PHYTIUM 飞腾

腾锐 D2000 安全 CPU

TEE 编程接口手册

v1.0

飞腾信息技术有限公司

2021 年 6 月

更新记录

版本号	发布部门	作者	发布日期	备　注
1.0	飞腾通用软件部	邓强，粟梁虎	2021-06-10	初稿

目录

1 介绍

飞腾开发的 optee-ft 是按照 GP 规范开发的 TEE OS，支持飞腾腾锐 D2000 安全 CPU 系统，开发搭建方便，便于开发者开发自有的上层可信应用，且 optee-ft 提供了完整的 SDK，方便编译 TA 和 CA。

2 密码加速引擎接口

2.1 接口概述

腾锐 D2000 安全 CPU 集成了密码加速引擎（以下简称 SCTO），支持 SM2/SM3/SM4/SM9 算法硬件加速。我们通过 optee-ft 提供的接口标准，将 SCTO 的相关功能 API 添加到 libmbedtls 库，实现飞腾平台 TEE 环境的硬件 SM2/SM3/SM4 算法。

optee-ft 根据 GP 规范实现了常用的加解密、签名验签和计算摘要的密码学算法的基础框架。如果需要使用硬件的密码学引擎来实现这些算法，则只需要替换掉对应的底层算法实现接口即可。对于上层用户而言，无需修改任何代码，只需按照 GP 规范，调用对应的接口组合即可实现对数据的加解密、摘要计算和数据签名验签操作。如图 2.1，我们替换掉 SM2、SM3 和 SM4 底层 API，将其用 SCTO 实现，就可以通过 GP 标准接口调用密码学引擎的能力来实现对数据的 SM2、SM3 和 SM4 算法的加解密、摘要计算和数据签名验签操作。

目前对应 SM2 算法需要替换六类 API 接口，包括：密钥对生成、密钥对释放、加密、解密、签名和验签。对应 SM3 算法需要替换五类 API 接口，包括：初始化、更新数据、完成、描述符释放和描述符克隆。对应 SM4 算法需要替换五类 API 接口，包括：初始化、更新数据、完成、描述符释放和描述符克隆，其中更新数据接口针对 cbc、ecb 和 ctr 模式有不同的实现。

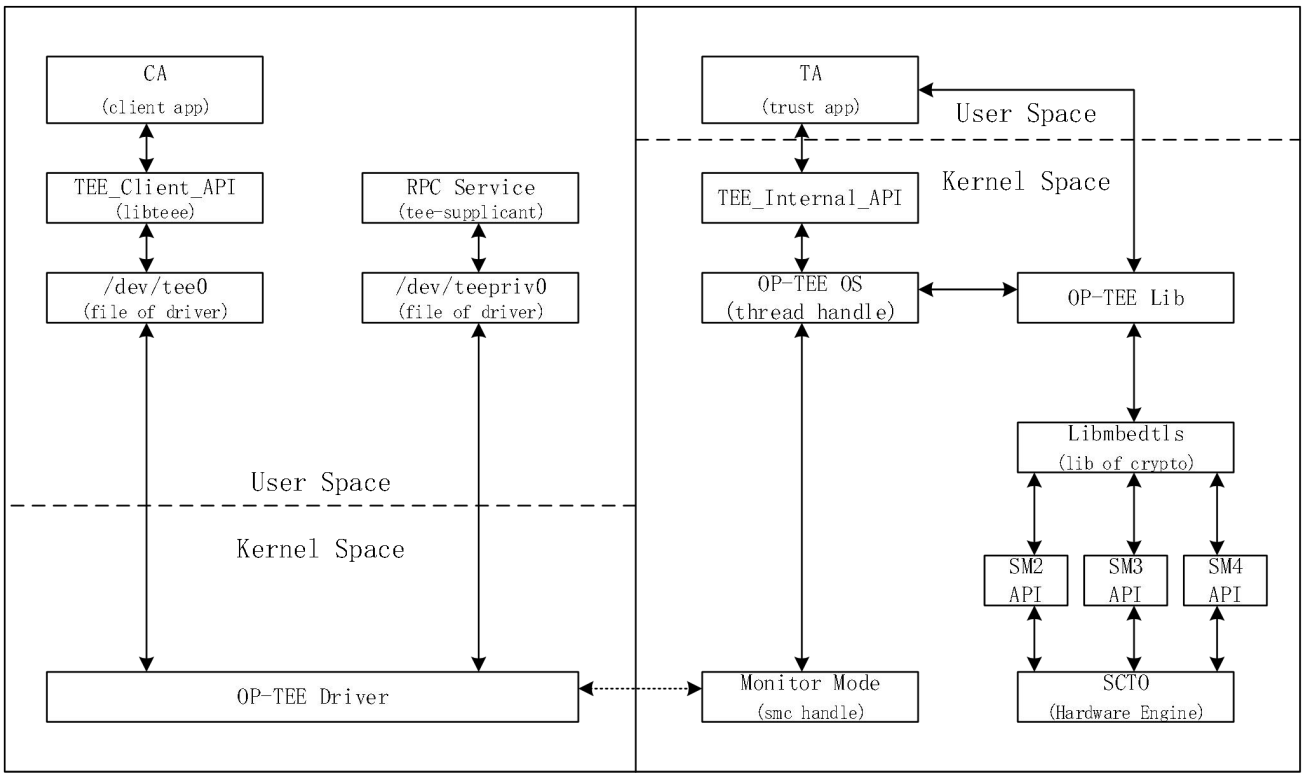


图 2.1 程序运行流程图

2.2 接口说明

2.2.1 SM2 算法类接口说明

在 optee-ft 当前版本的 SM2 底层算法实现接口中，对有些实现内容未做明确规定，需要我们根据实际需求去定义，特说明如下:

一、在 SM2 加解密接口，我们临时使用了 C1C2C3 的数据格式，应根据实际需求做调整，如 C1C3C2 格式。

二、在 SM2 签名验签接口，参数 msg 为签名数据的摘要，具体计算方法可参考示例程序 scto 中 scto_ta.c 文件中的 sm2_sign_digest 函数。

三、在 SM2 签名验签接口中，签名数据为 DER 格式，在签名时，会对生成的 r||s 整数对做 DER 格式编码再输出，在验签时，会对输入的签名做 DER 解码，所以要求验签时输入数据为 DER 编码格式。

```
static TEE_Result sm2_generate_keypair(struct ecc_keypair *key, size_t key_size)
```

功能：生成 SM2 密钥对

参数 key：用于保存密钥对

参数 key_size：密钥长度

返回结果：0 表示成功生成密钥对，其他为错误

```
static void sm2_free_public_key(struct ecc_public_key *s)
```

功能：释放 SM2 密钥对

参数 s：释放的密钥对

返回结果：成功释放密钥对

```
TEE_Result sm2_pke_encrypt(struct ecc_public_key *key, const uint8_t *src,
                           size_t src_len, uint8_t *dst, size_t *dst_len)
```

功能：使用 SM2 算法加密数据

参数 key：加密使用的 SM2 公钥

参数 src：需要加密的数据

参数 src_len：需要加密数据的长度

参数 dst：用于保存加密数据

参数 dst_len：入参时表示加密数据保存空间的大小，返回时表示加密后数据的长度

返回结果：0 表示加密成功，其他表示错误

```
TEE_Result sm2_pke_decrypt(struct ecc_keypair *key,const uint8_t *src,
                           size_t src_len, uint8_t *dst,size_t *dst_len)
```

功能：使用 SM2 算法解密数据

参数 key：解密使用的 SM2 密钥对

参数 src：需要解密的数据

参数 src_len：需要解密数据的长度

参数 dst：用于保存解密数据

参数 dst_len：入参时表示解密数据保存空间的大小，返回时表示解密后数据的长度

返回结果：0 表示解密成功，其他表示错误

```
TEE_Result sm2_dsa_sign(uint32_t algo,struct ecc_keypair *key, const uint8_t *msg,
                        size_t msg_len, uint8_t *sig, size_t *sig_len)
```

功能：使用 SM2 算法签名

参数 algo：算法类型

参数 key：签名使用的 SM2 密钥对

参数 msg：需要签名的数据摘要

参数 msg_len：需要签名的数据摘要的长度,目前标准为 32 字节

参数 sig：用于保存签名

参数 sig_len：入参时表示签名保存空间的大小，返回时表示签名的长度

返回结果：0 表示签名成功，其他表示错误

```
TEE_Result sm2_dsa_verify(uint32_t algo,struct ecc_public_key *key, const uint8_t *msg,
                          size_t msg_len, const uint8_t *sig, size_t sig_len)
```

功能：使用 SM2 算法验签

参数 algo：算法类型

参数 key：验签使用的 SM2 公钥

参数 msg：需要验签的数据摘要

参数 msg_len：需要验签的数据摘要的长度,目前标准为 32 字节

参数 sig：签名

参数 sig_len：签名的长度

返回结果：0 表示验签成功，其他表示错误

2.2.2 SM3 算法类接口说明

SM3 实现了使用 DMA 引擎操作的方式，但 DMA 引擎唯一，可根据实际情况确定由哪个算法使用。同时，对 SM3 增加了软件运算，在抢占不到硬件引擎的情况下使用软件运算，增加并发性能，可视情况使用。

```
void mbedtls_sm3_init( mbedtls_sm3_context *ctx )
```

功能：初始化 ctx 初始值和引擎硬件操作指针

参数 ctx：SM3 上下文

返回结果：成功完成初始化

```
int mbedtls_sm3_starts_ret(mbedtls_sm3_context *ctx )
```

功能：初始化 HASH 初始值和数据总长度

参数 ctx：SM3 上下文

返回结果：0 表示初始化成功，其他表示错误

```
int mbedtls_sm3_update_ret(mbedtls_sm3_context *ctx, const unsigned char *input, size_t ilen )
```

功能：更新 SM3 算法的数据

参数 ctx：SM3 上下文

参数 input：需要 SM3 算法运算的数据

参数 ilen：数据的长度

返回结果：0 表示成功，其他表示错误

```
int mbedtls_sm3_finish_ret( mbedtls_sm3_context *ctx, unsigned char output[32] )
```

功能：用于数据更新后完成 SM3 算法

参数 ctx：SM3 上下文

参数 output：用于返回算法的结果

返回结果：0 表示成功，其他表示错误

```
void mbedtls_sm3_free( mbedtls_sm3_context *ctx )
```

功能：用于释放 SM3 上下文的数据

参数 ctx：SM3 上下文

返回结果：成功释放 SM3 上下文数据

```
void mbedtls_sm3_clone( mbedtls_sm3_context *dst, const mbedtls_sm3_context *src )
```

功能：克隆 SM3 上下文

参数 dst：克隆的 SM3 上下文

参数 src：原始的 SM3 上下文

返回结果：成功克隆 SM3 上下文数据

2.2.3 SM4 算法类接口说明

SM4 实现了使用 DMA 引擎操作的方式，但 DMA 引擎唯一，可根据实际情况确定由哪个算法使用。同时，对 SM4 增加了软件运算，在抢占不到硬件引擎的情况下使用软件运算，增加并发性能，可视情况使用。

```
TEE_Result mbed_sm4_init(struct crypto_cipher_ctx *ctx,
                         TEE_OperationMode mode, const uint8_t *key1,
                         size_t key1_len, const uint8_t *key2 __unused,
```

```
                                        size_t key2_len __unused, const uint8_t *iv, size_t iv_len)
```

功能：SM4 算法初始化

参数 ctx：上下文

参数 mode：操作模式，这里可选加密或解密

参数 key1：加解密的密钥

参数 key1_len：密钥的长度

参数 key2：SM4 算法未用到

参数 key2_len：SM4 算法未用到

参数 iv：初始化向量

参数 iv_len：初始化向量的长度

返回结果：0 表示初始化成功，其他表示错误

```
TEE_Result mbed_sm4_cbc_update(struct crypto_cipher_ctx *ctx, bool last_block __unused,
                               const uint8_t *data, size_t len, uint8_t *dst)
```

功能：SM4 CBC 模式更新数据，根据上下文确定是加密还是解密

参数 ctx：上下文

参数 last_block：SM4 算法未用到

参数 data：需要处理的数据

参数 len：需要处理的数据的长度，要求 16 字节对齐

参数 dst：存放处理后的数据

返回结果：0 表示处理成功，其他表示错误

```
TEE_Result mbed_sm4_ecb_update(struct crypto_cipher_ctx *ctx, bool last_block __unused,
                               const uint8_t *data, size_t len, uint8_t *dst)
```

功能：SM4 ECB 模式更新数据，根据上下文确定是加密还是解密

参数 ctx：上下文

参数 last_block：SM4 算法未用到

参数 data：需要处理的数据

参数 len：需要处理的数据的长度，要求 16 字节对齐

参数 dst：存放处理后的数据

返回结果：0 表示处理成功，其他表示错误

```
TEE_Result mbed_sm4_ctr_update(struct crypto_cipher_ctx *ctx, bool last_block __unused,
                               const uint8_t *data, size_t len, uint8_t *dst)
```

功能：SM4 CTR 模式更新数据，加解密通用

参数 ctx：上下文

参数 last_block：SM4 算法未用到

参数 data：需要处理的数据

参数 len：需要处理的数据的长度，要求 16 字节对齐

参数 dst：存放处理后的数据

返回结果：0 表示处理成功，其他表示错误

```
void mbed_sm4_final(struct crypto_cipher_ctx *ctx)
```

功能：完成 SM4 算法，会释放上下文的数据

参数 ctx：上下文

返回结果：成功完成 SM4 算法

```
void mbed_sm4_free_ctx(struct crypto_cipher_ctx *ctx)
```

功能：释放 SM4 上下文

参数 ctx：上下文

返回结果：成功释放上下文

```
void mbed_sm4_copy_state(struct crypto_cipher_ctx *dst_ctx,
                         struct crypto_cipher_ctx *src_ctx)
```

功能：克隆 SM4 上下文

参数 dst_ctx：克隆的上下文

参数 src_ctx：原始上下文

返回结果：成功克隆上下文

2.3 测试实例

实现了上述底层算法接口后，还需要将这些接口注册到标准流程中，此外，还需要一些初始化的准备工作等。

测试采用的为 CentOS-8.2 系统，内核为 4.19.5，并使用了 phytium 针对该版本的补丁。同时，编译也

在该环境完成。

2.3.1 初始化

为了确保 SCTO 和软件锁有确定的初始状态，这里通过 optee-ft 的 driver_init 接口来完成，在启动过程中完成初始化。

这部分代码位于 optee_os/core/drivers/phytium/ft_scto.c 文件中，完成以下几部分工作：

一、注册 SCTO IO 地址段为安全 IO 地址，其中 SMX_RESERVE_BASE 为临时选用的一段地址，给软件锁使用，根据实际需求重新分配地址和大小，要求页对齐。

```
#define SRAM_BASE                                   (0x29800000)
#define   SMX_RESERVE_BASE          (SRAM_BASE + 0x0000UL)     //just for temporary use
#define SMX_RESERVE_SIZE            (0x1000)
#define SMX_BASE_ADDR                               0x28220000
#define SMX_BASE_SIZE                               0x7000
register_phys_mem_pgdir(MEM_AREA_IO_SEC, SMX_BASE_ADDR, SMX_BASE_SIZE);
register_phys_mem_pgdir(MEM_AREA_IO_SEC, SMX_RESERVE_BASE, SMX_RESERVE_SIZE);
```

二、将初始化函数注册到 driver_init 接口。

```
driver_init(phytium_sm_init);
```

2.3.2 SM2 接口注册

SM2 公钥相关接口注册

将封装好的 API 接入项目体系需要以下几步：

一、在 optee_os/core/tee/tee_svc_cryp.c 文件的 tee_obj_set_type 函数中增加 SM2 分支 crypto_acipher_alloc_sm2_public_key。

```
case TEE_TYPE_SM2_DSA_PUBLIC_KEY:
case TEE_TYPE_SM2_PKE_PUBLIC_KEY:
case TEE_TYPE_SM2_KEP_PUBLIC_KEY:
        res = crypto_acipher_alloc_sm2_public_key(o->attr, obj_type,
                                                  max_key_size);
        break;
```

二、在 optee_os/core/crypto/crypto.c 文件中实现 crypto_acipher_alloc_sm2_public_key。

三、在 optee_os/lib/libmbedtls/core/sm2.c 文件中实现 crypto_acipher_alloc_sm2_public_key，将已经封装好的 API 接入 optee-ft 项目体系。

SM2 密钥对相关接口注册

将封装好的 API 接入项目体系需要以下几步：

一、在 optee_os/core/tee/tee_svc_cryp.c 文件的 tee_obj_set_type 函数中增加 SM2 分支 crypto_acipher_alloc_sm2_keypair。

```
case TEE_TYPE_SM2_DSA_KEYPAIR:
case TEE_TYPE_SM2_PKE_KEYPAIR:
case TEE_TYPE_SM2_KEP_KEYPAIR:
        res = crypto_acipher_alloc_sm2_keypair(o->attr, obj_type,
                                               max_key_size);
        break;
```

二、在 optee_os/core/crypto/crypto.c 文件中实现 crypto_acipher_alloc_sm2_keypair。

三、在 optee_os/lib/libmbedtls/core/sm2.c 文件中实现 crypto_asym_alloc_sm2_keypair，将已经封装好的 api 接入 optee-ft 项目体系。

2.3.3 SM3 接口注册

HMAC 和 HASH 共用同一套底层接口，将封装好的 api 接入项目体系需要以下几步：

一、在 optee_os/core/crypto/目录 sm3-hash.c 和 sm3-hmac.c 文件中，注释掉原有的软件实现接口。

二、在 optee_os/lib/libmbedtls/core/目录 hash.c 和 hmac.c 文件中，分别增加 crypto_sm3_alloc_ctx 接口和 crypto_hmac_sm3_alloc_ctx 接口，将 SM3 选项注册到流程中去。

三、在 optee_os/lib/libmbedtls/mbedtls/library/md.c 文件中，增加 SM3 相关信息，并将封装好的 API 添加到标准流程对应的分支里去，分别为 mbedtls_md_starts、mbedtls_md_update、mbedtls_md_finish、mbedtls_md_free 和 mbedtls_md_clone。

2.3.4 SM4 接口注册

将封装好的 API 接入项目体系需要以下几步：

一、在 optee_os/core/crypto/目录下 sm4-cbc.c、sm4-ecb.c 和 sm4-ctr.c 三个文件中，注释掉原有的软件实现接口。

二、在 optee_os/lib/libmbedtls/core/sm4.c 文件中实现 crypto_sm4_cbc_alloc_ctx、crypto_sm4_ecb_alloc_ctx 和 crypto_sm4_ctr_alloc_ctx 接口，将该文件定义的封装好的 API 注册到标准流程中。

2.3.5 示例程序

在 scto/ta/scto_ta.c 文件中，按照 GP 标准接口调用，实现了 SM2 加解密和签名验签功能，SM3 HASH 和 HMAC 功能，SM4 CBC、ECB 和 CTR 模式的加解密功能，具体可参考该文件源代码。

同时，在 scto/host/scto.c 文件中，实现了对这些功能的调用。

2.3.6 编译

编译需要以下几步:

一、进入 optee_os 目录，执行如下命令编译 TEE OS。

```
./build_optee_os d2000
```

二、进入 optee_client 目录，执行如下命令编译 CA 端 API 库。

```
./build_optee_client_ft.sh
```

三、进入 hello_world 目录，执行如下命令编译示例程序。

```
./build_ta_scto_ft.sh
```

2.3.7 运行

一、编译完成的 TEE OS 为 out/tee-phytium-d2000.bin，需要将其打包入固件或者在 UEFI 下使用 tboot 功能将其加载，可参考《腾锐 D2000 安全 CPU-QuickStart》。

二、系统启动完成后，进入 SDK 目录，执行如下命令启动 CA 端后台程序。

```
./out/data/bin/tee-supplicant -d /dev/teepriv0 &
```

三、执行如下命令将 out/data/optee_armtz/目录下的 TA 程序拷贝到/data/optee_armtz/目录下。

```
mkdir -p /data/optee_armtz/
cp ./out/data/optee_armtz/*  /data/optee_armtz/
```

四、执行如下命令启动示例程序，默认启动 7 个线程，每个线程会循环执行所有流程，直到遇到错误停止。

```
./out/data/bin/scto
```

除 SM2 相关算法因随机数原因只有自验证外, SM3 和 SM4 相关算法的结果还有与软件算法的结果的对比，以验证计算结果是否正确。(SM2 算法有与软件相互加解密测试过，这里只是因没有固定结果不好每轮对比)

测试结果如下表所示:

表 2.1 测试结果

测试项目	3 小时运行结果
SM2 加解密	无错误
SM2 签名验签	无错误
SM3 HASH	无错误
SM3 HMAC	无错误
SM4 CBC 加解密	无错误
SM4 ECB 加解密	无错误
SM4 CTR 加解密	无错误

3 片内 efuse 读写接口

3.1 接口概述

飞腾平台芯片提供 efuse 功能，本软件实现 TEE OS 下 efuse_rw 模块功能，efuse_rw 模块目前支持 efuse 中 ICVFLAG 和 HBK 两个区域的读写操作，未来可以扩展为全区域读写操作的支持。

TEE 环境中，User 层运行的应用程序 TA（Trust Application）可以直接调用 efuse_rw 模块中的接口操作 efuse 硬件，但注意 efuse 是一次性写入硬件，以及 efuse 的一些读写限制特性。

调用关系如下图:

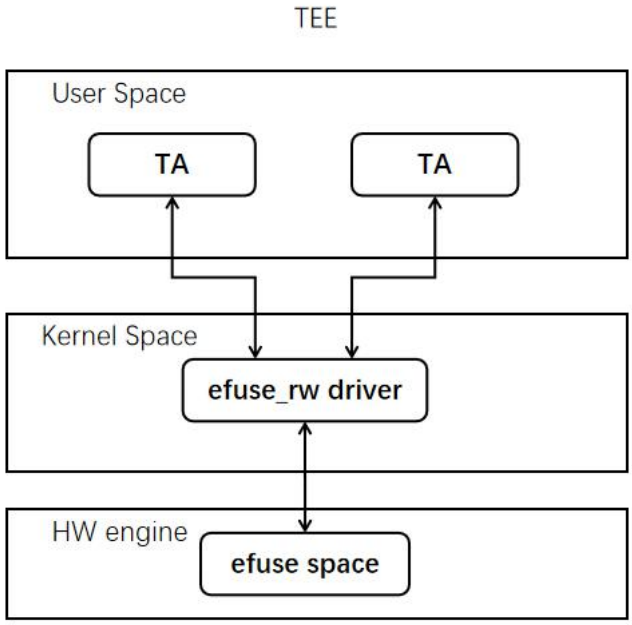


图 3.1 efuse 读写调用关系图

3.2 接口说明

```
int efuse_hbk_write(void)
```

功能：efuse 硬件中 ICVFLAG 和 HBK 两个区域的 write 操作。

返回：0　　成功

-1　　失败

注意：efuse 是一次性写入硬件，写入数据通过初始化变量的方式固化在代码中，写入 ICVFLAG 区域的数据固化在变量 ICVFLGA_val 中，写入 HBK 区域的数据固化在变量 HBK_val 中。

```
void efuse_hbk_print(void)
```

功能：efuse 硬件中 ICVFLAG 和 HBK 两个区域数据的输出操作，输出到 console 口显示。

返回：无

3.3 测试实例

本次功能设计完成的同时，提供一个基于 REE 和 TEE 环境交互的自测功能，并在飞腾 FT2000/4 硬件平台上进行了自测。

3.3.1 测试控制流程

测试流程如下图所示，控制流的路线如下图中实线所表明。

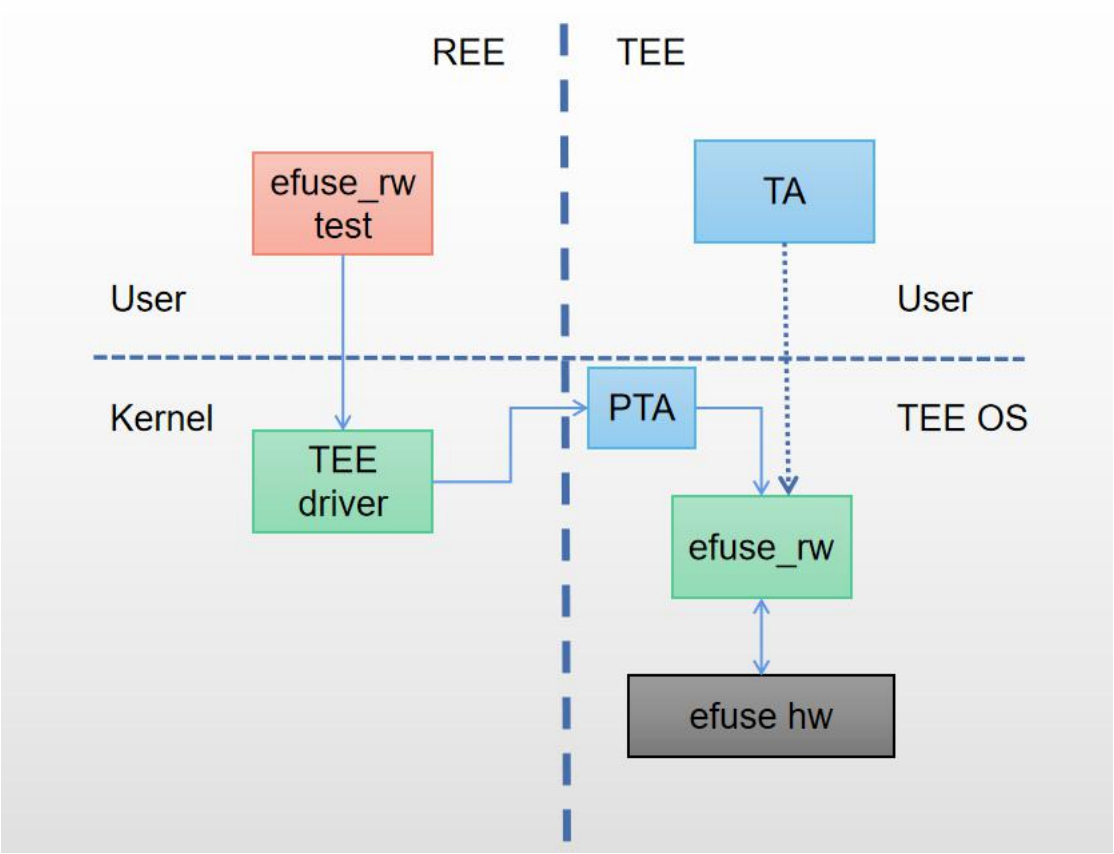


图 3.3 efuse 读写接口测试关系图

测试步骤如下:

步骤 1：REE 端 User 层应用 efuse_rw_test，源码为 efuse_rw_test.c，通过编译后的执行文件 efuse_rw_test 存放在 REE 文件系统中相应的 bin 目录下，通过脚本调用该应用程序。

步骤 2：脚本可以发起读和写两种操作，无论哪种操作都会触发 Linux TEE driver 中，包括 open session，invoke command 等标准的接口操作，调用顺序根据 GP 标准操作。读和写会对应两种不同的 command 命令发送给 TEE 侧相应的 PTA 程序。

步骤 3：PTA（Pseudo TA）作为一种伪 TA，可以模拟 TA 与 REE 端进行标准的 GP 交互操作，也可以直接访问 TEE OS 中的所有接口。根据传递过来的不同的 command 命令，对应的 efuse PTA 会调用不同的操作接口。

步骤 4：最后通过 efuse_rw 模块的接口调用，完成 efuse 硬件的写入操作和读出显示操作。

3.3.2 测试 LOG

在所有 DEBUG 信息打开的状态下，数据写入成功的提示如下:

```
Invoking TA before
F/TC:? 0 invoke_command:61 command entry point for pseudo ta "efuse_rw_invoke.pta"
D/TC:? 0 efuse_hbk_write:54 Unlock efuse success!
D/TC:? 0 efuse_hbk_write:86 Waiting for!
D/TC:? 0 efuse_hbk_write:91 reg[MAP_INT]          5!
D/TC:? 0 efuse_hbk_write:86 Waiting for!
D/TC:? 0 efuse_hbk_write:91 reg[MAP_INT]          5!
D/TC:? 0 efuse_hbk_write:86 Waiting for!
D/TC:? 0 efuse_hbk_write:91 reg[MAP_INT]          5!
D/TC:? 0 efuse_hbk_write:86 Waiting for!
D/TC:? 0 efuse_hbk_write:91 reg[MAP_INT]          5!
D/TC:? 0 efuse_hbk_write:86 Waiting for!
D/TC:? 0 efuse_hbk_write:91 reg[MAP_INT]          5!
D/TC:? 0 efuse_hbk_write:86 Waiting for!
D/TC:? 0 efuse_hbk_write:91 reg[MAP_INT]          5!
D/TC:? 0 efuse_hbk_write:86 Waiting for!
D/TC:? 0 efuse_hbk_write:91 reg[MAP_INT]          5!
D/TC:? 0 efuse_hbk_write:86 Waiting for!
D/TC:? 0 efuse_hbk_write:91 reg[MAP_INT]          5!
Invoking TA after
```

在所有 DEBUG 信息打开的状态下，数据读取成功的提示如下:

```
Invoking TA before
F/TC:? 0 invoke_command:61 command entry point for pseudo ta "efuse_rw_invoke.pta"
D/TC:? 0 efuse_hbk_print:115 EFUSE[ICVFLAG] 00000000
D/TC:? 0 efuse_hbk_print:117 EFUSE[HBK]:
D/TC:? 0 efuse_hbk_print:120 c4:01234567
D/TC:? 0 efuse_hbk_print:120 c8:89abcdef
D/TC:? 0 efuse_hbk_print:120 cc:00112233
D/TC:? 0 efuse_hbk_print:120 d0:44556677
D/TC:? 0 efuse_hbk_print:120 d4:8899aabb
D/TC:? 0 efuse_hbk_print:120 d8:ccddeeff
D/TC:? 0 efuse_hbk_print:120 dc:00011122
D/TC:? 0 efuse_hbk_print:120 e0:23334445
Invoking TA after
```

测试结果符合代码描述。

4 SMC 调用接口

4.1 接口概述

腾锐 D2000 安全 CPU 支持 TEE 环境通过 SMC 指令调用 EL3 安全态软件提供的各类软硬件信息查询功能。具体服务可以参考 www.phytium.com.cn 网址上的“技术支持/下载中心”的文档《Phytium Base Firmware 接口规范》。

接口调用关系如下图所示:

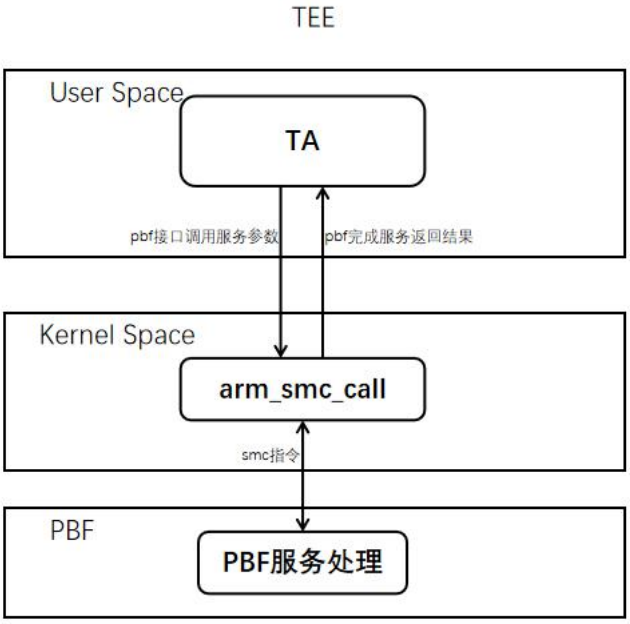


图 4.1 SMC 调用关系图

4.2 接口说明

```
void thread_arm_smc_call(struct thread_smc_args* in, struct thread_smc_args* out)
```

功能：发起 SMC 调用，并保存返回结果。

参数 in：输入上下文，TEE 标准结构类型，依据 SMC 调用规范。

参数 out：输出上下文，TEE 标准结构类型，依据 SMC 调用规范。

返回：无

4.3 测试实例

该调用接口可以在 TEE 运行的任意位置被调用，所以测试选取在 TEE 串口初始化后，直接调用该接口查询底层 PBF 版本信息。

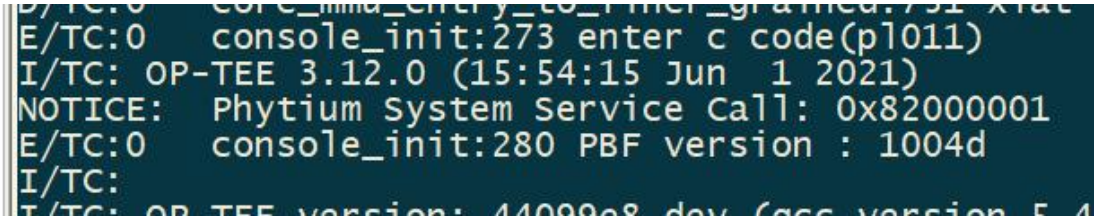
测试代码如下：

```
struct thread_smc_args in = {0};
struct thread_smc_args out = {0};

in.a0 = 0x82000001;
```

```
thread_arm_smc_call(&in, &out);
EMSG("PBF version: %lx", out.a0);
```

测试 LOG 如下：



可以看出 PBF 调用参数，回复的版本号是 1004d，查询文档得知，1 为大版本号，4d 为小版本号，4d 的十进制是 77，所以 pbf 的版本号查询结果是 1.77，符合最开始 pbf 的版本打印 log，测试结果无误。