

展锐 TEE 指纹 TA 开发指南

Release Date	2019/6/4
Document No.	
Version	V4.0
Document Type	UserGuide
Platform	UMS312,SC9863A,SC9832E,SC7731E
OS Version	Android 9.0

Unisoc Confidential For hiar

声明 Statement

本文件所含数据和信息都属于紫光展锐所有的机密信息，紫光展锐保留所有相关权利。本文件仅为信息参考之目的提供，不包含任何明示或默示的知识产权许可，也不表示有任何明示或默示的保证，包括但不限于满足任何特殊目的、不侵权或性能。当您接受这份文件时，即表示您同意本文件中内容和信息属于紫光展锐机密信息，且同意在未获得紫光展锐书面同意前，不使用或复制本文件的整体或部分，也不向任何其他方披露本文件内容。紫光展锐有权在未经事先通知的情况下，在任何时候对本文件做任何修改。紫光展锐对本文件所含数据和信息不做任何保证，在任何情况下，紫光展锐均不负责任任何与本文件相关的直接或间接的、任何伤害或损失。

All data and information contained in or disclosed by this document is confidential and proprietary information of UNISOC and all rights therein are expressly reserved. This document is provided for reference purpose, no license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, and no express and implied warranties, including but without limitation, the implied warranties of fitness for any particular purpose, and non-infringement, as well as any performance. By accepting this material, the recipient agrees that the material and the information contained therein is to be held in confidence and in trust and will not be used, copied, reproduced in whole or in part, nor its contents revealed in any manner to others without the express written permission of UNISOC. UNISOC may make any changes at any time without prior notice. Although every reasonable effort is made to present current and accurate information, UNISOC makes no guarantees of any kind with respect to the matters addressed in this document. In no event shall UNISOC be responsible or liable, directly or indirectly, for any damage or loss caused or alleged to be caused by or in connection with the use of or reliance on any such content.

关键字 Keywords

指纹, TEE

Unisoc Confidential For hiar

版本历史 Revision history

版本 Version	日期 Date	作者 Author	描述 Description
V0.1	2017/03/20	Unisoc	draft
V0.2	2017/04/21	Unisoc	1.更新 API 列表 2.完善 TA Demo 说明
V0.3	2017/05/19	Unisoc	1.增加指纹架构章节 2.增加开发步骤、debug、发布章节
V0.4	2017/05/30	Unisoc	增加注意事项
V0.5	2017/06/20	Unisoc	增加不同平台差异点说明
V0.6	2017/07/06	Unisoc	增加多指纹兼容方案初稿
V0.7	2017/07/19	Unisoc	完善多指纹兼容方案说明
V0.8	2017/08/02	Unisoc	完善开发说明，工具支持删除/增加 TA
V1.0	2017/08/26	Unisoc	7.1 小节更新 TOS 环境说明
V1.1	2017/12/11	Unisoc	增加工厂测试、支付开发说明
V1.2	2018/01/18	Unisoc	12.1 小节 IFAA 增加一个接口
V2.0	2018/02/06	Unisoc	10.5 小节增加多指纹兼容注意事项 6 10.4.2 小节多指纹兼容 Linux 驱动说明修改

V3.0	2019/02/12	Unisoc	1.针对 android 9.0 更新、完善文档说明 2.更新架构图、流程图 3.更新/删除驱动相关说明 4.更新模板
V4.0	2019/6/4	Unisoc	1.更新模板 2.更改文档名 3.10.5 小节增加第 4 条注意项

Unisoc Confidential For hiar

前 言 Foreword

一 范围 Scope

本文档用于指导指纹厂商和客户基于展锐 TEE 方案开发、适配指纹功能。

二 内容定义 Details Definitions

N/A

三 参考文献 References

N/A

Unisoc Confidential For hiar

目 录 Contents

声明 Statement.....	2
关键字 Keywords.....	3
版本历史 Revision history	4
前 言 Foreword	6
1. 概览 Overview.....	10
2. 指纹方案软件架构	10
2.1 REE 方案	10
2.2 TEE 方案	11
3. Trusty API	11
3.1 原生 Trusty API	11
3.2 扩展 API.....	12
3.2.1 安全存储	12
3.2.2 Keymaster	12
4. 开发文档.....	12
5. SDK.....	13
5.1 目录结构	13
5.2 编译步骤	13
5.3 动态 TA 支持方式	14
5.4 动态 TA 签名	15
5.4.1 动态 TA 签名方法	15
5.4.2 更改默认动态 TA 密钥的方法.....	16
5.5 代码结构	16
5.6 CA(Client Application).....	17
5.7 TA(Trusted Application).....	18
5.8 TA manifest	20
5.9 编译配置	21
6. 开发步骤.....	22
6.1 TOS 环境.....	22
6.2 指纹 REE 侧驱动开发	23
6.3 指纹 CA、TA 开发.....	23
6.3.1 CA 开发.....	23
6.3.2 TA 开发	23
6.3.3 CA、TA 通讯	24

6.3.4	SPI 接口	25
6.3.5	Storage 接口	26
6.3.6	获取随机数函数接口	29
6.3.7	HMAC 接口	30
6.3.8	获取 auth token key 接口	30
6.3.9	Selinux 配置	31
7.	Debug	32
7.1	CA、TA log	32
7.2	TA 未成功加载	32
7.3	TA 未加载	33
7.3.1	TOS 不支持	33
7.3.2	tsuppllicant 模块未编译	33
7.4	Storage 存储失败	34
7.5	Spi 异常	34
7.6	Trusty crash	34
8.	发布	36
8.1	客户提供给指纹厂商	36
8.2	指纹厂商提供给客户	36
8.3	客户集成	37
9.	多指纹兼容方案	37
9.1	方案概述	37
9.2	软件架构	38
9.3	软件流程图	39
9.3.1	匹配指纹初始化成功流程	39
9.3.2	匹配指纹初始化失败流程	40
9.4	代码修改	40
9.4.1	BiometricsFingerprint	40
9.4.2	Linux 指纹驱动	41
9.4.3	指纹 CA、TA 修改	41
9.4.4	CA、TA 集成方式修改	42
9.5	注意事项	42
10.	工厂测试	42
10.1	接口说明	43
10.2	Unisoc PC 工具以及手机工厂测试程序需求说明	43
10.3	工厂测试库 release 要求	43
10.4	工厂测试中的多指纹兼容	44
10.4.1	BBAT 工具测试	44
10.4.2	Native MMI test	44
10.4.3	MMI apk test	45
10.5	注意事项	46

11. 支付	46
11.1 IFAA	47
11.2 SOTER.....	49
12. 参考代码.....	50
12.1 工厂测试 fingersor.c.....	50

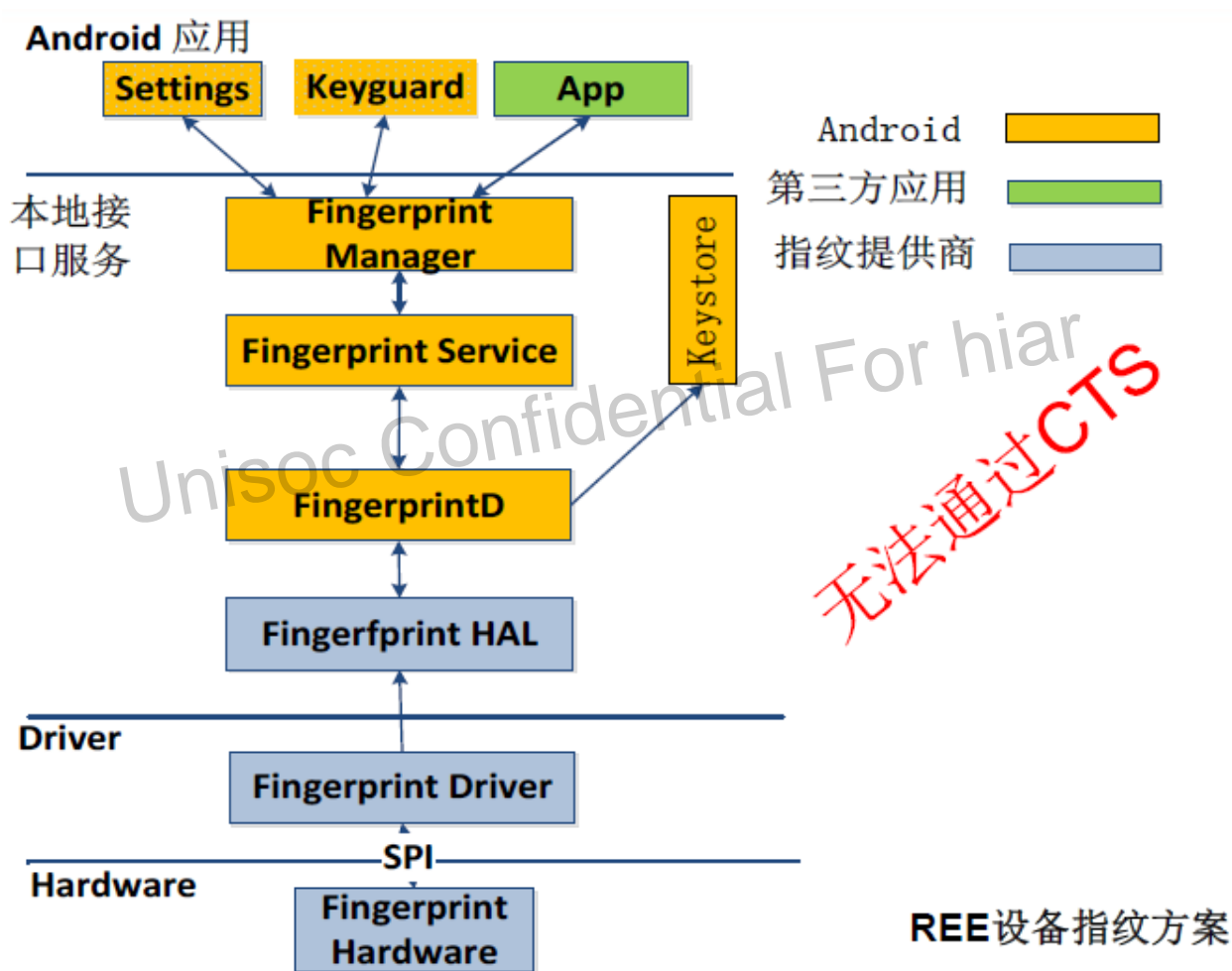
Unisoc Confidential For hiar

1. 概览 Overview

本文档用于指导指纹厂商和客户基于展锐 TEE 方案开发、适配指纹功能。

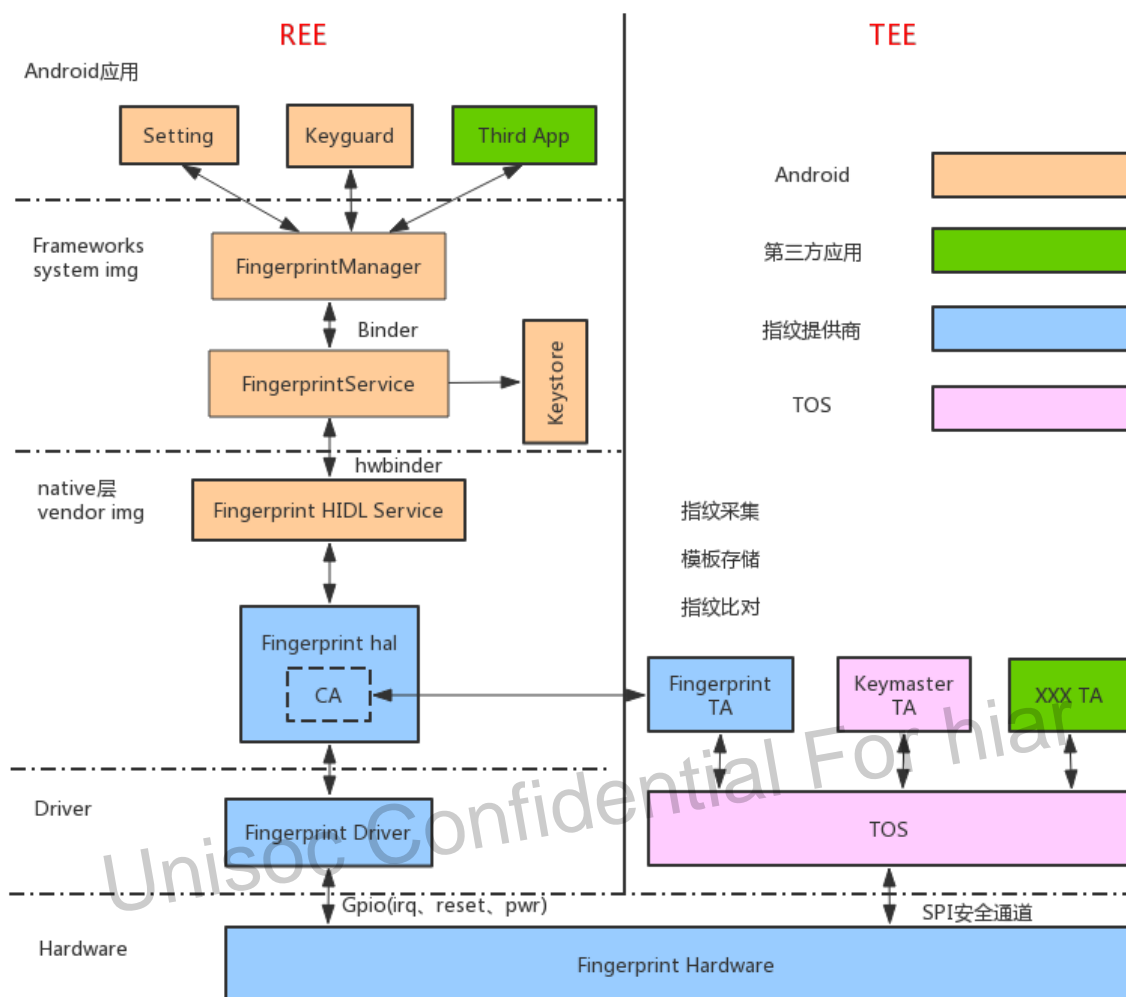
2. 指纹方案软件架构

2.1 REE 方案



图片- 1 指纹 REE 方案架构图

2.2 TEE 方案



图片- 2 指纹 TEE 方案架构图

相对 REE 方案的更改点：

- 1) 需要开发 CA、TA，算法、SPI、模板保存读取都在 TA 内部；
- 2) REE 侧指纹驱动只负责电源和中断。

3. Trusty API

3.1 原生 Trusty API

主要为 IPC 类 API，请参考 <https://source.android.com/security/trusty/trusty-ref>

3.2 扩展 API

3.2.1 安全存储

主要包括以下接口，具体使用方法请参考 [6.3.5](#) 章节。

~~storage_open_session~~（不推荐使用）

storage_open_session_async

storage_open_file

storage_close_session

storage_close_file

storage_get_file_size

storage_read

storage_write

storage_delete_file

storage_end_transaction

3.2.2 Keymaster

主要包括以下接口：

keymaster_open

keymaster_close

keymaster_get_auth_token_key

4. 开发文档

请参考 Google Trusty 官网：<https://source.android.com/security/trusty>

5. SDK

5.1 目录结构

```
├─ Makefile          -> Root makefile
├─ headers
│   └─ external      -> LK header files
│       └─ headers
│           └─ lk
│               └─ openssl
│   └─ lib            -> User lib header files
│       └─ include
│           └─ interface
│               └─ lib
│   └─ lk              -> Trusty header files
│       └─ trusty
│   └─ system
│       └─ gatekeeper
│           └─ keymaster
├─ make               -> Make scripts
├─ prebuilts          -> Prebuild modules to be linked
│   └─ lib             -> User libraries
└─ source              -> Make scripts
    └─ external
        └─ lk
            └─ lk
                └─ trusty
```

5.2 编译步骤

1. 设置 GCC 工具链路径，推荐使用 Android 代码包中自带的 arm-eabi-4.8 工具链：

```
export PATH=$PATH:<AOSP>/prebuilts/gcc/linux-x86/arm/arm-eabi-4.8/bin
```

2. 编译

编译命令格式为：

```
make M="[MOD-1] [MOD-2]:[TA] ..."
```

编译 TA 模块时需加上参数 :TA，如编译 tademo 的命令为：

```
make M="app/demo/tademo:TA"
```

编译完成后会在 SDK build 目录下生成 TA。

5.3 动态 TA 支持方式

Android 9.0 将只以动态加载的方式支持指纹 TA，TA 文件放在 android 侧，开机不会默认执行，只有当 CA 调用 TA 后，trusty 才会将 TA 加载到 tos 内存中，并随后运行此 TA。注意：动态 TA 需要预先签名，请参考 5.4 章节。

相对于早期静态加载方式的指纹 TA，从代码的角度，指纹 TA 无需做任何修改。

需要修改的地方如下：

1. TA 不再打包进 tos，直接放置在手机中的如下目录：

/vendor/firmware/

可以参考如下.mk 中的方式：

```
# xxxxxx TA

PRODUCT_COPY_FILES                                     +=
vendor/sprd/partner/xxxxx/xxxxx/vendor/firmware/signed/xx_trusty.elf:/vendor/firmware/xx_
trusty.elf
```

2. TA 名称的命名规则：

动态 TA 的加载是根据 CA、TA 之间 IPC 的 port 名来查找的，规则如下：

将 port 名以.分隔出的最后一部分+ .elf

比如代码中定义#define TADEMO_PORT "com.android.trusty.tademo"

则 TA 需要命名成 tademo.elf

动态 TA 加载 log:

```
1. s9863a1h10:/ # cat proc/kmsg|grep trusty
2. <6>[ 68.183174] c4 246 trusty: tam_load_request:932: com.android.trusty.tademo
3. <6>[ 68.183198] c4 246 trusty: handle_conn_req:406: failed (-2) to send response
4. //这个 fail 表示没有找到 TA,需要动态加载,属于正常 log
5. <6>[ 68.183207] c4 246 trusty: ta_manager_wait_load:382: ta_manager_wait_load com.android.trusty.tademo
6. <6>[ 68.185949] c4 246 trusty: ta_manager_write_ta:485: ta_manager_write_ta: new ta!
7. //开始读取 ta elf 到 tos 内存
8. <6>[ 68.188528] c0 181 trusty: ta_manager_write_ta:573: ta_manager_write_ta, load com.android.trusty.tademo accomplished!
9. //动态加载完成
```

5.4 动态 TA 签名

由于 tos 中开启了动态 TA 的签名校验,需要对 5.2 章节中编译生成的指纹 TA 先做签名,如果没有签名或签名的密钥不匹配会导致动态 TA 加载失败。

当动态 TA 没有签名或密钥不匹配时, ta_manager_handle_request 将会返回 ERR_NOT_ALLOWED, 即-17:

```
1. <6>[ 30.866766] c1 trusty: ta_manager_write_ta:538: ta_manager_write_ta: new ta!
2. <6>[ 30.999062] c0 trusty: ta_manager_handle_msg:760: ta_manager_handle_request failed -17!
```

5.4.1 动态 TA 签名方法

密钥位置: vendor/sprd/proprieties-

source/packimage_scripts/signimage/sprd/config/dynamic_ta_privatekey.pem

使用 vendor/sprd/proprieties-source/packimage_scripts/signimage/dynamicTA/ 目录下的 signta.py 进行签名。

运行方法：

```
python signta.py --uuid {UUID} --key “私钥” --in “未签名的 TA 文件名” --out “已签名的 TA 文件名”
```

以 demo 为例，执行如下命令：

```
python signta.py --uuid 4304bef636e54d9094b01ea4cd51d40b --key ../sprd/config/dynamic_ta_privatekey.pem --in tademo.elf.org --out tademo.elf
```

5.4.2 更改默认动态 TA 密钥的方法

5.4.1 章节中“密钥位置”默认放置的是平台测试用的密钥，从安全角度考虑，客户需要自己生成密钥替换掉此密钥。

密钥生成方法：

```
openssl genrsa -out dynamic_ta_privatekey.pem 1024
```

用生成的密钥替换掉上面目录的密钥，密钥名称需保持与默认的一致。

更换密钥后需要：

1. 重新编译 vendor img (目前 faceid TA 签名机制在编译阶段完成签名，保证 faceid TA 使用新密钥签名)；
2. 把原始的 tos.bin 放入 out 目录，重新 make 生成 tos-sign.bin；
3. 用新密钥对指纹 TA 签名。

5.5 代码结构

其中 nativetademo 是本地 TA，指纹功能用不到，指纹 CA、TA 开发参考 cademo 和 tademo。

└─ cademo


```

|   |— Android.mk
|   |— cademo.cpp
|   |— cademo.h
|   |— cademo_ipc.c
|   |— cademo_ipc.h
|— nativetademo
|   |— manifest.c
|   |— nativetademo.c
|   |— rules.mk
|— readme
|— tademo
|   |— ipc
|   |   |— rules.mk
|   |   |— tademo_ipc.cpp
|   |   |— tademo_ipc.h
|   |— LICENSE
|   |— manifest.c
|   |— rules.mk
|   |— trusty_tademo.cpp
|   |— trusty_tademo.h

```

5.6 CA(Client Application)

CA 主要包含两个文件，cademo_ipc.c 和 cademo.cpp 。

cademo_ipc 实现了 IPC 部分代码，包括 trusty_cademo_call、trusty_cademo_connect、trusty_cademo_disconnect 函数，供 cademo 调用，自定制 CA 需要修改 cademo_ipc.h 中的几个定义，其中 tademo_message 只需要修改名称，结构体本身的定义无需修改。

trusty_cademo_call 参数说明：

```

int trusty_cademo_call(uint32_t cmd, void *in, uint32_t in_size, uint8_t *out,
                      uint32_t *out_size)

```

```
// cmd: uint32_t - cmd id, 用来标识请求的具体类别 ;

// in: void * - 包含要发送给 TA 的具体内容的 buffer ;

// in_size: uint32_t - in buffer 的大小, 最大可为 4076 ( 详情可参考通讯大小限制说明 ) ;

// out: uint8_t * -是用来接收从 TA 回传数据的 buffer , 需保证实现申请的 out buffer 的空间不
小于实际要接受的有效数据+ sizeof(struct tademo_message) ;

// out_size: uint32_t * -此参数既是入参, 也是出参, 作为入参时设置的大小为 out buffer 所
申请空间的大小, 最大有效值为 4080 ( 详情可参考通讯大小限制说明 ) , 此处传入地址, 在里面将
可能会根据 TA 传回数据的大小作进一步修改, 应保证可能被修改的实传的大小不大于作为入参时
传入的大小。
```

注意：

1. CA TA 之间多次 IPC 通讯, 每次 IPC 通讯, CA 接收和发送的 buffer 都需要重新 malloc , 不能多次 IPC 使用同一个 buffer , 否则可能会导致通讯失败。
2. 当 CA 调用 connect 的时候, 动态 TA 机制会加载 TA , 跑 TA 的 main 函数, 当 CA 调用 disconnect 的时候, TA 进程会退出。故而在一个生命周期中, 无需每进行一次 ipc 通信就分别 connect、disconnect 一次, 如构造函数中 connect 一次后, 可进行 n 次 ipc 通信, 最后在析构函数中保证 disconnect ;
3. 当出现 “Message too long” 报错时, 首先对照上述参数说明检查 trusty_cademo_call 中的参数 : out 和 out_size。

5.7 TA(Trusted Application)

tademo_ipc 实现了 TEE 侧 IPC 部分代码, 除了名字修改外, 针对具体业务, 主要需要修改 :

handle_request 函数，其中 in_buf 为 CA 传过来的数据，out_buf 为 TA 返回给 CA 的数据，这个函数需要把 CA 传过来的数据在 TA 中处理后将返回的数据放到 out_buf 中返回给 CA。

注意：

1. tademo_ipc_init 中分别给出了与其它 TA 通信的 secure port 和与 CA 通信的 non-secure port 的创建 demo，但若没有与其它 TA 直接通信的需求，无需创建 secure port，目前指纹 TA 即是如此，无需保留类似如下代码：

```
/* Initialize secure service , other TA will use this*/

rc = port_create(TADEMO_SECURE_PORT, 1, TADEMO_MAX_BUFFER_LENGTH,
                IPC_PORT_ALLOW_TA_CONNECT);

if (rc < 0) {
    TLOGE("Failed (%d) to create port %s\n", rc, TADEMO_SECURE_PORT);
}

ctx->port_secure = (handle_t)rc;

rc = set_cookie(ctx->port_secure, &tademo_port_evt_handler_secure);

if (rc) {
    TLOGE("failed (%d) to set_cookie on port %d", rc, ctx->port_secure);
    close(ctx->port_secure);
    return rc;
}
```

-
2. TA main 函数的 while 循环中, 当 wait_any 返回错误时, 一定要 break , 不然 TA 无法退出。

5.8 TA manifest

TA manifest 定义了 TA UUID, TA 所使用的 heap 和 stack 的内存大小 (指纹 TA 推荐 heap 设置为 6M , stack 设置为 1M)。

不建议人为修改已有 uuid , 推荐使用 linux uuid 生成工具 uuidgen(random-based uuid) :

```
uuidgen [-r] [-t]
```

下面的 uuid 只是示意, 不要直接使用, 如果做多指纹兼容方案, 每个指纹的 uuid 必须不同。

```
trusty_app_manifest_t TRUSTY_APP_MANIFEST_ATTRS trustworthy_manifest =
```

```
{
```

```
/* UUID : {4304bef6-36e5-4d90-94b0-1ea4cd51d40b} */
```

```
{ 0x4304bef6, 0x36e5, 0x4d90,
```

```
    { 0x94, 0xb0, 0x1e, 0xa4, 0xcd, 0x51, 0xd4, 0x0b } },
```

```
/* optional configuration options here */
```

```
{
```

```
    TRUSTY_APP_CONFIG_MIN_HEAP_SIZE(6*1024*1024),
```

```
    TRUSTY_APP_CONFIG_MIN_STACK_SIZE(1 * 1024 * 1024),
```

```
},
```

```
};
```

5.9 编译配置

CA 代码的编译配置可以参考 Android native 程序。

Trusty TA 使用的 makefile 文件为 rules.mk, 类似 Android.mk。有几个主要的 makefile 变量必须定义, 如下:

MODULE :	TA 在 Trusty 中的 module 名称
MODULE_SRCS :	所有代码文件
MODULE_DEPS :	依赖的内部代码模块
MODULE_DEPS_STATIC :	依赖的内部预编译模块
MODULE_INCLUDES :	依赖的头文件

此外还必须同时包含 module-user_task.mk 和 module.mk 两个 makefile

实例文件:

```
LOCAL_DIR := $(GET_LOCAL_DIR)

MODULE := $(LOCAL_DIR)

MODULE_SRCS += \
    $(LOCAL_DIR)/manifest.c \
    $(LOCAL_DIR)/trusty_tademo.cpp \

IPC := ipc

MODULE_DEPS += \
    app/trusty

MODULE_DEPS_STATIC := \
    libc \
```

```

    libc-trusty \
    libstdc++-trusty

MODULE_INCLUDES += \
    $(LOCAL_DIR) \

include $(LOCAL_DIR)/$(IPC)/rules.mk

include make/module-user_task.mk

MODULE_DEPS_STATIC += <外部预编译库文件>

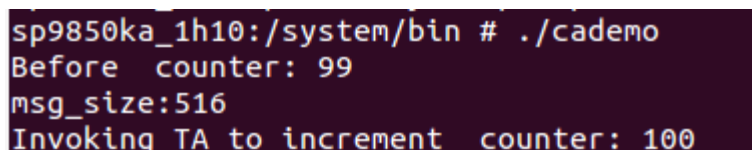
include make/module.mk

```

6. 开发步骤

6.1 TOS 环境

1. 参考 8.1 章节所列清单，获取完整的开发环境；
2. 环境准备：SDK 环境以及客户完整代码编译环境；
3. 参考 5.2 5.4 5.3 章节编译、签名并 push tademo 动态 TA；
4. 在客户完整代码环境中编译 CA，编译出 cademo，把 cademo push 进手机；也可以尝试使用 NDK 编译，NDK 编译需要使用 libtrusty.so 库和头文件 system/core/trusty/libtrusty/include/trusty/tpic.h（这两个文件从客户代码环境中取，lib 有 32 位和 64 位，注意区分）。推荐使用完整代码环境编译；
5. adb 进去执行 cademo，看是否正常。CA、TA 工作流程为 android 侧的 CA 发送数字 99 到 TA，TA 中对它加 1 然后返回，出现下图表明 CA、TA 工作正常，tos 环境 ok。



```

sp9850ka_1h10:/system/bin # ./cademo
Before counter: 99
msg_size:516
Invoking TA to increment counter: 100

```

6.2 指纹 REE 侧驱动开发

REE 侧驱动开发请与指纹供应商联系。REE 侧通常只有中断驱动，SPI 相关的驱动在 TEE 中实现。

开发完成后通过查看中断确认功能是否 OK。

注意：指纹设备节点的权限配置请放在 on post-fs-data 的触发中，否则可能会导致权限配置失败。如：

vendor/sprd/partner/sunwave/init.sunwave.rc

```
on post-fs-data
#add for sunwave fingerprint
    chmod 660 /dev/sunwave_fp
    chown system system /dev/sunwave_fp
```

6.3 指纹 CA、TA 开发

6.3.1 CA 开发

CA 是在 android 侧的，用 android.mk 编译，一般指纹 CA 会编译成 so 库文件，是上层 android.hardware.biometrics.fingerprint@2.1-service 和 TA 之间的通路，同时中断也在这里接收，转发到 TA 处理。这里需要注意要访问 dev 下面的 trusty 设备节点，selinux 等权限需要加上。这部分开发需要在客户的完整代码环境中进行。

6.3.2 TA 开发

1 编译

TA 使用 trusty SDK 编译，编译后会在 build 目录中生成 tademo.elf，签名并 push 到手机的/vendor/firmware/目录下调试。

CA、TA 之间的 IPC 通讯参考 cademo 和 tademo。API 文档：

<https://source.android.com/security/trusty/trusty-ref>

2 uuid 说明

在指纹 TA 的 manifest 中定义 uuid，参考下面的数值，多指纹方案不同指纹 TA uuid 必须不同。

```
uuid_t fpsensor_uuid = {  
    0x4304bef6, 0x36e5, 0x4d91, { 0x94, 0xb0, 0x1e, 0xa4, 0xcd, 0x51, 0xd4,  
    0x0c }};
```

3 TA 加载静态库

如下例 TA 加载.a 库：

```
include make/module-user_task.mk  
MODULE_DEPS_STATIC += $(shell pwd)/app/fp_ta/lib/xxx.a  
include make/module.mk
```

6.3.3 CA、TA 通讯

1. Port 说明

CA、TA 通讯定义相同的端口，比如（建议各个厂商使用自己的名字，避免冲突）：

"com.android.trusty.xxxxx"

修改如下 port 名字：

```
#define TADEMO_PORT "com.android.trusty.tademo"
```

目前 tademo 中定义了两个 port，secure port 是给其他 TA 访问 tademo 用的，CA 调用 TA 的场景属于 non secure，在没有与其它 TA 直接通信需求的情况下，只需创建 non-secure port 即可，secure port 的创建无需保留。详见 5.7 章节。

2. 通讯大小限制

CA TA 之间通讯有 size 限制，算上消息处理头等，整体大小限制为 4KB，故 tademo_ipc.h

中的 TADEMO_MAX_BUFFER_LENGTH 最大可定义为 (1024*4), 但相应的实际可用于 CA TA 间有效传输的数据大小限制为 4076 Byte , 也即 cademo.cpp 中的 SEND_BUF_SIZE 的最大限制值为 4076 (1024*4 - sizeof(struct tademo_message) - sizeof(struct tipc_msg_hdr) = 1024*4 - 4 - 16), 不同于 send_buf 只是用于组装结构体 tademo_message 对象中的不定长 payload 部分, recv_buf 是用于接收完整结构体 tademo_message 对象的, 故 RECV_BUF_SIZE 的最大有效值为 4080 (1024*4 - sizeof(struct tipc_msg_hdr) = 1024*4 - 16), tademo 中的默认设置 TADEMO_MAX_BUFFER_LENGTH 为 1024 , 若需要增大请修改创建 port 时的指定 size(即 demo 中的 TADEMO_MAX_BUFFER_LENGTH)以及 manifest 中堆栈大小。在实际传输时, 当 TADEMO_MAX_BUFFER_LENGTH 配置为最大值 (1024*4) 时, handle_request 中的 *out_buf_size 最大可为 4076。当要传输的数据大于最大限制时请考虑分批传输。

6.3.4 SPI 接口

spi 使用 ioctl 接口。

参考代码：

```
#include <sprd_pal_fp_default.h>
```

```
#include <io_device_def.h>
```

```
int32_t ret;
```

```
struct WRITE_THEN_READ_STR wr;
```

```
wr.max_speed_hz = 6000000;

wr.chip_select = 0;

wr.mode = 0;

wr.bits_per_word = 8;

wr.number = 2;

wr.len = 3;

wr.rxbuf = (uint8_t*)malloc(wr.len);

wr.txbuf = (uint8_t*)malloc(wr.len);

wr.debug = 0;

memset(wr.txbuf,0,wr.len);


ret = ioctl(IO_DEVICE_FP, SPI_WRITE_AND_READ, &wr);


free(wr.rxbuf);

free(wr.txbuf);
```

WRITE_THEN_READ_STR 结构体说明： max_speed_hz 设置频率； mode 设置 spi 模式； number 设置哪个 spi，比如 spi2，则 number 为 2； debug 是调试开关，建议在调试时设置为 1，合入版本时更改为 0。

6.3.5 Storage 接口

rule.mk 中增加 deps，如下：

```
MODULE_DEPS_STATIC += \  
    storage \
```

参考代码：

```
#include <lib/storage/storage.h>  
  
bool test_secure_storage_write() {  
    storage_session_t session;  
  
    uint32_t i;  
  
    uint8_t record[5000];  
  
    TLOGE("test_secure_storage write begin \n");  
  
    for(i=0;i<5000;i++)  
        record[i] = 88;  
  
    int rc = storage_open_session_async (&session, STORAGE_CLIENT_TD_PORT, 100);  
  
    if (rc < 0) {  
        TLOGE("Error: [%d] failed to open storage session\n", rc);  
  
        return false;  
    }  
  
    file_handle_t handle;
```

```

rc = storage_open_file(session, &handle, "file_test", STORAGE_FILE_OPEN_CREATE, 0);

if (rc < 0) {

    TLOGE("Error: [%d] failed to open storage object\n", rc);

    storage_close_session(session);

    return false;

}

rc = storage_write(handle, 0, record, 5000*sizeof(uint8_t), 0);

storage_close_file(handle);

storage_end_transaction(session,true);

storage_close_session(session);

if (rc < 0) {

    TLOGE("Error:[%d] writing storage object.\n", rc);

    return false;

}

TLOGE("test_secure_storage write end \n");

return true;

}

```

注意这里用到的堆栈超过了 tademo 中的 manifest 默认配置 如需要测试这部分 sample code , 需要修改 manifest 配置

关于文件性能及同步，建议如下：

@opflags, 设置为 0,不要设置为 STORAGE_OP_COMPLETE

```
int storage_delete_file(storage_session_t session, const char *name, uint32_t opflags)
```

@opflags, 设置为 0,不要设置为 STORAGE_OP_COMPLETE (get size 前的 open 时设置为 STORAGE_OP_COMPLETE)

```
int storage_open_file(storage_session_t session, file_handle_t *handle_p, const char *name, uint32_t flags, uint32_t opflags)
```

@opflags, 设置为 0,不要设置为 STORAGE_OP_COMPLETE

```
ssize_t storage_write(file_handle_t fh, storage_off_t off,
                      const void *buf, size_t size, uint32_t opflags)
```

文件有修改时，统一在 storage_close_file()后用 storage_end_transaction()进行提交；

使用 storage_open_session_async 替换 storage_open_session，且需在调用 storage_open_session_async 成功后于结束前调用 storage_close_session。

6.3.6 获取随机数函数接口

rule.mk 中增加 deps，如下：

```
MODULE_DEPS_STATIC += \
    rng \
    openssl \
```

参考代码：

```
#include <lib/rng/trusty_rng.h>
```

```
uint8_t data[512];

trusty_rng_secure_rand(data, 32);
```

6.3.7 HMAC 接口

```
#include <openssl/hmac.h>

uint8_t buf[32];

size_t buf_len;

HMAC(EVP_sha256(), key, key_length, message, length, buf, &buf_len);
```

6.3.8 获取 auth token key 接口

rule.mk 中增加 deps , 如下 :

```
MODULE_DEPS_STATIC += \
    keymaster \
```

参考代码 :

```
#include <lib/keymaster/keymaster.h>

bool GetAuthTokenKey(uint8_t **auth_token_key, size_t *length){

    *length = 0;

    *auth_token_key = NULL;

    long rc = keymaster_open();

    if (rc < 0) {

        return false;
```

```

    }

    keymaster_session_t session = (keymaster_session_t) rc;

    uint8_t *key = NULL;

    uint32_t local_length = 0;

    rc = keymaster_get_auth_token_key(session, &key, &local_length);

    keymaster_close(session);

    if (rc == NO_ERROR) {
        *auth_token_key = key;
        *length = local_length;

        return true;
    } else {
        return false;
    }
}

```

6.3.9 Selinux 配置

Trusty 相关的配置目录：

vendor/sprd/proprieties-

source/sprdtrusty/vendor/sprd/modules/common/sepolicy_androidp/

指纹厂商的 selinux 配置放在各自的目录下，确认.mk 中有将相应目录添加到 BOARD_SEPOLICY_DIRS 即可。如在 vendor/sprd/partner/xxxxx/sharklE/xxxxx.mk 中：

```
BOARD_SEPOLICY_DIRS += vendor/sprd/partner/xxxxx/sepolicy
```

举例增加 vendor/sprd/partner/xxxxx/sepolicy/hal_fingerprint_default.te 文件：

```
.....  
  
# allow HAL module to access '/dev/trusty-ipc-dev0'  
  
allow hal_fingerprint_default teetz_device:chr_file {open ioctl read write};  
  
.....
```

7. Debug

7.1 CA、TA log

CA log 打印在 android log 中，TA log 打印在 kernel log 中，导出 ylog 查看：

```
adb pull /storage/emulated/0/ylog ./ylog
```

必要情况下可以抓串口 log。

7.2 TA 未成功加载

如 kernel log 中有加载 TA，但最终加载失败，最常见的错误即-17，此种情况基本是以下一种或几种原因所致：

- TA 未签名；
- TA 未正确签名；

- 更改密钥后未更新 tos-sign.bin ;
- 更改密钥后未更新 vendor img (此种情况影响 faceid TA 的成功加载)

```
#define ERR_NOT_ALLOWED          (-17)
```

```
1.  <6>[ 30.407326] c3 trusty: tam_load_request:1009: com.android.trusty.tademo
2.  <6>[ 30.407331] c3 trusty: handle_conn_req:406: failed (-2) to send response
3.  <6>[ 30.407336] c3 trusty: ta_manager_wait_load:414: ta_manager_wait_load com.android.trusty.tademo
4.  <6>[ 30.866766] c1 trusty: ta_manager_write_ta:538: ta_manager_write_ta: new ta!
5.  <6>[ 30.999062] c0 trusty: ta_manager_handle_msg:760: ta_manager_handle_request failed -17!
```

解决方法请参考 5.4 章节。

7.3 TA 未加载

7.3.1 TOS 不支持

这种情况在 kernel log 中搜索 TA 包名中的关键字以及 “ta_manager” 均没有结果，因为 tos 中没有包含 tamanager，无法支持动态 TA，需确认取用的 tos.bin 是否是对应的 BOARD_TEE_LOW_MEM 为 true 的版本，必须取用 BOARD_TEE_LOW_MEM 为非 true 的版本。

```
1.  <6>[ 5.553425] c4 51 trusty: initializing ta_manager enter
```

7.3.2 tsuppllicant 模块未编译

tsuppllicant 是 tamanager 的 CA 实现，故缺少 tsuppllicant 也无法支持动态 TA，可通过命令行查看：

```
$ adb shell ps -A | grep 'tsuppllicant'
system          387      1  11124   3240 tipc_read_iter 7434bafccc S tsuppllicant
```

若无此进程请在 device/sprd/xxxxx/仓库中搜索 BOARD_TEE_LOW_MEM，若其被定义为 true 则不支持编译 tsuppllicant 模块，需切换到 BOARD_TEE_LOW_MEM 为非 true 的 board 中开

发，不建议直接在 board 中更改此值，dts 中 tos-mem 配置不匹配时可能会出现不能开机的问题。

device/sprd/xxxxx/common/security_feature.mk

```
ifneq ($(BOARD_TEE_LOW_MEM),true)
```

```
PRODUCT_PACKAGES += \
```

```
    tsuppllicant
```

```
endif
```

7.4 Storage 存储失败

1. 检查 android 侧 sprdstorageproxyd 进程是否启动正常；
2. 查看 android log 中是否有权限等异常打印；
3. 查看 TA log，查看调用 storage 接口后是否有返回，是否有异常打印。

7.5 Spi 异常

1. 首先确认软件，修改传入参数 debug 为 1，复测，查看 spi setup 和 spi sync 的 ret 是否为 0，0 表示正常；
2. 如果软件正常，则飞线查看硬件信号确认失败原因；
3. 确认 pinmap 是否配对。

7.6 Trusty crash

1. 在 kernel log 中搜索 “HALT” 关键字；

crash 现场：

```
1.  <6>[ 53.217730] c0 trusty: trusty_tademo: 276: float test enter =====
```

```

2. <6>[ 53.217737] c0 trusty: undefined abort, halting
3. <6>[ 53.217745] c0 trusty: r0 0x00000030 r1 0x80000000 r2 0x40a94d99 r3 0x00000000
4. <6>[ 53.217753] c0 trusty: r12 0x709f8100 usp 0x00fffee8 ulr 0x000089cd pc 0x000089cc
5. <6>[ 53.217760] c0 trusty: spsr 0x60000030
6. <6>[ 53.217768] c0 trusty: *usr r13 0x00fffee8 r14 0x000089cd
7. <6>[ 53.217775] c0 trusty: fiq r13 0xbc646bde r14 0x986ba721
8. <6>[ 53.217782] c0 trusty: irq r13 0x00000000 r14 0x000089c0
9. <6>[ 53.217789] c0 trusty: svc r13 0x8fb9e860 r14 0x8f600b11
10. <6>[ 53.217796] c0 trusty: und r13 0x00000000 r14 0x000089ce
11. <6>[ 53.217804] c0 trusty: sys r13 0x00fffee8 r14 0x000089cd
12. <6>[ 53.217810] c0 trusty: HALT: (reason = 9)

```

其中 pc 指针地址为 0x000089cc。

或：

```

1. <6>[ 136.273070] c5 6 trusty: sync_exception
2. <6>[ 136.273077] c5 6 trusty: iframe 0xfffffffffe05fd360:
3. <6>[ 136.273089] c5 6 trusty: x0 0x 0 x1 0x 5f x2 0x 9b9a8 x3 0x
0
4. <6>[ 136.273096] c5 6 trusty: x4 0x 9b9a8 x5 0x 0 x6 0xfffffffffe0000001 x7
0x 0
5. <6>[ 136.273108] c5 6 trusty: x8 0xfffffffffe0009b9a8 x9 0x fffeac x10 0xfffffffffe00ab08c
x11 0x 9bda8
6. <6>[ 136.273120] c5 6 trusty: x12 0xfffffffffe0000001 x13 0x fff888 x14 0x 8d30 x15
0xfffffffffe022dba8
7. <6>[ 136.273127] c5 6 trusty: x16 0x 1 x17 0x 0 x18 0x 0 x19
0xfffffffffe0320518
8. <6>[ 136.273141] c5 6 trusty: x20 0x 0 x21 0x 0 x22 0x 0 x23
0x 0
9. <6>[ 136.273152] c5 6 trusty: x24 0x 0 x25 0x 0 x26 0x 0 x27
0x 0
10. <6>[ 136.273163] c5 6 trusty: x28 0x 0 x29 0xfffffffffe05fd470 lr 0xfffffffffe0002184 sp
0xfffffffffe05fd450
11. <6>[ 136.273170] c5 6 trusty: elr 0x c9c8
12. <6>[ 136.273178] c5 6 trusty: spsr 0x 60000010
13. <6>[ 136.273186] c5 6 trusty: ESR 0x92000007: ec 0x24, il 0x1, iss 0x7
14. <6>[ 136.273193] c5 6 trusty: HALT: (reason = 9)
15. <6>[ 136.273204] c5 6 trusty: HALT: terminate user thread trusty_tapp_1_4304bef6-36e5-4d9

```

其中 elr 指针地址为 0xc9c8。

2. 找到这个指针对应的代码。

从上面的 log 或 uuid 看出是在 tademo 出错，用 tademo 的符号表来解析。

找到出错版本对应的 tademo.syms.elf，执行 addr2line 找到出错的行：

arm 的相关 bin 在工程 prebuilts/gcc/linux-x86/arm/arm-eabi-4.8/bin 目录中

```
./arm-eabi-addr2line -f -e tademo.syms.elf 0x000089cc  
handle_request  
/vendor/sprd/proprieties-  
source/sprdtrusty/app/tademo/ipc/tademo_ipc.cpp:278
```

可以看到出错的是在 tademo_ipc.cpp 的 278 行。

8. 发布

8.1 客户提供给指纹厂商

1. SDK；
2. catademo；
3. 完整代码编译环境或 CA 用到的库文件：
 - system/core/trusty/libtrusty/include/trusty/tipc.h
 - out/target/product/xxxxx_xxx/vendor/lib(64)/libtrusty.so
4. 告知应基于哪个 board 开发，且确认此 board 中 BOARD_TEE_LOW_MEM 宏定义为非 true 值，且提供的 pac 包在集成时用到的 tos.bin 取用的应当是 BOARD_TEE_LOW_MEM 为非 true 的 board 的对应版本。

8.2 指纹厂商提供给客户

1. 指纹 TA fp_ta.elf 和 fp_ta.syms.elf（带符号表，用于 debug）；
2. 指纹 CA，上层指纹开关，selinux 权限；

-
3. 指纹驱动。

8.3 客户集成

1. 通过 patch 的形式合入指纹厂商的修改；
2. 将指纹厂商提供的 TA 做签名后，通过.mk 的方式 copy 到/vendor/firmware/目录，或手动 push 到手机中的该目录，具体参考 5.4 和 5.3 章节。

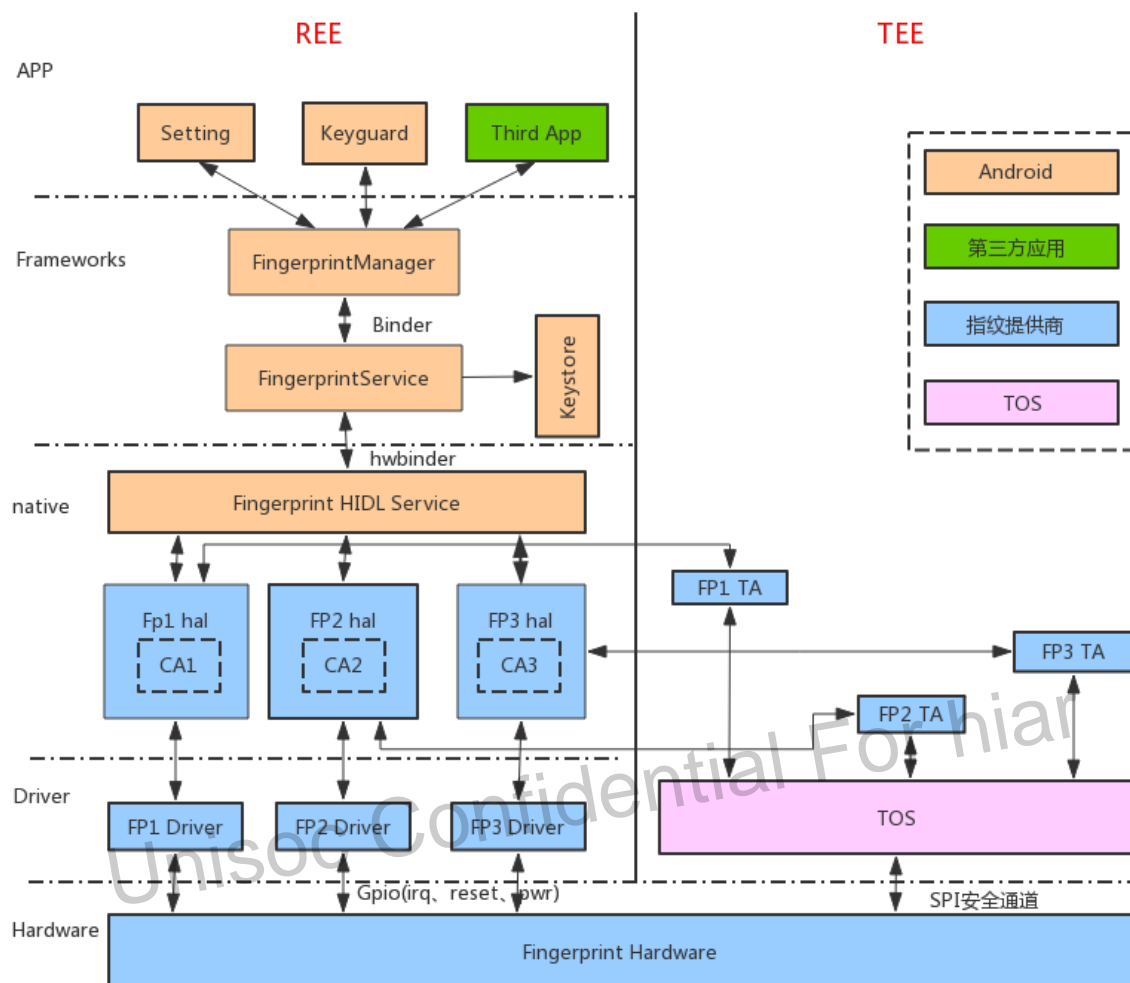
9. 多指纹兼容方案

9.1 方案概述

在同一软件方案的条件下，可支持根据实际安装的指纹模组硬件实现软件上的自动匹配的功能，从而达到兼容多个指纹硬件的目的。

开机后多个 kernel 侧指纹驱动启动并创建不同的设备节点；指纹 native 层的 android.hardware.biometrics.fingerprint@2.1-service 启动，按数组中的次序尝试加载相应指纹厂商的 hal 库，通过其 hal 库继续调用到 CA，CA 通过 ioctl 配置驱动初始化参数，然后对其相应的指纹 TA 发起通信请求，相应 TA 动态加载后通过 spi 读取 chip id，若读取失败则表示当前安装的硬件不匹配，将结果返回给 CA，CA 通过 ioctl 销毁驱动侧的相关设备，且发起 disconnect 请求以卸载相应 TA，CA 再进一步通知到 android.hardware.biometrics.fingerprint@2.1-service，android.hardware.biometrics.fingerprint@2.1-service 继续尝试加载数组中定义的下一家指纹厂商的 hal 库，直到匹配成功或数组轮询结束为止。匹配成功后 CA 通过 ioctl 注册 irq，后面流程和正常单指纹流程相同。

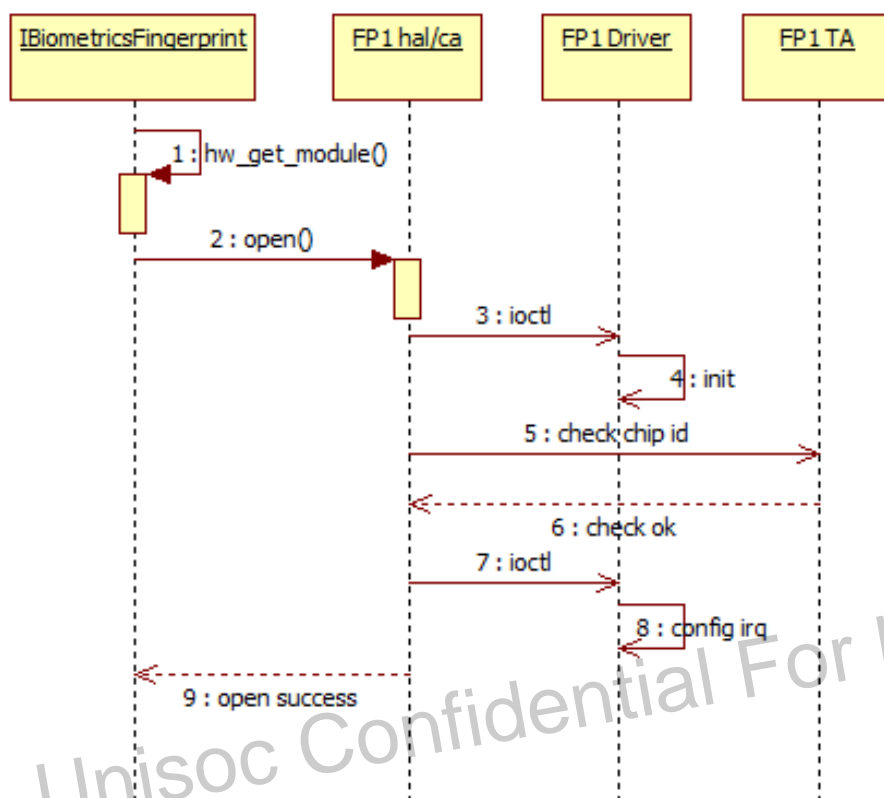
9.2 软件架构



图片-3 多指纹兼容架构图

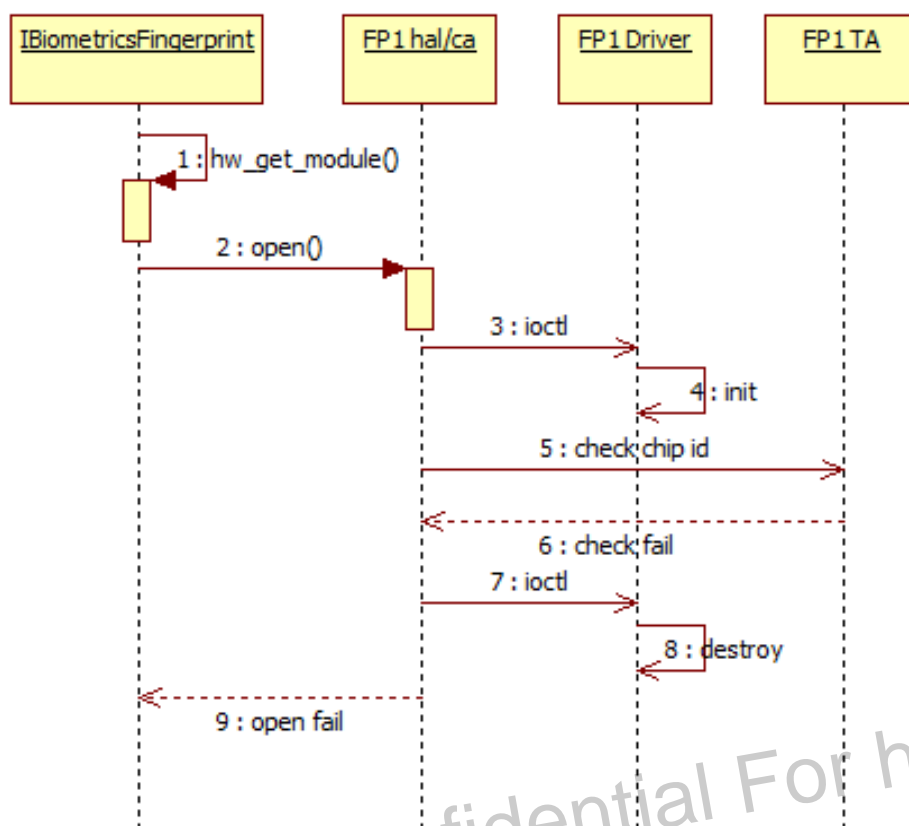
9.3 软件流程图

9.3.1 匹配指纹初始化成功流程



图片- 4 成功匹配指纹序列图

9.3.2 匹配指纹初始化失败流程



图片- 5 匹配指纹失败序列图

9.4 代码修改

9.4.1 BiometricsFingerprint

平台已在 BiometricsFingerprint::openHal 方法中增加了 for 循环 ,可尝试 load 多家指纹 hal 库文件，每家指纹厂商的库文件以不同名字来区分，相关指纹厂商 HAL MOUDLE 的配置需要在客户 code 中自行修改。

hardware/interfaces/biometrics/fingerprint/2.1/default/BiometricsFingerprint.cpp

```
/*Fp load hal to support multi fingerprint*/
```



```

#define SUNWAVE_FINGERPRINT_HARDWARE_MODULE_ID  "swfingerprint"

static const char *variant_keys[] = {

    FINGERPRINT_HARDWARE_MODULE_ID,

    SUNWAVE_FINGERPRINT_HARDWARE_MODULE_ID,

};

static const int FP_VARIANT_KEYS_COUNT =

    (sizeof(variant_keys)/sizeof(variant_keys[0]));

```

在 variant_keys[] 数组中配置相关指纹厂商的 MODULE_ID，具体值取决于指纹 hal/CA 库名，如库名为 swfingerprint.default.so，则 MODULE_ID 为“ swfingerprint”。

9.4.2 Linux 指纹驱动

这部分需要指纹厂商修改：

1. 提供 init ioctl 给 CA 调用，init ioctl 中做初始化工作，主要是 gpio 配置以及正常指纹上电，确保 TEE 侧可以通过 spi 正常读取 chip id；
2. 提供 destory ioctl 给 CA 调用，destory ioctl 中做反初始化工作，确保 chip id check fail 的情况下，将 gpio 相关配置还原，并销毁设备节点等，不影响后续指纹模块的流程；
3. 提供 config ioctl，当指纹 chip id check ok 后，CA 可以通过 config ioctl 配置 irq 等；
4. 驱动 probe 或者 module init 接口中不做配置 gpio、上电以及配置 irq 等工作，只做创建设备节点、注册设备驱动等工作，避免多个指纹驱动开机时相互影响。

9.4.3 指纹 CA、TA 修改

指纹 CA 需要增加接口到 TA 去读取 chip id，并根据 chip id 返回结果，通过 ioctl 来配置驱动相

关参数。不同的指纹 TA 使用不同的 uuid，并且 CA TA 的 IPC PORT 每个厂商需要定义成自己的，不能重复。

uuid 的生成方法请参考 5.8 章节。

9.4.4 CA、TA 集成方式修改

保证将多个指纹厂商的 CA 库均集成到/vendor/lib(64)/hw/目录中；

将多个指纹厂商的 TA 分别签名，将签名后的 TA 均放置到/vendor/firmware/目录下。

9.5 注意事项

1. 指纹占用空间的大小约等于指纹堆、栈、elf 大小之和，在单指纹占用空间合理的情况下，且忽略 tos 内存碎片的影响，理论上支持的指纹 TA 数量不受限制；
2. 为最大限度减少 tos 内存碎片的影响，建议将对 ram 空间需求大的指纹厂商优先适配，即在 variant_keys[]数组中优先适配相应指纹厂商；
3. 多个指纹 TA 需要使用不同的 UUID；
4. CA TA 的 IPC PORT 每个厂商需要定义成自己的，不能重复；
5. 指纹厂商在现有 TEE 指纹基础上，需要按照此方案修改指纹 CA、TA、驱动、BiometricsFingerprint 适配。
6. 多指纹方案在工厂测试中同样也需要做适配，具体将在第 10 章节讲述。

10. 工厂测试

支持 Unisoc 平台指纹工厂测试需要实现的接口以及事项说明。

10.1 接口说明

指纹厂商需要在工厂测试的 xxxfingerprint_factory.so 库中实现如下接口：

- factory_init()：工厂测试初始化。若不需要初始化，实现可为空；
- spi_test()：spi 测试，包括能读取到正确的 chip id；
- interrupt_test()：中断测试；
- deadpixel_test()：死点测试；
- finger_detect()：手指检测测试；
- factory_exit()：工厂测试结束，释放相关资源等。若无相关资源释放，实现可为空。

以上所有接口返回值为 0 表示成功；-1 表示失败。

10.2 Unisoc PC 工具以及手机工厂测试程序需求说明

PC 测试工具主要测试指纹硬件通路是否 ok，调用 spi_test 方法检测是否有指纹硬件，因此 spi_test 建议需要读取 chip id，能读到 id（检测到硬件通路 ok）返回 0，读不到返回 -1。此工具测试时间要求在 6s 内，因此请确保 spi_test 在 6s 内返回。请指纹厂商确认 PC 工具可以单独调用 spi_test 方法。若是单独调用此方法实现困难，请给出明确说明。

手机工厂测试程序会调用 10.1 章节中的所有方法，目的是确保此手机安装的指纹 sensor 功能 ok，请指纹厂商确保以上方法已涵盖指纹必要的测试。若无法涵盖请给出明确说明，无说明则默认已涵盖，通过这些接口可以确保指纹 sensor 的功能正常。

10.3 工厂测试库 release 要求

Unisoc 平台工厂测试调用 demo code 请参考 12.1 章节的 fingersensor.c。指纹厂商实现后需要按照 Unisoc 提供的 demo code 进行验证，然后将库以及自测试报告结果同步 release 给客户。

自测试报告需要包含测试 case 以及结果，可参考如下形式，但不限于此：

- case1 :正常开机 按照 12.1 章节中的调用方式编写测试程序 验证指纹工厂测试接口 pass ；
- case2 :验证过工厂测试后，进入设置中进行指纹录入，然后使用指纹解锁，工作正常 pass ；
- case3 :录入指纹后，再次使用工厂测试程序验证工厂测试接口，应可正常工作 pass ；
- case4 :测试程序单独调用 spi_test 接口，可以正常检测硬件通路 ok pass ；
-

10.4 工厂测试中的多指纹兼容

10.4.1 BBAT 工具测试

测试方法：在 PC 端运行工具测试。

指纹兼容 code 路径：

vendor/sprd/proprieties-source/autotest/interface/fingerprint/fingerprint_sensor.cpp

修改 fp_lib[] 数组，将不同指纹厂商的测试库的库名添加在此数组中。

```
static const char *fp_lib[] = {  
    "libfactorylib.so",  
};  
  
static const int FP_LIB_COUNT =  
    (sizeof(fp_lib)/sizeof(fp_lib[0]));
```

10.4.2 Native MMI test

测试方法：关机模式下按 power+volume up 键进入测试。

指纹兼容 code 路径：

vendor/sprd/proprieties-

source/autotest/interface/fingerprint/fingerprint_sensor_mmi.cpp

修改 fp_lib[]数组，将不同指纹厂商的测试库的库名添加在此数组中。

```
static const char *fp_lib[] = {  
  
    "libfactorylib.so",  
  
};  
  
static const int FP_LIB_COUNT =  
  
    (sizeof(fp_lib)/sizeof(fp_lib[0]));
```

10.4.3 MMI apk test

测试方法：在拨号盘输入“*##83789#*”进入测试 apk。

apk code 路径：

vendor/sprd/platform/packages/apps/ValidationTools/src/com/sprd/validationtools/fingerprint/
erprint/

指纹兼容 code 路径：

vendor/sprd/interfaces/fingerprintmmi/1.0/default/Fingerprintmmi.cpp

修改 fp_lib[]数组，将不同指纹厂商的测试库的库名添加在此数组中。

```
static const char *fp_lib[] = {  
  
    "libfactorylib.so",  
  
};
```

```
static const int FP_LIB_COUNT =  
  
(sizeof(fp_lib)/sizeof(fp_lib[0]));
```

10.5 注意事项

1. 做多指纹兼容方案时，建议将指纹工厂测试库 “xxxfingerprint_factory.so” 中的 xxx 以指纹厂商的名称命名用以区分；平台工厂测试代码中默认使用的工厂测试库名称为 libfactorylib.so，若项目不需要做多指纹兼容，可将指纹厂商提供的工厂测试库直接命名为 libfactorylib.so，这样就无需修改平台工厂测试代码。
2. Unisoc 不同模式或不同平台工厂测试程序既有 32 位的也有 64 位的，因此请同时提供 32 位和 64 位的.so 库。
3. Unisoc 平台会同时支持 factory mode 和正常开机模式的工厂测试程序，正常开机需要同时支持指纹工厂测试以及指纹录入解锁等功能，因此工厂测试接口不能与正常使用指纹产生冲突。
4. 上述 BBAT、Native MMI、MMI apk 三种测试方式都分别运行在自己的进程中，因此合入指纹厂商实现的测试库后，还需要添加相应的 selinux 权限配置。由于每个指纹厂商实现不同，平台无法提供 selinux 权限配置的统一方案，建议客户和指纹厂商根据 log 进行配置。

11. 支付

指纹厂商 trusty TEE 方案在指纹基础上对支付功能的支持。

11.1 IFAA

IFAA 需要实现三个接口，第一个是获取上一次鉴权成功的指纹模板 id，第二个是获取已注册的指纹模板集，第三个是保存指纹厂商编号，请帮忙按下述 sample 及描述实现：

```
enum fp_default_cmd_info{  
  
    SPI_WRITE_AND_READ = 0,  
  
    IFAA_SET_LAST_IDENTIFIED = 1,  
  
    IFAA_GET_LAST_IDENTIFIED = 2,  
  
    IFAA_SET_FP_ENROLLED_ID_LIST = 3,  
  
    IFAA_GET_FP_ENROLLED_ID_LIST = 4,  
  
    IFAA_SET_FP_VENDOR_ID = 7 ,  
  
    CMD_MAX  
  
};
```

1. 请在指纹鉴权成功后，调用 ioctl (cmd id 为 IFAA_SET_LAST_IDENTIFIED) 将指纹模板 id 写入 trusty 内核：

```
uint32_t lst_id;  
  
//added for test begin  
  
uint32_t read_val = 0;  
  
lst_id = 12345;  
  
ioctl(IO_DEVICE_FP, IFAA_SET_LAST_IDENTIFIED, &lst_id);  
  
ioctl(IO_DEVICE_FP, IFAA_GET_LAST_IDENTIFIED, &read_val);  
  
LOG_DBG("[LIYUN]-->lst_id = %d, read_val = %d\n", lst_id, read_val);  
  
//added for test end
```

-
2. 请在开机后、指纹模板有变化时调用 `ioctl (cmd id 为 IFAA_SET_FP_ENROLLED_ID_LIST)`
更新指纹模板集到 `trusty` 内核，数据格式为前 4 个字节放指纹模板的个数，后面依次每 4 个字节放一个指纹模板 id。
 3. 在开机后指纹 TA 检测到指纹模组后，将指纹厂商编号（2 个字节）写到 `trusty` 内核，具体编号值请参看下表，如若不在表格中请联系我们。

```
uint16_t fp_vendor_id = 0x0C; //集创北方（根据下面编号表格设置）  
  
ioctl(IO_DEVICE_FP, IFAA_SET_FP_VENDOR_ID, &fp_vendor_id);
```

指纹厂商编号：

指纹厂商	编号
FPC	0x01
汇顶	0x02
新思	0x03
神盾	0x04
義隆電	0x05
高通	0x06
联芯	0x07
迈瑞微	0x08
IDEX	0x09
信伟	0x0A
联咏科技	0x0B
集创北方	0x0C

思立微	0x0D
费恩格尔	0x0E
芯启航	0x0F

11.2 SOTER

SOTER 需要获取指纹厂商 name、version、fid ；

可参考如下数据结构：

```
#define SENSOR_VERSION_MAX_LENGTH 64
```

```
#define SENSOR_NAME_MAX_LENGTH 64
```

```
+typedef struct {
```

```
+  uint32_t fid;
```

```
+  unsigned char sensor_version[SENSOR_VERSION_MAX_LENGTH + 1];
```

```
+  unsigned char sensor_name[SENSOR_NAME_MAX_LENGTH + 1];
```

```
+} soter_fp_info_t;
```

其中 sensor_version 和 sensor_name 是用来表示指纹 sensor 的，数据由指纹厂商自己确定；

fid 为指纹鉴权成功后的指纹模板 id ；

请在指纹鉴权成功后调用 ioctl (cmd id 为 SOTER_SET_FP_INFO) 将上述数据写入 TEE 内核：

```
+ SOTER_SET_FP_INFO = 5,
```

注意事项：

新增指纹要更新 authtoken 中 authenticator_id (和之前一个不同的值), 其他场景保持不变。

12. 参考代码

12.1 工厂测试 fingersor.c

```
#include <stdio.h>

#include <unistd.h>

#include <dlfcn.h>

#define LIB_CACULATE_PATH "/system/lib/libfactorylib.so"

typedef int (*CAC_FUNC)(void);

enum {

    PASS = 1,

    OPEN_FAIL = -1,

    NO_SYMBOL = -2,

    TEST_FAIL = -3,

};

int main(void)

{
```

```
int ret = -1;

int test_result = -1;

int times = 50;


void *handle;

CAC_FUNC  factory_init = NULL;

CAC_FUNC  spi_test = NULL;

CAC_FUNC  interrupt_test = NULL;

CAC_FUNC  deadpixel_test = NULL;

CAC_FUNC  finger_detect = NULL;

CAC_FUNC  factory_exit = NULL;


fflush(stdout);

setvbuf(stdout, NULL, _IONBF, 0);


printf("dlopen fingerprint lib %s\n", LIB_CACULATE_PATH);

handle = dlopen(LIB_CACULATE_PATH, RTLD_LAZY);

if (!handle)

{

    printf("fingersor lib dlopen failed! %s, %d IN\n", dlerror(), __LINE__);

    test_result = OPEN_FAIL;

    return test_result;

}
```

Unisoc Confidential For hiar

```
}

printf("do factory_init\n");

*(void **) (&factory_init) = dlsym(handle, "factory_init");

if(!factory_init)

{

    printf("could not find symbol 'factory_init', %d IN\n", __LINE__);

    test_result = NO_SYMBOL;

    return test_result;

}

else

{

    ret = (*factory_init)();

    if(ret != 0)

    {

        printf("factory_init fail, ret = %d\n", ret);

        test_result = TEST_FAIL;

        return test_result;

    }

}

printf("do spi_test\n");
```

```
spi_test = dlsym(handle, "spi_test");

if(!spi_test)

{

    printf("could not find symbol 'spi_test', %d IN\n", __LINE__);

    test_result = NO_SYMBOL;

    return test_result;

}

else

{

    ret = (*spi_test)();

    if(ret != 0)

    {

        printf("spi_test fail, ret = %d\n", ret);

        test_result = TEST_FAIL;

        return test_result;

    }

}
```

```
printf("do interrupt_test\n");

interrupt_test = dlsym(handle, "interrupt_test");

if(!interrupt_test)

{
```

```
    printf("could not find symbol 'interrupt_test', %d IN\n", __LINE__);

    test_result = NO_SYMBOL;

    return test_result;
}

else

{

    ret = (*interrupt_test)();

    if(ret != 0)

    {

        printf("interrupt_test fail, ret = %d\n", ret);

        test_result = TEST_FAIL;

        return test_result;
    }
}
```

```
printf("do deadpixel_test\n");

deadpixel_test = dlsym(handle, "deadpixel_test");

if(!deadpixel_test)

{

    printf("could not find symbol 'deadpixel_test', %d IN\n", __LINE__);

    test_result = NO_SYMBOL;

    return test_result;
}
```

```
}

else

{

    ret = (*deadpixel_test)();

    if(ret != 0)

    {

        printf("deadpixel_test fail, ret = %d\n", ret);

        test_result = TEST_FAIL;

        return test_result;

    }

}

printf("do finger_detect\n");

finger_detect = dlsym(handle, "finger_detect");

if(!finger_detect)

{

    printf("could not find symbol 'finger_detect', %d IN\n", __LINE__);

    test_result = NO_SYMBOL;

    return test_result;

}

else

{
```

```
while(times > 0)

{

    ret = (*finger_detect)();

    if(ret == 0)

    {

        test_result = PASS;

        break;

    }

    times--;

    printf("not detect fingerprint, ret = %d, try again...\n", ret);

    usleep(1000*100);

}

if(ret != 0)

{

    printf("finger_detect several times but failed\n");

    test_result = TEST_FAIL;

    return test_result;

}

}

printf("do factory_exit\n");
```

```
factory_exit = dlsym(handle, "factory_exit");

if(!factory_exit)

{

    printf("could not find symbol 'factory_exit', %d IN\n", __LINE__);

    test_result = NO_SYMBOL;

    return test_result;

}

else

{

    ret = (*factory_exit)();

    if(ret != 0)

    {

        printf("factory_exit fail, ret = %d\n", ret);

        test_result = TEST_FAIL;

        return test_result;

    }

}

return test_result;

}
```