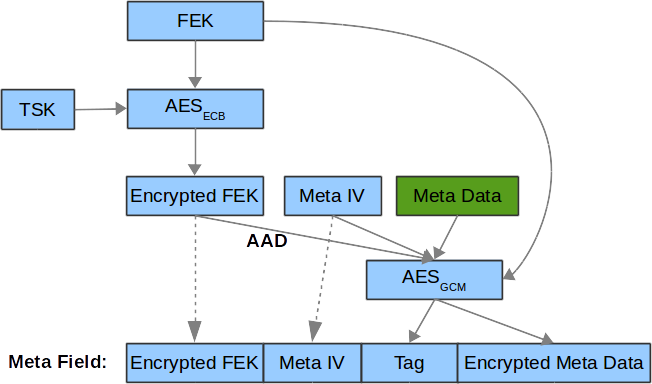
读取文件的情况相反，首先生成派生key, 然后读文件的fat entry，使用派生key解密得到FEK，然后读文件数据并使用FEK解密。

出于安全考虑，Secure Store中每个TA的派生key最好不同， 可以使用TA的UUID来派生key。

下图是Optee Ree Fs文件格式：



图Ree Fs Block data格式



图Ree Fs Meta data格式

### Sirius Version Control设计

版本控制主要是防止高版本回退到低版本（anti-rollback），可以采用Secure Store保存版本信息的方式：

* 系统或者TA运行，从Secure Store读取该软件的版本信息。
* 如果找不到版本信息，说明是第一次运行该软件，把当前软件的版本信息写到Secure Store中。
* 如果当前软件版本高于Secure Store中保存的版本，则更新软件的版本信息到Secure Store中。
* 如果当前软件版本低于Secure Store中保存的版本停止运行并系统报错。
* 升级过程同样处理。

### Sirius Secure Debug设计

默认secure debug disable，可以通过特殊应用程序或数据，需要经过系统验证后，才开放debug权限（打开相应的引脚及接口），即客户通过可以工作的接口，如网络接口，将数据文件传入文件系统，通过命令对数据文件进行解密验签（在Secure World中调用tRoot或SPAcc/PKA，验证通过即设置相应的GPIO，通过不同阶段不同GPIO的逻辑运算，CPU可知此时的验证结果），如果验证通过，则TEE OS打开相应的调试接口，并通知Normal world。

### 其他

#### Libtomcrypt， Libtommath

Libtomcrypt是开源的加密库，opteeOS中使用了这个lib，它依赖libtommath。Libtomcryt可以用来在pc上实现一些加解密，签名，hash等应用程序。

Libtommath编译方法：make, 然后copy头文件到libtomcrypt目录下

Libtomcrypt编译方法：./build libtomcrypt.a –DUSE\_LTM makefile –DLTM\_DESC ./libtomcrypt.a

#### PC端工具

## 参考文档：

Sirius Security系统设计文档V1.0.docx

DDI0464F\_cortex\_a7\_mpcore\_r0p5\_trm.pdf

\optee\_os-master\documentation\optee\_design.md

PRD29-GENC-009492C\_trustzone\_security\_whitepaper.pdf