Rockchip Linux Secure Boot Developer Guide

ID: RK-KF-YF-379

Release Version: V3.0.0

Release Date: 2022-4-30

Security Level: □Top-Secret □Secret □Internal ■Public

Rockchip Electronics Co., Ltd.

No.18 Building, A District, No.89, software Boulevard Fuzhou, Fujian,PRC

Website: www.rock-chips.com

Customer service Tel: +86-4007-700-590

Customer service Fax: +86-591-83951833

Customer service e-Mail: fae@rock-chips.com

Preface

Overview

This document mainly introduces the steps and notices of Secure Boot under RK Linux platform, which is convenient for customers to do secondary development base on it. Secure Boot function is designed to ensure devices with correct and valid firmware, and unsigned or invalid firmware will not boot.

Product Version

Chipset	Kernel verify	Kernel Version
RK3308/RK3399/RK3328/RK3326/PX30	AVB	4.4
RK3588/RK356X	FIT	5.10

Intended Audience

This document (this guide) is mainly intended for:

Technical support engineers

Software development engineers

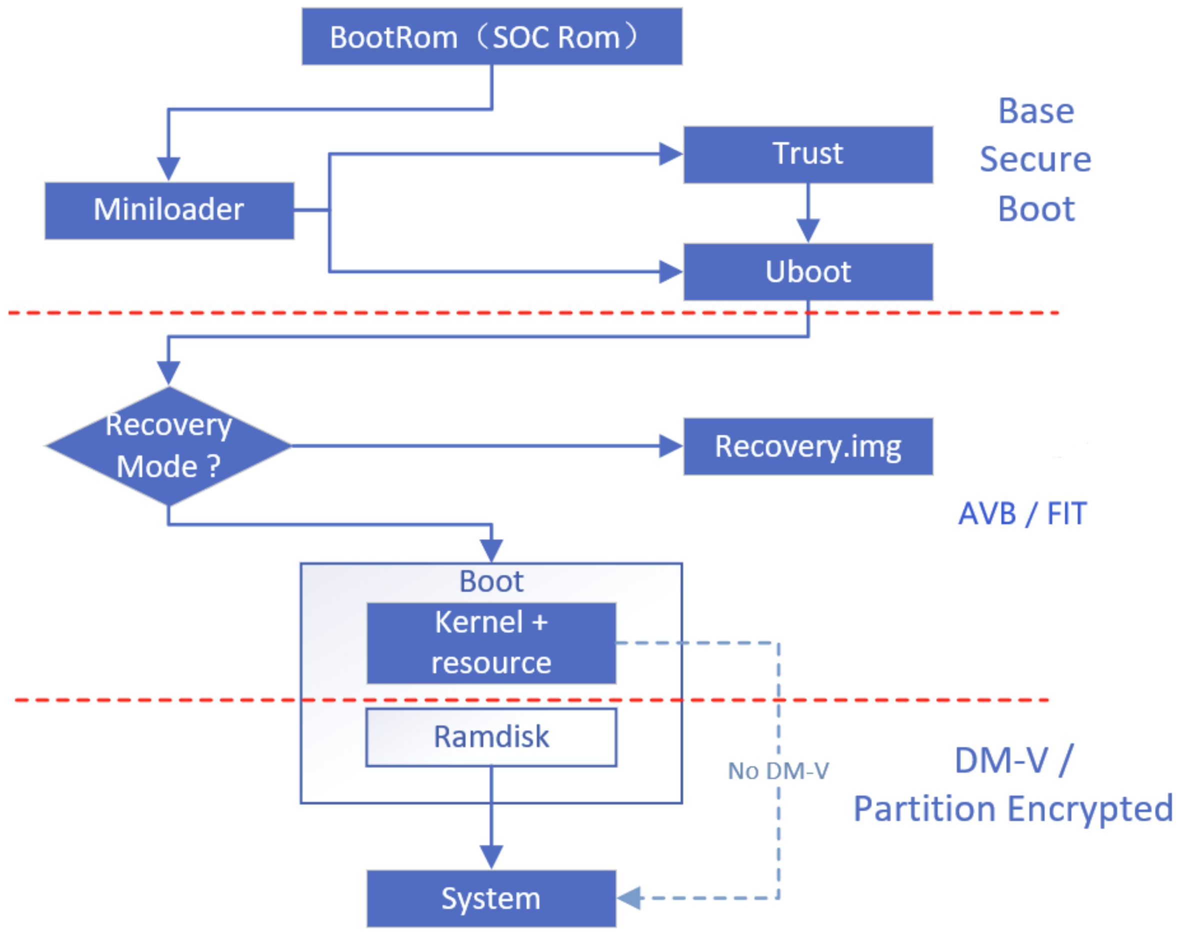
Revision History

Version	Author	Date	修改说明
V1.0.0	WZZ	2018-10-31	Initial version
V1.0.1	WZZ	2018-12-17	Fix vbmeta to security
V2.0.0	WZZ	2019-06-03	Sign_Tool is compatible with AVB boot.img, Update device-mapper related introduction
V2.0.1	Ruby Zhang	2020-08-10	Update the company name and document format
V3.0.0	WZZ	2022-04-30	Add RK3588 FIT supported Fixed AVB key storage description Add more reference documents Update AVB description Update tool links
V3.0.1	WZZ	2022-09-27	Add RK356X FIT supported statement

Contents

1. Secure Boot Introduction

1.1 Secure Boot Process



As shown in the above figure, starting from BootRom, a reliable security verification solution is established step by step on Linux platform. And it is divided into three parts in order. Customers can choose verification contents according to their own need.

Base Secure Boot: start with BootRom and verify Miniloader/Trust/Uboot step by step.

AVB / FIT: start with Uboot and verify Boot and Recovery (Optional).

DM-V: verify or decrypt system partition by Ramdisk tool that is packaged in Boot.

Note: the main difference between the above process and Android platform is DM-V process. Android takes fs_mgr mechanism to implement DMV verification in kernel. But Linux uses Ramdisk to verify.

1.2 Secure Boot Secure Storage

The following secure storage areas are included in Linux platform:

Storage area	Note
OTP / eFuse	Located in SOC, it is a fuse mechanism and irreversible flashing. OTP can be flashed by Miniloader but eFuse can only be flashed by PC tool (Refer to Section 2.2 Secure Information Flahing). Medias are different in different SOC. Currently, Linux platforms mainly includes: eFuse: RK3399 / RK3288 OTP： RK3308 / RK3326 / PX30 / RK3328 (See Section 1.3 Reference Resources Rockchip-Secure-Boot-Application-Note-V1.9.pdf）
RPMB	It is a physical partition of eMMC, and it is not available in file system and requires SOC authorizing access (that is, it can only be accessed by TEE). Generally it is considered to be a secure area.
Security Partition	The logical partition of storage medium is a temporary partition added to supplement the absence of RPMB on Flash. The partition contents are encrypted and stored and cannot be mounted, but may be forcibly erased. It can only be accessed by TEE (if you force to erase, TEE access will fail and Secure Boot will not start properly).

Note: Since OTP (eFuse) is mainly used internally by Rockchip, for security information, customers should give priority to other areas such as RPMB/Security. If you have a special needs, please apply for related materials by business.

Secure information and storage location of each process:

Security Information	Storage area
Base Secure Boot	Public Key Hash is stored in OTP/eFuse
AVB	In OTP device： permanent_attributes.bin hash is stored in OTP In eFuse device： permanent_attributes.bin is stored in RPMB/Security Partition permanent_attributes_cer.bin is stored in RPMB/Security Partition (permanent_attributes.bin is verified by Base Secure Boot Key)
DM-V	Root Hash is stored in Boot's Ramdisk, and Boot contents are verified by AVB to ensure accurate

1.3 Reference Resources

Reference documents:

Rockchip_Developer_Guide_UBoot_Nextdev_CN.pdf

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

Rockchip-Secure-Boot2.0.pdf

SDK/tools/linux/Linux_SecurityAVB/Readme.md

SDK/kernel/ Documentation/device-mapper/

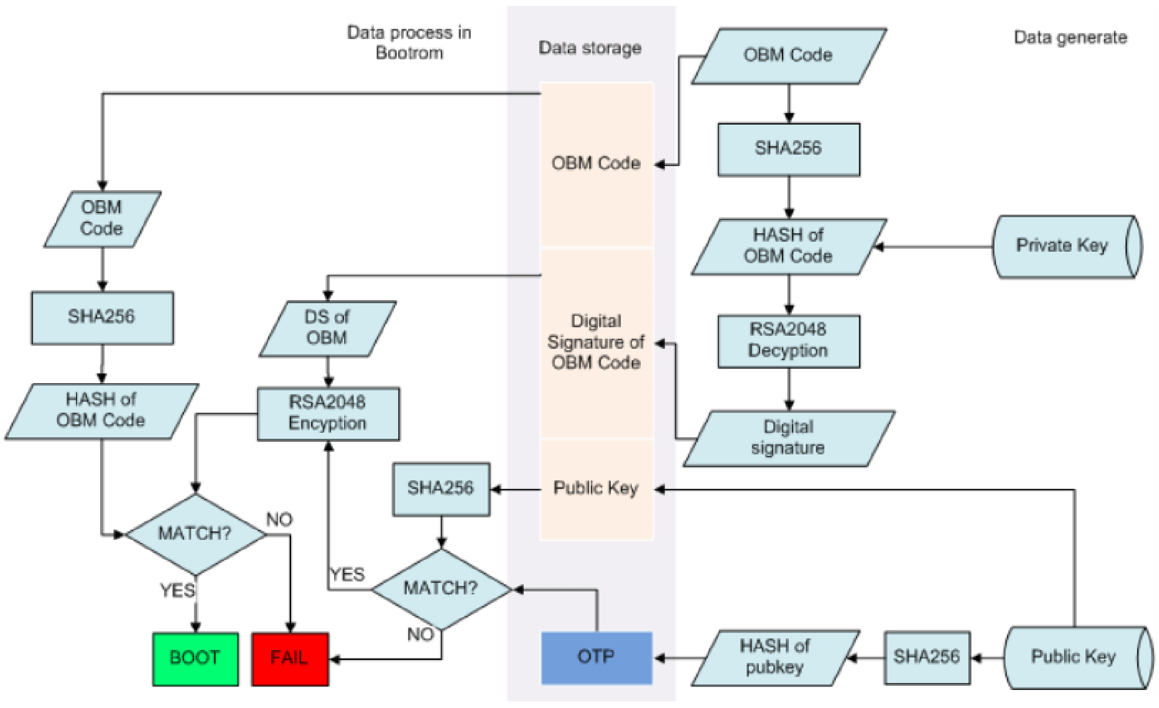
https://android.googlesource.com/platform/external/avb/+/master/README.md

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

2. Base Secure Boot

Base Secure Boot provides basic security to U-boot (loader/trust/uboot)

The start process is as follows:



In short, a signed firmware includes Firmware(OBM Code) + Digital Signature + Public key

Digital Signature + Public Key are added by signature tool.

About memory, the signed firmware is stored in eMMC or Flash, and the Public Key Hash is stored in OTP (eFuse) of a chip.

When starting, public key of the end of a firmware is verified by the Hash in OTP, and then the digital signature is verified by the public key to achieve the binding effect of a chip and signature code.

See Rockchip-Secure-Boot-Application-Note-V1.9.pdf for details.

If device using FIT, skip this part, because FIT image making has contained Base Secure Boot operation, do not repeat the operation manually

2.1 Signature Tools

2.1.1 UI tool (Windows)

SDK/tools/windows/SecureBootTool_v1.94 or see the enterprise network disk (see Section 1.3 Reference Resources)

1. Modify configurations

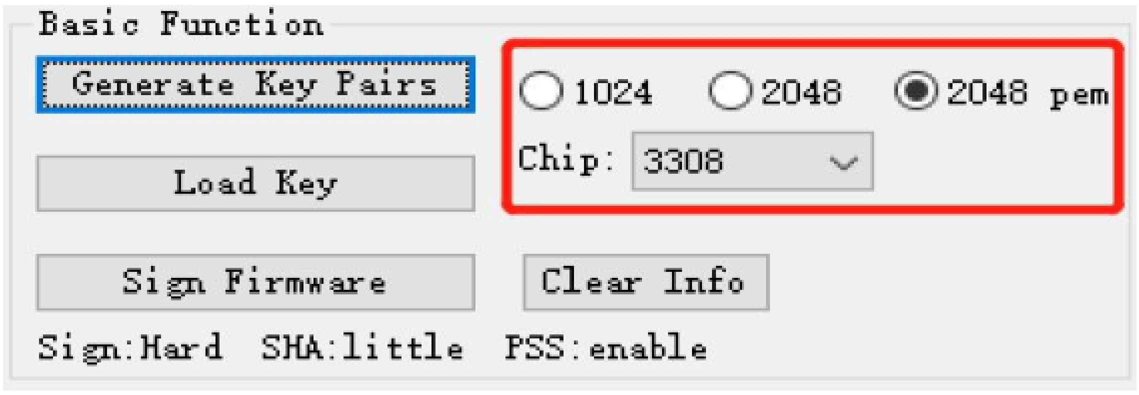
Open setting.ini in the tool：

If AVB is needed, modify exclude_boot_sign = True。

If the chip uses OTP to enable Secure Boot, set sign_flag=0x20. (bit 5: loader OTP write enabled, boards that have been written OTP, or eFuse chips, this flag should be set to null.)

2. Generate public and private keys

Select Chip and Key formats (pem is common format), click “Generate Key Pairs” to generate PrivteKey.pem and PublicKey.pem. (**Keys are generated randomly. Please save these two keys properly. After the security function is enabled, if the two keys are lost, the device will not be able to upgrade**.)



○ 1024 ○ 2048 ● 2048 pem

Chip: 3308

Sign:Hard SHA:little PSS:enable

3. Load key

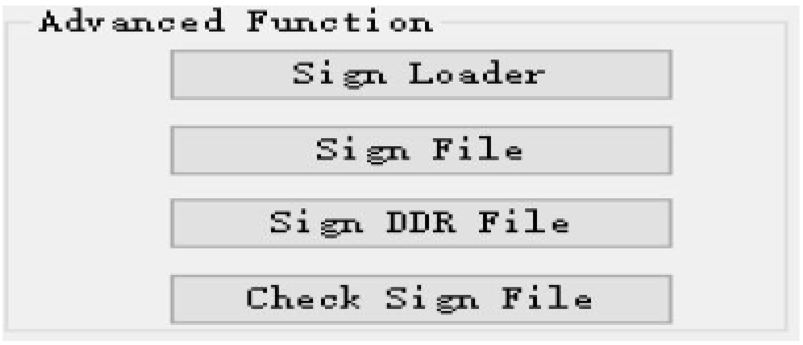
Select “Load Key” to load the public and private keys follow the prompts.

4. Signature

There are two ways to sign: only sign update.img and independent sign.

If you have already packaged update.img, you can sign update.img directly by “Sign Firmware”.

Independent sign need to press “CTRL + R +K” to open “Advanced Function”:



“Sign Loader” is used to sign Miniloader.bin

“Sign File” is used to sign trust.img and uboot.img

(Actually, sign update.img is unpack update.img, and then sign each firmware separately, and then packaged as a whole, at last sign the whole again)

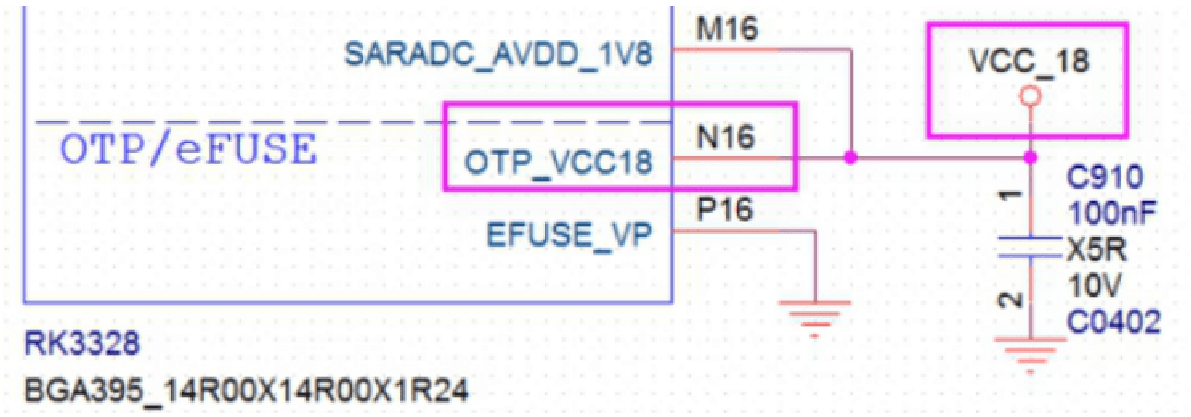
2.1.2 Command Line Tool

```
# step1: Generate rsa public private key (if you already have a key, skip this step)
 ./rk_sign_tool kk --out .
# step2: Load the public and private keys (can be load only once) and automatically saved to setting.ini
./rk_sign_tool lk --key privateKey.pem --pubkey publicKey.pem
# step3: Select the chip to determine signature  solution
./rk_sign_tool cc --chip 3326
# stpe4: Open setting.ini
# Set sign_flag = 0x20
# If the platform uses OTP to store security information, set sign_flag = 0x20, enable RKloader OTP writing function, empty boards must be enabled; otherwise, this item will be cleared.
# If AVB is needed, change exclude_boot_sign = True
# stpe5: Overall sign( independent sign skip this step).
./rk_sign_tool sf --firmware update.img
# stpe6: Sign loader, overall sign, skip steps 6-8.
./rk_sign_tool sl --loader  rk3326loader.bin
# stpe7: Sign uboot, for versions before v1.3, RK3326/RK3308 need to add–pss; others do not need.
./rk_sign_tool si --img uboot.img
# stpe8: Sign trust, for version before v1.3, RK3326/RK3308 need to add--pss; others do not needed.
./rk_sign_tool si --img trust.img
```

2.2 Secure Information Flashing

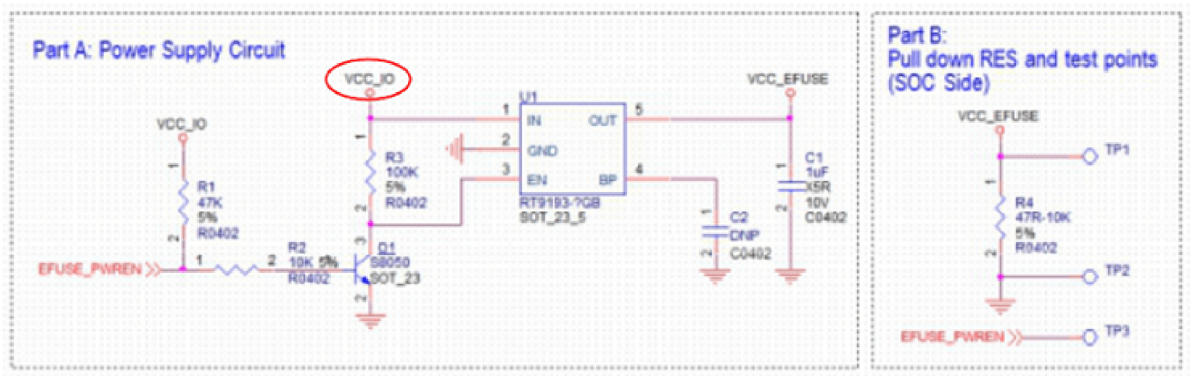
2.2.1 OTP

If a chip uses OTP to enable Secure Boot function, ensure that OTP pin of the chip is powered during Loader process. Download firmware directly through AndroidTool (Windows) / upgrade_tool (Linux). The first time you restart, Loader will be responsible for writing Hash of the Key to OTP and activating Secure Boot. Restart again and then the firmware is protected.



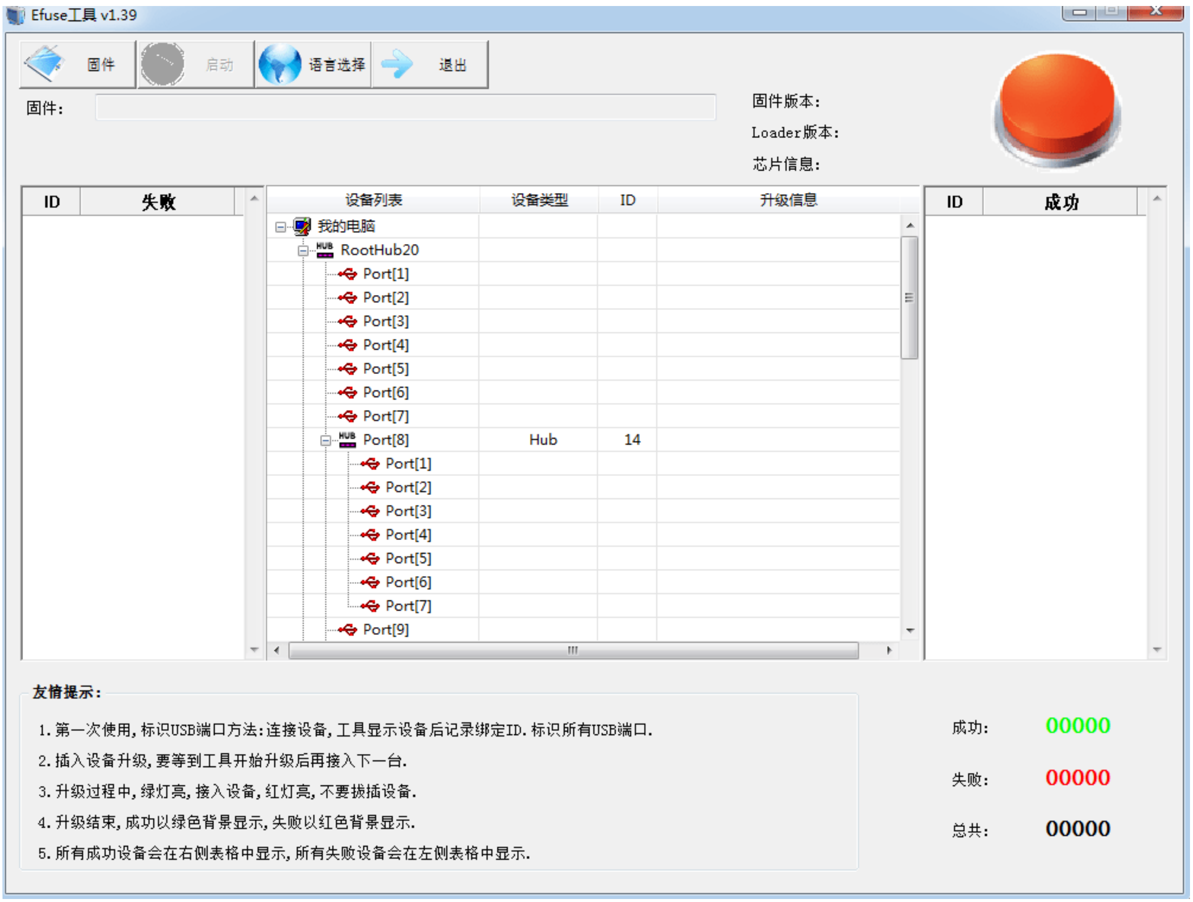
2.2.2 eFuse

If a chip uses eFuse to enable Secure Boot function, please ensure that the hardware connection is correct, because kernel has not been started when downloading eFuse, so please ensure that VCC_IO was powered in MaskRom state.



The board enters MaskRom state by using tools/windows/eFusetool_vXX.zip,.

Click "Firmware", select the signed update.img, or Miniloader.bin, click "Start" to start download eFuse.



固件:

固件版本:
Loader版本:
芯片信息:

ID	失败

设备列表	设备类型	ID	升级信息
我的电脑			
RootHub20			
Port[1]			
Port[2]			
Port[3]			
Port[4]			
Port[5]			
Port[6]			
Port[7]			
Port[8]	Hub	14	
Port[1]			
Port[2]			
Port[3]			
Port[4]			
Port[5]			
Port[6]			
Port[7]			
Port[9]			

ID	成功

友情提示:

1. 第一次使用，标识USB端口方法：连接设备，工具显示设备后记录绑定ID。标识所有USB端口。
2. 插入设备升级，要等到工具开始升级后再接入下一台。
3. 升级过程中，绿灯亮，接入设备，红灯亮，不要拔插设备。
4. 升级结束，成功以绿色背景显示，失败以红色背景显示。
5. 所有成功设备会在右侧表格中显示，所有失败设备会在左侧表格中显示。

成功: 00000
失败: 00000
总共: 00000

After eFuse is successfully downloaded, power off and restart, enter MaskRom, use AndroidTool to download other sign firmware to the board.

2.3 Verify

After secure boot takes effect, there are logs similar to the following output during Loader process.

```
SecureMode = 1
Secure read PBA: 0x4
SecureInit ret = 0, SecureMode = 1
```

3. Kernel Verify

3.1 AVB

AVB requires U-boot to work together. AVB on Linux is used to guarantee the integrity of uboot.img (including boot.img and recovery.img).

The corresponding tool is in tools/linux/Linux_SecurityAVB.

For detailed usage, please refer to tools/linux/Linux_SecurityAVB/Readme.md

(If there is a conflict, the Linux_SecurityAVB/Readme.md shall prevail)

3.1.1 Notice

About device lock & unlock：

When a device is in unlock state, program will still verify the whole boot.img. If the firmware has an error, the program will report what the error is, but **the device starts normally**. If a device is in lock state, the program will verify the whole boot.img. If the firmware has an error, the next level of firmware will not be started. Therefore, setting the device to unlock state during debugging is more convenient.

3.1.2 Firmware Configuration

3.1.2.1 Trust：

Endter rkbin/RKTRUST, take RK3308 as an example, find RK3308TRUST.ini and change:

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

to

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

TOS trust.img has enabled `BL32_OPTION` as default， such as `RK3288TOS.ini`

3.1.2.2 U-boot：

U-boot requires the following features:

```
# OPTEE support
CONFIG_OPTEE_CLIENT=y
CONFIG_OPTEE_V1=y       #RK312x/RK322x/RK3288/RK3228H/RK3368/RK3399 and V2 are
mutually
CONFIG_OPTEE_V2=y       #RK3308/RK3326 and V1 are mutually exclusive

# CRYPTO support
CONFIG_DM_CRYPTO=y          # eFuse device required
CONFIG_ROCKCHIP_CRYPTO_V1=y # For eFuse devices, like RK3399/RK3288
CONFIG_ROCKCHIP_CRYPTO_V2=y # For eFuse devices, like RK1808

# 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        # open when rpmb is not
available, not open as default
CONFIG_ROCKCHIP_PRELOADER_PUB_KEY=y                 # just for eFuse device
CONFIG_RK_AVB_LIBAVB_ENABLE_ATH_UNLOCK=y

#fastboot support
CONFIG_FASTBOOT=y
CONFIG_FASTBOOT_BUF_ADDR=0x2000000                  # it varies between
platforms, refer to default configuration
CONFIG_FASTBOOT_BUF_SIZE=0x08000000                 # it varies between
platforms, refer to default configuration
CONFIG_FASTBOOT_FLASH=y
CONFIG_FASTBOOT_FLASH_MMC_DEV=0
```

Use `./make.sh xxxx`, to generate uboot.img, trust.img, loader.bin

3.1.2.3 Parameter：

`vbmeta` and `system` partitions is required in `parameter.txt`, `security` partitions is optional:

- `vbmeta` is stored partitions signed information, it needs 1M size.
- `system` is named `rootfs` in buildroot. In order to fit more system, call it `system` in AVB. So if used buildroot, do not forget to rename your `rootfs` to `system`.
- `security` has the same functions to RPMB, 4M size, you have to add this partition if storage medium was not eMMC.

The following is an example of AVB parameter:

```
# 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

# AVB and A/B parameter:
0x00002000@0x00004000(uboot),0x00002000@0x00006000(trust),0x00004000@0x00008000(
misc),0x00010000@0x0000c000(boot_a),0x00010000@0x0001c000(boot_b),0x00010000@0x0
002c000(backup),0x00020000@0x0003c000(oem),0x00300000@0x0005c000(system_a),0x003
00000@0x0035c000(system_b),0x00000800@0x0065c000(vbmeta_a),0x00000800@0x0065c800
(vbmeta_b),0x00002000@0x0065d000(security),-@0x0065f00(userdata:grow)
```

When downloading, the name on the tool should be modified synchronously. After modification, reload parameter.

3.1.3 AVB Key

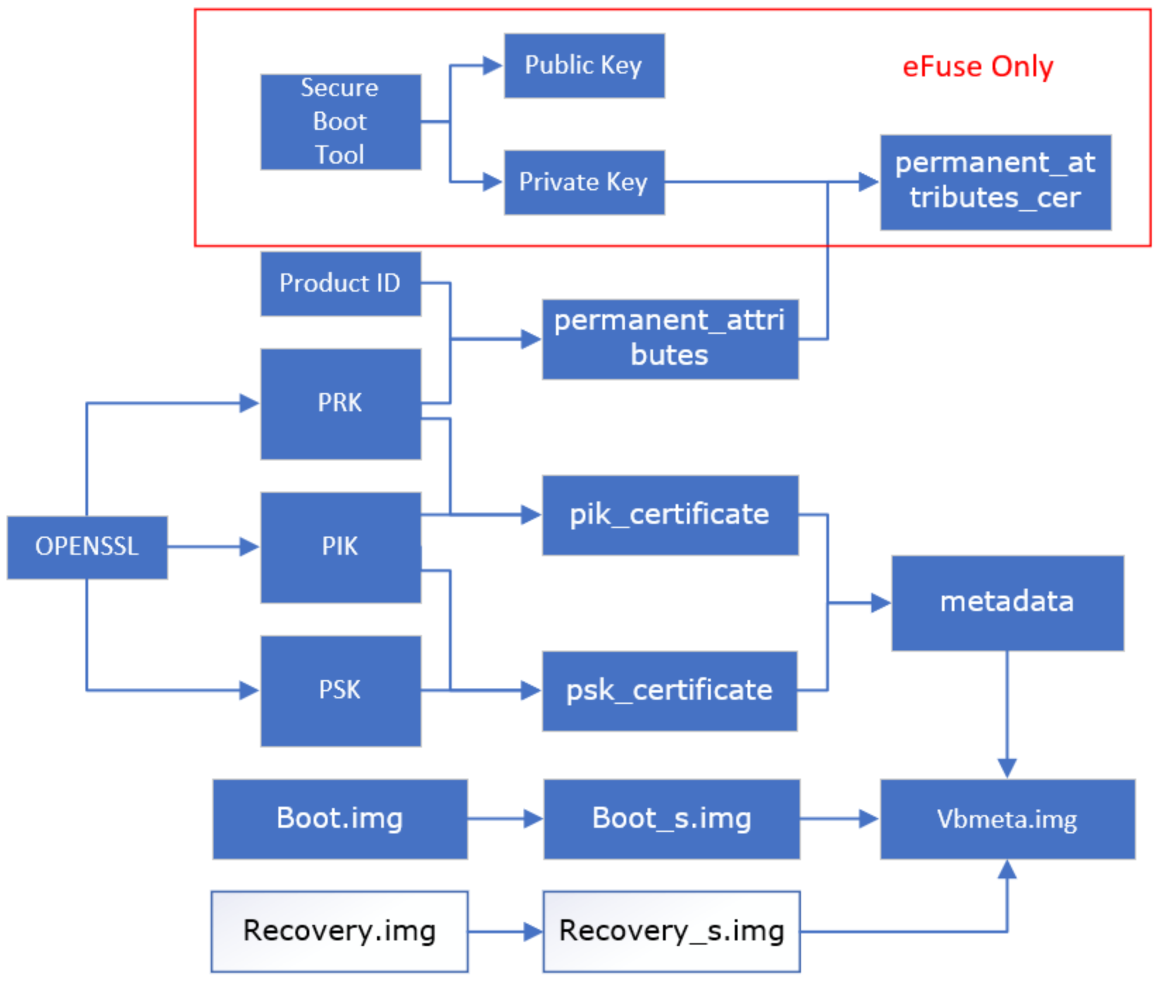
The main information of AVB contains the following four Keys:

Product RootKey (PRK): root Key of AVB, in eFuse devices, related information is verified by Base Secure Boot Key. In OTP devices, PRK-Hash information pre-stored in OTP is directly read and verified;

ProductIntermediate Key (PIK): intermediate Key;

ProductSigning Key (PSK): used to sign a firmware;

ProductUnlock Key (PUK): used to unlock a device.



A series of files are generated based on these 4 Keys, as shown in the figure. Refer to the Google AVB open source documents for details. https://android.googlesource.com/platform/external/avb/+/master/README.md

Compared with the original AVB, Linux grabs of main firmware verification function of AVB. In order to adapt to RK platform, additional permanent_attributes_cer.bin is generated in eFuse, in other words, it unnecessary to store permanent_attributes.bin in eFuse, permanent_attributes.bin information is verified directly by Base Secure Boot Key to save eFuse space.

There is already a set of test certificates and key in this directory. If you need a new Key and certificate, you can generate it yourself by the following steps:

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

For eFuse devices, you also need to generate additional permanent_attributes_cer.bin (OTP devices can skip this step).

```
./avb_user_tool.sh -f < /Path/to/PrivateKey.pem >
```

When generating `permanent_attributes_cer.bin`, `.setting` file would be generated at same time, and marks the device as eFuse device in the file. If the target device is changed, do not forget to change device type here.

Please keep the generated files properly, otherwise you will not be able to unlock after locking, and the device will not be able to flashed.

3.1.4 Sign the firmware

If AVB is enabled, device only can boot with sign firmware.

AVB can verify the boot.img and recovery.img, sign them with this command:

```
./avb_user_tool.sh -s -b < /path/to/boot.img > -r < /path/to/recovery.img > #
Remove -r option if no recovery.img
```

The signed images and vbmeta.img generated in `out` directory.

There are 2 ways to signed firmware(update.img) with 2.1 Signature Tools:

- use signature tools to sign loader.bin / uboot.img / trust.img, pack all signed images to update.img
- pack signed boot.img / recovery.img / vbmeta.bin and other images to update.img, and sign the update.img. (Set exclude_boot_sign=True firstly)

> `security` partition do not need to pack to update.img, it would initialized by U-Boot.

3.1.5 Download firmware

Download tool is upgrade_tool in Linux， RKDevTool in Windows.

You can directly download update.img or flash it one by one according to the partition.

If you try to flash it one by one, some images should be replaced:

- boot.img and recovery.img used `out` directory generated above.
- vbmeta should be downloaded.
- MiniloaderAll.bin / uboot.img / trust.img should use signed images.
- parameter.txt should be fixed with 3.1.2.3 Parameter

You should add vbmeta for RKDevTool, blank the address, and reload parameter, RKDevTool will update the address by itself.

After downloading firmware, the device is unlock state, it can not stop booting invalid image.

3.1.6 AVB Lock & Unlock

AVB supports Lock and Unlock states:

- Lock: verify the image next stage to U-Boot like boot.img and recovery.img, and **stop booting** invalid image, report error if image is invalid
- Unlock: verify the image next stage to U-Boot like boot.img and recovery.img, report error if image is invalid, **but still boot**.

Therefore, setting the device to unlock state is convenient for debugging.

All of AVB user operations use fastboot. There are some ways to enter device fastboot mode:

- Press "fastboot" key at booting if there is "fastboot" button in board.
- Run `reboot fastboot` in system
- Run `fastboot usb 0` in U-Boot console. (set CONFIG_BOOTDELAY in U-Boot, it supports specific delay to wait Ctrl-C to enter U-Boot console)

Anyway, device must communicate to PC with fastboot.

Store user password for fastboot which run with supper user, it helps us run fastboot without input user password manually.

```
./avb_user_tool.sh --su_pswd < /user/password >
```

Download AVB root information (must be done before "Lock" and "Unlock"):

```
./avb_user_tool.sh -d
```

Lock the device:

```
./avb_user_tool.sh -l # reboot device after finishing lock.
```

Unlock the device:

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

If everything goes well, the log below will be shown if the device is LOCKED.

```
ANDROID: reboot reason: "(none)"
Could not find security partition
read_is_device_unlocked() ops returned that device is LOCKED
```

3.2 FIT

FIT (flattened image tree) is a new type boot images scheme supported by u-boot, which can pack and verify any number of images. FIT use its (image source file) file to describe image, and finally forms an itb(flattened image tree BLOB) image with mkimage tool. its file use the syntax rules of DTS, it's flexible, and can directly run libfdt library and related tools. At the same time, it comes with a new set of safety inspection methods.

Main differences between FIT and AVB:

- FIT complation and signature are done by one command. AVB is compiled first and then signed.
- Only one Key Hash of FIT is stored in OTP/eFuse. AVB should store two.
- After FIT signature, the signature information is carried on the target images itself. AVB signature information is stored in vbmeta.bin。
- FIT has no trust.img, incorporated in uboot.img, AVB is not care about trust.img.

FIT supports below features：

- sha256 + rsa2048 + pkcs-v2.1(pss) padding
- Firmware rollback prevention
- Firmware re-signature(Remote signature)
- Crypto hardware acceleration

For now, security verify mode only supports sha256+rsa2048+pkcs-v2.1(pss) padding。

For more details， see Rockchip_Developer_Guide_UBoot_Nextdev.pdf chapter 12。

3.2.1 Keys generation

FIT Keys and Base Secure Boot keys are the same set of keys, but it's generation and using integrated into compilation, and actually it calls rk_sign_tool to generate the key and sign the firmware as well.

Execute the following three commands in the u-boot project to generate the RSA keys. Usually, it only needs to be generated once. After that, the keys are used to sign and verify the firmware. Please keep it properly.

```
# 1. key stored directory: keys
mkdir -p keys

# 2. use rk_sign_tool to generate RSA privateKey.pem and publicKey.pem, and
rename them: keys/dev.key and keys/dev.pubkey
../rkbin/tools/rk_sign_tool kk --bits 2048 --out .

# 3. use -x509 and private key to generate signature certificate: keys/dev.crt
openssl req -batch -new -x509 -key keys/dev.key -out keys/dev.crt
```

> If console report error below:
>
> 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
>
> Please create it manually first: touch ~/.rnd

```
# ls keys/
dev.crt  dev.key  dev.pubkey
```

> Note: The names "keys", "dev.key", "dev.crt" ,"dev.pubkey" are fixed. These names has defined in its file. If name was changed, it would fail to pack images.

3.2.2 Configuration

These features should be enabled in U-Boot:

```
# required
CONFIG_FIT_SIGNATURE=y
CONFIG_SPL_FIT_SIGNATURE=y

# option
CONFIG_FIT_ROLLBACK_PROTECT=y       # boot.img rollback prevention
CONFIG_SPL_FIT_ROLLBACK_PROTECT=y   # uboot.img rollback prevention
```

> It is recommended to select configuration by `make menuconfig`, and `make savedefconfig` to update original defconfig file. In this way, you can avoid some errors due to wrong dependency relationship.

3.2.3 U-Boot compilation and signature

U-Boot would sign boot.img / recovery.img / loader.bin / uboot.img at the end of compilation. So we should determine the location of boot.img and recovery.img before compilation.

（1）Basic command（No rollback prevention）：

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

result：

```
# ......
# After compilation, generated signed uboot.img, boot.img and 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）Extension command 1：

Enabled rollback prevention, add rollback parameter to basic command：

```
// Specify the minimum version number of uboot.img, boot.img and recovery.img is
10 and 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
```

result：

```
......

// After compilation, generated signed uboot.img, boot.img and recovery.img, and
enabled rollback prevention.
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）Extension command 2：

Burn public key hash to OTP/eFuse， add `--burn-key-hash` to (1) or (2)：

```
# Specify the minimum version number of uboot.img, boot.img and recovery.img is
10 and 12
# Public key is burned to OTP/eFUSE at SPL stage.
./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
```

> System would stop booting if new version number is less that old when enabled rollback prevention.

result：

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

# After compilation, generated signed uboot.img, boot.img and recovery.img, and
enabled rollback prevention.
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
```

> SPL print `RSA: Write key hash successfull` at booting

（4）**Note:**

- `--boot_img`: optional. Specify target boot.img to be signed.
- `--recovery_img`: optional. Specify target recovery.img to be signed.
- `--rollback-index-uboot`, `--rollback-index-boot`, `--rollback-index-recovery`: optional. Specify rollback version number.
- `--spl-new`: If the compilation command does not take this parameter, the SPL file in rkbin is used to package and generate `loader` by default; Otherwise, package the loader with the currently compiled SPL file.

Because security u-boot-spl.dtb compilation requires RSA public key (generated by user), the official SDK does not submit SPL file with security feature to `rkbin` repository. However, users can also submit their own security SPL to the `rkbin` repository. After that, when compiling the firmware, they can no longer specify this parameter and use the stable SPL file.

Three firmware will be generated after compilation: loader, uboot.img, boot.img. As long as the RSA key is not replaced, it is allowed to update any firmware separately.

3.2.4 Boot Log

```
BW=32 Col=10 Bk=8 CS0 Row=15 CS=1 Die BW=16 Size=1024MB
out
U-Boot SPL board init
U-Boot SPL 2017.09-gacb99c5-200408-dirty #cjh (Apr 09 2020 - 20:51:21)
unrecognized JEDEC id bytes: 00, 00, 00

Trying to boot from MMC1
// SPL signature verification is done.
sha256,rsa2048:dev+
// rollback prevention: now uboot.img version number is 10, device minimum
version number is 9. Rollback index: 10 >= 9, OK
// SPL checked images hash.
### 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


U-Boot 2017.09-gacb99c5-200408-dirty #cjh (Apr 09 2020 - 20:51:21 +0800)

Model: Rockchip RV1126 Evaluation Board
PreSerial: 2
DRAM:  1023.5 MiB
Sysmem: init
Relocation Offset: 00000000, fdt: 3df404e0
Using default environment

dwmmc@ffc50000: 0
Bootdev(atags): mmc 0
MMC0: HS200, 200Mhz
PartType: EFI
boot mode: normal
conf: sha256,rsa2048:dev+
resource: sha256+
DTB: rk-kernel.dtb
FIT: signed, conf required
HASH(c): OK

I2c0 speed: 400000Hz
PMIC:  RK8090 (on=0x10, off=0x00)
vdd_logic 800000 uV
vdd_arm 800000 uV
vdd_npu init 800000 uV
```

```
vdd_vepu init 800000 uV
......

Hit key to stop autoboot('CTRL+C'):  0
### Booting FIT Image at 0x3d8122c0 with size 0x0052b200
Fdt Ramdisk skip relocation
### Loading kernel from FIT Image at 3d8122c0 ...
   Using 'conf' configuration
   // uboot signature verification is done
   Verifying Hash Integrity ... sha256,rsa2048:dev+ OK
   // rollback prevention: now boot.img version number is 22, device minimum
version number is 21. Rollback index: 22 >= 21, OK
   Verifying Rollback-index ... 22 >= 21, OK
   Trying 'kernel' kernel subimage
     Description:  Kernel for arm
     Type:         Kernel Image
     Compression:  uncompressed
     Data Start:   0x3d8234c0
     Data Size:    5349248 Bytes = 5.1 MiB
     Architecture: ARM
     OS:           Linux
     Load Address: 0x02008000
     Entry Point:  0x02008000
     Hash algo:    sha256
     Hash value:
64b4a0333f7862967be052a67ee3858884fcefebf4565db5c3828a941a15f34a
   Verifying Hash Integrity ... sha256+ OK  // checked kernel hash
### Loading ramdisk from FIT Image at 3d8122c0 ...
   Using 'conf' configuration
   Trying 'ramdisk' ramdisk subimage
     Description:  Ramdisk for arm
     Type:         RAMDisk Image
     Compression:  uncompressed
     Data Start:   0x3dd3d4c0
     Data Size:    0 Bytes = 0 Bytes
     Architecture: ARM
     OS:           Linux
     Load Address: 0x0a200000
     Entry Point:  unavailable
     Hash algo:    sha256
     Hash value:
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
   Verifying Hash Integrity ... sha256+ OK  // checked ramdisk hash
   Loading ramdisk from 0x3dd3d4c0 to 0x0a200000
### Loading fdt from FIT Image at 3d8122c0 ...
   Using 'conf' configuration
   Trying 'fdt' fdt subimage
     Description:  Device tree blob for arm
     Type:         Flat Device Tree
     Compression:  uncompressed
     Data Start:   0x3d812ec0
     Data Size:    66974 Bytes = 65.4 KiB
     Architecture: ARM
     Load Address: 0x08300000
     Hash algo:    sha256
     Hash value:
8fb1f170766270ed4f37cce4b082a51614cb346c223f96ddfe3526fafc5729d7
   Verifying Hash Integrity ... sha256+ OK // checked fdt hash
```

```
    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 ...

[    0.000000] Booting Linux on physical CPU 0xf00
[    0.000000] Linux version 4.19.111 (cjh@ubuntu) (gcc version 6.3.1 20170404
(Linaro GCC 6.3-2017.05)) #28 SMP PREEMPT Wed Mar 25 16:03:27 CST 2020
[    0.000000] CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=10c5387d
```

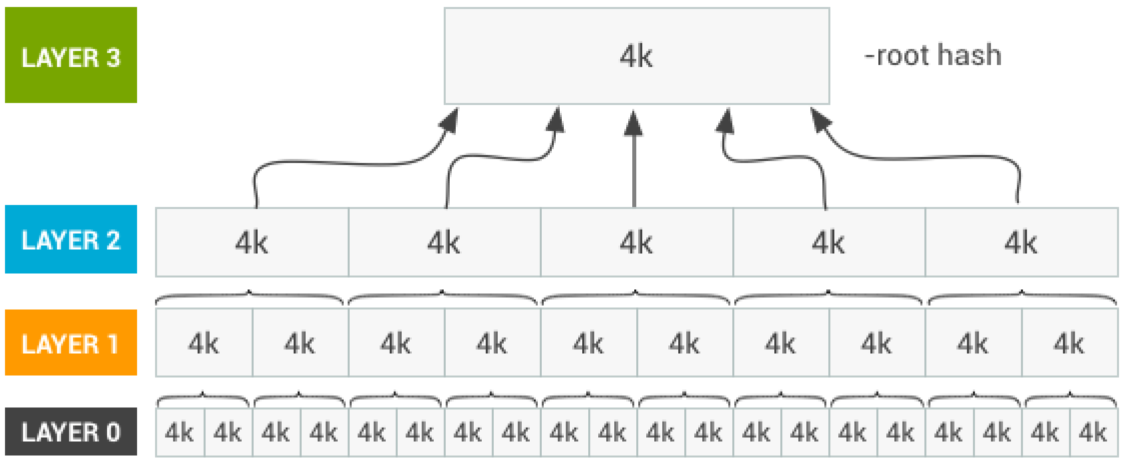
4. DM-V

One precondition of using DM-V is that boot.img must be secure. This solution will package a ramdisk in boot.img. The security of ramdisk is guaranteed by AVB. The veritysetup tool is used to verify mounted firmware in ramdisk.

The advantage of DM-V is that verification speed is fast, the larger the firmware, the more obvious the effect.

The disadvantage is that DM-V can only work in read-only file systems, Boot and System firmware will become larger.

The basic theory is Device-Mapper-Verity technology, which will do 4K segmentation on a verification firmware and hash calculation for each 4K data slice, iterate multiple layers, and generate corresponding Hash-Map (within 30M) and Root- Hash. When creating a virtual partition based on DM-V, Hash-Map is verified to ensure that the Hash-Map is correct.


LAYER 3
4k
-root hash
LAYER 2
4k 4k 4k 4k 4k
LAYER 1
4k 4k 4k 4k 4k 4k 4k 4k 4k 4k
LAYER 0
4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k 4k

After the partition is mounted, when there is data accessing, it is going to do hash verification of the 4K partition where the data is located. When verification errors occur, I/O errors are returned, and the corresponding location cannot be used, which is similar to file system corruption.

Please refer to Documentation/device-mapper/ under Kernel for details.

Or refer to https://source.android.google.cn/security/verifiedboot/dm-verit.

4.1 Sign Firmware

DM-V function can be used when kernel opens related resources.

Pay attention to open CONFIG_DM_VERITY when Compiling Kernel.

For related tools, refer to the network disk file (Section 1.3 Reference Resources)
Linux_SecurityDM_v1_01.tar.gz

The above compressed files should be decompressed in a Linux environment. It contains soft links and will be expanded to original file size, causing the file to become larger under windows.

Decompress the file to get:

```
|── config
|── mkbootimg
|── mkdm.sh
└── ramdisk.tar.gz
```

First configure the config file, please fill in the corresponding information according to actual situation.

```
ROOT_DEV=   #the partition location of actual root firmware in flash, such as
/dev/mmcblk2p8
INIT=       #the first script that actual root runs, generally is /init or
/sbin/init
ROOTFS_PATH=    #rootfs firmware that needs to be signed
KERNEL_PATH=        #Kernel Image location, generally is
kernel/arch/arm(64)/boot/Image
RESOURCE_PATH=  #kernel resource.img location, generally is kernel/resource.img
```

Then run `./mkdm.sh -m dmv -c config (--debug)`, the script will automatically package Root-Hash into Ramdisk, and package with kernel, resource to boot.img. Hash-Map is attached to rootfs.img.

Obtain boot.img and rootfs_dmv.img from output directory.

Download the two firmware to the board instead of the original boot.img and rootfs.img.

4.1.1 Version upgrade

In order to make Ramdisk more compatible with chips, Ramdisk is compiled instead of using the fixed Ramdisk for new version.

```
$COMMON_DIR/mk-ramdisk.sh ramboot.img $RK_CFG_RAMBOOT
```

- $COMMON_DIR is `<SDK>/device/rockchip/common/`
- $RK_CFG_RAMBOOT is SDK predefined compilation configuration. Only supported by RK3588 now.
- System should be built first, some information about system will be packed to Ramdisk through `mk-ramdisk.sh`.
- This command is also used for encryption, encryption or verification depends on parameter `RK_SYSTEM_CHECK_METHOD` in `<SDK>/device/rockchip/.BoardConfig.mk`, DM-V for verification, DM-E for encryption.

5. Partition Encryption

Partition encryption is also based on device-mapper technology, except each partition block treatment. Refer to Chapter 4 DM-V

Advantages: high security, free file system, readable and writable.

Disadvantages: when encrypting partitions, they cannot be compressed; reading and writing data must be calculated by encryption and decryption, which affects the efficiency of reading and writing to some extent.

5.1 rootfs Encryption

Like DM-V, partition encryption also requires Kernel to open related resources:

```
CONFIG_BLK_DEV_DM
CONFIG_DM_CRYPT
CONFIG_BLK_DEV_CRYPTOLOOP
```

Share a set of tools with DM-V (Linux_SecurityDM_v1_01.tar.gz)

Need to configure the following items in the config file:

```
ROOT_DEV=   #the location of actual root firmware in flash, such as
/dev/mmcblk2p8
INIT=    #the first script that the actual root runs, generally is /init or
/sbin/init
KERNEL_PATH=     #Kernel Image location, generally is
kernel/arch/arm(64)/boot/Image
RESOURCE_PATH=  #Kernel resource.img location, generally is kernel/resource.img
inputimg=   #firmware that needs to be encrypted
cipher= #aes-cbc-plain by default
key= #note the format size, the key should match with cipher
```

Use `./mkdm.sh -m fde-s -c config`

to generate boot.img and encrypted.img in output directory.

Download these two firmware to the board instead of the original boot.img and rootfs.img.

This part is just a demo, the key of system stored in `init`, this is weak security. Bacause customer's key is private, it's better to design code by customer to store the keys. If There is no scheme of key processing, you can refer to Keybox

5.1.1 Version upgrade

See 4.1.1 Version upgrade

5.2 Non System Firmware Encryption

There are many open source tools for firmware encryption and decryption. We are talking about dmsetup (consistent with tools in chapter 5.1) here. To use this tool, you need to open the following configurations:

```
# Kernel:
CONFIG_BLK_DEV_DM
CONFIG_DM_CRYPT
CONFIG_BLK_DEV_CRYPTOLOOP

# Buildroot:
BR2_PACKAGE_LUKSMETA
```

Share a set of tools with DM-V(Linux_SecurityDM_v1_01.tar.gz)

Need to configure the following items in the config file

```
inputimg=   #firmware that need to be encrypted, or use inputfile to encrypt folders
cipher= # aes-cbc-plain by default
key= # note the format size, the key should match with cipher
```

Using `./mkdm.sh -m fde -c config` to generate encrypted.img and encrypted_info in the output directory.

Encrypted.img is the encrypted file, which can be mounted and virtualized to a partition device by dmsetup. The encrypted_info here is encrypted information, such as:

```
#dmsetup create encfs-4284779680572201071 --table "0 550912 crypt aes-cbc-plain 000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f 0 TARGET_PARTITION 0 1 allow_discards"
sectors=550912
cipher=aes-cbc-plain
key=000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
```

The method to mount encrypted files to /mnt:

```
source encrypted_info
loopdevice=`losetup -f`
losetup ${loopdevice} encrypted.img
dmsetup create encrypt-file --table "0 $sectors crypt $cipher $key 0 $loopdevice 0 1 allow_discards"
mount /dev/mapper/encrypt-file /mnt
```

Umount method:

```
umount /mnt
dmsetup remove encrypt-file
losetup -d ${loopdevice}
```

6. Keybox

When using partition or system encryption, a key storage location is required. Generally, the key is stored in RPMB or security partition through OPTEE. In principle, this part needs to be designed by the manufacturer, since the code design is in direct contact with the manufacturer's private key. Considering that some customers don't know about OPTEE, a Keybox demo code is added to the SDK and actually used in [7. Security demo](#).

(It's recommended to refactor the code by customer)

OPTEE related codes， in `<SDK>/external/security/`， include `bin` and `rk_tee_user`:

Bin stores part of the pre compiled non-public code and header files and is the basic library for optee operation.

- `bin` is pre-built private code and head files, and is the basic library for OPTEE.
- `rk_tee_user` is a project to compile the customer's own CA/ TA, which comes with some demo test projects.

This chapter mainly introduces how to use OPTEE in `buildroot`. For other systems, you can also refer to the compilation rules in buildroot for compilation. More information about OPTEE, refers to **`Rockchip_Developer_Guide_TEE_SDK_CN.pdf`** .

`extra_app` is a weak subproject of `rk_tee_user`, if this compilation directory exists, it will be compiled followed `rk_tee_user`, otherwise skip the compilation. Therefore, there is a demo of keybox stored in `buildroot/packege/rockchip/tee-user-app/extra_app`, when `buildroot` building, the code of keybox will be added to `rk_tee_user`, and then compile `rk_tee_user` to get customer's `keybox_app` and TA program.

The relevant codes of builderoot are located in `<SDK>/builderoot/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
```

> `keybox_ App` can write key in any cases, but can only read key in ramdisk and PID = 1. Ramdisk is packed in boot.img, and it verified by FIT, that make sure `keybox_app` read key in safely. This requires customers to use this code with FIT or AVB.

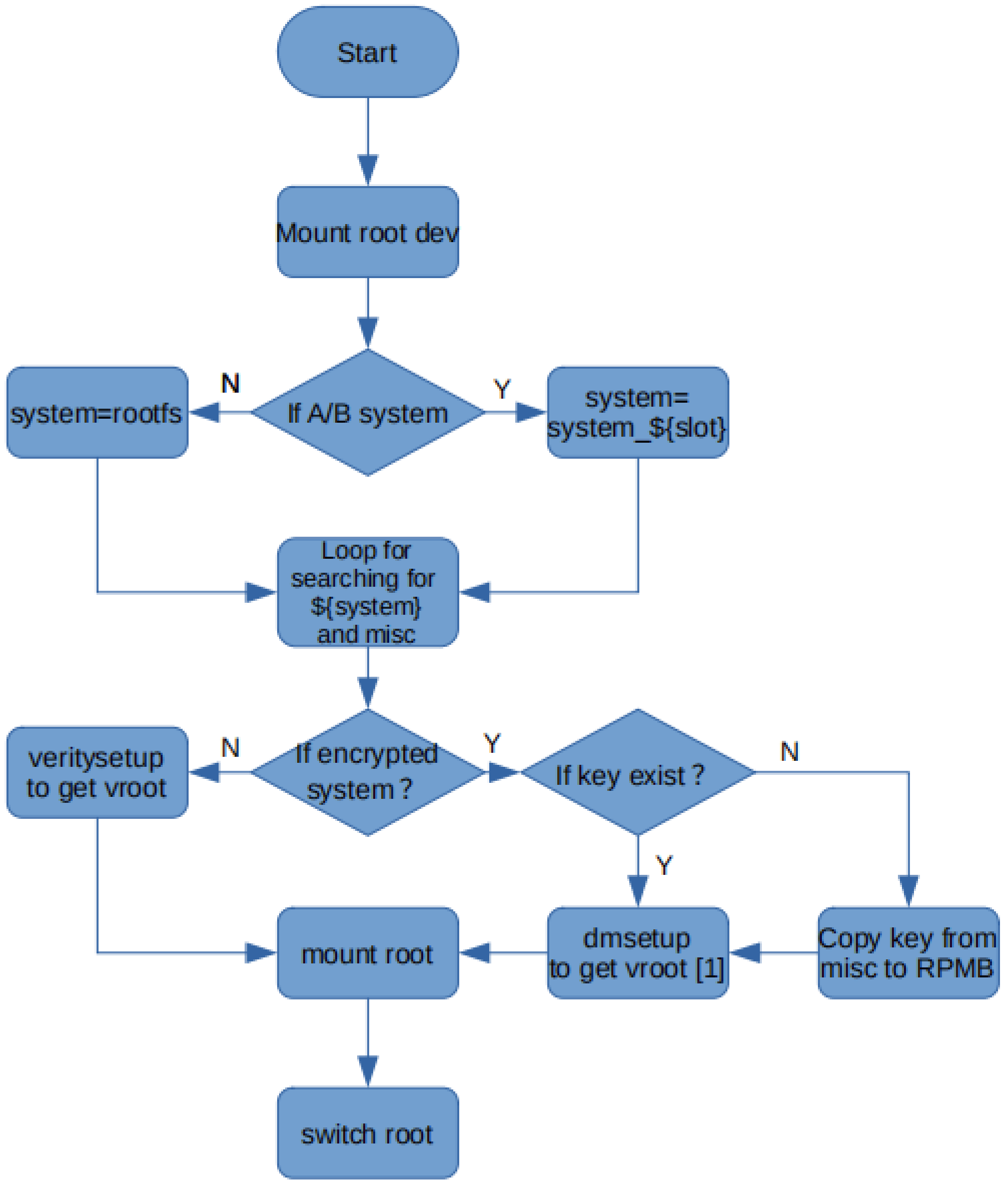
7. Security Demo

The Linux SDK has released a set of scripts to fast build a security firmware through several simple configurations.

(At present, this building environment is only applicable to FIT. For the support list, refer to [Product Version](#))

7.1 Ramdisk startup process

This demo uses FIT + Ramdisk, and ramdisk guides the system to startup. The most critical part of ramdisk is its `init` script. In ramdisk, the system is decrypted or verified according to the type of system packaging, and unified into `vroot` device. The ramdisk takes into account the A/B system, system verification and system encryption.



> [1] In the encryption system, the file system will be expanded according to the actual size of the system partition.

7.2 System configuration

If SDK supports Security Demo, BoardConfig-security-base.mk and BoardConfig-ab-base.mk can be found in `<SDK>/device/rockchip/.target_product`

```
# cat BoardConfig-security-base.mk
export RK_LOADER_UPDATE_SPL=true
# ramboot config
export RK_CFG_RAMBOOT=rockchip_rk3588_ramboot
# Set ramboot image type
```

```
export RK_RAMBOOT_TYPE=CPIO
# Define enable RK SECUREBOOT
export RK_RAMDISK_SECURITY_BOOTUP=true
export RK_SECURITY_OTP_DEBUG=true                    # Default enabled, easier to
debug, do not write public key hash to OTP.
export RK_SYSTEM_CHECK_METHOD=DM-V                   # DM-V for system verified,
DM-E for system encrypt.

# cat BoardConfig-ab-base.mk
export RK_PACKAGE_FILE_AB=rk3588-package-file-ab
export RK_MISC=blank-misc.img
export RK_PARAMETER=parameter-ab.txt
export RK_CFG_RECOVERY=                              # There is no recovery.img in
A/B.
```

> **RK_SECURITY_OTP_DEBUG is enabled by default, OTP key does not write public key hash to OTP, in other word, loader.bin is safe in default. Please turn off this option when mass producting**.

Enable Security Demo, `BoardConfig-security-base.mk` should be included to `<SDK>/device/rockchip/.BoardConfig.mk`, like:

```
diff --git a/rk3588/BoardConfig-rk3588-evb1-lp4-v10.mk b/rk3588/BoardConfig-
rk3588-evb1-lp4-v10.mk
index 5fa455d..753bb6c 100644
--- a/rk3588/BoardConfig-rk3588-evb1-lp4-v10.mk
+++ b/rk3588/BoardConfig-rk3588-evb1-lp4-v10.mk
@@ -59,3 +59,5 @@ export RK_DISTRO_MODULE=
 export RK_BOARD_PRE_BUILD_SCRIPT=app-build.sh
 # Define package-file
 export RK_PACKAGE_FILE=rk3588-package-file
+REALDIR=`dirname $(readlink -f ${BASH_SOURCE[0]})`
+source ${REALDIR}/BoardConfig-security-base.mk
```

> Append `BoardConfig-ab-base.mk` if enabled A/B system.

Now, run `./build.sh` to build security firmware. In building, some errors may appear, according to the prompts, fix the errors. Or continue to follow this document below and modify the configuration.

7.3 Detailed configuration

- Generate FIT Key

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

- Save user password to U-Boot if using system encryption.

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

  > Some operation of building encrypted system need root permission, which means that the system cannot be compiled in the server environment (Normally, common server user can not have root permission).

- Modify Kernel configuration and add the following configuration:

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

CONFIG_TEE=y                        # Required in encrypted system.
CONFIG_OPTEE=y                      # Required in encrypted system.
```

- If using encrypted system, modify Kernel dts and add the following node:

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

- Modify U-Boot configuration and add the following configuration:

```
CONFIG_FIT_SIGNATURE=y
CONFIG_SPL_FIT_SIGNATURE=y
CONFIG_ANDROID_AB=y         # Just for A/B system.
```

- Modify ROOTFS configuration and add the following configuration:

```
BR2_ROOTFS_OVERLAY="board/rockchip/common/security-system-overlay"
BR2_PACKAGE_RECOVERY=y                          # Required in encrypted system.
BR2_PACKAGE_RECOVERY_UPDATEENGINEBIN=y          # Required in encrypted system.
BR2_PACKAGE_RECOVERY_BOOTCONTROL=y              # Required in encrypted system,
and depends on two configurations above.
```

If everything goes well, the system can be started and in protection. System can't started if any image was replaced.

7.4 Debugging method

If Ramdisk `init` can't switch to system, try to enable debug log, and use `DEBUG` function to add private log. Determine it's operation following [7.1 Ramdisk startup process]

```
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
 }
```