Rockchip Linux Secure Boot 开发指南

文件标识：RK-KF-YF-379

发布版本：V4.0.0

日期：2024-1-11

文件密级：□绝密 □秘密 □内部资料 ■公开

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

地址： 福建省福州市铜盘路软件园A区18号

网址： www.rock-chips.com

客户服务电话： +86-4007-700-590

客户服务传真： +86-591-83951833

客户服务邮箱： fae@rock-chips.com

前言

概述

本文档主要介绍 RK Linux 平台下，Secure Boot 的使用步骤和注意事项，方便客户在此基础上进行二次开发。安全启动功能旨在保护设备使用正确有效的固件，非签名固件或无效固件将无法启动。

产品版本

芯片名称	Kernel 检验方式	安全分区
RK3308/RK3328/RK3326/PX30/RK3358	AVB	OTP
RK3399	AVB	EFUSE
RK3588/RK356X	FIT	OTP

读者对象

本文档（本指南）主要适用于以下工程师：

技术支持工程师

软件开发工程师

修订记录

版本号	作者	修改日期	修改说明
V1.0.0	WZZ	2018-10-31	初始版本
V1.0.1	WZZ	2018-12-17	修改笔误 vbmeta->security
V2.0.0	WZZ	2019-06-03	Sign_Tool 兼容 AVB boot.img， 修改 device-mapper 相关使用说明
V2.0.1	Ruby Zhang	2020-08-10	调整格式，更新公司名称
V3.0.0	WZZ	2022-04-30	添加RK3588 FIT支持 修改AVB Key存储描述 修改参考文献目录 更新工具链接 升级AVB描述
V3.0.1	WZZ	2022-09-27	添加RK356X FIT支持声明
V4.0.0	WZZ	2024-01-11	适配Linux SDK最新make规则 将AVB和FIT的编译方法和SDK make规则耦合 DM-V/E改为system-verity/encryption，并修改编译方法 删除windows UI签名工具支持，统一改为Linux工具签名 提供KeyBox代码

目录

7. 参考资料

1. Linux Secure Boot 简介

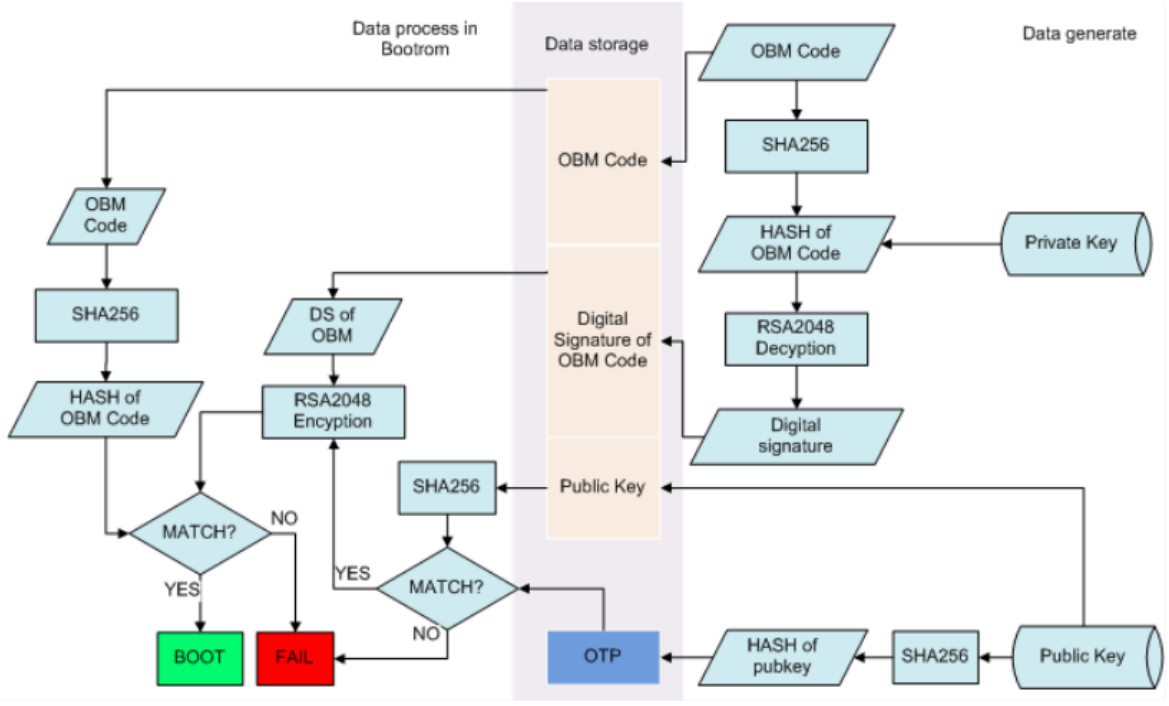
Secure Boot 是为了确保设备上运行固件完整且合法的一套校验流程，主要目的是防止非法第三方用户恶意注入软件的一种保护措施。使用Secure Boot后，保护分区被篡改后，设备都会报错，必要时会停止运行。

Secure Boot 主要保护系统前启动分区的安全。System和用户分区，该文档提供了分区加密和分区校验方案。

	System-Verity	System-Encryption
落盘信息	明文落盘	密文落盘
系统密钥	明文公钥，存储在Boot.img固件中	加密密钥，加密存储在RPMB分区
防抄袭	否	是
防篡改	是	是

阅读本文，先按产品版本了解芯片平台的分类，不同平台的芯片资源不一样，采用的安全策略也不一样，按分类阅读相关章节。本文基于通用SDK基础上，帮助客户从零搭建固件安全方案。在此基础上，也提供一些脚本，帮助客户管理、简化搭建固件安全的流程。

1.1 Linux Secure Boot 流程



以上流程图，是Bootrom校验loader的过程，其他固件校验，也是类似的流程。

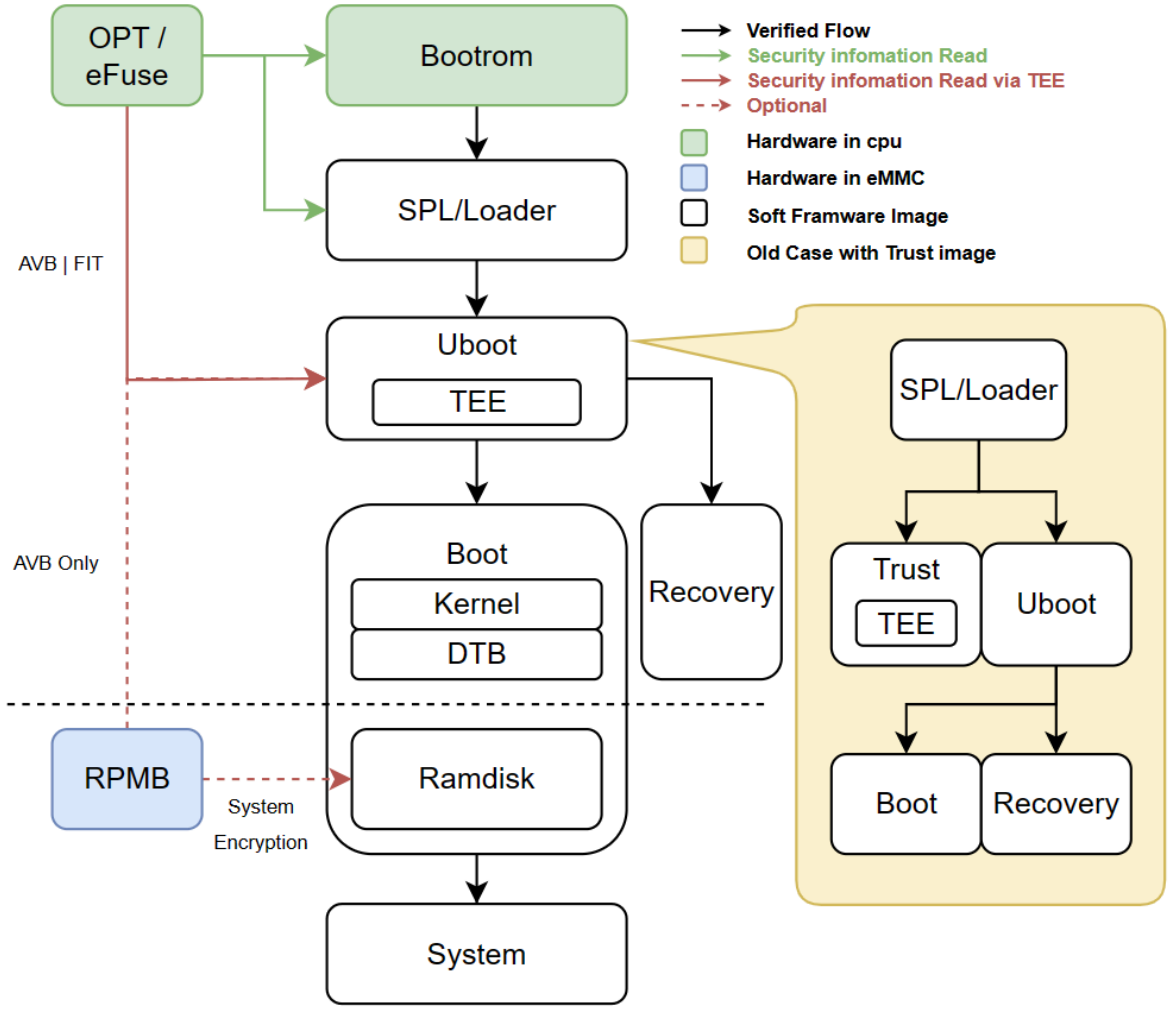
简略说，一个签名固件包括 Firmware(OBM Code) + Digital Signature + Public key

其中 Digital Signature + Public Key 都由签名工具添加。

存储上，签名固件放在 eMMC 或 Flash 上，Public Key Hash 放在芯片的 OTP(eFuse)上。

启动的时候，通过 OTP 中的 Hash 校验固件中的公钥，再用公钥校验数字签名的方式，达到芯片与签名代码绑定的效果。

详见 Rockchip-Secure-Boot-Application-Note-V1.9.pdf



Note: Trust固件与Uboot固件平级，都是由SPL或Loader校验，校验流程与Uboot一致。新芯片平台上，已经将Trust固件打包进Uboot固件中。

基于以上原理，各个固件使用同一套密钥签名，就只需要向OTP/eFuse中写一个Public Key Hash，然后各个固件逐级使用Public Key Hash校验后级固件，形成一个可信链路，确保链路中的固件是完整且合法的。

Secure Boot的主要代码由U-boot提供，保证到U-boot后面一级固件的安全性。**System的校验或者加密，由Ramdisk负责搭建(详见 "系统安全"章节).**

1.2 Secure Boot 安全存储

为了加强可信链路安全性，可信链路的可信根需要存在硬件模块中，并和硬件形成强绑定状态。

除了可信根存储外，Linux 平台还提供了以下几个安全存储区域，供客户存储自定义信息：

存储区域	说明
OTP / eFuse	位于 SOC 上，都是熔断机制的不可逆烧写。 OTP 可由 Miniloader / SPL 烧写，eFuse 只能通过 PC 工具烧写 不同的 SOC 采用不同的介质，目前 Linux 平台主要有： eFuse： RK3399 / RK3288 OTP： RK3308 / RK3326 / PX30 / RK3328 每个芯片内部的OTP/eFuse空间大小不一，但都不会太大，一般都只够放一两个Keys。
RPMB	位于 eMMC 上的一块物理分区，文件系统上不可见， 需要 SOC 鉴权访问（即只能由 TEE 访问）， 分区内容加密存放，无法挂载， 一般认为是安全区域。
Security Partition	位于存储介质上的逻辑分区，是为弥补 Flash 介质上无 RPMB 而加入的临时分区。分区内容加密存放， 无法挂载，但可能被强制擦除。同样只能由 TEE 访问 （强制擦除，TEE 访问报错，Secure Boot 无法正常启动）。

Note: 由于 OTP（eFuse）主要由 Rockchip 内部使用，客户安全信息请优先考虑 RPMB/Security 等其他区域。如有硬性需求，请向业务申请对应资料。

2. Secure Boot

Linux 系统中，提供了新旧2种Secure Boot方案，FIT和AVB方案，效果是一样的，两种方案不可混用，各个芯片具体选用那种方案，请参考产品版本。

2.1 FIT

FIT（flattened image tree）是U-Boot支持的一种新固件类型的引导方案，支持任意多个image打包和校验。FIT 使用 its (image source file) 文件描述image信息，最后通过mkimage工具生成 itb (flattened image tree blob) 镜像。its文件使用 DTS 的语法规则，非常灵活，可以直接使用libfdt 库和相关工具。同时自带一套全新的安全检验方式。

更多FIT细节信息，参考Rockchip_Developer_Guide_UBoot_Nextdev文档第十二章。

2.1.1 Keys 生成

U-Boot工程下执行如下三条命令可以生成签名用的RSA密钥对。通常情况下只需要生成一次，此后都用这对密钥签名和验证固件，请妥善保管。

```
# 1. 放key的目录：keys
mkdir -p keys

# 2. 使用RK的"rk_sign_tool"工具生成RSA2048的私钥privateKey.pem和publicKey.pem，分别更名存放为：keys/dev.key和keys/dev.pubkey。命令为：
../rkbin/tools/rk_sign_tool kk --bits 2048 --out .
ln -s privateKey.pem keys/dev.key
ln -s publicKey.pem keys/dev.pubkey

# 3. 使用-x509和私钥生成一个自签名证书：keys/dev.crt （效果本质等同于公钥）
openssl req -batch -new -x509 -key keys/dev.key -out keys/dev.crt
```

> 如果报错用户目录下没有.rnd文件：
>
> Can't load /home4/cjh//.rnd into RNG
> 140522933268928:error:2406F079:random number generator:RAND_load_file:Cannot open file:../crypto/rand/randfile.c:88:Filename=/home4/cjh//.rnd
>
> 请先手动创建：touch ~/.rnd

`ls keys/`查看结果：

```
dev.crt  dev.key  dev.pubkey
```

> 注意：上述的"keys"、"dev.key"、"dev.crt" 、"dev.pubkey"名字都不可变。因为这些名字已经在its文件中静态定义，如果改变则会打包失败。

2.1.2 配置

U-Boot的defconfig打开如下配置：

```
# 必选。
CONFIG_FIT_SIGNATURE=y
CONFIG_SPL_FIT_SIGNATURE=y

# 可选。
CONFIG_FIT_ROLLBACK_PROTECT=y       # boot.img防回滚
CONFIG_SPL_FIT_ROLLBACK_PROTECT=y   # uboot.img防回滚
```

> 建议通过make menuconfig的方式选中配置后，再通过make savedefconfig更新原本的defconfig文件。这样可以避免因为强加defconfig配置而导致依赖关系不对，进而导致编译失败的情况。

2.1.3 U-boot编译以及固件签名

U-boot编译的时候，同时对boot.img / recovery.img / loader.bin / uboot.img进行签名。因此在编译前，需要确定要签名的boot.img和recovery.img的位置。

（1）基础命令（不防回滚）：

```
./make.sh rv1126 --spl-new --boot_img boot.img --recovery_img recovery.img
```

编译结果：

```
# ......
# 编译完成后，生成已签名的uboot.img,boot.img和recovery.img。
start to sign rv1126_spl_loader_v1.00.100.bin
# ......
sign loader ok.
# ......
Image(signed, version=0):  uboot.img (FIT with uboot, trust...) is ready
Image(signed, version=0):  recovery.img (FIT with kernel, fdt, resource...) is
ready
Image(signed, version=0):  boot.img (FIT with kernel, fdt, resource...) is ready
Image(signed):  rv1126_spl_loader_v1.05.106.bin (with spl, ddr, usbplug) is ready
pack uboot.img okay! Input: /home4/cjh/rkbin/RKTRUST/RV1126TOS.ini

Platform RV1126 is build OK, with new .config(make rv1126-secure_defconfig)
```

（2）扩展命令1：

如果开启防回滚，必须对上述（1）追加rollback参数。例如：

```
// 指定 uboot.img,boot.img和recovery.img的最小版本号分别为10、12、12.
./make.sh rv1126 --spl-new --boot_img boot.img --recovery_img recovery.img --
rollback-index-uboot 10 --rollback-index-boot 12 --rollback-index-recovery 12
```

编译结果：

```
......

// 编译完成后，生成已签名的uboot.img,boot.img和recovery.img，且包含防回滚版本号。
start to sign rv1126_spl_loader_v1.00.100.bin
......
sign loader ok.
......
Image(signed, version=0, rollback-index=10):  uboot.img (FIT with uboot, trust)
is ready
Image(signed, version=0, rollback-index=12):  recovery.img (FIT with kernel, fdt,
resource...) is ready
Image(signed, version=0, rollback-index=12):  boot.img (FIT with kernel, fdt,
resource...) is ready
Image(signed):  rv1126_spl_loader_v1.00.100.bin (with spl, ddr, usbplug) is ready
```

（3）扩展命令2：

如果要把公钥hash烧写到OTP/eFUSE，必须对上述（1）或（2）追加参数`--burn-key-hash`。例如：

```
# 指定uboot.img,boot.img和recovery.img的最小版本号分别为10、12、12.
# 要求SPL阶段把公钥hash烧写到OTP/eFUSE中。
./make.sh rv1126 --spl-new --boot_img boot.img --recovery_img recovery.img --
rollback-index-uboot 10 --rollback-index-boot 12 --rollback-index-recovery 12 --
burn-key-hash
```

> 开启回滚后，更新的版本号不得低于前一版本（可以相等），否正系统会不让启动

编译结果：

```
# ......
# 使能 burn-key-hash
### spl/u-boot-spl.dtb: burn-key-hash=1

# 编译完成后，生成已签名的uboot.img,boot.img和recovery.img，且包含防回滚版本号。
start to sign rv1126_spl_loader_v1.00.100.bin
# ......
sign loader ok.
# ......
Image(signed, version=0, rollback-index=10):  uboot.img (FIT with uboot, trust)
is ready
Image(signed, version=0, rollback-index=12):  recovery.img (FIT with kernel, fdt,
resource...) is ready
Image(signed, version=0, rollback-index=12):  boot.img (FIT with kernel, fdt,
resource...) is ready
Image(signed):  rv1126_spl_loader_v1.00.100.bin (with spl, ddr, usbplug) is ready
```

> 上电开机时能看到SPL打印：RSA: Write key hash successfully。

注意：公钥hash是否烧写，只会影响Maskrom是否校验loader，且烧录后，无法撤销。loader验证uboot以及后续验证流程还是一致的。因此，调试阶段建议先不用burn-key-hash。

（4）扩展命令3：FIT 固件签名扩展命令3

为方便boot和recovery更新，U-boot也支持单独对boot或recovery固件进行签名。需要之前完整编译过一次签名的U-boot固件。

```
./make.sh rv1126 --spl-new
# 编译前，确定在开启签名后，已经至少编译过一次U-boot

./scripts/fit.sh --boot_img boot.img            # 单独签名 boot.img
./scripts/fit.sh --recovery_img recovery.img    # 单独签名 recovery.img
```

> 请确保U-boot包含`scripts: fit.sh: repack boot / recovery individually`补丁（Change-Id: Ib9c0396625971469e13e8092d233e2aea5714486）

（5）注意事项：

- `--boot_img`：可选。指定待签名的boot.img。
- `--recovery_img`：可选。指定待签名的recovery.img。
- `--rollback-index-uboot`、`--rollback-index-boot`、`--rollback-index-recovery`：可选。指定防回滚版本号。
- `--spl-new`：如果编译命令不带此参数，则默认使用rkbin中的spl文件打包生成loader；否则使用当前编译的spl文件打包loader。

因为u-boot-spl.dtb中需要包含RSA公钥（来自于用户），所以RK发布的SDK不会在rkbin仓库提交支持安全启动的spl文件。因此，用户编译时要指定`--spl-new`参数。但是用户也可以把自己的spl版本提交到rkbin工程，此后编译固件时就可以不再指定此参数，每次都使用这个稳定版的spl文件。
编译后会生成三个固件：loader、uboot.img、boot.img，只要RSA Key 没有更换，就允许单独更新其中的任意固件。

2.1.4 启动验证

固件配置正常，按"Key Hash"将Key Hash烧写入OTP，将会看到以下log。

```
......
Trying to boot from MMC1
// SPL完成签名校验
sha256,rsa2048:dev+
// 防回滚检测：当前uboot.img固件版本号是10，本机的最小版本号是9
rollback index: 10 >= 9, OK
// SPL完成各子镜像的hash校验
### Checking optee ... sha256+ OK
### Checking uboot ... sha256+ OK
### Checking fdt ... sha256+ OK

Jumping to U-Boot via OP-TEE
I/TC:
E/TC:0 0 plat_rockchip_pmu_init:2003 0
E/TC:0 0 plat_rockchip_pmu_init:2006 cpu off
E/TC:0 0 plat_rockchip_pmusram_prepare:1945 pmu sram prepare 0x14b10000 0x8400880
0x1c
E/TC:0 0 plat_rockchip_pmu_init:2020 pmu sram prepare
E/TC:0 0 plat_rockchip_pmu_init:2056 remap
I/TC: OP-TEE version: 3.6.0-233-g35ecf936 #1 Tue Mar 31 08:46:13 UTC 2020 arm
I/TC: Next entry point address: 0x00400000
I/TC: Initialized
......
### Loading kernel from FIT Image at 3d8122c0 ...
   Using 'conf' configuration
   // uboot完成签名校验
   Verifying Hash Integrity ... sha256,rsa2048:dev+ OK
   // 防回滚检测：当前boot.img固件版本号是22，本机的最小版本号是21
   Verifying Rollback-index ... 22 >= 21, OK
   Trying 'kernel' kernel subimage
......
   Verifying Hash Integrity ... sha256+ OK  // 完成kernel的hash校验
### Loading ramdisk from FIT Image at 3d8122c0 ...
   Using 'conf' configuration
   Trying 'ramdisk' ramdisk subimage
......
   Verifying Hash Integrity ... sha256+ OK  // 完成ramdisk的hash校验
   Loading ramdisk from 0x3dd3d4c0 to 0x0a200000
### Loading fdt from FIT Image at 3d8122c0 ...
   Using 'conf' configuration
   Trying 'fdt' fdt subimage
......
   Verifying Hash Integrity ... sha256+ OK // 完成fdt的hash校验
   Loading fdt from 0x3d812ec0 to 0x08300000
   Booting using the fdt blob at 0x8300000
   Loading Kernel Image from 0x3d8234c0 to 0x02008000 ... OK
   Using Device Tree in place at 08300000, end 0831359d
Adding bank: 0x00000000 - 0x08400000 (size: 0x08400000)
Adding bank: 0x0848a000 - 0x40000000 (size: 0x37b76000)
Total: 236.327 ms
```

```
Starting kernel ...
```

替换任意非签名固件，系统将无法正常启动。

2.2 AVB

AVB（Android Verified Boot）主要运用于Android系统，围绕U-Boot固件，建立了一个完整的可信链，从硬件保护的可信根开始，一直延伸到引导加载程序，以及 U-Boot 前后的分区（比如boot 或者recovery分区）。

Linux系统上，由于缺少Android的系统组件，因此，对AVB进行裁剪，仅使用AVB完成boot和recovery分区的校验。Loader 到 Uboot的校验由RK私有方案完成。

2.2.1 Keys 生成

Loader 到 Uboot的校验，由RK私有方案完成，需要生成一组RSA密钥对，同FIT方案一致，参考FIT Keys 生成.

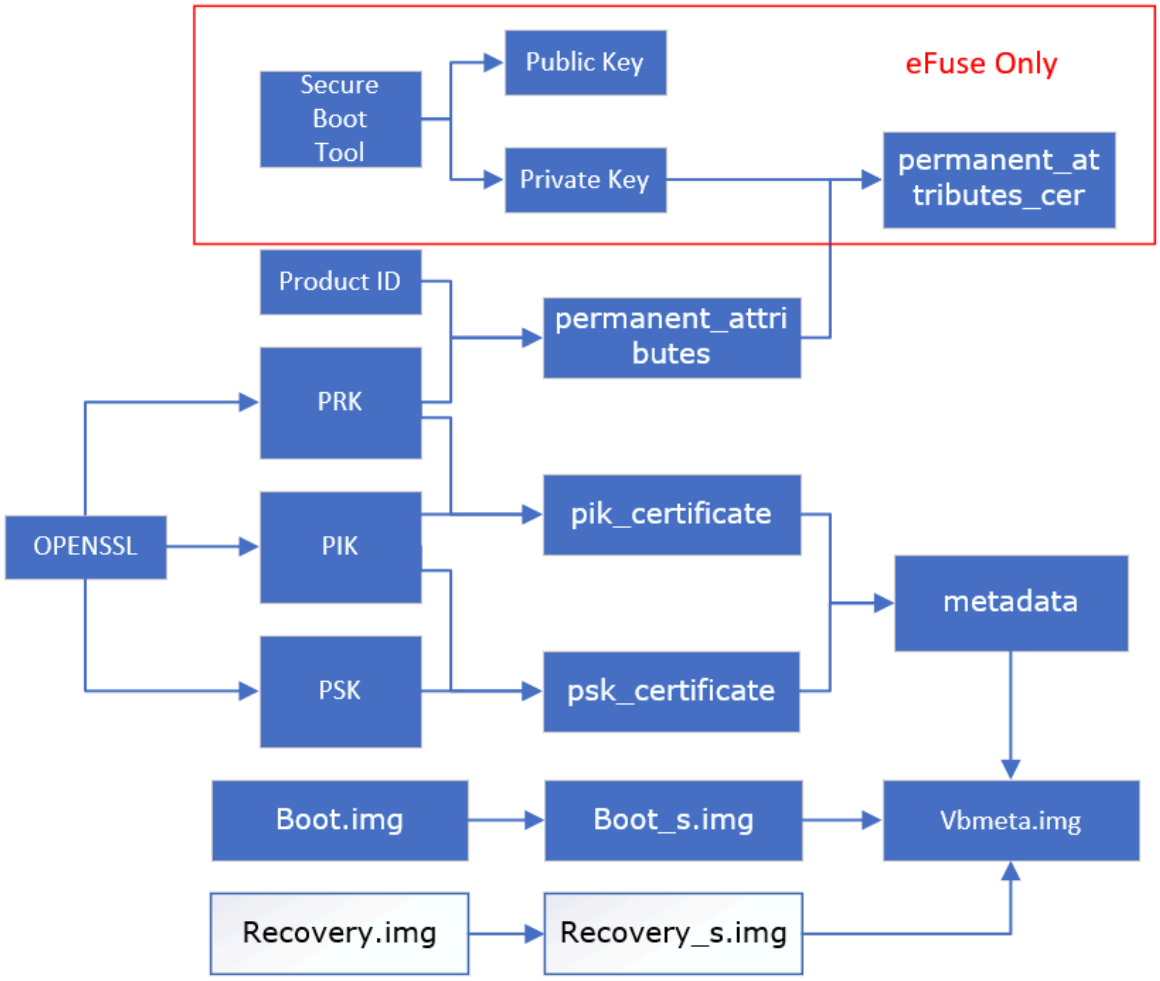
除此外，AVB 还包含以下 4 把 Key：

Product RootKey (PRK)：AVB 的 Root Key，eFuse 设备中，由 Base Secure Boot 的 Key 校验相关信息。OTP 设备中，直接读取预先存放于 OTP 中的 PRK-Hash 信息校验；

ProductIntermediate Key (PIK)：中间 Key，中介作用；

ProductSigning Key (PSK)：用于签名固件的 Key；

ProductUnlock Key (PUK)：用于解锁设备。



AVB 根据这 4 把 Key 派生出一系列的文件，如图所示，具体信息参 Google AVB 开源https://android.googlesource.com/platform/external/avb/+/master/README.md

和原版 AVB 相比，Linux 下主要截取了 AVB 最主要的固件校验功能，其中为适配 Rockchip 平台，在 eFuse设备 上，额外产生了 permanent_attributes_cer.bin，这使得我们可以不必将 permanent_attributes.bin 存入 eFuse 内，直接由 loader 的公钥校验 permanent_attributes.bin 信息，达到节省 eFuse 空间的作用。

`tools/linux/Linux_SecurityAVB/avb_keys` 下已经有一套测试的证书和 Key，调试阶段可以先使用这一套密钥，方便调试和管理。正式产品中，请替换成自己的私有密钥。可以按下面步骤自行生成。

```
cd tools/linux/Linux_SecurityAVB/
touch avb_keys/temp.bin

# Input 16 bytes product id, like "0123456789ABCDE"
# Or use random id,
# head -c 16 /dev/urandom | od -An -t x | tr -d ' \n' > avb_keys/product_id.bin
echo -n $PRODUCT_ID > avb_keys/product_id.bin

# Generate test keys.
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -outform PEM -out
avb_keys/testkey_prk.pem
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -outform PEM -out
avb_keys/testkey_psk.pem
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -outform PEM -out
avb_keys/testkey_pik.pem
```

```
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -outform PEM -out
avb_keys/testkey_puk.pem

# Generate certificate.bin and metadata.
python scripts/avbtool make_atx_certificate --output=avb_keys/pik_certificate.bin
--subject=avb_keys/temp.bin --subject_key=avb_keys/testkey_pik.pem --
subject_is_intermediate_authority --subject_key_version 42 --
authority_key=avb_keys/testkey_prk.pem
python scripts/avbtool make_atx_certificate --output=avb_keys/psk_certificate.bin
--subject=avb_keys/product_id.bin --subject_key=avb_keys/testkey_psk.pem --
subject_key_version 42 --authority_key=avb_keys/testkey_pik.pem
python scripts/avbtool make_atx_certificate --output=avb_keys/puk_certificate.bin
--subject=avb_keys/product_id.bin --subject_key=avb_keys/testkey_puk.pem --
usage=com.google.android.things.vboot.unlock --subject_key_version 42 --
authority_key=avb_keys/testkey_pik.pem
python scripts/avbtool make_atx_metadata --output=avb_keys/metadata.bin --
intermediate_key_certificate=avb_keys/pik_certificate.bin --
product_key_certificate=avb_keys/psk_certificate.bin

# Generate permanent_attributes.bin
python scripts/avbtool make_atx_permanent_attributes --
output=avb_keys/permanent_attributes.bin --product_id=avb_keys/product_id.bin --
root_authority_key=avb_keys/testkey_prk.pem

# If eFuse device used, generate permanent_attributes.bin.
openssl dgst -sha256 -out avb_keys/permanent_attributes_cer.bin -sign dev.key
avb_keys/permanent_attributes.bin
```

所以，AVB 方案Keys包括以下两个部分，分布于两个目录：

```
PC:~/SDK$ tree u-boot/keys/
.
├── dev.key
└── dev.pubkey

PC:~/SDK$ tree tools/linux/Linux_SecurityAVB/avb_keys
.
├── metadata.bin
├── permanent_attributes.bin
├── pik_certificate.bin
├── product_id.bin
├── psk_certificate.bin
├── puk_certificate.bin
├── testkey_pik.pem
├── testkey_prk.pem
├── testkey_psk.pem
└── testkey_puk.pem
# 其中，testkey_*.pem 是AVB的密钥，其余bin文件都是AVB密钥根据具体用途派生出的密钥证书。
```

2.2.2 配置

2.2.2.1 Trust

进入 rkbin/RKTRUST，以 RK3308 为例，找到 RK3308TRUST.ini，修改

```
[BL32_OPTION]
SEC=0
```

改为：

```
[BL32_OPTION]
SEC=1
```

TOS 格式的trust，默认已经开启了`BL32_OPTION`，比如`RK3288TOS.ini`

2.2.2.2 U-boot

U-boot 需要以下特性支持：

```
# OPTEE support
CONFIG_OPTEE_CLIENT=y
CONFIG_OPTEE_V1=y            #RK312x/RK322x/RK3288/RK3228H/RK3368/RK3399 与V2互斥
CONFIG_OPTEE_V2=y            #RK3308/RK3326 与V1互斥

# CRYPTO support
CONFIG_DM_CRYPTO=y           # eFuse 设备必须项
CONFIG_ROCKCHIP_CRYPTO_V1=y # eFuse 设备，比如 RK3399/RK3288， 按芯片选择
CONFIG_ROCKCHIP_CRYPTO_V2=y # eFuse 设备，比如 RK1808，按芯片选择

# AVB support
CONFIG_AVB_LIBAVB=y
CONFIG_AVB_LIBAVB_AB=y
CONFIG_AVB_LIBAVB_ATX=y
CONFIG_AVB_LIBAVB_USER=y
CONFIG_RK_AVB_LIBAVB_USER=y
CONFIG_AVB_VBMETA_PUBLIC_KEY_VALIDATE=y
CONFIG_ANDROID_AVB=y
CONFIG_OPTEE_ALWAYS_USE_SECURITY_PARTITION=y # 非eMMC设备打开，必须存在security分区
CONFIG_ROCKCHIP_PRELOADER_PUB_KEY=y          # 只适用于eFuse设备
CONFIG_RK_AVB_LIBAVB_ENABLE_ATH_UNLOCK=y

#fastboot support
CONFIG_FASTBOOT=y
CONFIG_FASTBOOT_BUF_ADDR=0x2000000           # 必要时，可修改
CONFIG_FASTBOOT_BUF_SIZE=0x08000000          # 必要时，可修改
CONFIG_FASTBOOT_FLASH=y
CONFIG_FASTBOOT_FLASH_MMC_DEV=0
```

> 部分平台默认U-boot配置不开fastboot。正常开机的时候，按住`Ctrl-M`可以看到当前的内存分布，手动配置`CONFIG_FASTBOOT_BUF_*`，起始地址和大小需要避免和当前内存配置冲突。

使用`./make.sh xxxx`，生成 uboot.img, trust.img 和 loader.bin。

2.2.2.3 Parameter

vbmeta是分区表中必有的项目，security分区需要按硬件情况可选添加：

- vbmeta分区是存放目标分区的签名信息的文件，大小1M
- security分区，一般是作为RPMB的替代品，如果在非eMMC设备上，没有RPMB，就需要加入这个分区，大小4M，如果开启`CONFIG_OPTEE_ALWAYS_USE_SECURITY_PARTITION`，则强制使用security分区。

因此，一般分区表可以这样分配：

```
# AVB parameter:
0x00002000@0x00004000(uboot),0x00002000@0x00006000(trust),0x00002000@0x00008000(
misc),0x00010000@0x0000a000(boot),0x00010000@0x0001a000(recovery),0x00010000@0x0
002a000(backup),0x00020000@0x0003a000(oem),0x00300000@0x0005a000(system),0x00000
800@0x0035a000(vbmeta),0x00002000@0x0035a800(security),-
@0x0035c800(userdata:grow)
uuid:system=614e0000-0000-4b53-8000-1d28000054a9
```

下载的时候，下载工具上的分区名称要同步修改，修改后，重载 parameter.txt。

2.2.3 固件签名

Loader、Uboot、Trust签名使用`rk_sign_tool`：

```
export RK_SIGN_TOOL=${SDK_DIR}/rkbin/tools/rk_sign_tool
export KEYS=${SDK_DIR}/u-boot/keys    # Keys 生成章节，默认key存放再u-boot/keys下
${RK_SIGN_TOOL} cc --chip $CHIP
${RK_SIGN_TOOL} lk --key $KEYS/dev.key --pubkey $KEYS/dev.pubkey

${RK_SIGN_TOOL} sl --loader < path to loader.bin >
${RK_SIGN_TOOL} si --img < path to trust.img or uboot.img >
```

> 默认签名不烧写OTP Public Key Hash，方便调试
>
> 需要烧写OTP Public Key Hash，需要在`${SDK_DIR}/rkbin/tools/setting.ini`中，加入`sign_flag=0x20`

Boot 和 Recovery 签名使用 `avbtool`：

```
# Sign boot.img
IMAG=boot.img # path to boot.img
SIZE=`ls ${IMAGE} -l | awk '{printf $5}'`
SIZE=$[(SIZE / 4096 + 18) * 4096] # Align 4K
python scripts/avbtool add_hash_footer --image ${IMAGE} --partition_size ${SIZE}
--partition_name boot --key avb_keys/testkey_psk.pem --algorithm SHA512_RSA4096
# 生成带签名信息boot.img
```

```
python scripts/avbtool make_vbmeta_image --public_key_metadata
avb_keys/metadata.bin --include_descriptors_from_image ${IMAGE} --algorithm
SHA256_RSA4096 --rollback_index 0 --key avb_keys/testkey_psk.pem  --output
out/vbmeta.img
# 生成和boot.img配套的out/vbmeta.img

# Sign recovery.img
IMAG=recovery.img # path to recovery.img
SIZE=$(get_from_parameter) # 从parameter获取recovery分区的size。分区大小必须4K对齐。
python scripts/avbtool add_hash_footer --image ${IMAGE} --partition_size ${SIZE}
--partition_name boot --key avb_keys/testkey_psk.pem --algorithm SHA512_RSA4096 -
-public_key_metadata avb_keys/metadata.bin
# 生成带签名信息recovery.img,recovery 不需要和vbmeta绑定，自带metadata信息。
```

2.2.4 AVB Keys 烧写流程

AVB 固件允许设备处于Lock和Unlock两种状态：

- Lock: 设备进行严格的固件校验，一旦固件校验失败，则停止启动，适用于量产。
- Unlock: 设备进行宽松的固件校验模式，发现固件校验失败，发出错误警告，但不停止启动，适用于调试。

AVB的两种锁定状态可以在`fastboot`模式中修改。因此，在AVB 固件烧写后，还需要设备重启，进入fastboot模式，搭配PC fastboot命令，将AVB Keys烧录到设备中，并锁定设备，才能起到防护效果。

依照U-boot版本不同，部分版本在AVB第一次启动时，直接进入fastboot，部分客户则会直接启动进系统，如果设备直接进系统，可以使用`reboot fastboot`命令让设备进入fastboot模式。

fastboot 模式下，使用以下命令烧录AVB Keys并锁定设备。

```
cd ${SDK_DIR}/tools/linux/Linux_SecurityAVB

# download permanent_attributes.bin
./scripts/fastboot stage avb_keys/permanent_attributes.bin
./scripts/fastboot oem fuse at-perm-attr

# If eFuse device:
./scripts/fastboot stage avb_keys/permanent_attributes_cer.bin
./scripts/fastboot oem fuse at-rsa-perm-attr

# Lock device
./scripts/fastboot oem at-lock-vboot
```

锁定成功，并重启后，可以看到以下log：

```
ANDROID: reboot reason: "(none)"
read_is_device_unlocked() ops returned that device is LOCKED
```

如果需要解锁设备，使用以下命令：

```
./scripts/fastboot oem at-get-vboot-unlock-challenge
./scripts/fastboot get_staged raw_unlock_challenge.bin
python ./scripts/avb-challenge-verify.py raw_unlock_challenge.bin
avb_keys/product_id.bin # Generate unlock_challenge.bin
python ./scripts/avbtool make_atx_unlock_credential --
output=unlock_credential.bin --
intermediate_key_certificate=avb_keys/pik_certificate.bin --
unlock_key_certificate=avb_keys/puk_certificate.bin --
challenge=unlock_challenge.bin --unlock_key=$KEYS/testkey_puk.pem
./scripts/fastboot stage unlock_credential.bin
./scripts/cmd_fastboot oem at-unlock-vboot
```

2.2.5 启动Log

完整烧录AVB固件，并锁定设备后，可以看到如下输出：

```
DDR Version V1.31 20210118
...
Boot1 Release Time: Mar 29 2021 14:15:15, version: 1.27
...
StorageInit ok = 10694
SecureMode = 1                          # OTP / eFuse 已经被烧写
Secure read PBA: 0x4
# Head flags=0x20b3                     # | 仅第一次烧录带Sign_flag = 0x20的签名
loader时
# SecureCheckHeader 0, 0, 0             # | 会打印 OTP 烧写Log，之后再开机，无此Log
# SecureWriteOTP                        # |
# enable secure flag!                   # |
# otp write key success! !!             # |
atags_set_pub_key: ret:(0)              # public key 校验有效
SecureInit ret = 0, SecureMode = 1      # 使用 SecureMode 启动
...
U-Boot 2017.09-gfae486e407-210107 #zain (Aug 18 2021 - 17:34:57 +0800)
...
Hit key to stop autoboot('CTRL+C'):  0
ANDROID: reboot reason: "(none)"
Vboot=1, AVB images, AVB erify
read_is_device_unlocked() ops returned that device is LOCKED # 设备处于Lock状态
Fdt Ramdisk skip relocation
Booting IMAGE kernel at 0x00680000 with fdt at 0x1f00000...  # kernel正常加载，没有
报错
...
```

2.2.6 快捷脚本

为简化上述AVB操作步骤，`tools/linux/Linux_SecurityAVB/`中提供了管理脚本，`avb_user_avb_user_tool.sh`提供了以下快捷功能：

```
# Generate AVB Keys
./avb_user_tool.sh -n <Product ID> #The size of Product ID is 16 bytes.
```

```
# Sign boot and recovery, final image in `out`.
./avb_user_tool.sh -s -b < /path/to/boot.img > -r < /path/to/recovery.img > #
Remove -r option if no recovery.img
# Loader / Uboot / Trust still use `rk_sign_tool`.

# Lock and Unlock device
# Save root password for fastboot operation.
./avb_user_tool.sh --su_pswd < /user/password >
# Mark eFuse tool, and generate permanent_attributes_cer.bin. OTP device skip
this step.
./avb_user_tool.sh -f < /path/to/privatekey.pem >
# download AVB Keys
./avb_user_tool.sh -d
# Lock device
./avb_user_tool.sh -l # reboot device after finishing lock.

# Unlock device
./avb_user_tool.sh -u # reboot device after finishing unlock.
```

2.2.7 AVB 功能简化修改

常规AVB 的使用，需要在正常的固件下载流程外，再进行AVB密钥下载，这会使烧写流程复杂化，增加生产成本。因为正常的AVB密钥下载用的是AVB的公钥，公钥只用作校验用，因此，只要保证公钥不被篡改，可以将公钥明文保留在固件中。基于这个原理，本章节提供一种将AVB公钥打包到U-boot代码中的方案，U-boot会正常被loader校验，确保公钥是合法有效的。对比正常AVB方案，可以省略掉AVB密钥`fastboot`下载步骤。

该方案和常规AVB流程不同，属于非标准修改，因此代码只能手动应用，无法合并到主线上。

```
diff --git a/common/android_bootloader.c b/common/android_bootloader.c
index 5525fe620e8..7d59257633f 100644
--- a/common/android_bootloader.c
+++ b/common/android_bootloader.c
@@ -801,6 +801,7 @@ static AvbSlotVerifyResult android_slot_verify(char
*boot_partname,
        if (ops->read_is_device_unlocked(ops, (bool *)&unlocked) !=
AVB_IO_RESULT_OK)
                printf("Error determining whether device is unlocked.\n");

+       unlocked = 0; /* Lock State Fixed */
        printf("read_is_device_unlocked() ops returned that device is %s\n",
               (unlocked & LOCK_MASK)? "UNLOCKED" : "LOCKED");

diff --git a/lib/avb/libavb_user/avb_ops_user.c
b/lib/avb/libavb_user/avb_ops_user.c
index 18c0c1fb237..bb96a4bade2 100644
--- a/lib/avb/libavb_user/avb_ops_user.c
+++ b/lib/avb/libavb_user/avb_ops_user.c
@@ -42,6 +42,7 @@
 #include <android_avb/avb_vbmeta_image.h>
 #include <android_avb/avb_atx_validate.h>
 #include <boot_rkimg.h>
+#include <android_avb/avb_root_pub.h>
```

```
 static void byte_to_block(int64_t *offset,
                           size_t *num_bytes,
@@ -192,21 +193,17 @@ validate_vbmeta_public_key(AvbOps *ops,
                            size_t public_key_metadata_length,
                            bool *out_is_trusted)
 {
-/* remain AVB_VBMETA_PUBLIC_KEY_VALIDATE to compatible legacy code */
-#if defined(CONFIG_AVB_VBMETA_PUBLIC_KEY_VALIDATE) || \
-    defined(AVB_VBMETA_PUBLIC_KEY_VALIDATE)
-       if (out_is_trusted) {
-               avb_atx_validate_vbmeta_public_key(ops,
-                                                  public_key_data,
-                                                  public_key_length,
-                                                  public_key_metadata,
-                                                  public_key_metadata_length,
-                                                  out_is_trusted);
+       if (!public_key_length || !out_is_trusted || !public_key_data) {
+               printf("%s: Invalid parameters\n", __func__);
+               return AVB_IO_RESULT_ERROR_IO;
        }
-#else
-       if (out_is_trusted)
+
+       *out_is_trusted = false;
+       if (public_key_length != avb_root_pub_bin_len)
+               return AVB_IO_RESULT_ERROR_IO;
+       if (memcmp(public_key_data, avb_root_pub_bin, public_key_length) == 0)
                *out_is_trusted = true;
-#endif
+
        return AVB_IO_RESULT_OK;
 }

@@ -558,21 +555,17 @@ AvbIOResult validate_public_key_for_partition(AvbOps *ops,
                                              bool *out_is_trusted,
                                              uint32_t
*out_rollback_index_location)
 {
-/* remain AVB_VBMETA_PUBLIC_KEY_VALIDATE to compatible legacy code */
-#if defined(CONFIG_AVB_VBMETA_PUBLIC_KEY_VALIDATE) || \
-    defined(AVB_VBMETA_PUBLIC_KEY_VALIDATE)
-       if (out_is_trusted) {
-               avb_atx_validate_vbmeta_public_key(ops,
-                                                  public_key_data,
-                                                  public_key_length,
-                                                  public_key_metadata,
-                                                  public_key_metadata_length,
-                                                  out_is_trusted);
+       if (!public_key_length || !out_is_trusted || !public_key_data) {
+               printf("%s: Invalid parameters\n", __func__);
+               return AVB_IO_RESULT_ERROR_IO;
        }
-#else
-       if (out_is_trusted)
+
```

```
+       *out_is_trusted = false;
+       if (public_key_length != avb_root_pub_bin_len)
+               return AVB_IO_RESULT_ERROR_IO;
+       if (memcmp(public_key_data, avb_root_pub_bin, public_key_length) == 0)
                *out_is_trusted = true;
-#endif
+
        *out_rollback_index_location = 0;
        return AVB_IO_RESULT_OK;
 }
```

生成公钥文件，并放入U-boot中，等待编译。

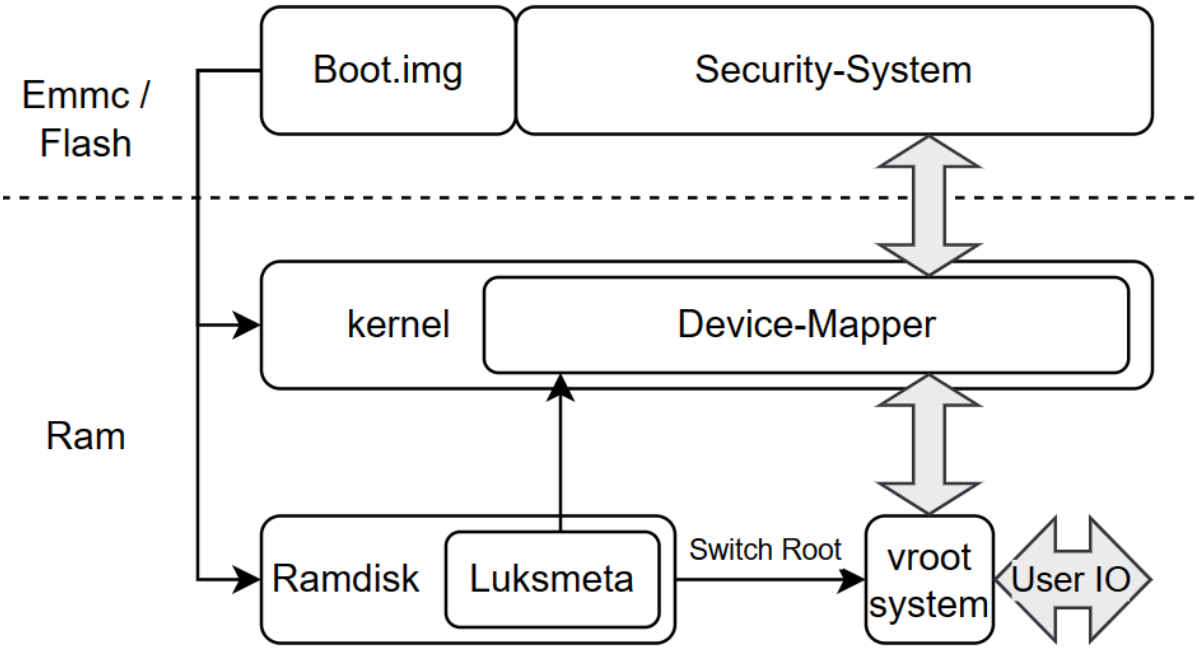
```
./scripts/avbtool extract_public_key --key avb_keys/testkey_psk.pem --output
avb_root_pub.bin

echo "#ifndef __AVB_ROOT_PUB_H_" > avb_root_pub.h
echo "#define __AVB_ROOT_PUB_H_" >> avb_root_pub.h
echo "" >> avb_root_pub.h
xxd -i avb_root_pub.bin >> avb_root_pub.h
echo "" >> avb_root_pub.h
echo "#endif /* __AVB_ROOT_PUB_H_ */" >> avb_root_pub.h

cp avb_root_pub.h $UBoot/include/android_avb/.
```

按AVB U-boot 配置,并编译签名U-boot。

3. 系统安全



上述框图大致描述了boot.img是如何校验或解密系统的：

1. Boot.img 必须处于在Secure Boot的保护下，确保这套机制的基础是安全的。

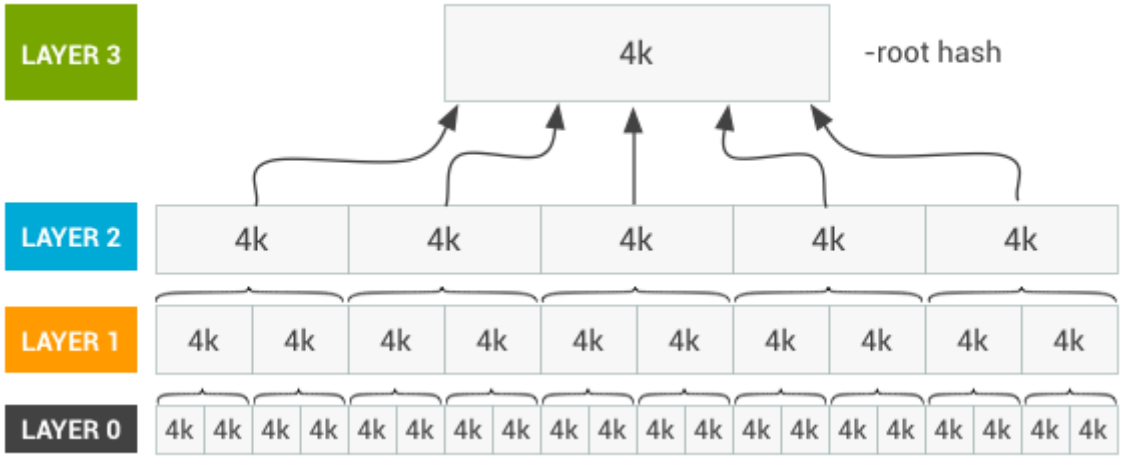
2. Ramdisk 使用Luksmeta工具集，调用Kernel的Device-Mapper特性，将Security-System虚拟化成vroot system节点，之后用`switch_root`命令切换到最终系统。
3. 最终系统或用户对根文件系统的读写，都通过Device-Mapper反馈到Security-System分区上。
4. 一旦发现系统分区内容被篡改，系统将返回IO错误，篡改部分将无法正常使用。

3.1 DM 介绍

系统分区的体积普遍要比前面几个分区大很多，如果要对整个分区计算哈希校验，开机时就会占用大量时间。因此，系统分区的校验，需要在Kernel中，开启Device-Mapper特性，可以节省大量开机耗时。

Device-Mapper技术和Crypt相结合，能完成对大体积固件的快速校验或加解密，以下以快速校验为例进行说明。

Device-Mapper-Verity 机制会对校验固件进行 4K 切分，并对每个 4K 数据片进行 hash 计算，迭代多层，并生成对应 Hash-Map和 Root-Hash。



Root-Hash先对Hash-Map 进行校验，保证 Hash-Map 无误，之后通过Device-Mapper-Verity创建虚拟分区并挂载。使用分区时，Kernel实时对访问I/O所在的 4K 分区单独进行 Hash 校验，并和Hash-Map比对。当校验出错时，返回 I/O 错误，与文件系统损坏现象一样。

同理，加解密即将Hash算法改成对称加密算法。

Device-Mapper 的优点在于，校验速度快，固件越大，效果越明显。缺点是存储读速度会稍受影响。

具体可参考 Kernel 底下 Documentation/device-mapper/ 或者 https://source.android.google.cn/security/verifiedboot/dm-verity。

3.1.1 配置

Device-Mapper 特性需要 Kernel 开启以下配置：

```
CONFIG_BLK_DEV_DM=y
CONFIG_DM_CRYPT=y
CONFIG_BLK_DEV_CRYPTOLOOP=y
CONFIG_DM_VERITY=y
```

> 此时，Kernel 编译生成的boot.img不带有主动校验或解密系统的能力。如系统安全的框图所示，需要将Ramdisk和Kernel打包在一起，才具有主动校验或解密的能力，如Ramdisk配置说明。

3.2 系统处理

按需求，二选一操作。

3.2.1 System-Verity

系统校验使用第三方工具`veritysetup`完成，`veritysetup`是 Linux 系统上用于设置和管理 dm-verity 的命令行工具。dm-verity (Device-Mapper-Verity) 是 Linux 内核的一个模块，基于Device-Mapper再封装，用于提供块级别的数据完整性保护，通常用于防止文件系统或系统文件被未经授权的修改。

`veritysetup`会根据目标系统镜像生成Hash-Map和Root-Hash，Hash-Map可以使用工具直接附在系统镜像的后面，Root-Hash打包进`boot.img`的Ramdisk中。启动时，Ramdisk负责使用Root-Hash将目标镜像通过dm-verity虚拟成`vroot`块设备，挂载后，使用`switch_root`命令切换至目标系统。

在PC上，对系统镜像做处理，在系统镜像后面附加Hash-Map并保存Root-Hash。

```
target_image=$(realpath $path_to_system_img)              # 输入系统镜像，校验信息直接
附在输入镜像上

sectors=$(ls -l "$target_image" | awk '{printf $5}')      # 获取镜像大小
hash_offset=$[(sectors / 1024 / 1024 + 2) * 1024 * 1024] # 系统以1M对齐，计算Hash-
Map放置的偏移，系统镜像后面的2M空间
result=$(veritysetup --hash-offset=$hash_offset format "$target_image"
"$target_image")
# 保存数据供后续操作使用
echo "root_hash=$(echo ${result##*:})" > security.info
echo "hash_offset=$hash_offset" >> security.info
```

> Root-Hash存放在`security.info`中，后续将在Ramdisk 初始脚本配置中使用。
>
> 注意：此方法打包出来的镜像，只比原来的镜像多了一段Hash-Map的数据，他仍然可以被当成普通系统，不经过`veritysetup`的处理，直接挂载使用。

3.2.2 System-Encryption

系统加密使用第三方工具`dmsetup`完成，`dmsetup` 是 Linux 系统上用于管理设备映射（Device Mapper）的命令行工具。它允许用户创建和管理虚拟块设备，这些设备可以进行各种操作，如合并、分割、重映射、加密等。

PC上，`dmsetup`会使用Device Mapper创建一个加密容器，将需要加密的系统放入容器中，完成加密。

```
target_image=$(realpath $path_to_system_img)     # 输入系统镜像
security_system=${path_to_output_system}         # 输出的加密系统
key=$(cat ${path_to_key_file})                   # 读取加密key,密钥长度要和下面算法吻合
cipher=${cipher_name}                            # 对称加密算法名，比如aes-cbc-plain

sectors=$(ls -l "$target_image" | awk '{printf $5}')
```

```
sectors=$[(sectors + (21 * 1024 * 1024) - 1) / 512] # remain 20M for partition
info / unit: 512 bytes

#创建加密容器
loopdevice=$(losetup -f)
mappername=encfs-$(shuf -i 1-10000000000000000000 -n 1)
dd if=/dev/null of="$security_system" seek=$sectors bs=512
sudo losetup $loopdevice "$security_system"
sudo dmsetup create $mappername --table "0 $sectors crypt $cipher $key 0
$loopdevice 0 1 allow_discards"

#将系统放入加密容器中
sudo dd if="$target_image" of=/dev/mapper/$mappername conv=fsync

sync && sudo dmsetup remove $mappername
sudo losetup -d $loopdevice

# 保存数据供后续操作使用
echo "cipher=$cipher" > security.info
echo "key=$key" >> security.info
```

打包出来的加密系统即为密文，无法直接解析使用。同系统校验一样，加密系统使用时，也是需要Ramdisk参与，提供加密的算法和密钥，使用`dmsetup`工具将系统分区虚拟成`/dev/mapper/vroot`节点，并挂载使用。

Ramdisk配置参考Ramdisk 初始脚本配置章节。和System-Verity对比，System-Encryption的加密密钥不能直接存储在Ramdisk固件中。

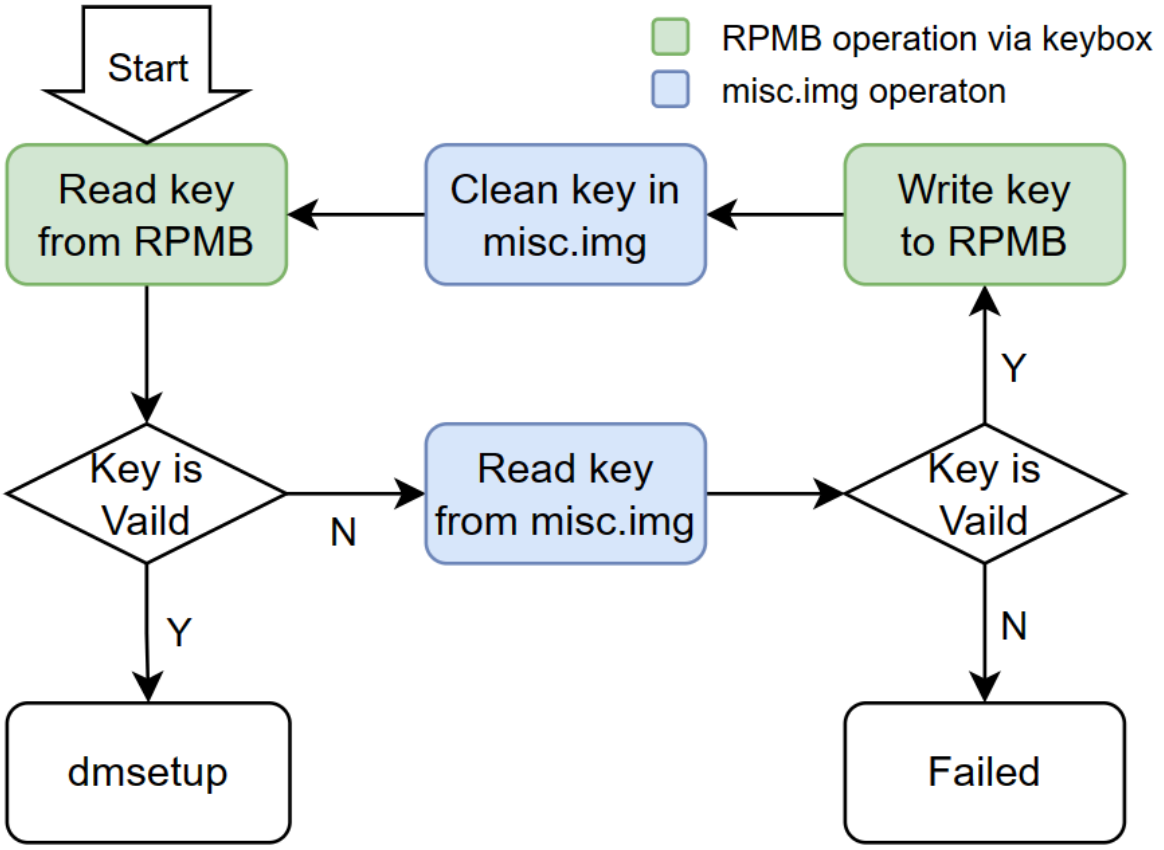
3.2.2.1 密钥存储

和System-Verity对比，System-Encryption的加密密钥不能直接存储在Ramdisk固件中。SDK默认将密钥存储在`misc.img`，通过Ramdisk读取misc.img的密钥，并转存到安全存储中（RPMB或Security分区），同时清空misc.img。

PC上，将密钥打包进`misc.img`使用以下命令：

```
# 密钥默认存放在misc镜像偏移10K的位置
MISC=misc.img                                                    # 原misc文件
key="af16857cf77982040e6dbeee739d00619847b0e16a25c8a9589b4ce93aced6b1"
dd if="$MISC" of=security-misc.img bs=1k count=10
echo -en "\x40\x00" >> security-misc.img                         # 声明密钥长度位64 byte
（0x40）
echo -n "$key" >> security-misc.img                              # 写入密钥
skip=$[10 * 1024 + 64 + 2]
dd if="$MISC" of=security-misc.img seek=$skip skip=$skip bs=1
# 生成代码密钥的security-misc.img
```

Ramdisk中，对密钥的处理流程如图：



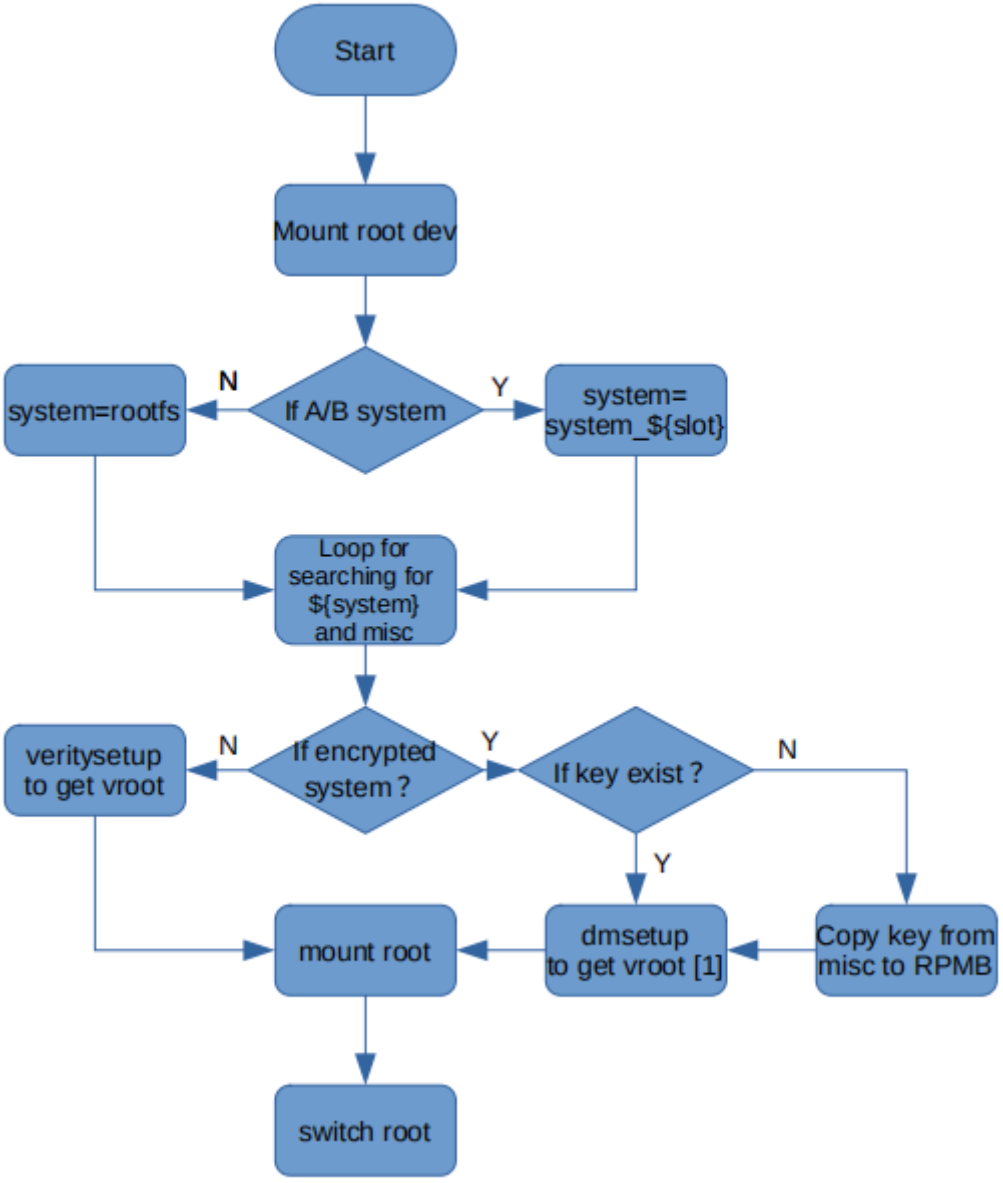
图中，从RPMB中读写密钥的操作都通过KeyBox软件完成。另外，为防止KeyBox从Ramdisk中被烤出来使用，还做了部分运行限制。比如必须是PID=1的程序中运行，必须是在Ramdisk中运行。KeyBox详情可以参考KeyBox章节。

3.3 Ramdisk 配置

Linux SDK 中，Ramdisk 是由Buildroot生成的一个小系统，和Kernel一同打包成boot.img。在系统安全方案中，主要负责系统的解密或校验，完成系统解密或校验后，使用`switch_root`切换至目标系统并释放自己。根据使用目的不同，需要做不同的配置。

3.3.1 Ramdisk 初始脚本配置

Ramdisk中，所有工作都在`/init (buildroot/output/rockchip_${chip}_ramboot/target/init)`脚本完成。`init`脚本根据系统的打包类型，对系统进行解密或者验签，并统一生成`vroot`设备。大致流程如下图，兼顾了A/B系统，系统校验和系统加密。



> [1] 加密系统下，会根据系统分区的实际大小对system文件系统进行扩容。

`init`脚本由`buildroot/board/rockchip/common/security-ramdisk-overlay/init.in`修改生成。操作之前，请按System-Verity 或System-Encryption 的要求，生成对应的`security.info`，以下操作会用到：

```
cp buildroot/board/rockchip/common/security-ramdisk-overlay/init.in \
   buildroot/board/rockchip/common/security-ramdisk-overlay/init
init_file=buildroot/board/rockchip/common/security-ramdisk-overlay/init
optee_storage=RPMB # 根据硬件实际情况，optee_storage填充RPMB或SECURITY（security分区）
source security.info

# 系统加密或系统校验二选一操作：
# - 系统校验
sed -i "s/ENC_EN=/ENC_EN=false/" $init_file
sed -i "s/OFFSET=/OFFSET=$hash_offset/" $init_file
sed -i "s/HASH=/HASH=$root_hash/" $init_file
sed -i "s/# exec busybox switch_root/exec busybox switch_root/" "$init_file"

# - 系统加密
sed -i "s/ENC_EN=/ENC_EN=true/" $init_file
sed -i "s/CIPHER=/CIPHER=$cipher/" $init_file
```

```
sed -i "s/SECURITY_STORAGE=RPMB/SECURITY_STORAGE=$optee_storage/" $init_file
sed -i "s/# exec busybox switch_root/exec busybox switch_root/" "$init_file"
```

3.3.2 Ramdisk 编译配置

默认的Ramdisk配置一般为`rockchip_<chip_name>_ramboot_defconfig`，配置是以系统校验（System-Verity）为目的设置的，如果客户有系统加密（System-Encryption）需求，额外添加以下配置：

```
BR2_PACKAGE_TEE_USER_APP=y # 参考KeyBox章节配置，部分SDK可以直接 include
"tee_aarch64_v2.config"
BR2_PACKAGE_TEE_USER_APP_COMPILE_CMD="6432"
BR2_PACKAGE_TEE_USER_APP_TEE_VERSION="v2"
BR2_PACKAGE_TEE_USER_APP_EXTRA_TOOLCHAIN="$(PATH_TO_CROSS_COMPILE)"

BR2_PACKAGE_RECOVERY=y
BR2_PACKAGE_RECOVERY_UPDATEENGINEBIN=y # 加密Key是存放在misc中的，Recovery 选项是为了
添加misc解析程序
```

> 默认配置都会包含`BR2_ROOTFS_OVERLAY+="board/rockchip/common/security-ramdisk-overlay/"`，该配置会将Ramdisk 初始脚本配置的`init`脚本放入Ramdisk中。
>
> `BR2_PACKAGE_TEE_USER_APP`相关配置含义，参考KeyBox章节，请按实际需求灵活配置。

Ramdisk的本质，也是一个完整的中间层系统，因此它的编译和正常的Buildroot编译一样：

```
cd buildroot
./build/envsetup.sh rockchip_${chip}_ramboot_defconfig   # chip 根据实际填写
make # 固件生成在 output/rockchip_${chip}_ramboot/images/rootfs.cpio.gz
```

客户也可以自行准备Ramdisk，并按如下要求进行自定义：

1. Ramdisk 必须包含Luksmeta工具集，至少保证，在检验系统中有veritysetup可用，在加密系统中有dmsetup工具可用。
2. 在加密系统中，需要移植TEE程序，用以完成加密密钥的读写；移植updateEngine程序，用以读取misc中的密钥，也可以自行剥离这部分，独立程序，或另外找地方传递密钥参数。
3. Ramdisk的init启动脚本需要参考Ramdisk 初始脚本配置 修改，完成Ramdisk挂载新系统的要求。

3.3.3 Ramdisk 打包

生成的Ramdisk系统需要和Kernel固件一起打包成`boot.img`，根据使用方案不同，打包方式也不一样：

```
# 和kernel一起打包成boot.img, FIT / AVB 方案二选一
cd ${RK_SDK_DIR} # SDK 根目录
# FIT 方案下
./device/rockchip/common/scripts/mk-fitimage.sh boot.img \
    device/rockchip/${chip}/boot4recovery.its \
    kernel/arch/arm64/boot/Image \
    kernel/arch/arm64/boot/dts/rockchip/${RK_KERNEL_DTB} \
    kernel/resource.img \
```

```
    buildroot/output/rockchip_${chip}_ramboot/images/rootfs.cpio.gz

# AVB方案下
./kernel/scripts/mkbootimg \
    --kernel kernel/arch/arm64/boot/Image \
    --ramdisk buildroot/output/rockchip_${chip}_ramboot/images/rootfs.cpio.gz \
    --second kernel/arch/arm64/boot/dts/rockchip/${RK_KERNEL_DTB} \
    -o boot.img
```

签名上面生成的`boot.img`，参考FIT 固件签名扩展命令3或者AVB 固件签名

4. KeyBox

在使用分区或系统加密的时候，需要一个密钥存储的位置，通常的做法是通过OPTEE，将密钥存储在RPMB或者Security分区。由于这部分代码设计是和厂家私钥直接接触，原则上是需要厂家自行设计。考虑到部分客户对OPTEE不了解，SDK中添加了一个KeyBox的实例代码，并且在System-Encryption中进行了实际使用，供客户参考。**（推荐客户重构代码）**

OPTEE相关代码，位于`<SDK>/external/security/`下，包括bin和rk_tee_user两个仓库：

- bin存放预编译的部分非公开代码和头文件，是OPTEE运行的基础库。
- rk_tee_user是编译用户自己的CA/TA的工程，里面自带部分demo test工程。

OPTEE如何使用，参考**Rockchip_Developer_Guide_TEE_SDK_CN.pdf**文档，里面有详细的介绍，这里就不在赘述。这里主要讲KeyBox程序如何编译。

4.1 独立编译

KeyBox 代码位于`<SDK>/buildroot/package/rockchip/tee-user-app`

```
# tree buildroot/package/rockchip/tee-user-app/
buildroot/package/rockchip/tee-user-app/
├── Config.in
├── extra_app
│   ├── host
│   │   ├── main.c
│   │   └── Makefile
│   └── ta
│       ├── include
│       │   ├── ta_keybox.h
│       │   └── user_ta_header_defines.h
│       ├── keybox.c
│       ├── Makefile
│       └── sub.mk
└── tee-user-app.mk
```

编译时，需要将`extra_app`下的`host`和`ta`的分别放到`rk_tee_user`中，例如：

```
# tree rk_tee_user/v2/host/extra_app/ rk_tee_user/v2/ta/extra_app/
rk_tee_user/v2/host/extra_app/
├── main.c
└── Makefile
rk_tee_user/v2/ta/extra_app/
├── include
│   ├── ta_keybox.h
│   └── user_ta_header_defines.h
├── keybox.c
├── Makefile
└── sub.mk
```

`rk_tee_user`的编译同时生成CA(`keybox_app`)和TA程序，这两个程序是运行在不同环境中的。

- CA，运行在系统的用户空间中，编译链和系统编译链保持一致。
- TA，运行在TrustZone环境，编译链需要和BL32(TEE)保持一致，没有特殊声明的情况下，默认为32位编译链。

因此，`rk_tee_user`编译可能需要两个不同的编译链配置。

以系统为64位系统，TA为32位系统为例：

```
cd rk_tee_user/v2 # v1 同理，只是编译目录不同
./build.sh 6432
ARM32_TOOLCHAIN=${ARM32_CROSS_COMPILER_PATH} \
AARCH64_TOOLCHAIN=${AARCH64_CROSS_COMPILER_PATH} \
./build.sh 6432                    # 编译64位CA，32位TA
# 生成程序：
# out/ta/extra_app/8c6cf810-685d-4654-ae71-8031beee467e.ta
# out/extra_app/keybox_app
```

4.2 tee_user_app 编译

KeyBox程序在Buildroot中，使用`tee_user_app`包编译。以64位系统搭配32位TA编译为例，配好以下内容，即可实现编译。

```
BR2_PACKAGE_TEE_USER_APP=y
BR2_PACKAGE_TEE_USER_APP_COMPILE_CMD="6432"
BR2_PACKAGE_TEE_USER_APP_TEE_VERSION="v2"
BR2_PACKAGE_TEE_USER_APP_EXTRA_TOOLCHAIN="$(PATH_TO_CROSS_COMPILE)"
```

生成的KeyBox程序和TA自动集成到系统中。

4.3 Kernel 使能 OPTEE

KeyBox 基于OPTEE功能开发完成，使用时，在Kernel中加入OPTEE支持。

```
CONFIG_TEE=y
CONFIG_OPTEE=y
```

同时在dts中，配置optee节点

```
        optee: optee {
                compatible = "linaro,optee-tz";
                method = "smc";
                status = "okay";
        }
```

4.4 KeyBox 修改建议

KeyBox章节中列举了一个密钥存储的通用做法，且该方案属于开源做法，同步给所有客户。为增加密钥存储的安全性，请客户自行修改Keybox，提供几种常用的修改建议：

- Keybox的使用环境必须处于安全环境中，要搭配FIT安全方案或者AVB安全方案使用。
- TA文件可以按Rockchip_Developer_Guide_TEE_SDK_CN.pdf，第5章节 **TA签名**，对ta文件进行签名，签名完成后，第三方无法在客户机器上运行自定义TA。
- 严格限制CA使用环境，避免CA被读取后，在其他环境下将密钥读取出来。默认代码中，KeyBox可以在任何情况下，进行写Key操作，但是只能在Ramdisk下，且PID=1的程序中读Key。
- 修改CA-TA交互，默认代码中，CA-TA做了一些简单的随机校验的动作，一旦CA-TA交互出出错，直接输出随机密钥。

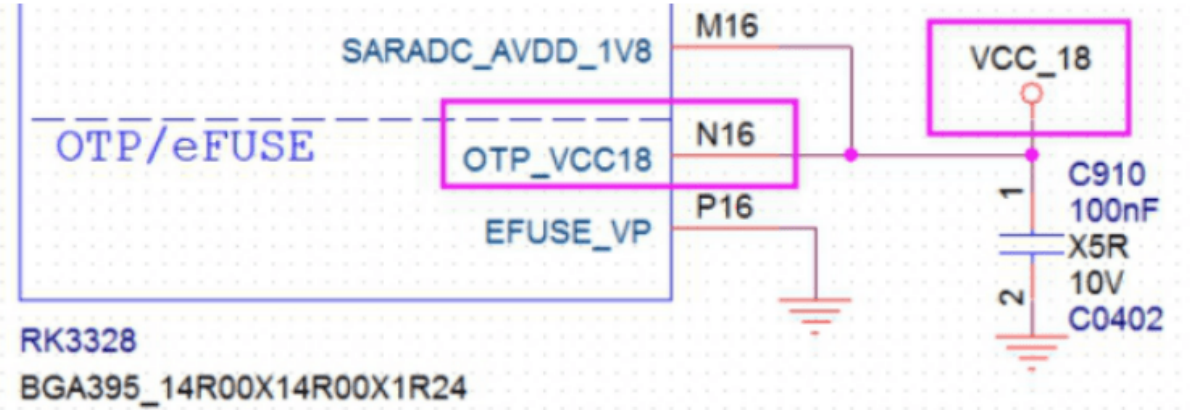
5. 安全信息烧写

5.1 Key Hash

不同芯片安全存储的介质不同，根据产品版本确认芯片使用的存储介质，并按要求操作。

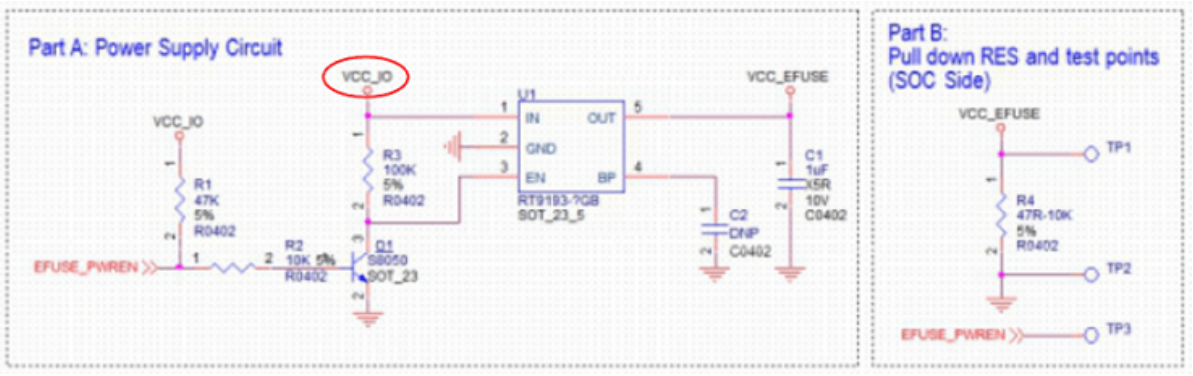
5.1.1 OTP

如果芯片使用 OTP 启用 Secure Boot 功能，保证芯片的 OTP 引脚在 Loader 阶段有供电。直接通过 AndroidTool(Windows) / upgrade_tool(Linux)下载固件，第一次重启，Loader 会负责将 Key 的 Hash 写入 OTP，激活 Secure Boot。再次重启，固件就处于保护中了。



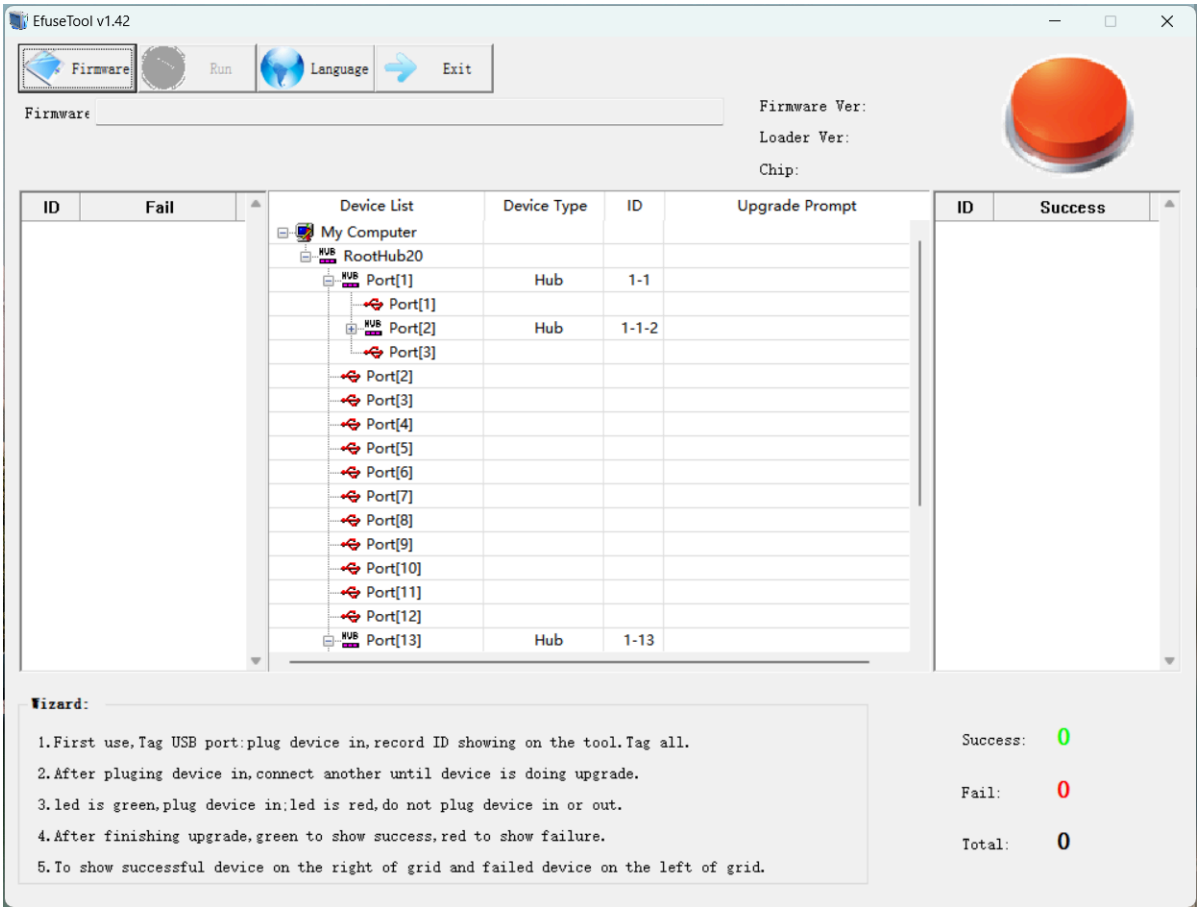
5.1.2 eFuse

如果芯片使用 eFuse 启用 Secure Boot 功能，请保证 VCC_IO 在 MaskRom 状态（硬件默认状态）下有电，且VCC_EFUSE需要以下电源控制电路。



使用 tools/windows/eFusetool_vXX.zip，板子进入 MaskRom 状态。

点击"固件"，选择签名的 update.img，或者 Miniloader.bin，点击运行"启动"，开始烧写 eFuse。



eFuse 烧写成功后，再断电重启，进入 MaskRom后，将其他签名固件下载到板子上。

5.2 自定义安全信息

客户自定义安全信息，建议使用OPTEE存储，参考Rockchip_Developer_Guide_TEE_SDK_CN.pdf文档。

6. Security Demo

前面的章节介绍了从零搭建固件安全方案的方法，为简化搭建流程，SDK提供了几个管理脚本，并耦合到通用编译脚本`build.sh`中，通过几个简单的配置，实现系统加密或者系统校验的目的。

注意：该章节不再提供具体的原理讲解和详细操作，需要了解操作的具体细节，请翻看前面的文档。关于简化脚本，基于通用SDK开发，因为SDK不断迭代且个别SDK版本与通用SDK存在冲突，无法保证每个SDK的简化脚本都能正常使用，如果简化脚本出错，可以先参考前面章节，尝试调试。

由于固件安全的敏感性，且该方案开源给所有客户，建议客户在快速搭建系统外，认真分析其原理，修改优化方案，做出符合自己标准的安全固件。

6.1 SDK配置

在SDK根目录下，可以使用`make menuconfig`进行SDK编译配置，安全相关配置都集中在`RK_SECURITY`下:

```
[*] security feature
      *** Security check method (system-verity) needs squashfs rootfs type ***
      Secureboot Method (avb)  --->
      Optee Storage (rpmb)  --->
      security check method (base|system-encryption|system-verity) (base)  --->
[ ]   burn security key (NEW)
```

具体配置如下：

```
RK_SECURITY                             # 安全功能使能

RK_SECUREBOOT_FIT                       # 基础安全功能方式，FIT / AVB 二选一
RK_SECUREBOOT_AVB

RK_SECURITY_OPTEE_STORAGE_RPMB          # OPTEE 安全存储使用介质，二选一
RK_SECURITY_OPTEE_STORAGE_SECURITY      # eMMC存储选RPMB，flash存储选SECURITY

RK_SECURITY_CHECK_SYSTEM_BASE           # 只校验 loader + uboot + boot
RK_SECURITY_CHECK_SYSTEM_VERITY         # 在base的基础上，增加rootfs校验
RK_SECURITY_CHECK_SYSTEM_ENCRYPTION     # 在base的基础上，增加rootfs加密

RK_SECURITY_INITRD_BASE_CFG             # 有rootfs校验或加密时，需要配置ramboot
RK_SECURITY_INITRD_TYPE
RK_SECURITY_FIT_ITS

RK_SECURITY_BURN_KEY                    # 使能public key hash 烧写功能
```

> **以上选项请特别注意RK_SECURITY_BURN_KEY，为了调试方便，默认状态都是关闭的，关闭情况下不会真正将Key Hash写入OTP，也就是默认loader.bin是安全的，可以被替换，但从loader开始，和烧过OTP的流程一致，都会对uboot进行严格校验。正式生产产品的时候，请打开该选项，进行Key Hash烧录**

以上配置结束后，可以使用`./build.sh`或`make`进行编译。过程中会出现报错，根据提示，排除错误即可。或者依照文档，继续进行修改配置。编译中的检查报错，只对关键配置进行检查，各SDK默认配置有差别，不保证编译出来的固件安全功能完善，以最终验证为准。如果有问题，可以先确定是哪个固件的校验出问题，再翻看文档的具体调试。

6.2 详细配置

6.2.1 密钥生成

- 密钥生成

```
./build.sh createkeys
```

- 如果使用的是系统加密，将带root权限的用户密码存储在U-Boot中

```
echo "Password for sudo" > u-boot/keys/root_passwd
```

> 编译加密系统时，会使用的系统的loopback特性，需要用户对编译的电脑有root的操作权限，这意味着该方法无法在服务器环境下编译（除非你的用户有root权限）。

6.2.2 U-Boot修改

U-Boot 配置按产品版本的"Kernel 检验方式"分为：

- FIT 方案需要添加

```
CONFIG_FIT_SIGNATURE=y
CONFIG_SPL_FIT_SIGNATURE=y
```

- AVB 方案需要添加

  AVB 的U-Boot改动比较多，具体参考AVB U-boot Configuration

6.2.3 Kernel 修改

Kernel需要添加OPTEE和Device Mapper配置

```
CONFIG_BLK_DEV_DM=y
CONFIG_DM_CRYPT=y
CONFIG_BLK_DEV_CRYPTOLOOP=y
CONFIG_DM_VERITY=y

CONFIG_TEE=y                    # 加密系统必须选
CONFIG_OPTEE=y                  # 加密系统必须选
```

DTS 文件需要添加OPTEE节点

```
        optee: optee {
                compatible = "linaro,optee-tz";
                method = "smc";
                status = "okay";
        }
```

6.2.4 Buildroot修改

Buildroot 系统配置，需要添加

```
BR2_ROOTFS_OVERLAY+="board/rockchip/common/security-system-overlay"
```

6.2.5 Ramdisk修改

Ramdisk的配置，默认是按系统校验配置的，如果需要改成系统加密，需要添加以下配置

```
BR2_PACKAGE_TEE_USER_APP=y                    # 参考KeyBox章节配置，部分SDK可以直接
include "tee_aarch64_v2.config"
BR2_PACKAGE_TEE_USER_APP_COMPILE_CMD="6432"
BR2_PACKAGE_TEE_USER_APP_TEE_VERSION="v2"
BR2_PACKAGE_TEE_USER_APP_EXTRA_TOOLCHAIN="$(PATH_TO_CROSS_COMPILE)"

BR2_PACKAGE_RECOVERY=y
BR2_PACKAGE_RECOVERY_UPDATEENGINEBIN=y       # 加密Key是存放在misc中的，Recovery 选项
是为了添加misc解析程序
```

6.3 验证方法

文档的安全方案，保护了loader，trust，uboot，boot，system这几级。

按文档方法，编译一个安全固件，正确下载密钥，并正常开机。如果开机异常，请先排除开机异常问题。

再准备一个非安全固件，或替换过key的固件。用第二套固件的任意镜像，替换安全的系统中受保护的分区，都会造成系统无法启动。

如果替换的过程中，发现某个分区保护失效，按Partition Security Process示意图查看是哪部分的校验失效了，再翻看文档具体章节，查看配置。

6.4 调试方法

如果init无法正常跳转系统，可以打开打印，并且使用`DEBUG`函数添加一些私有打印，按照Ramdisk 初始脚本配置确定它的运行状态。

```
diff --git a/board/rockchip/common/security-ramdisk-overlay/init.in
b/board/rockchip/common/security-ramdisk-overlay/init.in
index 756d0b5375..c3267e28b1 100755
--- a/board/rockchip/common/security-ramdisk-overlay/init.in
+++ b/board/rockchip/common/security-ramdisk-overlay/init.in
@@ -16,7 +16,7 @@ BLOCK_TYPE_SUPPORTED="
 mmcblk
 flash"

-MSG_OUTPUT=/dev/null
+MSG_OUTPUT=/dev/kmsg
 DEBUG() {
        echo $1 > $MSG_OUTPUT
 }
```

7. 参考资料

Rockchip_Developer_Guide_UBoot_Nextdev_CN.pdf

Rockchip-Secure-Boot-Application-Note-V1.9.pdf

Rockchip-Secure-Boot2.0.pdf

Rockchip_Developer_Guide_TEE_SDK_CN.pdf

SDK/kernel/ Documentation/device-mapper/

https://android.googlesource.com/platform/external/avb/+/master/README.md

https://source.android.google.cn/security/verifiedboot/dm-verity