



地平线
Horizon Robotics

X3J3平台系统软件 安全启动与加密使用说明

v1.3

2021-02

版权所有© 2018 Horizon Robotics

保留一切权利

免责声明

本文档信息仅用于帮助系统和软件使用人员使用地平线产品。本文档信息未以明示或暗示方式授权他人基于本文档信息设计或制造任何集成电路。

本文档中的信息如有更改，恕不另行通知。尽管本文档已尽可能确保内容的准确性，本文档中的所有声明、信息和建议均不构成任何明示或暗示的保证、陈述或担保。

本文档中的所有信息均按“原样”提供。地平线不就其产品在任何特定用途的适销性、适用性以及不侵犯任何第三方知识产权方面做出任何明示或暗示保证、陈述或担保。地平线不承担产品使用所引起的任何责任，包括但不限于直接或间接损失赔偿。

买方和正在基于地平线产品进行开发的其他方（以下统称为“用户”）理解并同意，用户在设计产品应用时应承担独立分析、评估和判断的责任。用户应对其应用（以及用于其应用的所有地平线产品）的安全性承担全部责任，并保证符合所有适用法规、法律和其它规定的要求。地平线产品简介和产品规格中提供的“典型”参数在不同应用下可能会不同，实际性能也可能随时间而变化。所有工作参数，包括“典型”参数，都必须由用户自己针对每项用户应用进行验证。

用户同意如因用户未经授权使用地平线产品或因不遵守本说明中的条款，造成任何索赔、损害、成本、损失和（或）责任，用户将为地平线及其代表提供全额赔偿。

© 2018 版权所有

北京地平线信息技术有限公司

<https://www.horizon.ai>

修订记录

修订记录列出了各文档版本间发生的主要更改。下表列出了每次文档更新的技术内容。

版本	修订日期	修订说明
1.0	2020-09	初始版本
1.1	2020-12	增加 UBoot AVB 以及 Kernel DM-verity 的详细说明
1.2	2020-12	更新 eufse 烧写部分，增加 jtag/bif sd/bif spi 和 socid 白名单内容
1.3	2020-12	更新 efuse 烧写章节，添加公钥 hash，key 的选择功能 更新 uboot 签名章节，添加密钥管理小节

目录

免责声明.....	I
修订记录.....	II
目录	III
1 引言	1
1.1 编写目的.....	1
1.2 术语约定.....	1
1.3 读者对象和阅读建议.....	1
2 总体概述.....	2
2.1 系统启动总体流程图.....	2
2.2 流程解释.....	2
3 UBOOT 签名过程.....	4
3.1 HEADER 格式	4
3.2 加密和签名工具.....	4
3.3 对 UBOOT IMAGE 进行签名	9
3.4 对 HEADER 签名	9
3.5 密钥管理.....	10
4 BPU 模型文件的保护.....	11
4.1 加解密与签名验证过程.....	11
4.2 运行时的保护	16
5 EFUSE 的烧写.....	17
5.1 EFUSE 的基本介绍	17
5.2 EFUSE 的配置文件	17
5.3 配置范例.....	20
6 AVB 和 DM-VERITY.....	21
6.1 BUILD 系统添加 AVB 以及 DM-VERITY 校验.....	22
6.2 UBOOT 校验 LINUX 内核.....	22
6.3 LINUX 内核使用 DM-VERITY 校验根文件系统	23

1 引言

1.1 编写目的

撰写该文档主要是对于 XJ3 芯片方案中关于安全启动与加密的使用说明。

1.2 术语约定

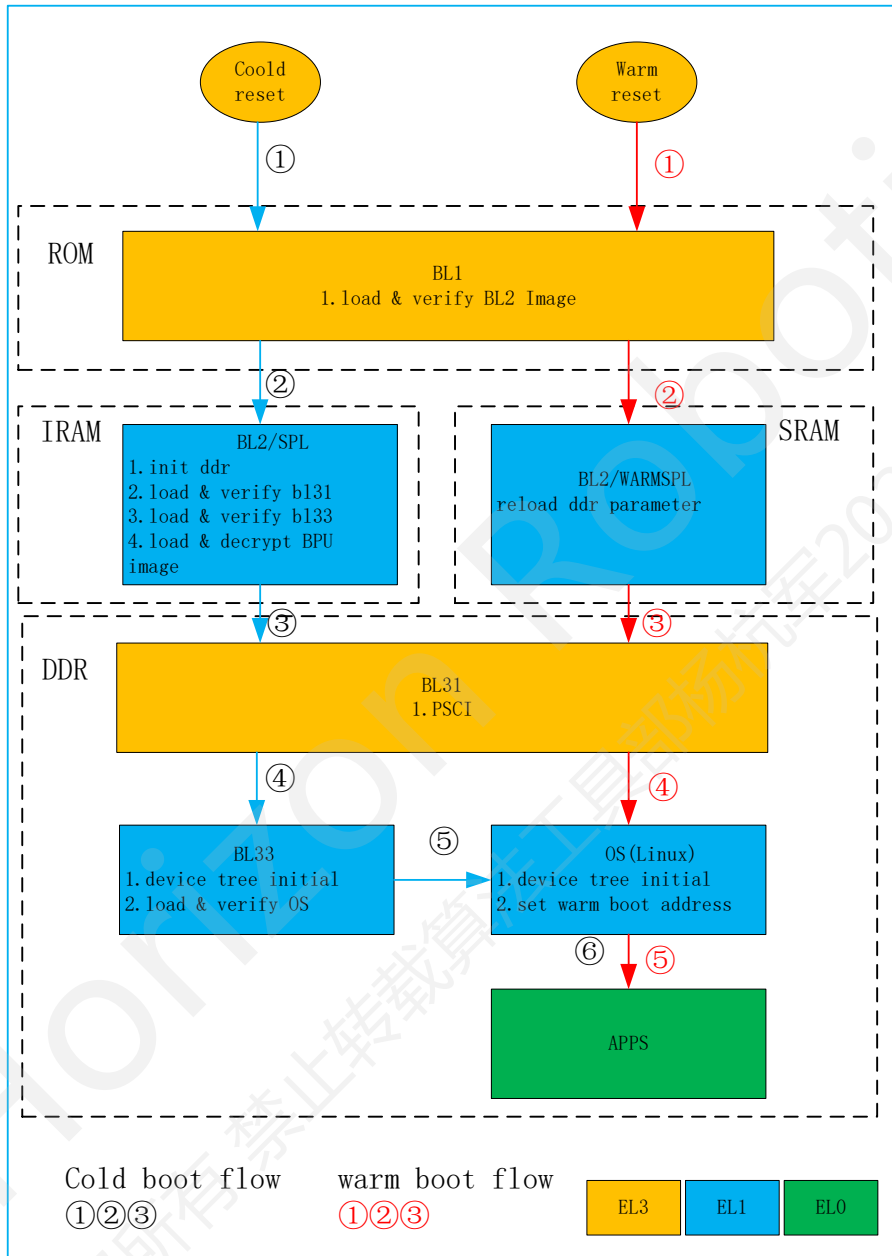
缩写	全称
SoC	System on Chip
BL[x]	Boot Loader Stage [x]
SPL	Secondary Program Loader
BPU	Brain Process Unit
AES	Advanced Encryption Standard
IV	Initial Vector
SHA	Secure Hash Algorithm
AVB	Android Verified Boot
PSCI	Power State Coordination Interface
SMC	Secure Monitor Cal
CMA	Contiguous Memory Allocator
HBM	HoBot Model

1.3 读者对象和阅读建议

该文档针对需要使用到安全启动的开发工程师及相关工程人员。文档提供了该部分软件使用上的各层面的信息以及大致的概念，如果涉及具体接口信息，请参考对应的详细设计手册。

2 总体概述

2.1 系统启动总体流程图



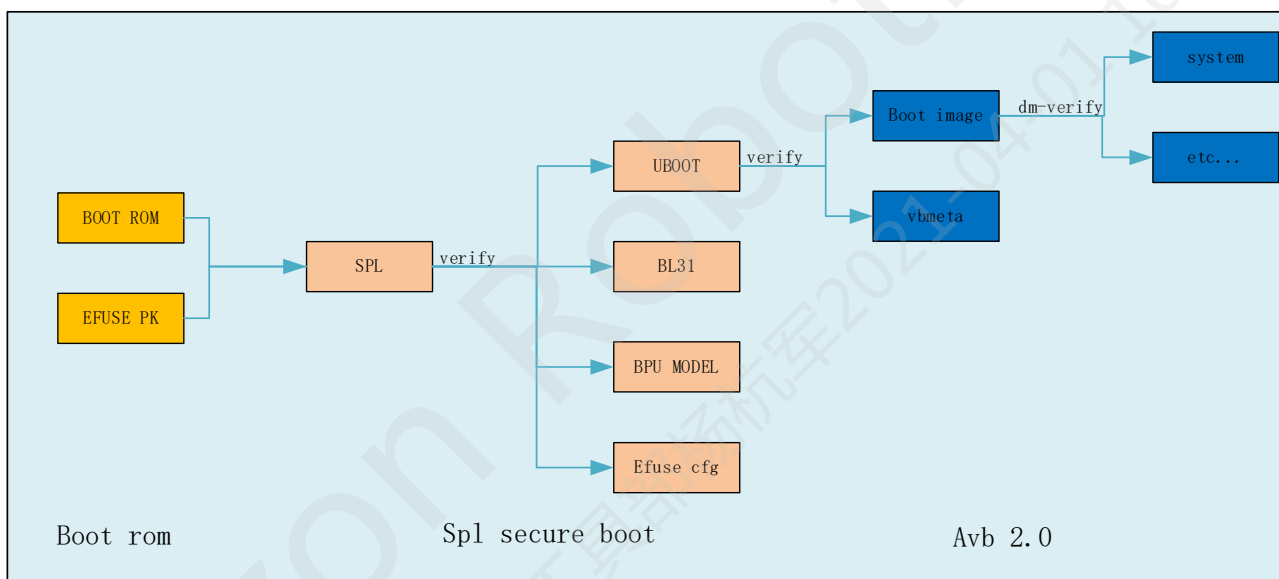
上电分为冷启动和热启动两种，热启动特指休眠唤醒流程，其他方式如掉电上电，reboot 命令等均为冷启动

2.2 流程解释

1. Power On Reset 时跳转到 BL1 执行，运行状态 EL3

2. BL1 将会跳转到 BL2 中去执行，运行状态切换到 EL1，冷启动时 BL2 为称 SPL, 热启动时称为 WARM SPL
3. BL2/SPL 负责初始化 DDR，加载 BL31、BL33 和 BPU 加密模型的镜像，并做校验。然后 BL2 通过 SMC 调用，跳转到 BL31，BL31 运行在 EL3 状态。WARMSPL 只负责从 sram 中重新 load ddr 参数。
4. BL31 初始化 PSCI、向量表后，跳转到 BL33，BL33 运行在 EL1 状态
5. BL33 做 device tree 初始化，加载和校验 OS，然后启动 OS，OS 运行在 EL1 状态
6. OS 启动完成，开始运行用户层的服务进程，用户层服务运行在 EL0 状态

在整体的启动流程中，系统的安全启动（Secure Boot）以层级的方式逐级向下校验，总体流程如下图所示



Secure Boot 的验证流程见上图所示，bootrom 对 SPL 镜像进行校验。SPL 校验通过之后，SPL 再去加载并校验 UBoot，BL31，BPU 模型；系统进入 UBoot 之后，再由 UBoot 来使用 AVB 校验 OS 内核镜像（boot image，包含 ramdisk，dtb 等）。在 OS 启动之后，通过 OS 支持的 dm-verity 来校验对应的系统分区。bootrom 校验 spl 和 spl 校验 bl31 是强制性的，其余的如 spl 校验 uboot 和 bpu，以及 uboot 通过 avb 校验 kernel，和 kernel 使用 dm-verity 校验 filesystem 都可以通过 eFUSE 进行配置，详细情况参考第五章“eFUSE 的烧写”。

3 uboot 签名过程

3.1 header 格式

bl31 和 uboot 的 header 格式如下:

```
1. typedef struct {
2.     uint8_t      signature_h[RSA_SIGN_SIZE];
3.     uint8_t      signature_i[RSA_SIGN_SIZE];
4.     uint32_t      magic; /* HBOT */
5.     uint32_t      image_load_addr;
6.     uint32_t      image_len;
7.     uint32_t      product_id;
8.     uint32_t      customer_id;
9.     uint8_t      reserve[HB_HEAD_SIZE - 2 * RSA_SIGN_SIZE - 20];
} hb_img_header;
```

- signature_h: header 中除了本部分外, 对其他内容的签名
- signature_i: 该 image 的签名
- magic: 幻数, HBOT
- image_load_addr: image 验证过后被 load 到 ddr 的地址
- image_len: image len
- customer_id: customer id, 预留给客户使用, 要与 efuse 中烧写的 customer id 保持一致
- product_id: 预留给客户自定义的 product id, 与 efuse 中烧写的 product id 要一致
- reserve: 保留

在编译镜像时, 会将 header 添加到 uboot.img 之前, 在安全启动时 spl 根据 header 中信息对 uboot 镜像验证。

3.2 加密和签名工具

在 build/tools/下, 有一个 key_management_toolkits, 这个文件夹下便是用来对各个 image 进行打包的工具, 其中 utils/util_auth.sh 是对文件加密和签名的脚本, 对其他 image 的打包会经常用到这个脚本

```
1. while getopts "ei:o:k:v:p:shr" opt; do
2.     case $opt in
3.         r) # get key from kms with real rom key
```



```
4.         get_rom_key=1
5.         fallback_key=0
6.         ;;
7.     i)
8.         input=$OPTARG
9.         ;;
10.    o)
11.        output=$OPTARG
12.        ;;
13.    k)
14.        key=$OPTARG
15.        ;;
16.    v)
17.        iv=$OPTARG
18.        ;;
19.    p)
20.        pad_size=$OPTARG
21.        ;;
22.    e)
23.        pad_image $input $pad_size
24.        encrypt_image $key $iv $input.pad
25.        exit
26.        ;;
27.    h)
28.        hash_image $output $input
29.        exit
30.        ;;
31.    s)
32.        sign_image $key $input $output
33.        exit
34.        ;;
35.    \?)
36.        echo "Invalid option: -$OPTARG" >&2
37.        ;;
38.  esac
39. done
```

- r: 表示使用真正的 key, 需要从服务器上拿到 key
- i: 输入
- o: 输出
- k: key
- v: init vector
- p: padding size

- e: 执行加密过程
- h: 执行 hash 过程
- s: 执行签名过程

util_auth.sh 的内容如下:

```
1.get_rom_key=0
2.
3. # $1: key
4. # $2: iv
5. # $3: name of image
6.encrypt_image()
7.{
8. ...
9.     openssl enc -nosalt -nopad -aes-128-cbc -in $image_name -out $image_name.enc \
10.                -K $key \
11.                -iv $iv
12.}
13.
14. # $1: image $2
15. # $2: pad size
16.pad_image()
17.{
18. ...
19.
20.}
21.
22.hash_size=32          #sha256 size is 256 -> 32 byte
23.hash_block_size=64    #sha256 block size is 512 bit -> 64byte
24.block_size=32768      #32K, because IP limiation (2^64-1 byte...)
25.hash_set_lv3_blk_sz=33554432 #32M,
26.
27.# $1 image
28.# $2 name output
29.process_hash_set_lv3()
30.{
31. ...
32.}
33.
34.process_hash_set()
35.{
36. ...
37.}
38.
```

```
39.# $1: prefix of hash value file
40.# $2: the image we want to do hash
41.hash_image()
42.{
43.    img_size=$(wc -c < $2)
44.
45.    if [ "$img_size" -gt "$hash_set_lv3_blk_sz" ]; then
46.        process_hash_set_lv3    $2 $1
47.    else
48.        process_hash_set        $2 $1
49.    fi
50.}
51.
52.# $1: priv key if using fallback key, or keep the number of real rsa key
53.# $2: input image
54.# $3: output image
55.sign_image()
56.{
57.    if [ "$get_rom_key" -eq "0" ]; then
58.        echo use fake key
59.        cat $2 | openssl rsautl -inkey $1 -sign > $3
60.    else
61.        echo use real key
62.
63.        folder_exist=0
64.        test -d key_list && folder_exist=1
65.
66.        if [ "$folder_exist" -eq "0" ]; then
67.            echo folder not exist, we pull new
68.            git clone git@10.104.34.40:/home/git/key_list.git
69.        fi
70.
71.        . key_list/rom/access_key.list
72.        . key_list/spl/access_key.list
73.
74.        if [ "$1" -eq "0" ]; then
75.            ./kms_util -u $account -a $ak_rom_rsa_key0 -s $2 -o $3
76.        fi
77.        ....
78.    fi
79.}
```

- encrypt_image: 对 image 进行加密

- hash image: 对 image 进行 hash 计算
- sign_image: 对 image 进行签名,本地测试的 key 存放在了 tools/key_management_toolkits/utlis/fallback_key_sets 路径下。客户在使用自己的 key 时, 对应替换掉这个文件夹下对应的 key 即可

打包 uboot 的工具为 pack_uboot_tool.sh, 位于 build/tools/key_management_toolkits, 打包的过程如下:

```
1.function pack_uboot()  
2.{  
3.    local uboot_sign_image="uboot.img.bin"  
4.    ...  
5.  
6.    #1, get uboot keys  
7.    get_key  
8.  
9.    #2, sign uboot Image  
10.   cp "$TARGET_UBOOT_DIR/$uboot_image" ./  
11.   enc_and_sign_uboot  
12.   rm $uboot_image  
13.  
14.   #3, sign Header  
15.   sign_header  
16.  
17.   #4, cat image / header together  
18.   cat uboot_header_with_sign      >> $uboot_sign_image  
19.  
20.   if [ x"$ENCRYPT_FLAG" = x"true" ];then  
21.       cat ${uboot_image}.pad.enc      >> $uboot_sign_image  
22.   else  
23.       cat ${uboot_image}.pad          >> $uboot_sign_image  
24.   fi  
25.   ...  
26.}
```

1. 获取 uboot 的 key
2. 对 uboot image 进行签名
3. 对 uboot 的 header 进行签名
4. 打包 uboot header 和 uboot image

3.3 对 uboot image 进行签名

```
1.function enc_and_sign_uboot()  
2.{  
3.    # encrypt uboot image  
4.    util_auth.sh -i $uboot_image -k $uboot_aes_key -v $uboot_aes_iv -p 64 -e  
5.  
6.    # hash uboot image  
7.    util_auth.sh -i ${uboot_image}.pad -o $uboot_image -h  
8.  
9.    # sign uboot hash  
10.   util_auth.sh -i ${uboot_image}_hash.bin -k $uboot_rsa_priv -o ${uboot_image}.sign -s  
11.}
```

uboot 签名的过程比较简单，加密作为备选，并未用到

1. 对 uboot image 计算 hash
2. 对 hash 进行签名，目前 uboot 签名使用本地的 key，客户在使用时将\$uboot_rsa_iv 对应的私钥文件替换即可，公钥用于在 spl 阶段对 uboot 镜像验证。

3.4 对 header 签名

```
1.function sign_header()  
2.{  
3.    local header_tmp="uboot_header_tmp"  
4.    local header_sign="uboot_header_with_sign"  
5.  
6.    image_size=$(wc -c < ${uboot_image}.pad)  
7.    echo "image size = $image_size"  
8.  
9.    hex2bin.sh $uboot_magic uboot_magic.bin  
10.   hex2bin.sh $uboot_image_load_addr uboot_image_load_addr.bin  
11.   hex2bin.sh $image_size uboot_image_size.bin  
12.   cat ${uboot_image}.sign > $header_tmp  
13.   cat uboot_magic.bin >> $header_tmp  
14.   cat uboot_image_load_addr.bin >> $header_tmp  
15.   cat uboot_image_size.bin >> $header_tmp  
16.   hex2bin.sh $USER_PRODUCT_ID product_id.bin  
    cat product_id.bin >> $header_tmp  
    hex2bin.sh $USER_CUSTOMER_ID customer_id.bin  
    cat customer_id.bin >> $header_tmp  
    rm customer_id.bin
```

```

17. dd if=/dev/zero of=${header_tmp}.pad bs=1 count=$uboot_header_size status=none
18. dd if=$header_tmp of=${header_tmp}.pad bs=1 count=$uboot_header_size conv=notrunc status=none
19.
20. util_auth.sh -i ${header_tmp}.pad -o $header_tmp -h
21. util_auth.sh -i ${header_tmp}_hash.bin -k $uboot_rsa_priv -o ${header_tmp}.sign -s
22.
23. cat ${header_tmp}.sign > $header_sign
24. cat ${header_tmp}.pad >> $header_sign
25.}

```

1. 首先按照 header 的布局，填充 header 内容
2. 对 header 计算 hash，并签名，header 和 image 的签名使用同一把 key
3. 最终将 header 的签名填充到 header 中，USER_PRODUCT_ID/USER_CUSTOMER_ID 即 header 格式中 product_id/customer_id,客户可自行进行修改，但要与烧到 efuse 中 product_id/customer_id 一致（efuse 相关内容见本文第 5 章），然后生成最终的 header

3.5 密钥管理

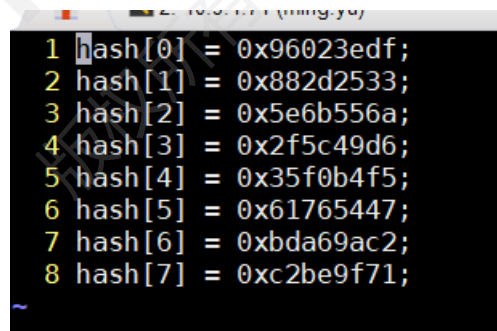
签名校验算法使用的是 RSA2048，私钥使用标准的 PEM PKCS#1 格式，客户将公钥和私钥覆盖 build/tools/key_management_toolkits/utis/fallback_key_sets 下的 spl_rsa_uboot_img_priv.pem 和 spl_rsa_uboot_img_pub.pem 文件，在编译时，便会对 uboot 进行签名。

spl 需要使用公钥进行验证，spl 有两种方式获取公钥。

第一种方式，需要客户将公钥交于地平线，地平线将公钥打包到 spl 的 keybank 中，地平线将会对 keybank 镜像加密和签名，以确保 keybank 的安全性。

第二种方式，客户可将公钥打包到 image 中，并需要将其 hash 烧写到 efuse 中，spl 将会计算镜像中的 hash，与 efuse 中烧写的 hash 进行比对。

默认情况下编译时，已经将公钥打包到了镜像中，客户只需要将 uboot 的公钥替换为自己的即可。在 build/tools/key_management_toolkits 下，pack_keyset_tool.sh 脚本用于打包公钥并计算 hash 值，执行该脚本后，将在 ddr 的输出目录产生 efuse_hash_data 文件，文件中存放了公钥的 hash。需要将 hash 烧写到 efuse 中。



```

1 hash[0] = 0x96023edf;
2 hash[1] = 0x882d2533;
3 hash[2] = 0x5e6b556a;
4 hash[3] = 0x2f5c49d6;
5 hash[4] = 0x35f0b4f5;
6 hash[5] = 0x61765447;
7 hash[6] = 0xbda69ac2;
8 hash[7] = 0xc2be9f71;

```

注：

- 1、烧写 efuse 的方法参见第 5 章节“EFUSE 的烧写”
- 2、使用 debug 版本编译。ddr 的输出目录为 out/horizon_x3-debug.64/target/deploy/ddr，使用 release 版本编译，ddr 的输出目录为 out/horizon_x3-release.64/target/deploy/ddr
- 3、公钥的 hash 并不是对 xxxx-pub.pem 文件直接计算 hash，而是对公钥 256 字节的元数据计算 hash，请使用 pack_keyset_tool.sh 产生公钥

4 BPU 模型文件的保护

注意： BPU 模型加密保护仅在系统版本 0922 及以上支持

BPU 模型文件的保护可以分为两个部分，

一是防止模型文件被离线修改，也就是对模型文件的源文件进行保护。在 PC 上进行加密和签名，在 SPL 中可以进行签名验证和解密，对 BPU 模型源文件实现保护。

二是防止模型文件被在线修改，也就是在运行过程中，模型文件区域被踩踏或恶意修改。通过 MPU 硬件对模型区域进行保护，只允许 BPU 可以访问模型区域

4.1 加解密与签名验证过程

4.1.1 bpu image 的布局

模型文件在 PC 端是先进行加密，然后再签名，在 SPL 中则是先验证签名再解密。

bpu image 的 header 格式如下

```
1.
2. typedef struct {
3.     hbm_info_t                hbm[4];
4.     uint32_t                  image_len;
5.     uint32_t                   bpu_range_start;
6.     uint32_t                   bpu_range_sz;
7.     uint32_t                   bpu_all_fetch_only_sz;
8.
9. } bpu_img_header;
```

- hbm: 模型文件信息，最多可以放 4 个模型文件
- image_len: image 的长度
- bpu_range_start: bpu 模型地址的起始
- bpu_range_sz: 所有 bpu 模型的大小

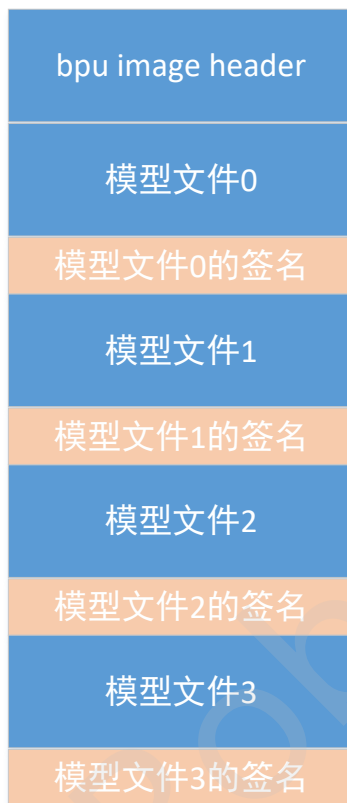
- `bpu_all_fetch_only_sz`: 所有 bpu 模型中指令数据的大小

每个模型文件 header 的格式如下

```
0. typedef struct hbm_info {  
1.     char name[100];  
2.     uint32_t hbm_size;  
3.     uint32_t inst_load_addr;  
4.     uint32_t inst_load_size;  
5.     uint32_t par_load_addr;  
6.     uint32_t par_load_size;  
7.     uint32_t inst_data_offset;  
8.     uint32_t par_data_offset;  
9.     uint32_t signature_offset;  
10.    uint32_t hbm_start_offset;  
11.    uint32_t sig_flag;  
12. }hbm_info_t;
```

- `name`: 模型的名称
- `hbm_size`: 模型文件的大小
- `inst_load_addr`: 指令数据 load 到内存的地址
- `inst_load_size`: 指令数据大小
- `par_load_addr`: 参数数据 load 到内存的地址
- `part_load_size`: 参数数据大小
- `inst_data_offset`: 指令数据的偏移
- `par_data_offset`: 参数数据的偏移
- `hbm_start_offset`: 模型文件的偏移
- `sig_flag`: 签名 flag, 0 表示地平线签名, 1 表示客户签名

bpu image 的布局如图所示:



4.1.2 bpu image 的签名

加密过程由编译器同时提供的 HBDK 完成，加密完成后，将生成一份加密后的模型文件和一份模型文件信息的 json 文件。

json 文件的形式如下：

```
{
  "inst": {
    "bpu_inst_offset": "0x10000",    //指令数据的偏移
    "bpu_inst_size": 65536          //指令数据的大小
  },
  "param": {
    "bpu_param_offset": "0x20000",   //参数数据的偏移
    "bpu_param_size": 65536         //参数数据的大小
  },
  "summary": {
    "aes_iv_crc32": "0xf97160f3",
    "aes_key_crc32": "0xcc58ebb9",
    "cipher": "aes-128-cbc",
    "file_name": "test3_enc.hbm",    //模型文件的名称
    "file_size": 262144,             //模型文件的大小
    "inst_ciphertxt_crc32": "0x789593ac",
    "param_ciphertxt_crc32": "0x231c332e"
  }
}
```

```
}  
}
```

签名由客户完成，使用摘要算法是 sha256，签名算法是 RSA2048，产生 256 字节的签名信息附在原模型文件的后面即可。

在 keymanagement_tools 中提供了一个可参考的签名脚本 bpu_only_sig.sh，支持对多个 HBM 文件进行签名

```
./bpu_only_sig.sh -b -i utils/fallback_key_sets -j test3_enc.hbm.json,test4_enc_hbm.json
```

- -b -i 表示是使用了本地的 key
- -j 表示模型信息 json 文件，可添加多个模型文件，以逗号间隔

执行后，源文件不变，会生成 xxxx-with-sign 文件，表示添加了签名信息的模型文件

./bpu_only_sig.sh 中使用的地平线的公钥私钥信息，若客户使用自己的私钥签名，需将 RSA 公钥交于地平线打包进 SPL 中。

下面是 bpu_only_sig.sh 中对 bpu image 签名的函数，同 uboot/bl31 的签名过程类似，使用 util_auth.sh 脚本对 image 计算 hash，然后进行签名，这里也使用的本地的 key

```
1.pack_mul_hbm()  
2.{  
3.  
4.    name=$(cat $1 | ${JQ_EXEC} .summary.file_name | sed 's/\\/\\/g')  
5.    if [ $name != "null" ]&& [ $name != "" ]; then  
6.        #encrypt bpu image, shall be delete in future #1  
7.        pad_image $name 64  
8.        util_auth.sh -i $name.pad -o $name.tmp -h  
9.  
10.       # sign bpu image hash  
11.       util_auth.sh -i $name.tmp_hash.bin -k spl_rsa_bpu_img_priv.pem -  
12.       o $name.tmp.sign -s  
13.       rm $name.tmp_*  
14.       cat $name.pad > $name-with_sign  
15.       cat $name.tmp.sign >> $name-with_sign  
16.  
17.  
18.}
```

4.1.3 bpu image 的打包

模型文件签名之后，需要对模型文件增加头信息，有两种方法：

PC 端在 keymanagement_tools 中提供了打包 bpu image 的工具 pack_bpu_img.py。

注意：此时要打包的文件是附带了签名的 HBM 文件。例如使用 bpu_only_sig.sh 生成了带签名

的 HBM 文件 xxx-with-sign 文件，应将此文件覆盖原来的 HBM 文件。

```
python3 pack_bpu_img.py -j test3_enc.hbm.json,test4_enc.hbm.json -s 0,1 -a 0x2c940000
```

- -j 模型信息 json 文件，可同时打包多个模型文件，json 文件以逗号间隔，最多可打包 4 份模型文件
- -s 签名标志位，0—表示由地平线签名，1—表示外部客户签名。有多份 HBM 文件时以逗号间隔，并与 json 文件一一对应。外部客户对模型文件签名后，需将公钥交于地平线，打包入 SPL 中。
- -a 模型文件 load 地址

由于模型文件被 load 到指定地址后，MPU 会将这块区域进行保护，只有 BPU 可以访问这块区域，所以这块区域需要提前预留。

默认 DTS 中为 ion 预留了一块区域，使用的是 CMA。起始地址为 0x4000000，大小为 0x2A000000，为了节省 DDR，并未给专门给 BPU 模型预留区域，因此这块区域可能被多个驱动模块使用。

```
ion_reserved: ion_reserved@0x4000000 {  
    compatible = "ion-pool";  
    reg = <0x0 0x4000000 0x0 0x2A000000>;  
    status = "disabled";  
};
```

BPU 尝试预留内存时，由于 CMA 的对齐为 4M，这可能出现 DDR 资源的浪费。因此需要使用 ion_reserved 区域。ion_reserved 默认处于未被使能的状态，我们首先把 CMA 区域替换成 ion_reserved，然后再借用 ion_reserved 的部分区域作为 BPU 预留区域。

通过 UBoot 命令行的方式从 ion 中末尾处截取一段区域作为 BPU 模型的区域：

UBoot 命令设置 BPU 模型区域，10 表示设置模型区域大小为 10M，最大为 64M

```
setenv ion_cma 0  
setenv model_reserved_size 10  
saveenv
```

在执行完上述 UBoot 命令后，模型区域的起始地址是 $0x4000000 + 0x2A000000 - 0xA00000 = 0x2D600000$ ，大小为 0xA00000

在板子上，提供了 hrut_hbm 工具也可以进行添加 hbm 文件的动作

```
hrut_hbm -j test3_enc.hbm.json -s 0 -a 0x2c940000
```

- -j: 模型文件的 json 文件
- -s: 签名的标志
- -a: load bpu 模型的地址，如果 emmc 或 flash 已经有了 bpu 模型，该参数可以不指定

4.2 运行时的保护

模型包括指令和参数两部分，两部分数据被 load 到内存保留的地址后，MPU 会将这两部分数据设置为 BPU 只读，其他模块不可读不可写。所以在 UBoot 中要设置 DTS 为 BPU 模型预留一块区域，以免出现 MPU 设置为 BPU 只读的区域被其他模块申请的情况。

5 EFUSE 的烧写

5.1 efuse 的基本介绍

注意：本章节的 eFUSE 烧写方法仅在系统版本 20210122 及以上支持。

eFUSE 中共有 31bank，每个 bank 有 32bits，bank31 的每个 bit 对应某个 bank 的 lock 标志，置 1 表示此 bank 将不能被烧写。bank7-bank11 的全部 32bit，以及 bank21[0:23]和 bank22[0:5]可供用户烧写。Bank21 可供用户烧写 customer id 和 product id。bank21 的结构如下

比特位	[0:15]	[16:23]
含义	CUSTOMER ID	PRODUCT ID

Bank22 的结构如下

比特位	[0]	[1]	[2]	[3]	[4]	[5]
含义	disable jtag	disable bif sd	disable bif spi	verify uboot	verify bpu model	select key

5.2 efuse 的配置文件

在 build/ddr/efuse 下提供了 efuse_cfg_outside.json 文件，可供用户修改 efuse 的值

```
"outside": {
  "bypass":1,
  "setlock":0,
  "disjtag":0,
  "disbifsd":0,
  "disbifspi":0,
  "veri_uboot":0,
  "veri_bpu":0,
  "sel_key":0,
  "debug_lock":0,
  "customerid":"0x0",
  "productid":"0x0",
  "id_lock":0,
  "normalbank": {
    "bank7":["0x0"],
    "bank8":["0x0"],
    "bank9":["0x0"],
    "bank10":["0x0"],
```

```

        "bank11":["0x0"]
    },
    "securebank": {
        "hash0":["0x0"],
        "hash1":["0x0"],
        "hash2":["0x0"],
        "hash3":["0x0"],
        "hash4":["0x0"],
        "hash5":["0x0"],
        "hash6":["0x0"],
        "hash7":["0x0"]
    },
    "socid":["0x0","0x0", "0x0", "0x0", "0x0","0x0", "0x0", "0x0", "0x0", "0x0",
        "0x0","0x0", "0x0", "0x0", "0x0","0x0", "0x0", "0x0", "0x0", "0x0",
        "0x0","0x0", "0x0", "0x0", "0x0","0x0", "0x0", "0x0", "0x0", "0x0",
        "0x0","0x0", "0x0", "0x0", "0x0","0x0", "0x0", "0x0", "0x0", "0x0",
        "0x0","0x0", "0x0", "0x0", "0x0","0x0", "0x0", "0x0", "0x0", "0x0",
        "0x0","0x0", "0x0", "0x0", "0x0","0x0", "0x0", "0x0", "0x0", "0x0",
        "0x0","0x0", "0x0", "0x0", "0x0","0x0", "0x0", "0x0", "0x0", "0x0",
        "0x0","0x0", "0x0", "0x0", "0x0","0x0", "0x0", "0x0", "0x0", "0x0",
        "0x0","0x0", "0x0", "0x0", "0x0","0x0", "0x0", "0x0", "0x0", "0x0"]
    }
}

```

- **bypass:** 1 表示 eFUSE 烧写不执行，0 表示执行此次 eFUSE 烧写，若要烧写 eFUSE，应置为 0。
- **setlock:** 1 表示写数据后上锁，0 表示不上锁
- **disjtag:** disable jtag， 1 表示 disable。0 表示 enable
- **disbifsd:** disable bif sd， 1 表示 disable。0 表示 enable
- **disbifspi:** disable bifspi， 1 表示 disable。0 表示 enable
- **veri_uboot:** 设置为 1 表示上电时校验 uboot，设置为 0，表示上电不校验 uboot
- **veri_bpu:** 如果 bpu 模型放到了镜像中，设置为 1 表示上电时校验 bpu 模型，设置为 0，表示不校验 bpu 模型
- **sel_key:** key 选择标志位，0 表示使用外部 key，1 表示使用 keybank 中的 key
- **debug_lock:** 1 表示更新 bank22 后上锁，0 表示不上锁
- **customerid:** 供用户烧写 customer id，最大 65535

- productid: 供用户烧写 productid, 最大 255
- id_lock: 写入 customer id/productid 后是否上锁, 上锁后 customer id 和 product id 不能再写入
- normalbank: bank7-bank11 的值
- securebank: uboot public key 的 hash 值
- socid: soc id 白名单, 每一颗 soc 都有个 unique id, 若 id 在这个白名单中, bif/bifspi/jtag 则会打开。

注:

1、jtag/bif sd/bif spi 在 spl 中默认会打开, soc id 白名单优先级最高, 在白名单中则会打开, 若不在, 根据 efuse bank22 判断, 若被置 1, 则关闭相应功能。

2、bif sd 和 bif spi 不能单独关闭, 可以单独打开, jtag 功能不能单独打开, 可以单独关闭。即若打开 jtag 功能, 三个功能就都使能。若关闭 jtag 功能, 可以单独打开 bif sd 或 bif spi

3、setlock 标志是针对 normal bank 数组和 secure bank 数组的整体开关, 设置为 1 时, 若 normal bank/secure bank 的某个 bank 不为 0, 烧写 efuse 后会上锁; bank 值为 0 不会上锁。为了灵活性, 每个 bank 也可以单独设置是否上锁, 只需在 bank 值后加上 "lock" "unlock" 字样即可, 并且这种方式的优先级更高, 也就是只要满足任意一个条件就会上锁: (1) "setlock" 设置为 1, bank 值不为 0; (2) bank 后添加 "lock" 关键字。如下所示, normal bank7 设置为 0, 并上锁, normal bank8 设置为 0x2, 但不上锁, normal bank9/10/11 都将上锁。secure bank 值都为 0, 且不上锁。

```
....  
    "setlock": 1,  
    "normalbank": {  
        "bank7": ["0x0", "lock"],  
        "bank8": ["0x2", "unlock"],  
        "bank9": ["0x3"],  
        "bank10": ["0x4"],  
        "bank11": ["0x5"]  
    },  
    "securebank": {  
        "hash0": ["0x0"],  
        "hash1": ["0x0"],  
        "hash2": ["0x0"],  
        "hash3": ["0x0"],  
        "hash4": ["0x0"],  
        "hash5": ["0x0"],  
        "hash6": ["0x0"],  
        "hash7": ["0x0"]  
    },  
.....
```

4、bank21 包含 customer id 和 product id 两个字段，id_lock 表示更新后是否 lock，lock 后就不能再次写入，所以 customer id 和 product id 建议一次配置完成

5、bank22 包含 disable jtag/disable bifsds/disable bifsps/verify uboot/verify bpu/select key 共计 6 个字段，debug_lock 表示设置后是否 lock bank22，lock 后不能再烧写 bank22，建议上述 6 个字段一次更新完成。

6、验证 uboot 还可以通过 pin 的方式进行调试，GPIO92 作为输入置 1 时，也将校验 uboot，用户在设置 efuse 之前，可以使用该 pin 作为测试方法

编译生成 disk.img 时，-q 表示会将 eFUSE 配置文件打包入 disk.img

```
./build.sh xxx xxx -q
```

烧写完成后，可以在 UBoot 中查看 eFUSE 的烧写情况，且只能查看 normal bank。

```
efuse dump
```

5.3 配置范例

客户若需要校验 uboot，并使用外部 key 的方式，需要将公钥的 hash 烧写到 eFUSE 中

```
{
  "outside": {
    "bypass":0,
    "setlock":1,
    "disjtag":0,
    "disbifsd":0,
    "disbifspi":0,
    "veri_uboot":1,
    "veri_bpu":0,
    "sel_key":0,
    "debug_lock":1,
    "customerid":"0x0",
    "productid":"0x0",
    "id_lock":0,
    .....
    "securebank": {
      "hash0":["0x1234"],
      "hash1":["0x5678"],
      "hash2":["0xabcd"],
      "hash3":["0x1234"],
      "hash4":["0x5678"],
      "hash5":["0xabcd"],
      "hash6":["0x1234"],
      "hash7":["0x5678"]
    }
  },
}
```



```
.....
}
```

- 1、bypass 置 0，表示将烧写 efuse
 - 2、set_lock 置 1，表示对应的烧写 hash 后，将会 lock
 - 3、verify_uboot:置 1，表示 spl 将校验 uboot
 - 4、sel_key: 置 0，表示使用外部 key。
 - 5、debug_lock:置 1，表示 bank22 将会被 lock，也就是下次 disjtag/disbifsd/disbifspi/verify_boot/verify_bpu/sel_key 将不能再次配置
 - 6、hash0--hash7: 公钥对应的 hash，产生公钥 hash 参见 3.5 章节
- 客户若使用 keybank 中的 key 校验 uboot，可参考如下配置

```
{
  "outside": {
    "bypass":0,
    "setlock":0,
    "disjtag":0,
    "disbifsd":0,
    "disbifspi":0,
    "veri_uboot":1,
    "veri_bpu":0,
    "sel_key":1,
    "debug_lock":1,
    "customerid":"0x0",
    "productid":"0x0",
    "id_lock":0,
    ...
  }
}
```

- 1、bypass 置 0，表示将烧写 efuse
- 2、verify_uboot:置 1，表示 spl 将校验 uboot
- 3、sel_key: 置 1，表示使用 keybank 中的 key；公钥还需要交于地平线打包到 spl 中

6 AVB 和 dm-verity

本章节主要介绍 X3J3 平台安全启动的校验流程开源部分。开源部分分为两步：1. UBoot 校验 Linux 内核，基于 Android Verified Boot (AVB) 移植并实现；2. Linux 内核使用 dm-verity 校验根文件系统，基于 Linux 内核的 Device Mapper 及 DM-verity (DM 指代 Device Mapper)

模块实现。

6.1 Build 系统添加 AVB 以及 DM-Verity 校验

在编译系统中，当 Secure 镜像被选择时自动调用对应脚本给 boot.img 以及 system.img 添加校验信息。boot.img 以及 system.img 的校验元数据均储存在 vbmeta.img 中，在启动后显示为 vbmeta 分区。涉及脚本如下均位于下方路径：

```
build/tools/avbtools/
```

其中，"build_avb_images.sh"为最上层脚本，负责整体镜像的生成。"build_avb_images"脚本执行过程中会自动生成 avb 镜像制作所需配置文件，储存为"out/system_image_info.txt"，生成示例如下：

```
mount_point=
fs_type=ext4
partition_size=156762112
partition_name=system
uuid=da594c53-9beb-f85c-85c5-cedf76546f7a
verity=true
ext_mkuserimg=make_ext4fs
avb_avbtool=/git/dinggao.pan/xj3/build/tools/avbtools/avbtool
verity_key=/git/dinggao.pan/xj3/build/tools/avbtools/keys/shared.priv.pem
verity_block_device=/dev/mmcblk0p9
skip_fsck=true
verity_signer_cmd=verity/verity_signer
```

如需修改各项参数，请对"build_avb_images.sh"中，"create_config_file"函数进行修改。其中，校验加密所使用的密钥，目前储存于上方"verity_key"变量中，请根据项目需求，自行替换密钥文件或者修改密钥所处路径。

如果有额外分区需要使用 dm-verity 进行验证，也可参照"build_boot_vbmeta.sh"或者"build_system_vbmeta.sh"进行分区镜像制作，并修改"build_vbmeta.sh"将对应分区校验信息添加至 vbmeta.img 中。

具体 AVB 以及 DM-Verity 介绍，请参考安卓以及 Linux 内核官方文档。

6.2 UBoot 校验 Linux 内核

在安全启动流程中，Linux 内核文件（包括 dtb.img – 目前包含所有可用 dtb 文件）会由 UBoot 做整体校验。涉及文件：

```
uboot/cmd/avb.c
uboot/common/avb_verify.c
```

```
uboot/common/bootm.c  
uboot/lib/libavb/
```

UBoot 命令中的"avb_verify"封装了"avb init"以及"avb verify"。"avb init"会初始化并寻找 AVB 所需镜像所处的储存介质。"avb verify"则负责将"vbmeta"以及"boot"分区内容读取出来并校验，具体实现请参考上述文件。

目前 DM-verity 仅在 eMMC 上实现，Flash(nand 或者 nor)均没有通过 mtd-block 虚拟层制作 DM，没有实现 DM-verity。在 eMMC 启动流程中，如果开启了 AVB 以及 DM-verity，则 UBoot 会在校验成功后，向 bootargs 添加 DM-verity 所需元数据以供 Kernel 初始化 DM-verity 使用。

6.3 Linux 内核使用 DM-verity 校验根文件系统

在 UBoot 校验 Linux 内核成功并向 Linux 内核提供 DM-verity 初始化所需元数据后，Linux 内核会初始化 DM-verity 并在使用根文件系统过程中对每一个读取出来的擦写块进行校验。有区别与普通启动，Secure 启动使能 DM-verity 后，根文件系统不再直接从 mmcblk 分区挂载，改为从 DM 设备(一般为/dev/dm-0)挂载。