

Rockchip Secure Boot Application Note

Version:1.9

Email:zyf@rock-chips.com hjh@rock-chips.com

Date:2018.06

Classification level:Publicity

Preface

Terms

Sector

Sector size is 512 bytes

eFuse

One-Time Programmable Memory IP in SOC

Introduction

This document describes how to implement Rockchip secure boot solution.

Secure boot mechanism is for verifying firmware validity, which aims to prevent invalid firmware upgrade and booting.

The device which had programmed eFuse will enable secure boot ROM, and could not boot from the un-signed firmware. So trying to upgrade un-signed firmware or unmatched key signed firmware will fail.

NOTE: The valid signed firmware can boot smoothly on fake copies of device circuit board or same CPU platform hardware. Secure boot will verify the validity of software, but not hardware.

This document applies to RK3126, RK3128, RK3228, RK3229, RK3288, RK3368, RK3399, RK3228H, RK3328 and RK3326, RK3308 and PX30.

Features of secure boot:

- Support secure boot ROM
- Support SHA256
- Support RSA2048
- Support eFuse hash to verify public key

The relative tool and loader revision:

- MiniloaderV2.19 or the latest revision
- Uboot V2.17or the latest revision
- Efuse tool V1.35 or the latest revision

- SecureBootTool 1.79 or the latest revision
- RKBatchTool 1.8 or the latest revision(deprecated, Use FactoryTool instead)
- FactoryTool 1.39 or the latest revision

History

Revision	Date	Description	Author
1.0	2014-11-05	Original document	ZYF
1.1	2015-12-21	Update secure boot tool	YBC
1.2	2016-02-02	Update secure boot tool	YHC
1.3	2016-09-29	Re-edit	ZYF
1.4	2016-11-15	Add detailed description of workflow	Joshua
1.5	2016-11-16	1. Add terms and definitions.2. Add eFuse layout.	Joshua
1.6	2017-02-15	Add RK3328 and RK3228H.	ZYF
1.7	2017-05-19	Add sequence chart and note	ZZJ
1.8	2017-10-30	Refactor the format and add hardware info	CW
1.9	2018-06-05	Add OTP program public key hash flow	CF

Contents

Rockchip Secure Boot Application Note

Preface

1 Architecture

1.1 Secure Boot Process

1.2 Secure Boot Sequence

1.3 MaskRom Boot to the First Loader (RKminiLoader/U-Boot)

1.4 First Loader boot to u-boot(Secondary Boot Loader,option)

1.5 U-Boot Boot to Boot Image with Linux kernel)

1.6 U-Boot Boot to Recovery

2 eFuse Layout

3 Overall Operation Flow

3.1 Make Update.img

3.2 Firmware Sign

3.3 Programming eFuse

3.4 Firmware Upgrade

3.5 Ensure that Secure Boot has been Enabled

4 Make Update.img

4.1 Generate Images

4.2 Make Update.img

5 Firmware Sign Flow

5.1 Generating RSA key

5.2 Save RSA key

- 5.3 Loading RSA key
- 5.4 Configuration
- 5.5 Sign Firmware
- 6 Programming eFuse
 - 6.1 Hardware Conditions
 - 6.1.1 eFuse Programming
 - 6.1.2 OTP Programming
 - 6.2 Tool UI
 - 6.3 Load the Signed Firmware
 - 6.4 Click 'run' Button to Start
 - 6.5 Programming eFuse
 - 6.6 Programming OTP
- 7 Firmware Upgrade
 - 7.1 Firmware Upgrade
- 8 Verification
 - 8.1 Check Secure Flag
 - 8.2 Secure Boot Test

1 Architecture

1.1 Secure Boot Process

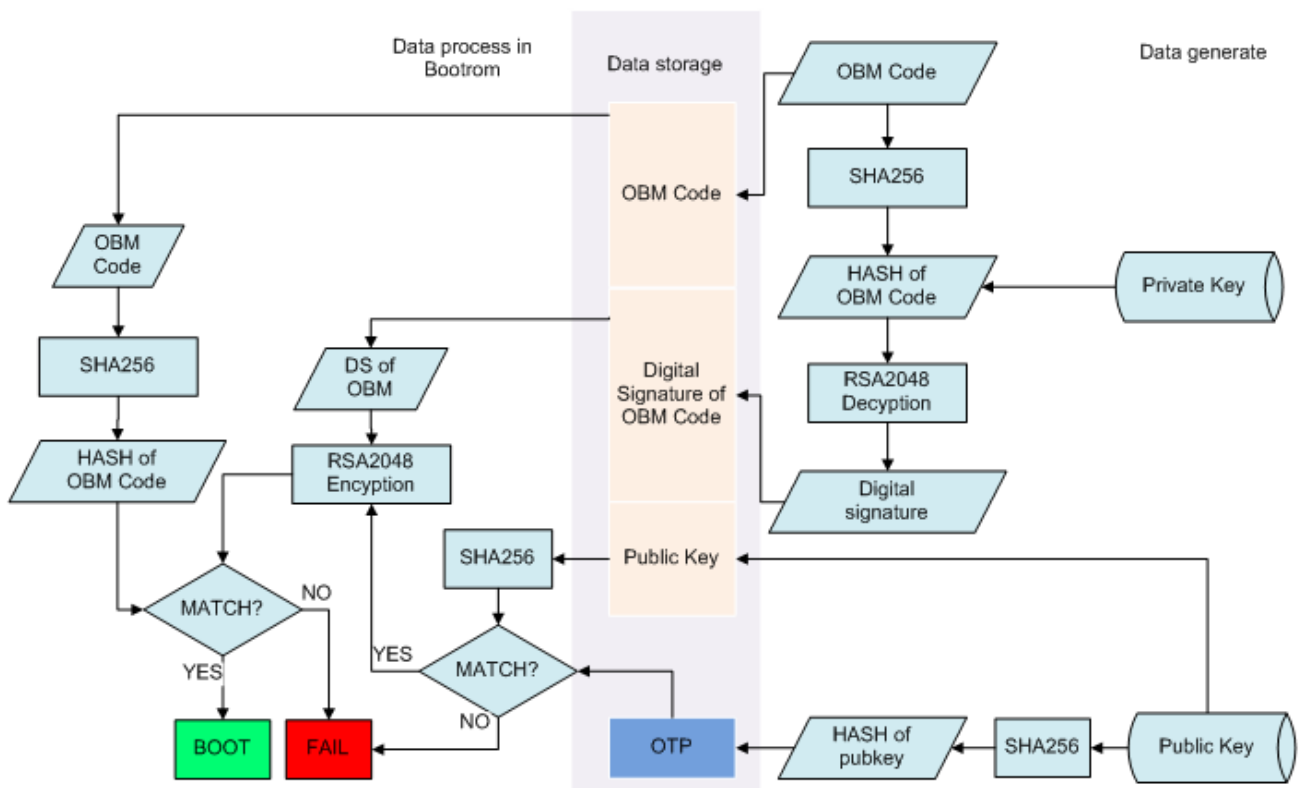


Figure 1-1 Secure boot process

1.2 Secure Boot Sequence

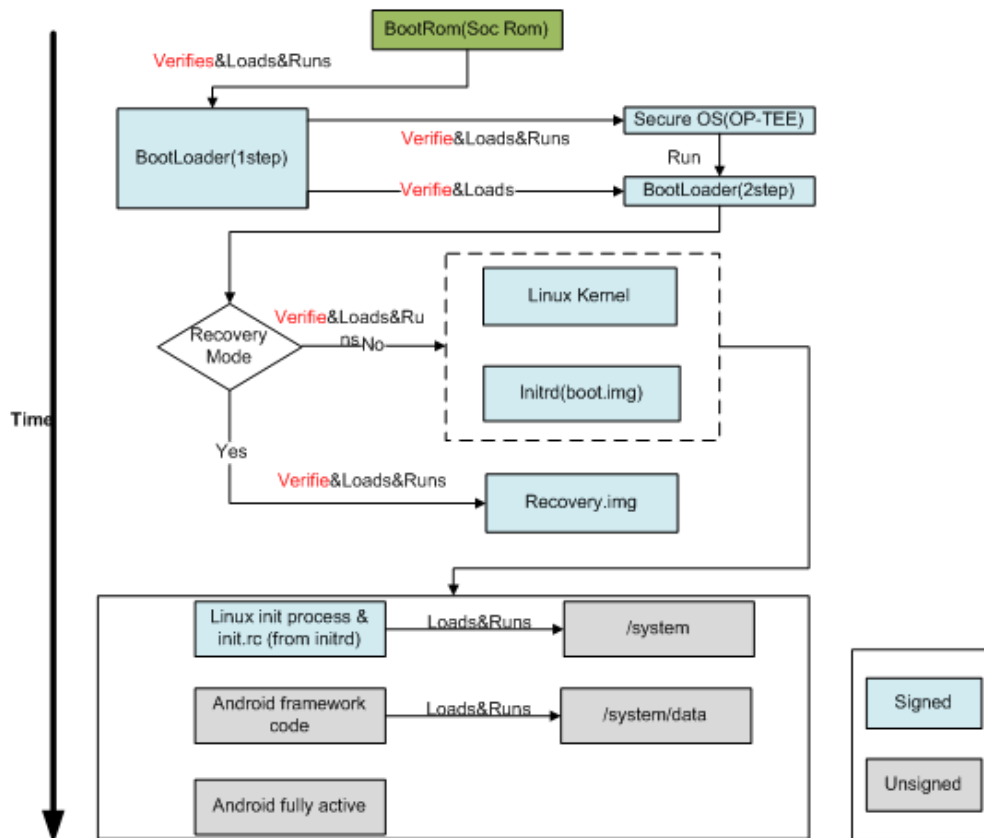


Figure 1-2 Secure boot sequence

1.3 MaskRom Boot to the First Loader (RKminiLoader/U-Boot)

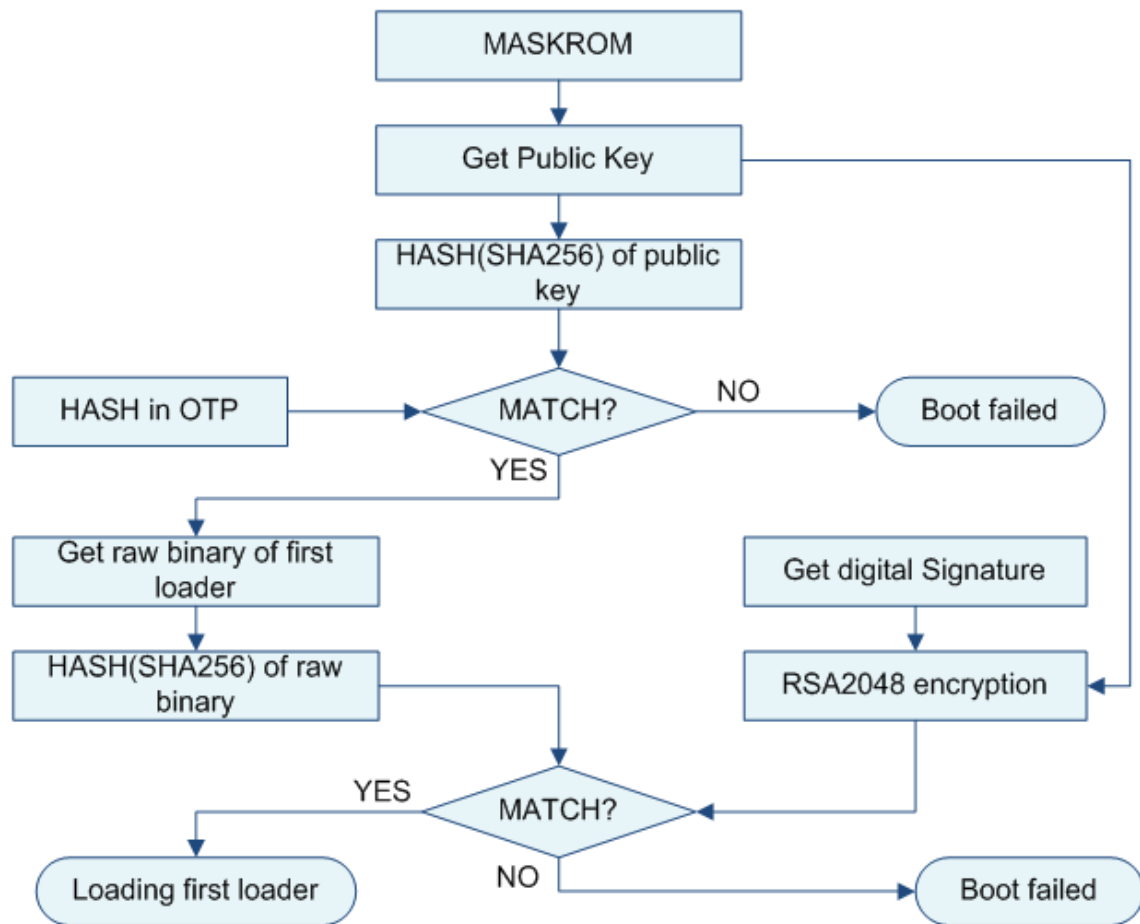


Figure 1-3-1 MaskRom to loader sequence

First loader layout in user partition of flash

Table 1-1 First loader data layout

0-63 sector	64 sector reverse
first loader(8128 sector)(5 copys)	Boot loader partition
0-2047	loader header
2048-4095	public key and digital signature
4096 -	raw binary
...	
Boot loader copy(4) partition	
0-2047	loader header
2048-4095	public key and digital signature
4096 -	raw binary

The structure of public key and digital signature layout at address 2048 to 4095:

```
typedef struct tagBOOT_HEADER↓
{↓
    uint32 tag;↓
    uint32 version;↓
    uint32 flags;↓
    uint32 size;↓
    uint32 reserved1[3];↓
    uint16 HashBits;↓
    uint16 RSABits;        /* length in bits of modulus */↓
    uint32 RSA_N[64];      /* RSA public key */↓
    uint32 RSA_E[64];↓
    uint32 RSA_C[64];↓
    uint32 HashData[(8+1)*2]; //loader hash↓
    uint32 signature[64];  /* digital signature */↓
}BOOT_HEADER, *PBOOT_HEADER;↓
```

Figure 1-3-2 first_loader_secure_header struct

Public key: uint32 RSA_N[64], RSA_E[64], RSA_C[64] ;

Digital signature: uint32 signature[64]

Step1: Get public key from first loader partition.

Step2: Calculate the hash(SHA256) of public key and compare it with the the hash stored in OTP.If mathed,load the first loader successfully, otherwise booting failed.

Step3: Calculate the hash(SHA256) of raw binary and compare it with RSA2048 decryption(have been obtaine in step1) of digital signature. If matched, load first loader successfully, otherwise booting failed.

1.4 First Loader boot to u-boot(Secondary Boot Loader,option)

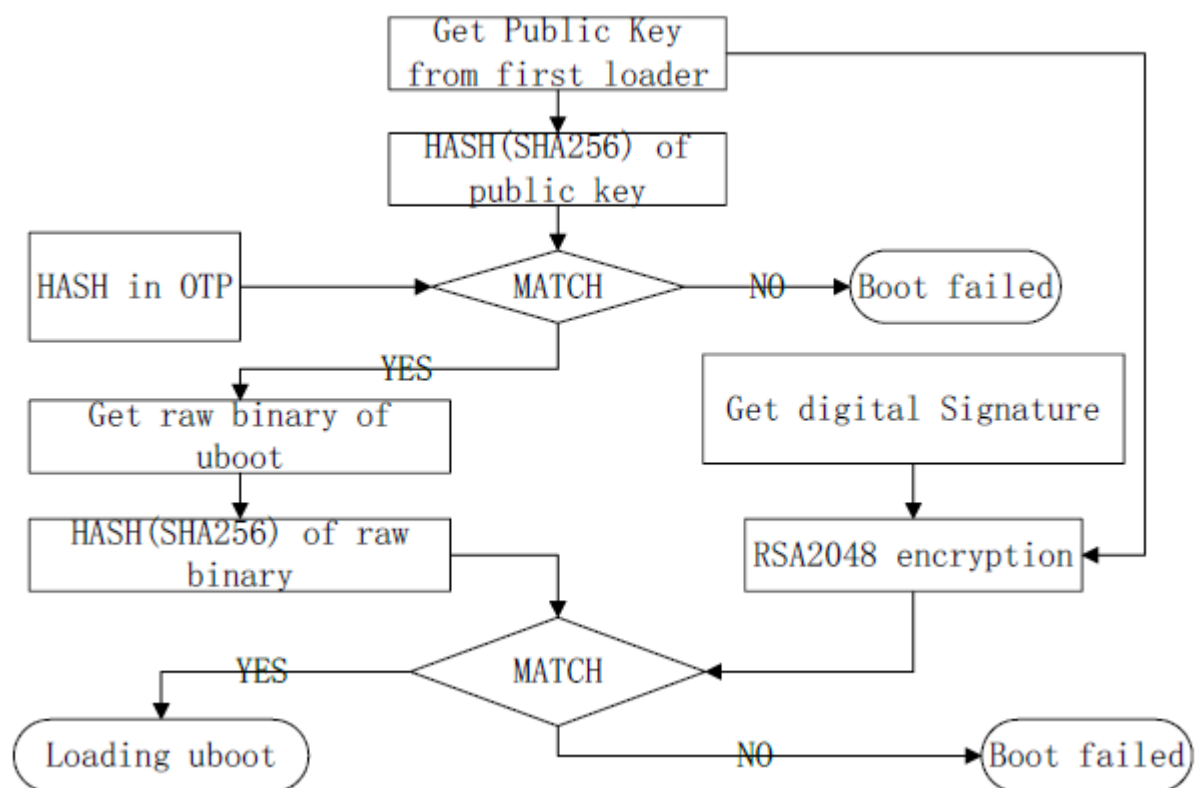


Figure 1-4-1 boot to -uboot flow

uboot (4MB, 4cops)	UBoot	
	0-2047	header, digital signature
	2048-	Raw binary
	...	
	UBoot copy(3)	
	0-2047	header, digital signature
	2048-	Raw binary

Table 1-4 u-boot layout in flash

The structure of header with digital digital signature layout at address 0 to 2047:

```

typedef struct tag_second_loader_hdr
{
    unsigned char magic[LOADER_MAGIC_SIZE]; // "LOADER "
    unsigned int loader_load_addr; /* physical load addr ,default is 0x60000000*/
    unsigned int loader_load_size; /* size in bytes */
    unsigned int crc32; /* crc32 */
    unsigned int hash_len; /* 20 or 32 , 0 is no hash */
    unsigned char hash[LOADER_HASH_SIZE]; /* sha */
    unsigned char reserved[1024-32-32];
    uint32 signTag; //0x4E474953, 'N' 'G' 'T' 'S'
    uint32 signlen; //256
    unsigned char rsaHash[256]; /* digital signature */
    unsigned char reserved2[2048-1024-256-8];
}second_loader_hdr; //Size:2K

```

Figure 1-4-2 Second_loader_hdr struct

Digital signature: unsigned char rsaHash[256];

I Step 1: Get public key from first loader partition

I Step 2: Calculate the hash (sha256) of public key and compare it with hash in OTP, if matched go to next step, otherwise booting failed.

I Step 3: Calculate the hash(SHA256) of raw binary and compare it with RSA2048 encryption (have been obtained in step 1) of digital signature, if matched, loading successfully and deliver the public key to U-Boot, otherwise booting failed.

1.5 U-Boot Boot to Boot Image with Linux kernel)

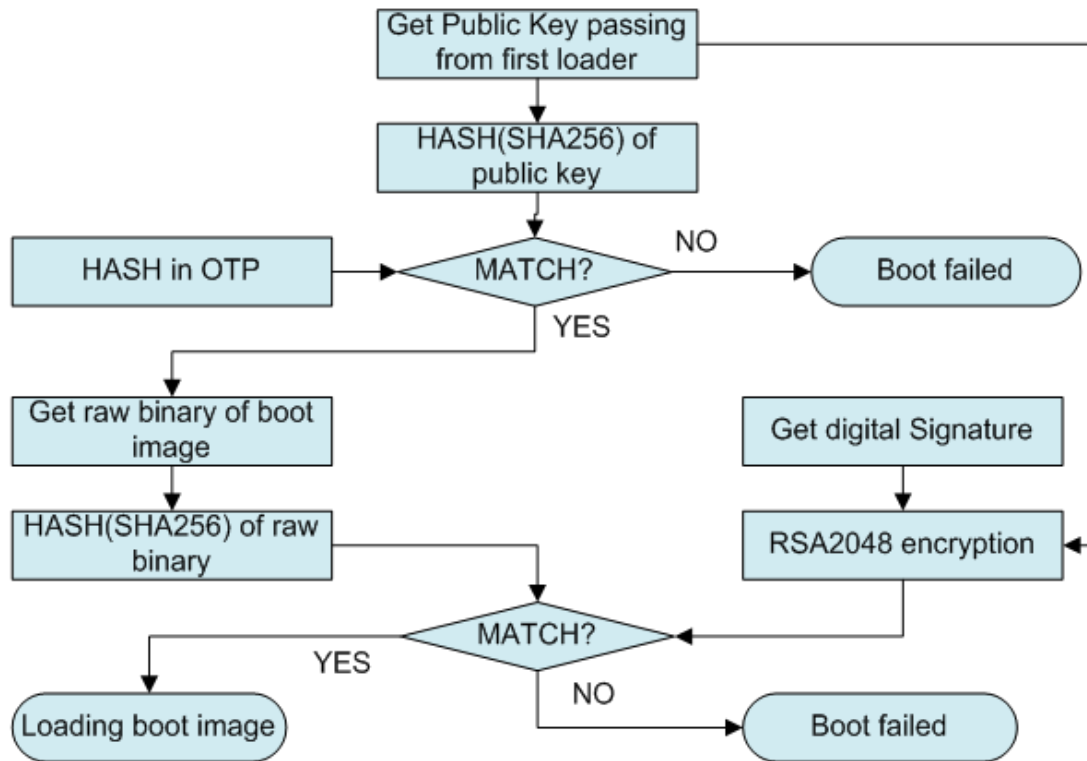


Figure 1-5 U-Boot to boot sequence

Table 1-2 Boot data layout

boot.img	0-2047	header
2048-4095	digital signature	
4096-	kernel,ramdisk,dtb...	
...		

The structure of layout 0-2047(header):

```

#define BOOT_MAGIC_SIZE 8
#define BOOT_NAME_SIZE 16
#define BOOT_ARGS_SIZE 512
typedef struct tag_boot_img_hdr{
    unsigned char magic[BOOT_MAGIC_SIZE]; /*"ANDROID!"*/
    unsigned int kernel_size; /* size in bytes */
    unsigned int kernel_addr; /* physical load addr */
    unsigned int ramdisk_size; /* size in bytes */
    unsigned int ramdisk_addr; /* physical load addr */
    unsigned int second_size; /* size in bytes */
    unsigned int second_addr; /* physical load addr */
    unsigned int tags_addr; /* physical addr for kernel tags */
    unsigned int page_size; /* flash page size we assume */
    unsigned int unused[2]; /* future expansion: should be 0 */
    unsigned char name[BOOT_NAME_SIZE]; /* asciiz product name */
    unsigned char cmdline[BOOT_ARGS_SIZE];
    unsigned int id[8]; /* timestamp / checksum / sha1 / etc */
    unsigned char reserved[0x400-0x260];
    uint32 signTag; //0x4E474953
    uint32 signlen; //128
    unsigned char rsaHash[128]; /* digital signature */
}boot_img_hdr;

```

Figure 1-6 Boot_img_hdr struct

Digital signature: unsigned char rsaHash[128];

I Step 1: U-Boot get public key obtained from first loader.

I Step 2: Calculate the hash (sha256) of public key and compare it with hash in OTP, if matched go to next step, otherwise booting failed.

I Step 3: Hash(SHA256) of raw binary and compare it with RSA2048 encryption(using public key get in step 1) of digital signature, if matched, boot to linux kernel, otherwise booting failed.

1.6 U-Boot Boot to Recovery

The same as boot to boot image, detail please refer to chapter1.4.

2 eFuse Layout

RK3368, RK3288, RK3229 and RK3228 used 1024 bits eFuse for secure boot, data layout:

Table 2-1 eFuse data layout

32-bit Word Addressing	Description
0x00	Security flagBits [7:0] security enable flag Bits [31:8] reserved
0x01-0x3	Reserved
0x04-0x07	Reserved
0x8-0xF	RSA public key hash
0x10-0x17	Reserved
0x18	Reserved
0x19-0x1A	Reserved
0x1B-0x1D	Reserved
0x1E	Reserved
0x1F	eFuse write lock bits

RK3228H and RK3328 used 7680 bits OTP for secure boot, data layout:

Table 2-2 OTP data layout

32-bit Word Addressing	Description
0-63	Public Key (N)
64-127	Public Key (E)
128	Security flagBits [7:0] 0xff: security enable flagBits [15:8] RSA_E size (word uint)Bits [31:16] Reserved
129	Trusted Firmware revocation counter (ID #0)
130-131	Non-trusted Firmware revocation counter (ID #1)
132-239	Reserved

3 Overall Operation Flow

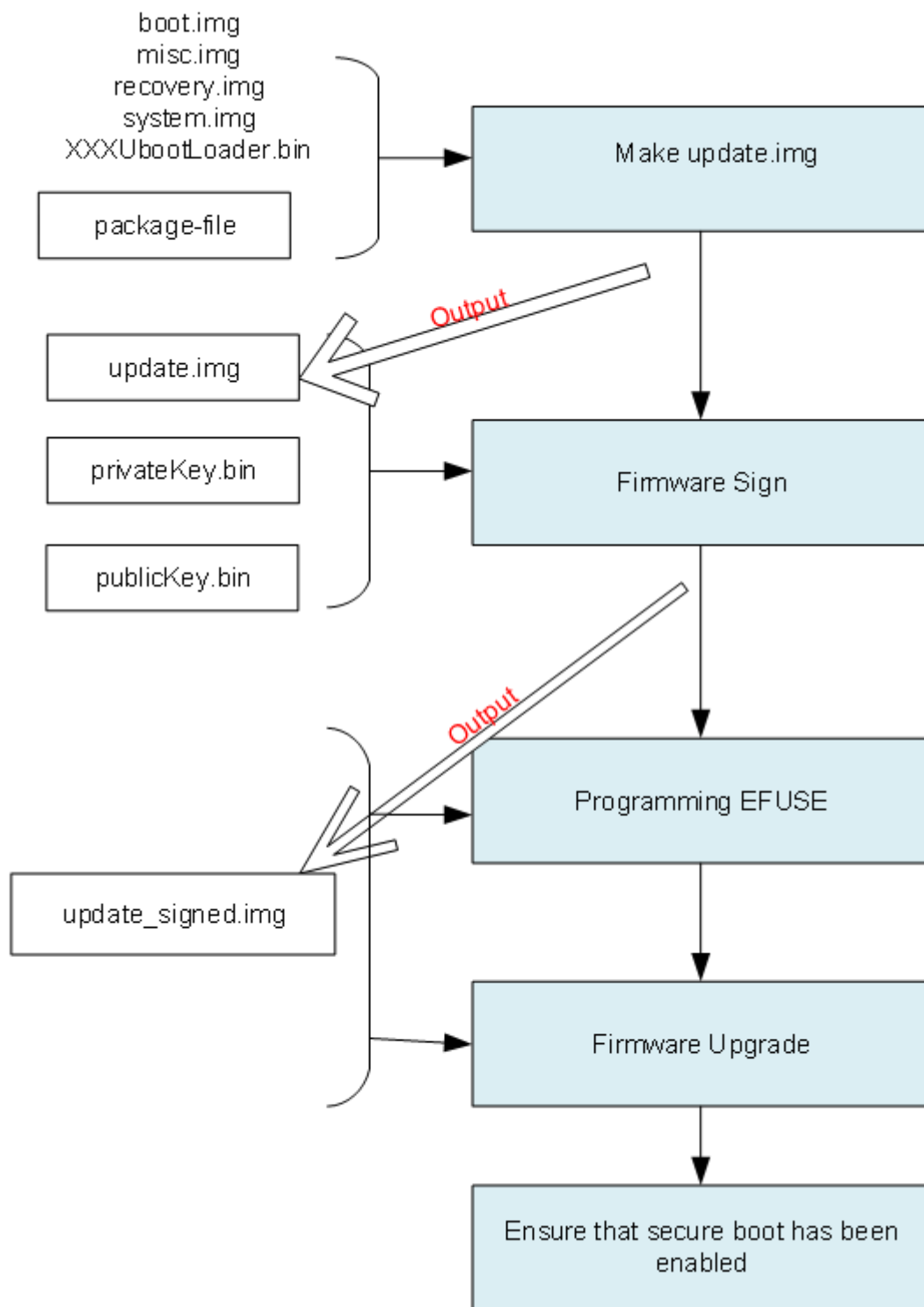


Figure 3-1 Secure boot operation process

3.1 Make Update.img

[See Make update.img](#)

3.2 Firmware Sign

[See Firmware Sign Flow](#)

3.3 Programming eFuse

[See Programming eFuse](#)

3.4 Firmware Upgrade

[See Firmware Upgrade](#)

3.5 Ensure that Secure Boot has been Enabled

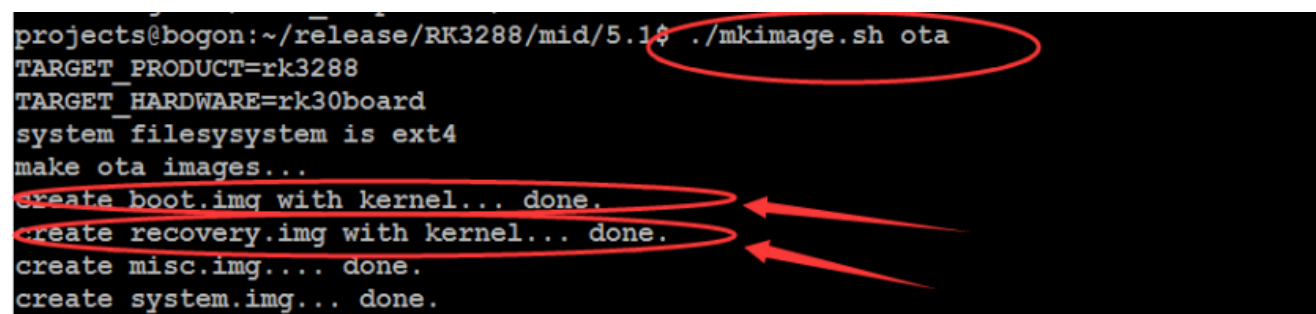
[See Verification](#)

4 Make Update.img

4.1 Generate Images

After build Android, use the following script to generate images:

`./mkimage.sh ota`



```
projects@bogon:~/release/RK3288/mid/5.1$ ./mkimage.sh ota
TARGET_PRODUCT=rk3288
TARGET_HARDWARE=rk30board
system filesystem is ext4
make ota images...
create boot.img with kernel... done.
create recovery.img with kernel... done.
create misc.img.... done.
create system.img... done.
```

Figure 4-1 Script to generate images

4.2 Make Update.img

Refer to RKTools/windows/AndroidTool/rockdev/package-file. This file controls which files will be packaged.

Take RK3288, for example. Change bootloader path, commentaries resource and kernel lines, set backup to RESERVED.

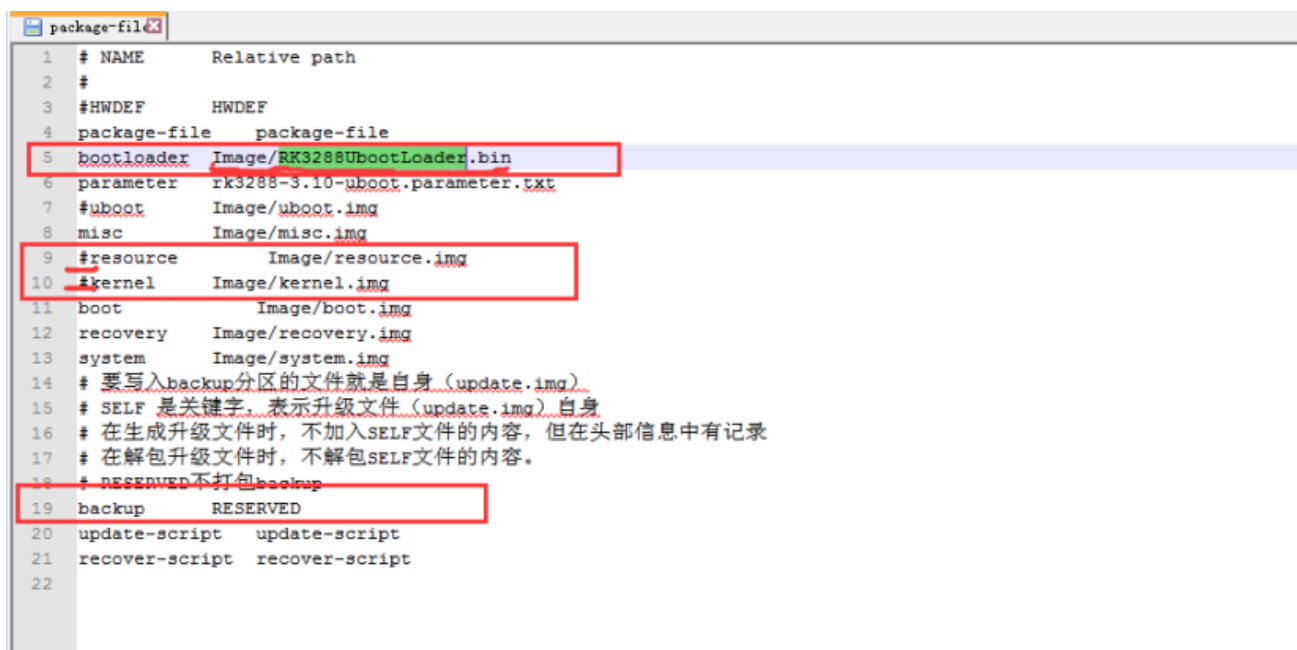


Figure 4-2 Package-file to control the packaging

Copy RKTools/windows folders to windows system, then run AndroidTool/rockdev/mkupdate.bat to generate the update.img.

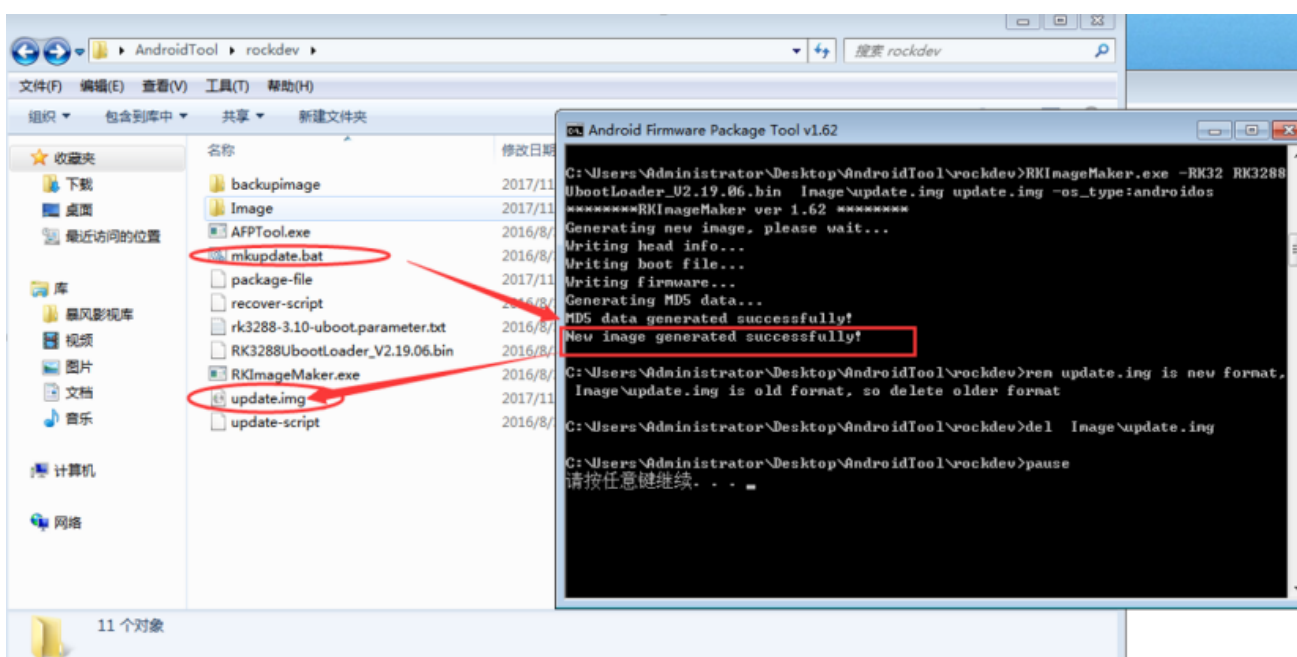


Figure 4-3 Script-to-generate-images

5 Firmware Sign Flow

This instruction is for Windows tools, while Linux has its own.

5.1 Generating RSA key

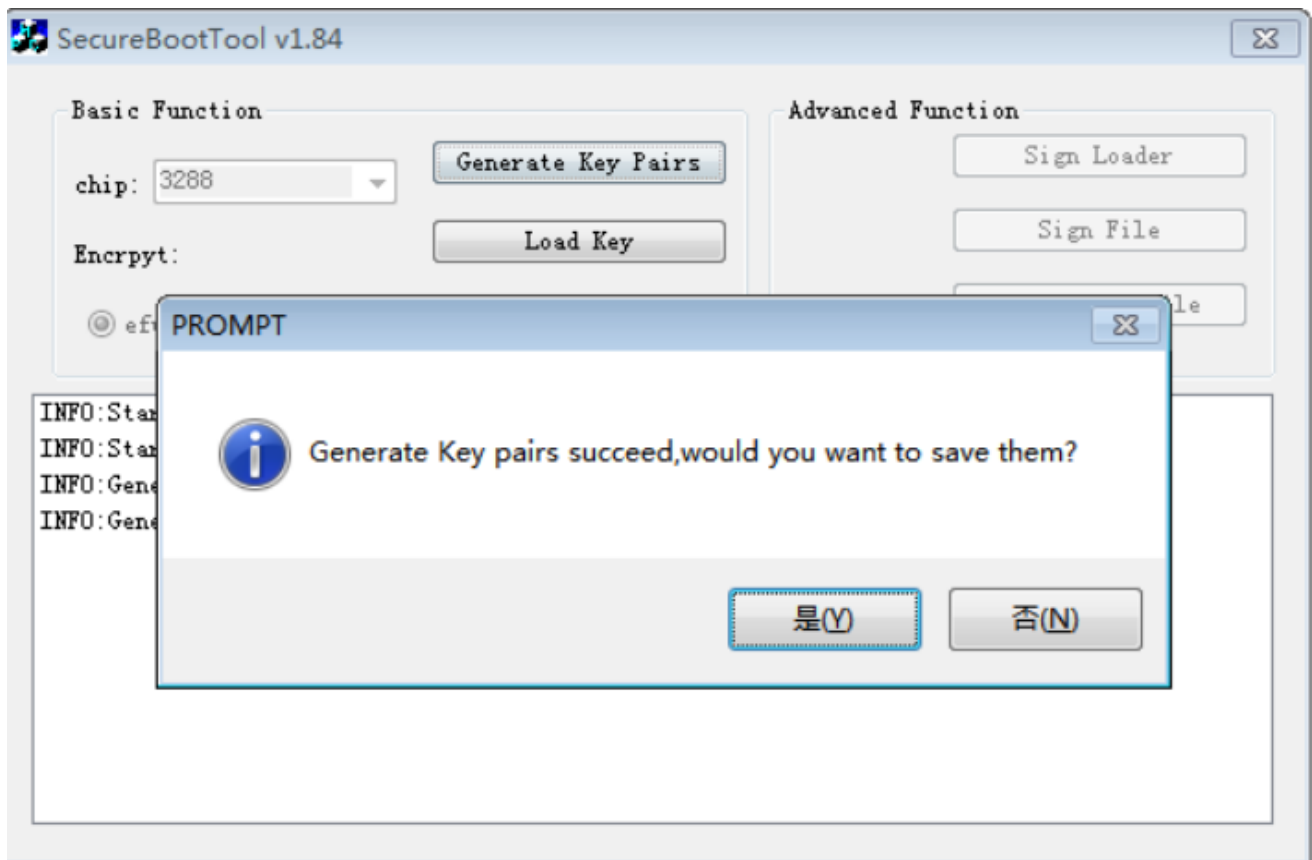


Figure 5-1 SecureBootTool generates RSA key

5.2 Save RSA key

This key will be used for signed firmware and for OTA, please back up to a secure storage.

NOTE: The keypair is VERY important! Make sure to save it securely. Once you lost it or leak it, your product will be exposed in high risk, also the old device will be unable to be updated anymore. It should be maintained through the whole product life cycle

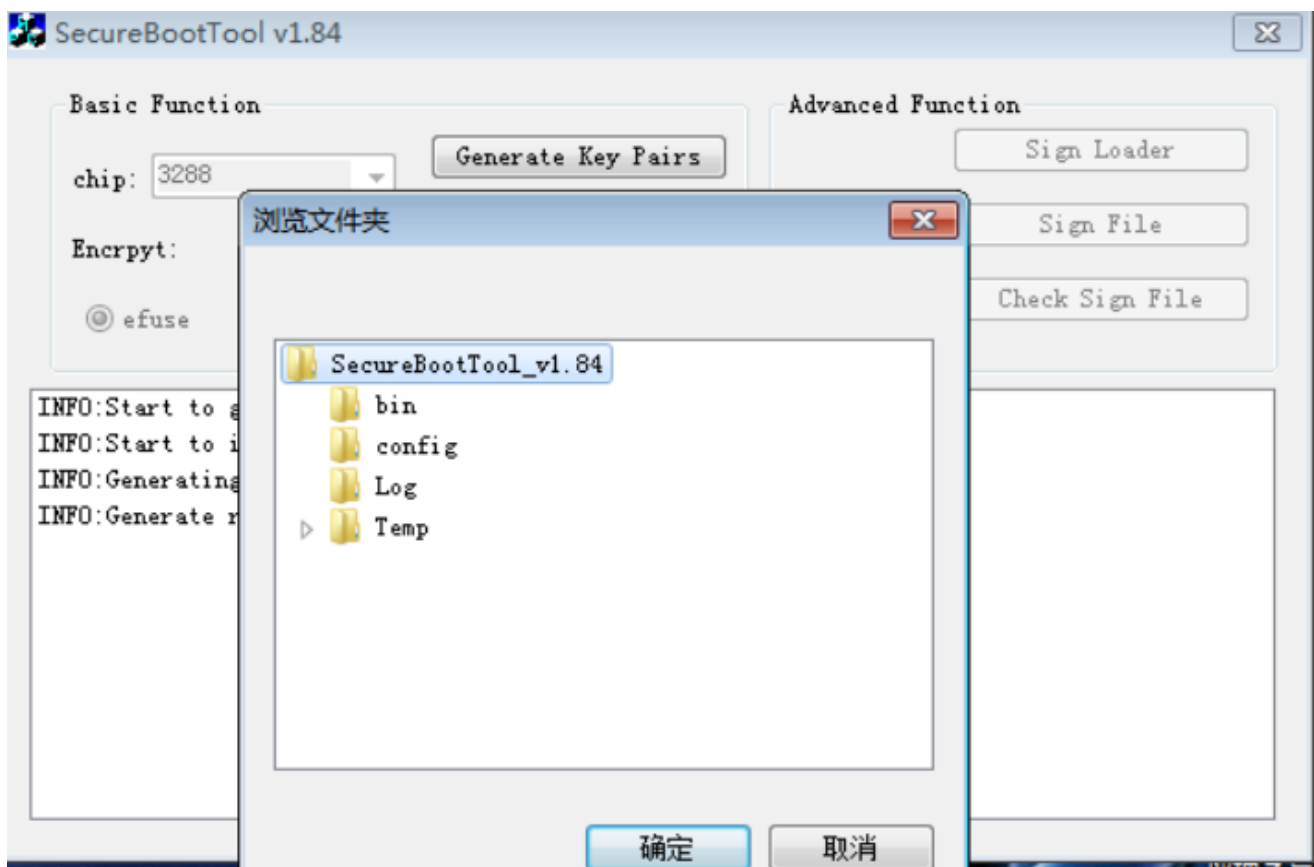


Figure 5-2 SecureBootTool saves RSA key

5.3 Loading RSA key

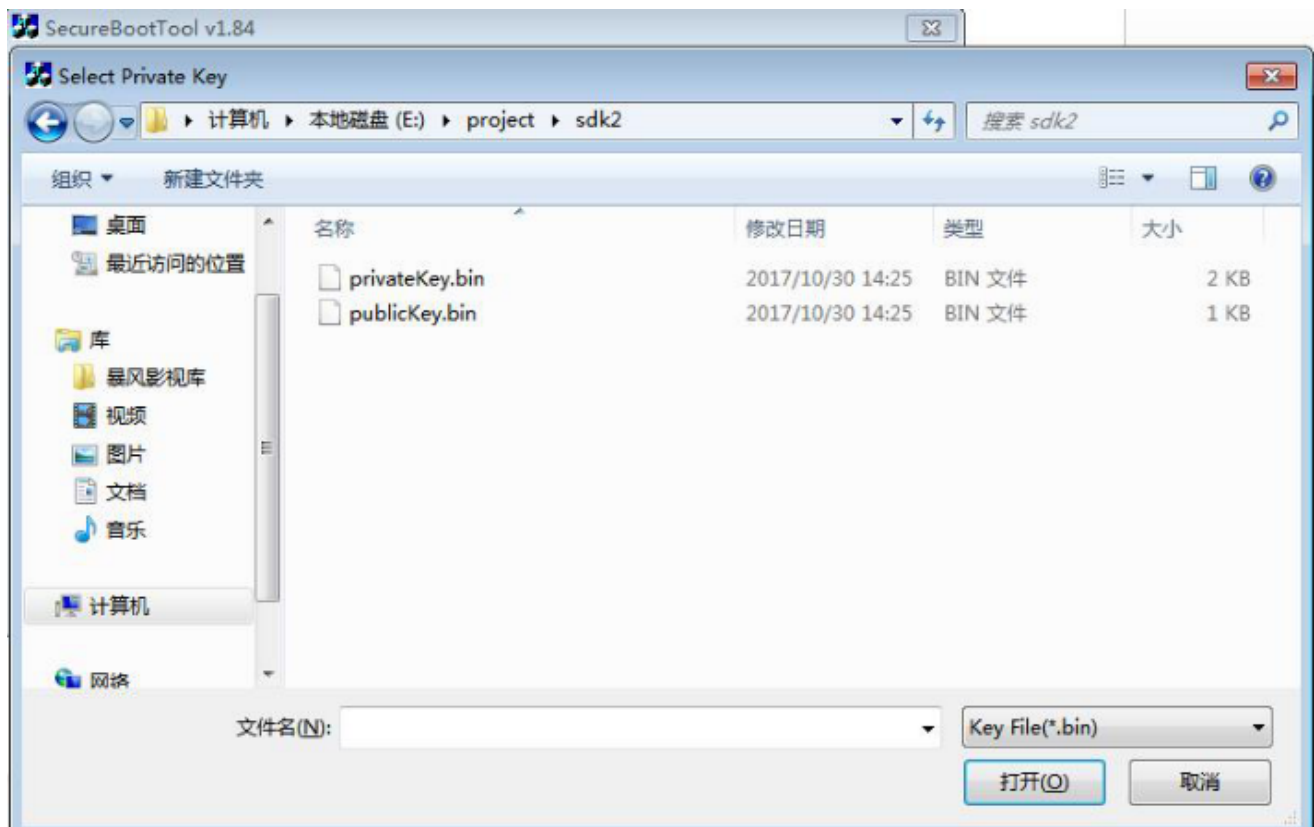
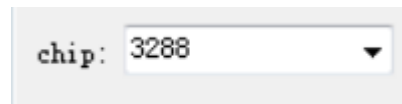



Figure 5-3 SecureBootTool loads RSA key

5.4 Configuration



chip: 3288 ▼

Choose SOC platform

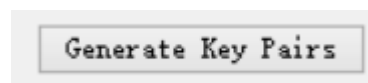


Encrypt:
☒ efuse ☐ soft

:

Option 'efuse' means using eFuse to store the hash of the RSA public key, and will enable secure boot ROM(recommended).

Option 'soft' is for some special applications, will not enable secure boot ROM, used RSA1024 and SHA160.



Generate Key Pairs

Every product model will generate RSA KEY only once, please backup in case that you cannot upgrade firmware or OTA again.



Load Key

Loading backup RSA key (support 'pem' file format generated by openssl)



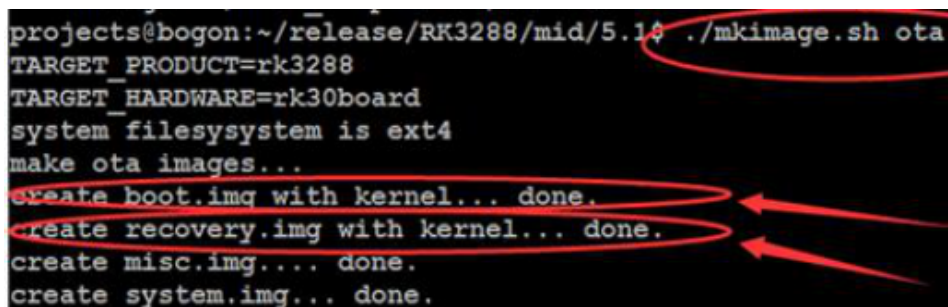
Sign Firmware

Sign firmware

5.5 Sign Firmware

Make sure the 'boot.img' and the 'recovery.img' are included in the kernel image.

Refer to the pack command:



```
projects@bogon:~/release/RK3288/mid/5.18$ ./mkimage.sh ota
TARGET_PRODUCT=rk3288
TARGET_HARDWARE=rk30board
system filesystem is ext4
make ota images...
create boot.img with kernel... done.
create recovery.img with kernel... done.
create misc.img... done.
create system.img... done.
```

Figure 5-4 Images'pack command

Open firmware image:

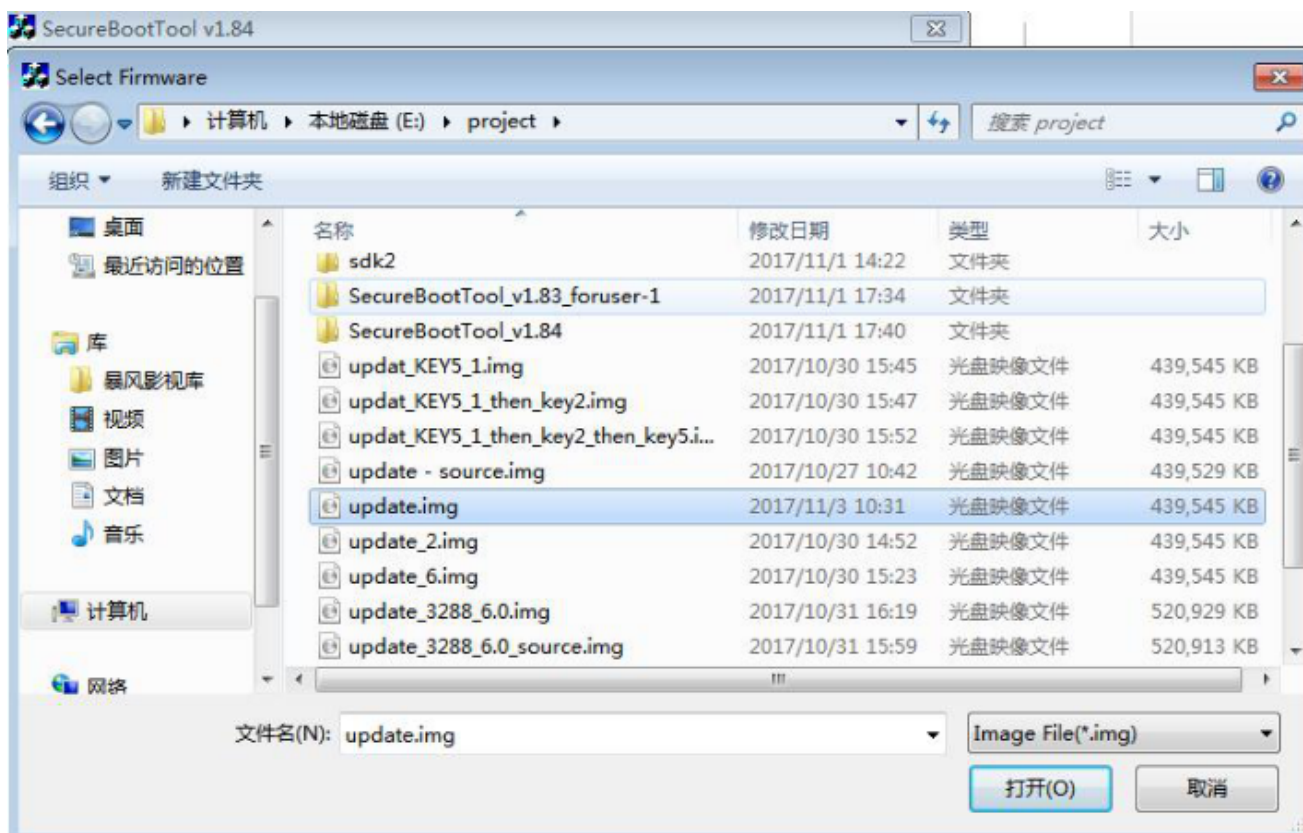


Figure 5-5 SecureBootTool selects firmware

Signed firmware:

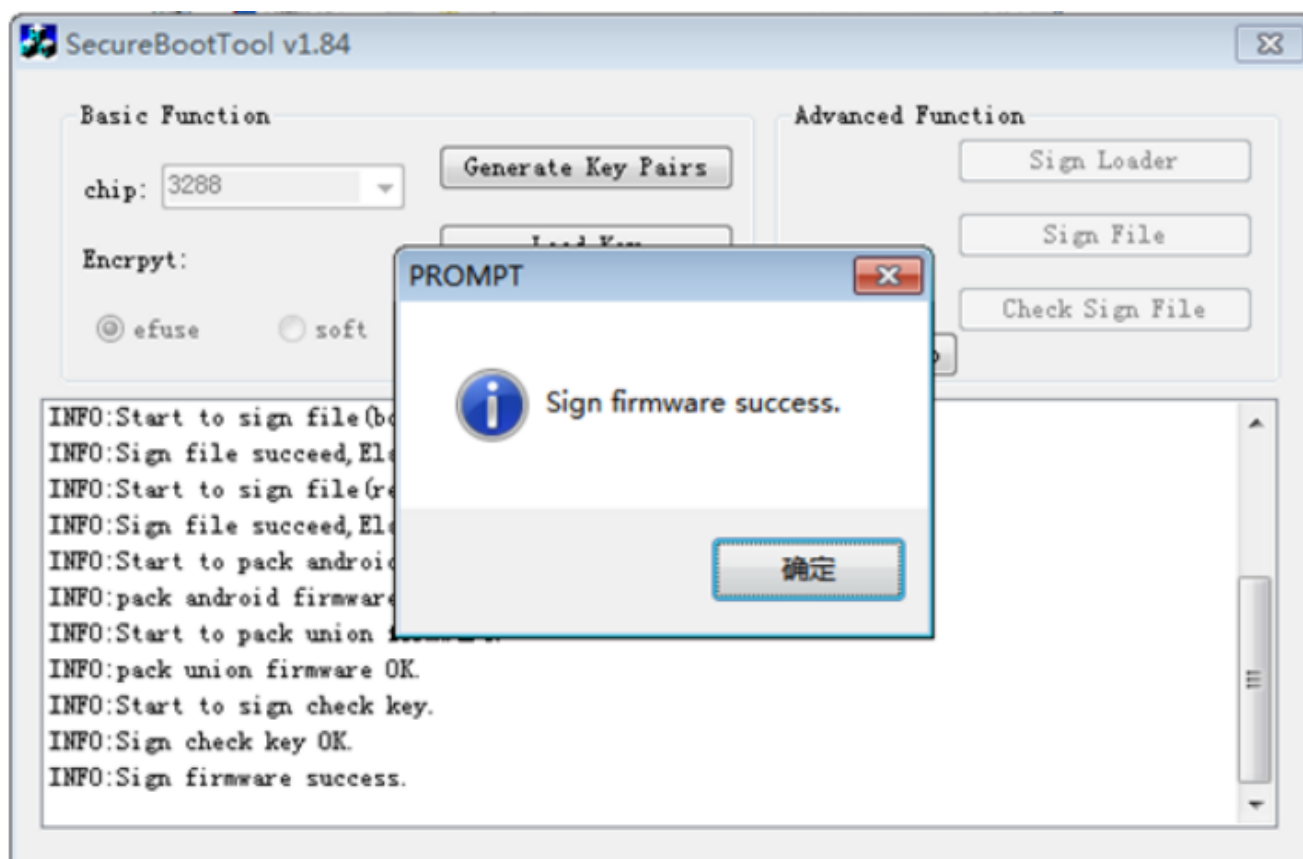


Figure 5-6 Secure Boot Tool-signed firmware

6 Programming eFuse

6.1 Hardware Conditions

For Rockchip AP series, there are two ways to program user secure data. One is "eFuse programming", the other is "OTP programming"(only few chips support). Following is the introduction.

6.1.1 eFuse Programming

RK3126, RK3128, RK3228, RK3229, RK3288, RK3368 and RK3399 support eFuse programming, following is the general requirements:

A. If products do not need eFuse data programming, we advise to connect eFuse Power Pin directly to GND. Avoiding eFuse data change caused by misoperation. (RK3126/RK3126C eFuse Power Pin is reused with SARADC function, so that it would not to be grounded.)

B. If products need eFuse programming, then connect a pull down resistance to GND on eFuse Power Pin, to make sure that eFuse power pin doesn't fluctuate in normal work condition. also to avoid eFuse data change caused by misoperation. This pull-down resistance value, please refer to each chip platform's reference schematics, generally it's at a range of 47Ω-10KΩ.

C. There are two types of power supply for eFuse programming:

a) Onboard power supply mode

- Advantage: PCBA socket test board is not needed, you can program eFuse data first, and then upgrade the firmware. When system works in normal condition, the firmware must make sure that eFuse power is not on,keep 0V to prevent misoperation.
- Disadvantage: Power supply circuit must placement on the board. The material cost is increased, and you need to make sure the firmware is no misoperation at any time.
- Apply to: This power supply mode applies to customers who don't want to add PCBA testing process. For example some BOX products, their interfaces and assembling are both simple, not need socket board to use on the PCBA test.

b) Power supply by PCBA test board(recommended)

- Advantage: Only test points needed. It is no power supply circuit on board so users can't crack through software too.
- Disadvantage: Increase PCBA test process, the test cost is higher.

I Apply to: Products like tablets, their assembling is complicated. If PCBA is abnormal, it 's more complicated to rework and replace, so these kinds of products usually have PCBA testing process, Programming eFuse on this process is reasonable.

D. Electronic circuit introduction:

Each chip platform's eFuse power supply voltage is different(such as 1.5/1.8/2.5V), power supply pin number and current requirement is also different.

we recommend that power supply capacity should be 50mA above, for detailed voltage and pull-down resistance value, you can refer to schematic diagram. Summarized advices are below:

Table 6-1 Hardware parameters

Chip Part Number	eFusePower	Programming Mode	VQPS Current Requirement	Pull-down Resistance Value	eFusePower Pin Number	Remark
RK3126、RK3126C	2.5V	Power by PCBA test board	>50mA	None	PIN68	Reused with ADC
RK3128	2.5V	Onboard or powered by external	>50mA	<=10K	R10	
RK3168、RK3188	1.5V	Onboard or powered by external	>50mA	<=510R	Y10	
RK3228、RK3229	1.6V	Onboard or powered by external	>50mA	<=100R	R10	
RK3288	1.5V	Onboard or powered by external	>50mA	<=510R	P19	
RK3368	1.5V	Onboard or powered by external	>50mA	<=47R	Y10	
RK3399	1.8V	Onboard or powered by external	>50mA	<=1K	AD23	

Recommended power supply mode is shown as below diagram.

a) PartA: eFuse power supply circuit, please choose suitable LDO part number according to the voltage requirement above, this part circuit can be placed on mainboard, and also can be placed on the PCBA test board.

b) PartB: eFuse power pin with pull down resistance R4(47R-10K), keep the voltage low level to avoid misoperation. If power supply circuit is placed on the PCBA test board, the SOC mainboard needs to add responding testing points, to facilitate fixture pin touch.

Attention:

a) RK3126C's eFuse power is reused with ADC function, so it can't connect pull-down resistance.

b) RK3228/RK3229's eFuse power supply is suggest to be adjusted to 1.55-1.6V, to be more stabled.

c) If the device uses onboard power supply mode, please make sure eFuse_PWREN, which is in the following diagram be distributed an independent GPIO to control the LDO. It must make sure there is no power output on VCC_eFuse PIN in normal work condition. Details refer to reference schematic that RK released, if there is no GPIO distributed, contact us or use external power supply mode.

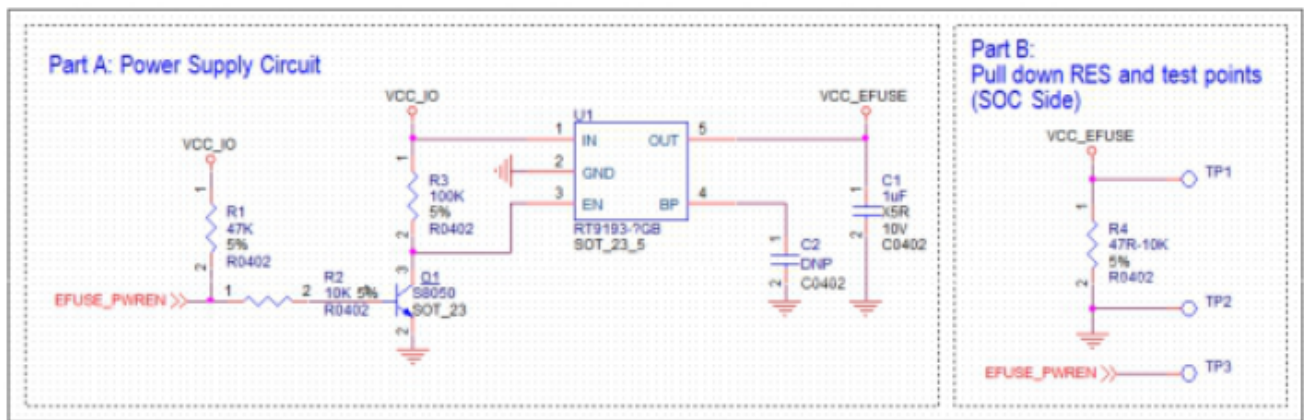


Figure 6-1 eFuse circuit

6.1.2 OTP Programming

RK3328 and RK3228H support OTP programming mode, this mode is no need external power supply circuit, OTP_VCC18(PIN16) is always powered by VCC_18. you only need to run the special time sequence for OTP programming, not need the additional changes about hardware.

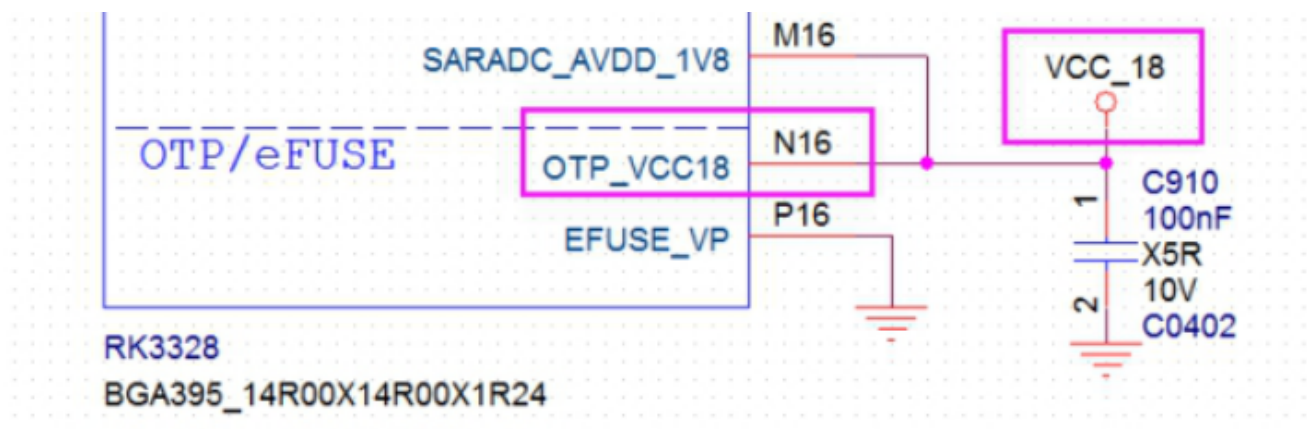


Figure 6-2 OTP circuit

6.2 Tool UI

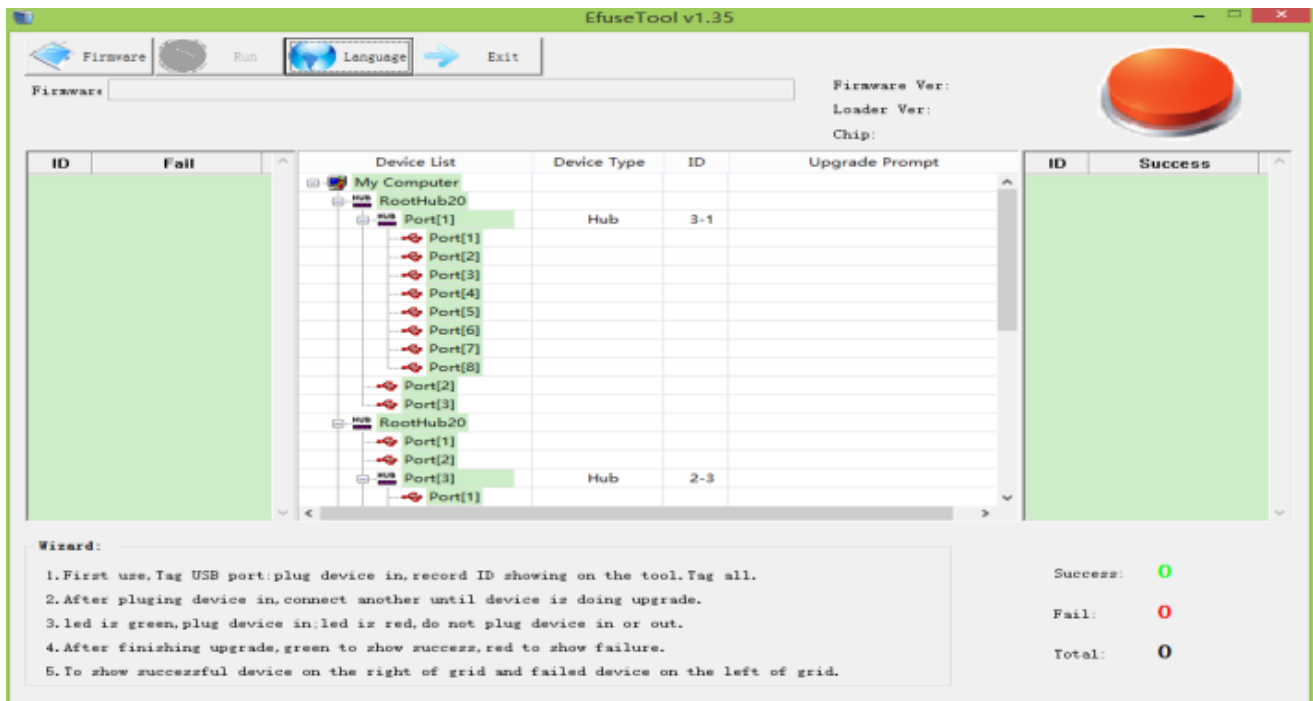


Figure 6-3 eFuse tool UI

6.3 Load the Signed Firmware

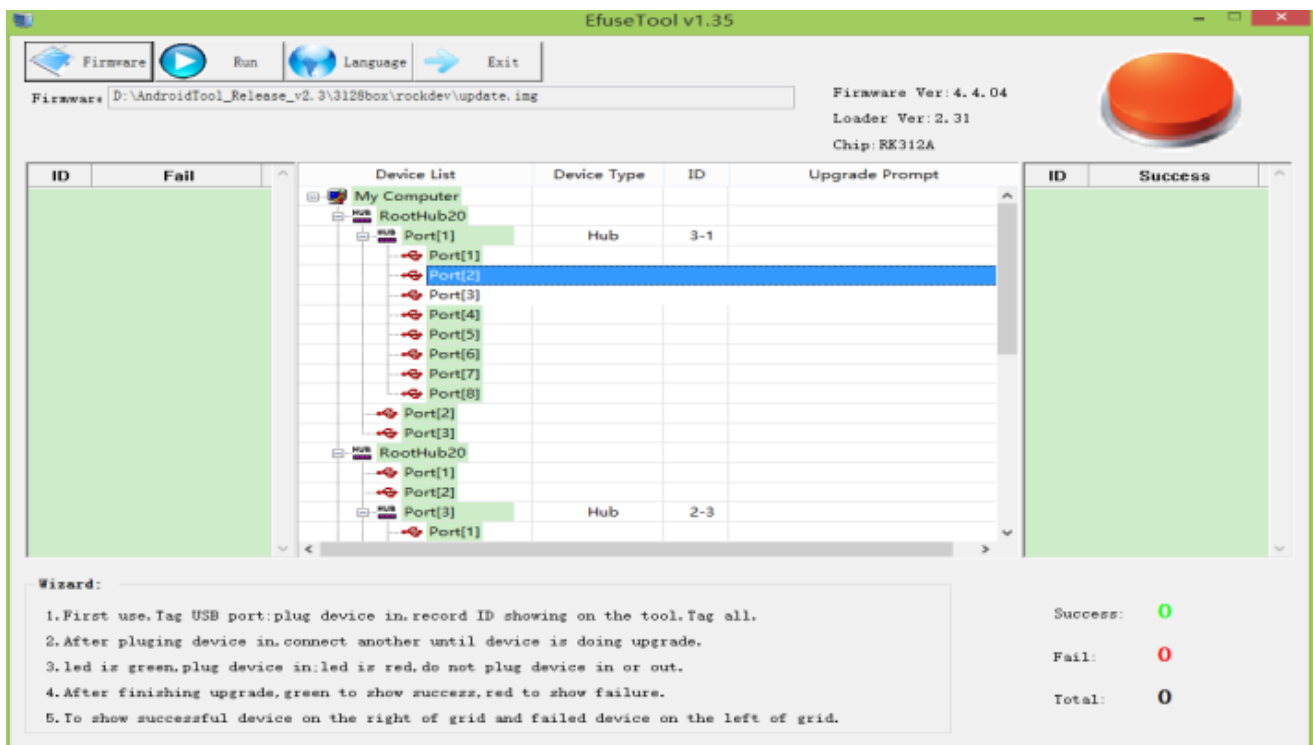


Figure 6-4 Load signed firmware

6.4 Click 'run' Button to Start

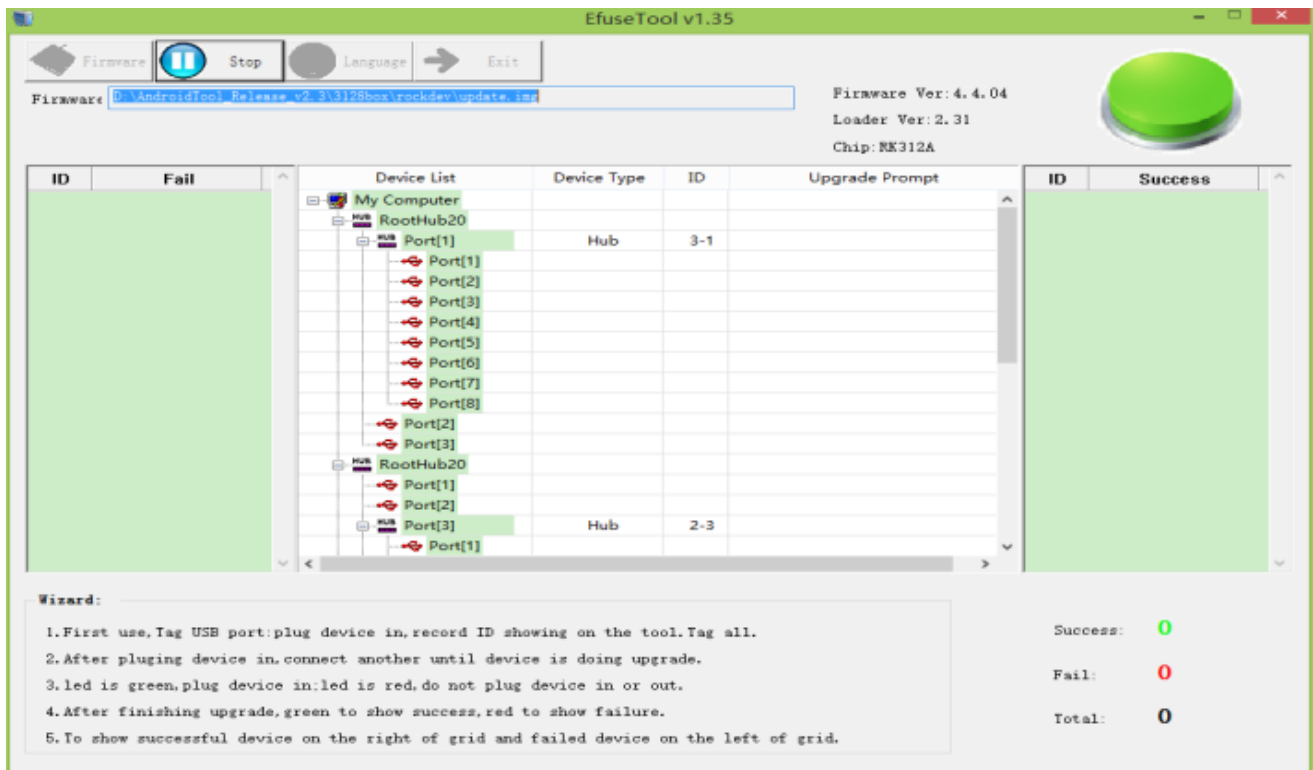


Figure 6-5 Programming the chip

6.5 Programming eFuse

Connect the device to the PC by USB cable; the tool will program the hash of RSA public key to eFuse automatically.

Programming eFuse needs an external power supply, the detail information please refer to SOC's DATASHEET.

Notice: RK3228H, RK3328, RK3336, RK3308 and PX30 don't need step [6.2](#) to [6.4](#). Programming will be done by upgrading firmware which has been signed.

6.6 Programming OTP

RK3228H, RK3328, RK3326, RK3308 and PX30 support OTP programming. Public key hash need program to OTP. Programming OTP performs are :

1. First, follow the above steps to burn signed firmware. If the machine can start normally, the signature process is correct. Then OTP can be programmed.

2. The signature tool uses version of SecureBootTool V1.9 or more. Open the config.ini file in the tools directory. Find "sign_flag=", set "sign_flag=0x20"(bit 5 set 1) which enable write OTP in RKLoader. Save config.ini file. Reopen SecureBootTool.exe to sign firmware or RKLoader.

本地磁盘 (D:) > work > SecureBootTool_v1.9

名称	修改日期	类型	大小
bin	2016/11/7 15:26	文件夹	
Log	2018/5/11 10:17	文件夹	
config.ini	2018/5/14 18:01	配置设置	2 KB
libcrypto-1_1.dll	2017/5/25 21:20	应用程序扩展	2,042 KB
libssl-1_1.dll	2017/5/25 21:20	应用程序扩展	365 KB
msvcr120.dll	2017/5/25 21:20	应用程序扩展	949 KB
PrivateKey.pem	2018/4/2 10:46	PEM 文件	2 KB
PublicKey.pem	2018/4/2 10:46	PEM 文件	1 KB
SecureBootTool.exe	2018/5/11 10:14	应用程序	1,130 KB

Figure 6-6-1 SecureBootTool

```

config.ini - 记事本
文件(F)  编辑(E)  格式(O)  查看(V)  帮助(H)

[System]
support_chip=3308|3326|3399|3228h|3229|3368|3228|3288|3128|3036
new_crypto=3308|3326
#using software to check signature,using sha160 ,belong to "soft_sign"
soft_sign=3128|3036
#using hardware to check signature,using big sha256,belong to "hard_sign"
hard_sign_big_hash=3228h|3368|3228|3288
#using hardware to check signature,using little sha256,belong to "hard_
hard_sign_litte_hash=3399
#using hardware to check signature,using pss padding ,at the beginning
hard_sign_pss=3308|3326|3229

sign_flag=0x20
sign_soft_version=
sign_nonce=
  
```

Figure 6-6-2 config.ini

3. Use re-signed firmware or RKLoader burning. After burning, restart the machine. The RKLoader will be responsible for generating hash of public key and writing it to OTP during startup and enable secure boot.

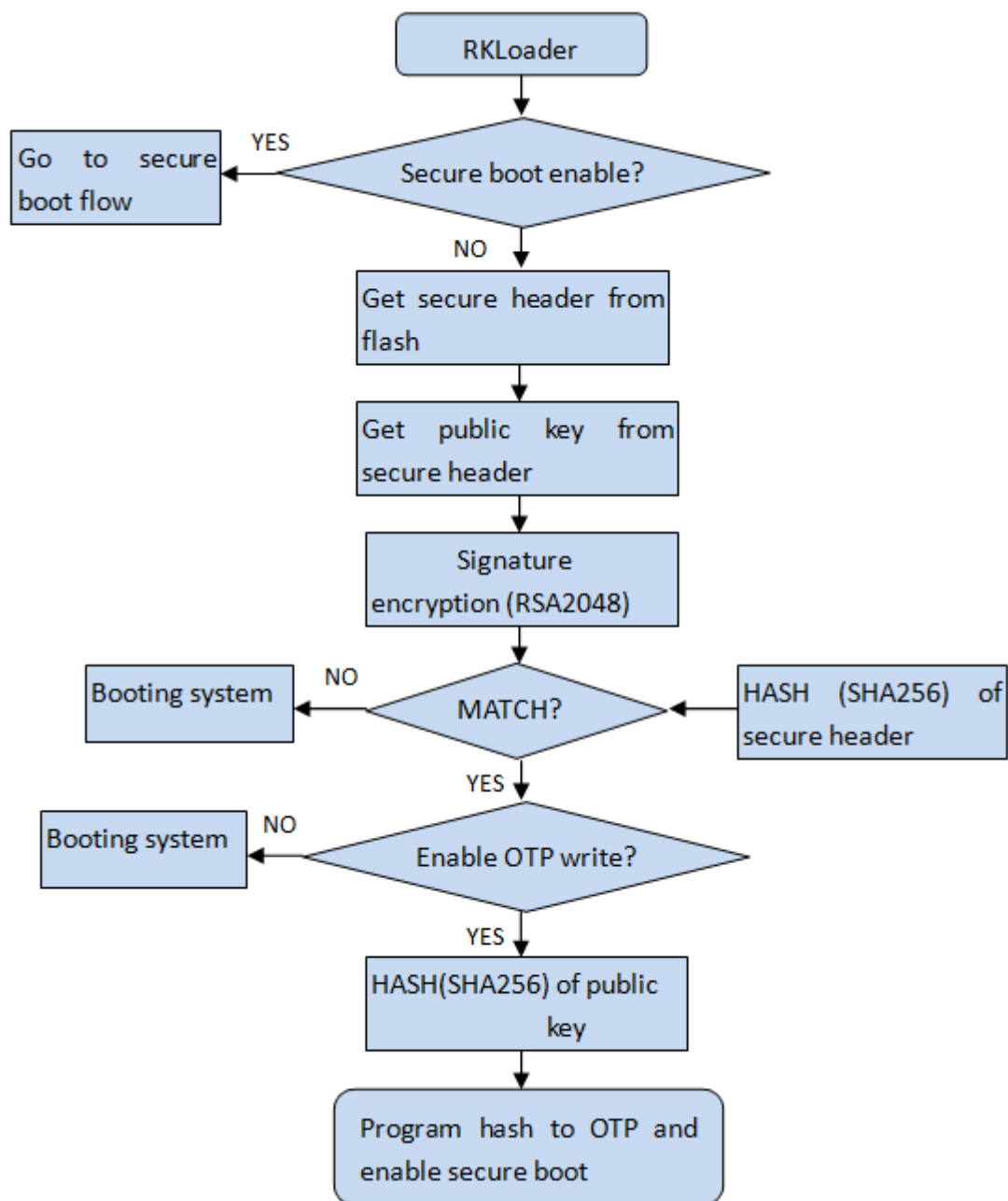


Figure 6-6-3 OTP program flow

4.If OTP program success, serial port print "otp write key success!!!!". If OTP program fail, serial port print"otp write error: !!!!".

7 Firmware Upgrade

7.1 Firmware Upgrade

Open the signed firmware and connect the device which has programmed eFuse to the PC by USB cable:

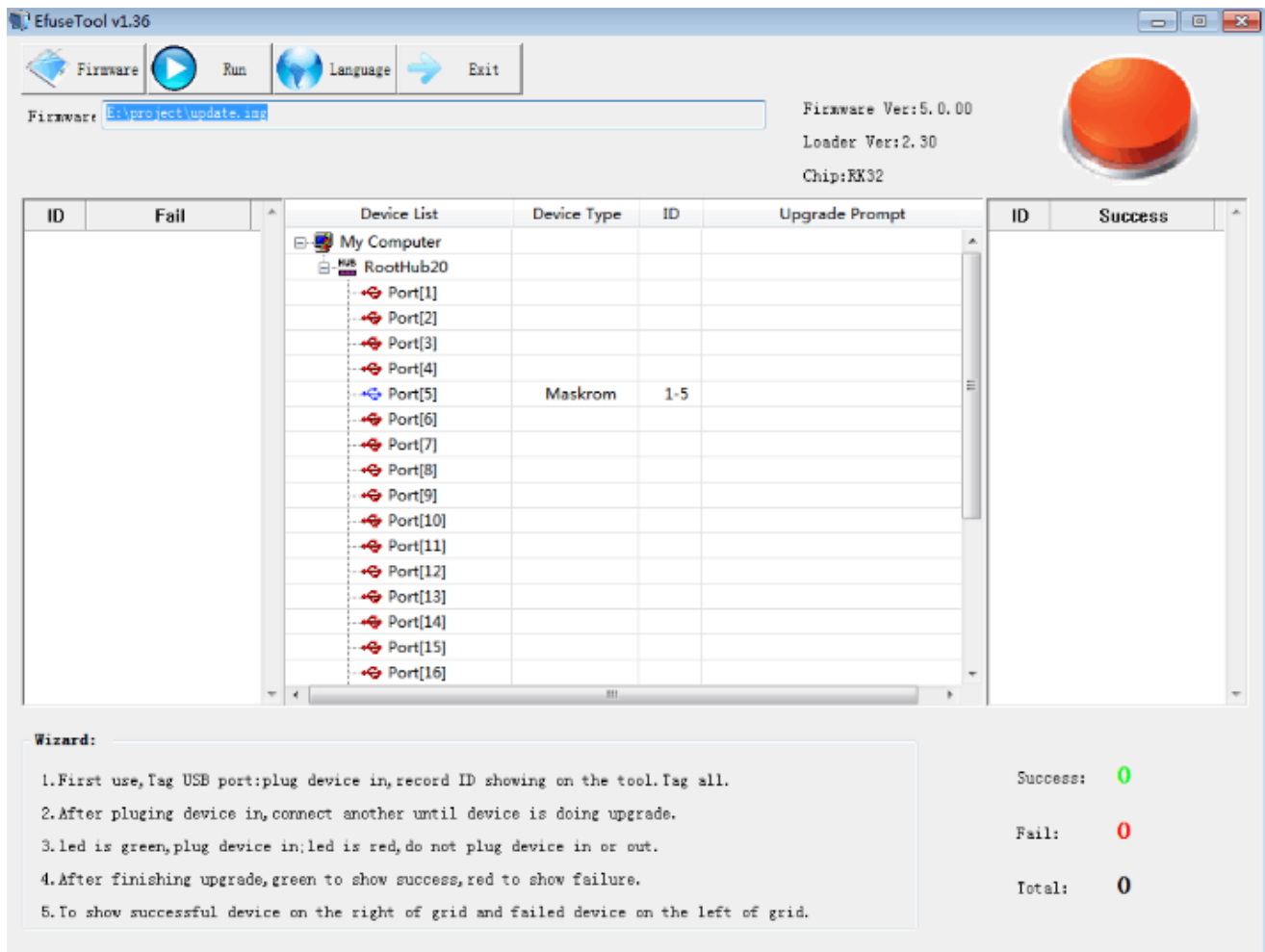


Figure 7-1 Upgrade tool 1

Click the 'Upgrade' button to start firmware upgrade and wait it to be completed:

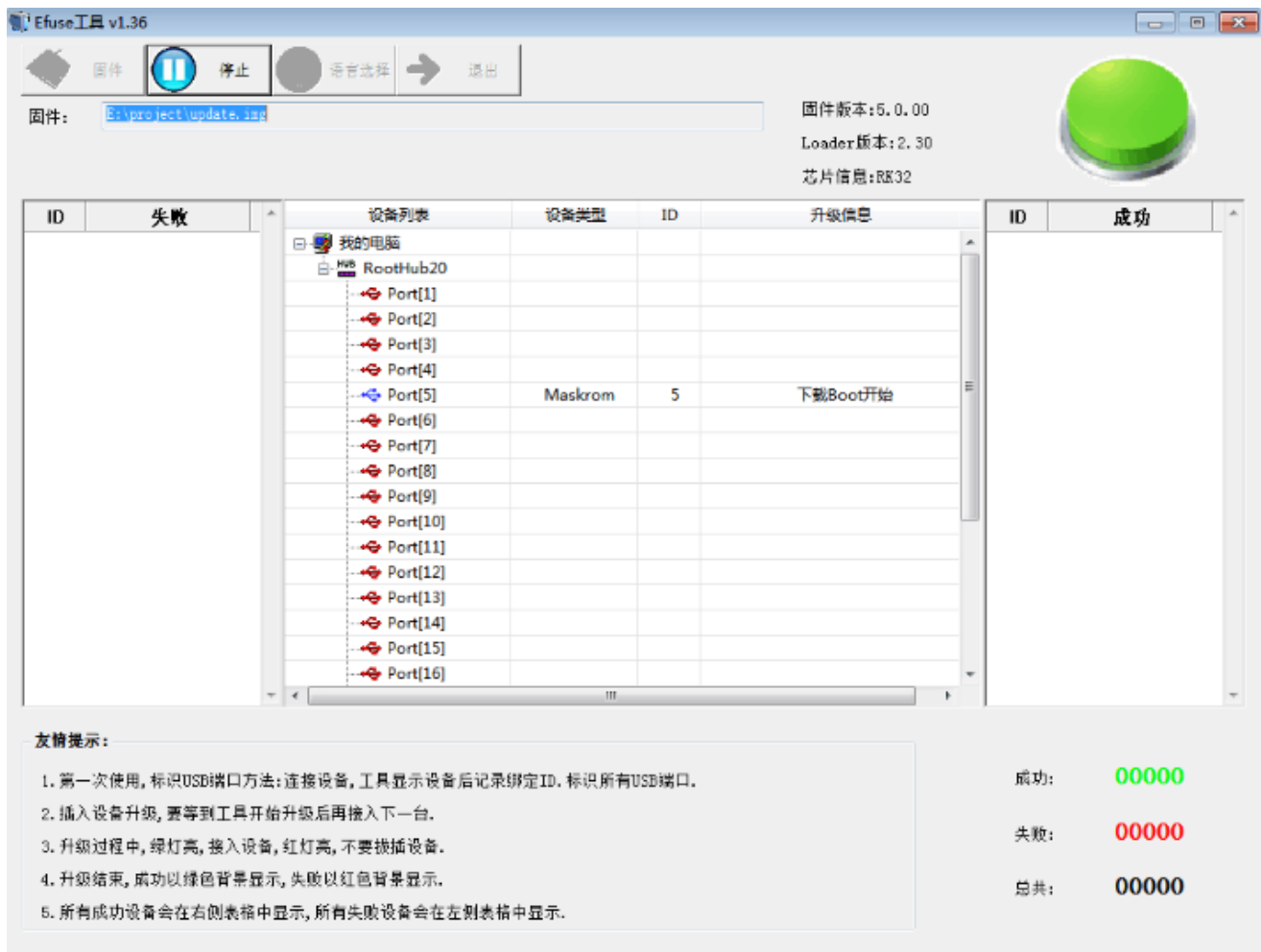


Figure 7-2 Upgrade tool 2

8 Verification

8.1 Check Secure Flag

Use serial port tools (e.g. SecureCRT) to get the log of system boot. These words show that the security boot is on:

Secure Boot Mode: 0x1

SecureBootEn = 1, SecureBootLock = 1

```

106
107 GetParam
108 check parameter success
109 Unknow param: MACHINE_MODEL:rk3288!
110 Unknow param: MACHINE_ID:007!
111 Unknow param: MANUFACTURER:RK3288!
112 Unknow param: FWR_HLD: 0,0,A,0,1!
113 power key: bank-0 pin-5
114 can't find dtg node for ricoh619
115 pmic:act8846
116 fg:cw201x
117 Secure Boot Mode: 0x1
118 SecureBootEn = 1, SecureBootLock = 1
119
120 #Boot ver: 2015-02-06#2.19
121 empty serial no.
122 checkKey
123 vbus = 0

```

Figure 8-1 Log of system boot

8.2 Secure Boot Test

The device which had programmed eFuse will enable secure boot rom, and could not boot from the unsigned firmware.

So try to upgrade un-signed firmware or unmatched key signed firmware will fail;

And upgrade matched signed firmware will boot success.

SOC RK3128 and RK3126 will fail at “wait for loader”:

The screenshot shows the FactoryTool v1.52 interface. At the top, there are buttons for Firmware, Stop, Upgrade, Restore, Demo, Language, and Exit. Below these, the Firmware path is set to 'E:\project\updat_KEYS 1 then key2 then key5.in'. The Firmware Ver is 5.0.00, Loader Ver is 2.30, and the Chip is RK32.

The main area displays a list of devices connected to the computer. The 'Device List' table shows the following data:

ID	Fail	Device List	Device Type	ID	Upgrade Prompt
5	Test Device Fail	My Computer			
		RootHub20			
		Port[1]			
		Port[2]			
		Port[3]			
		Port[4]			
		Port[5]	Maskrom	5	Test Device Fail
		Port[6]			
		Port[7]			
		Port[8]			
		Port[9]			
		Port[10]			
		Port[11]			
		Port[12]			
		Port[13]			
		Port[14]			
		Port[15]			
		Port[16]			

At the bottom, there is a 'Wizard' section with instructions:

- 1.First use, tag USB port: plug device in, record ID showing on the tool. Tag all.
- 2.After plugging device in, connect another until device is doing upgrade.
- 3.led is green, plug device in: led is red, do not plug device in or out.
- 4.After finishing upgrade, green to show success, red to show failure.
- 5.To show successful device on the right of grid and failed device on the left of grid.

On the right side, there is a summary of results:

- Success: 00000
- Fail: 00001
- Total: 00001

Figure 8-2 Upgrade fail 1

Other SOC will fail at “Download Boot”:

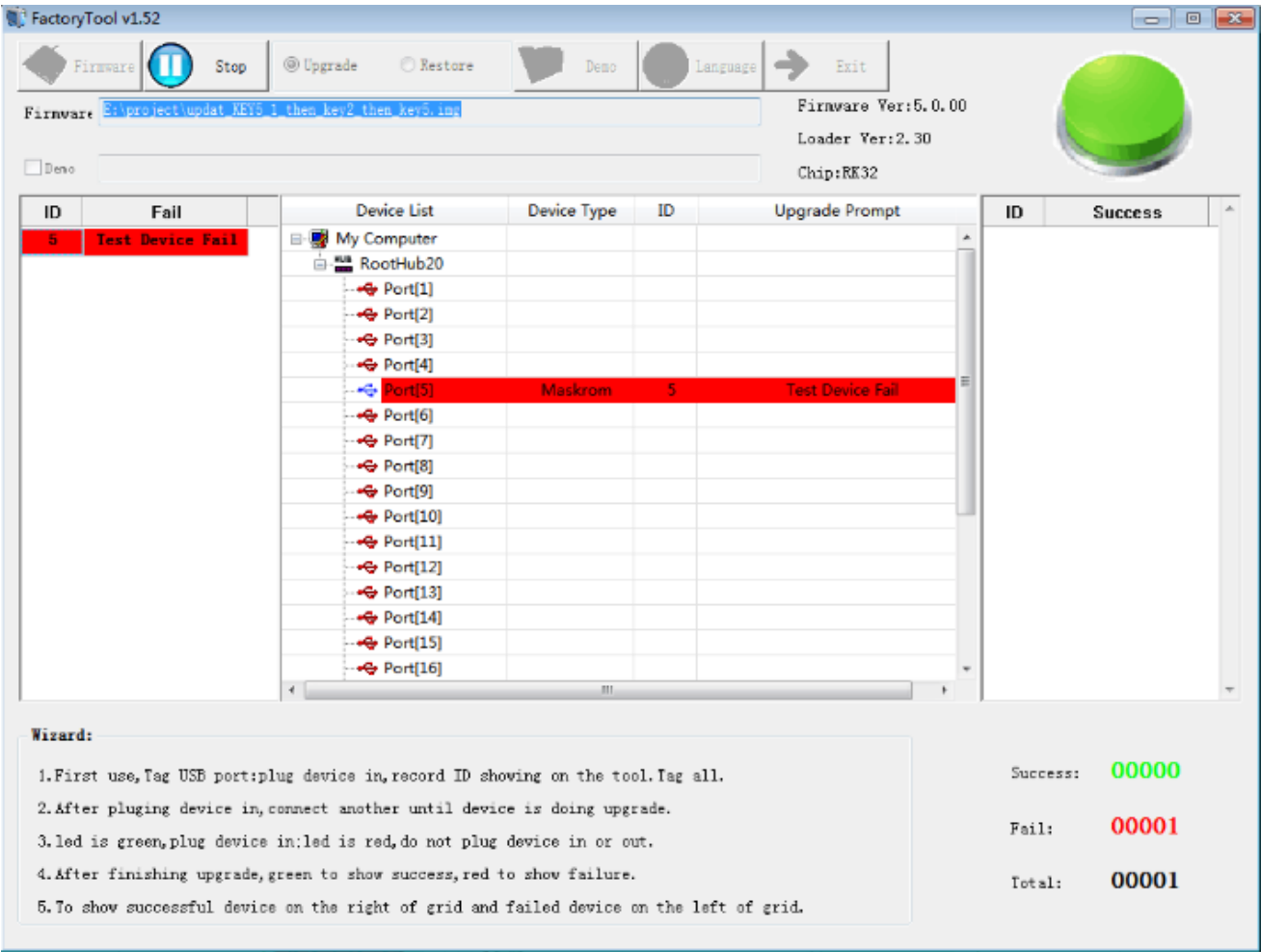


Figure 8-3 Upgrade fail 2