# i.MX 6 Linux High Assurance Boot (HAB) User's Guide

Contents

1 Overview....1
2 Mechanism....1
3 Implementation....3
4 Test Procedure....5
5 Dynamically Allocate HAB Data....11

## 1 Overview

Contemporary electronic gadgets often have opportunities to do system upgrade. The upgrade is so convenient that the end user can retrieve the updated image and do it themselves. But this also opens a back-door for malware writers. They may provide updated image which contains virus and rootkit. Once such system is upgraded and used, bad consequences can be expected, such as theft of critical personal information, anonymous monitor on personal activities, etc.

For security consideration, it is necessary that the hardware have some mechanism to ensure that the software it is running can be trusted. Freescale i.MX6 series chip provides **High Assurance Boot (HAB)** feature which meets such a requirement. OEM can utilize it to make their product reject any system image which is not authorized for running.

## 2 Mechanism

An asymmetric encryption is adopted to implement the HAB feature. The OEM can use an utility provided by Freescale to generate private key and corresponding public key pairs. For any system image they want to release, the private key is used to do the encryption. This encryption generates a unique identifier for the image which is called a **certificate**. The public key is also attached to the image. Before applying the

© 2013 Freescale Semiconductor, Inc.

freescale™

new system, the public key is used to decrypt the certificate. Then, a comparison is performed to check if the certificate and the image match. The image is considered "trusted" only if a match occurs. Otherwise the image is deemed as "unsafe" and will not be permitted to be loaded and run. This process is called **authentication**.

A hacker can only have access to the public key. Per the property of asymmetric encryption, private key, which the OEM uses, cannot be deduced from that. Without the private key, the hacker cannot attach a valid certificate for his malicious system image. HAB will reject it in a very early stage.

The OEM will burn the digest (hash) of the public key to the eFuses of i.MX6 chip. Once burned, it cannot be modified. This prevents the possibility that the hacker use another pair of private and public key to cheat.

Figures below provide a more precise illustration.

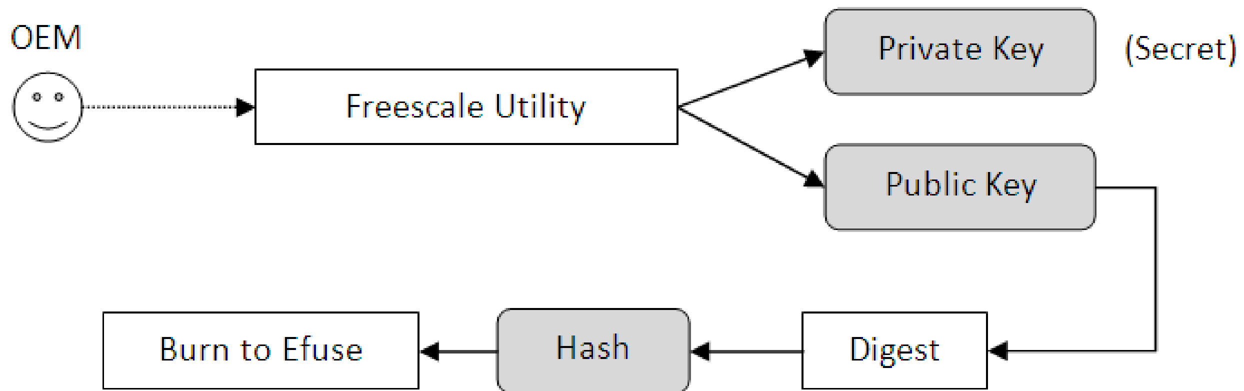


**Figure 1. Generation of Public/Private Key**

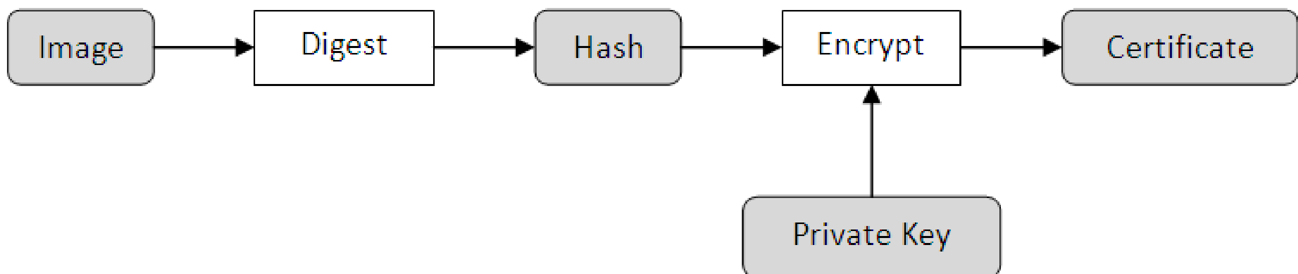


**Figure 2. Certificate Generation**

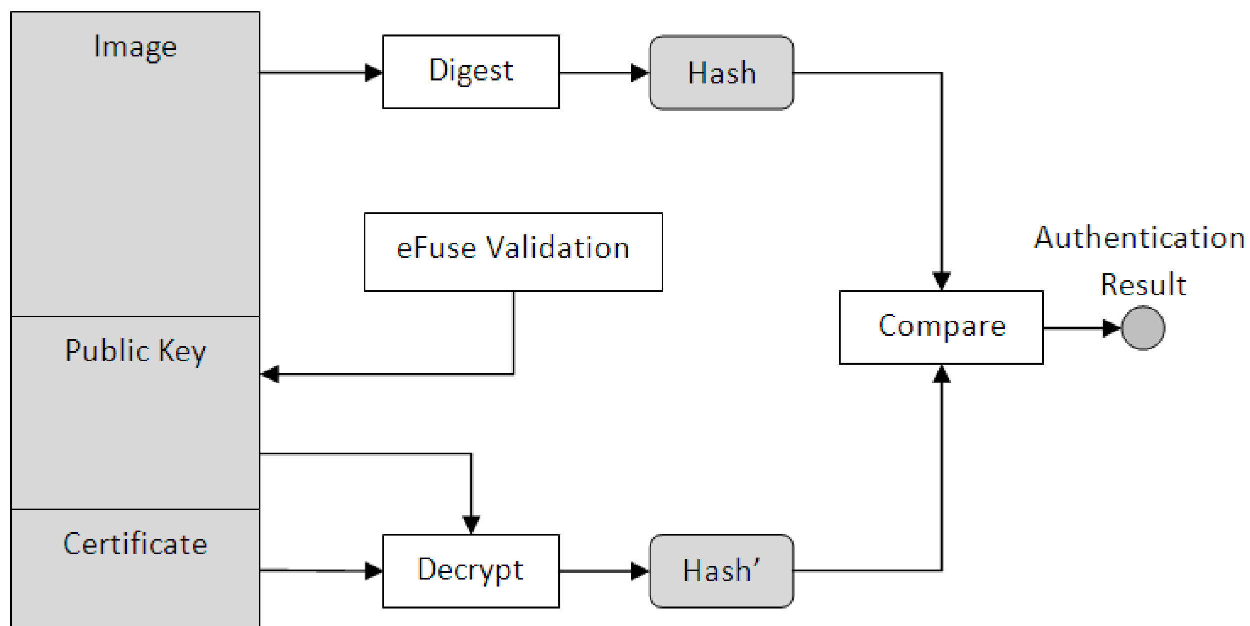


**Figure 3. Authenticate process (Run time)**

# 3 Implementation

The first stage of HAB is the authentication of U-Boot. A CST tool is used to generate the **CSF data**, which includes public key, certificate, and instruction of authentication process. The CSF data is attached to the original u-boot.bin. The process is called **Signature**.

The IVT should be modified to contain a pointer to the CSF data. The original u-boot.bin image size is around 0x27000 to 0x28000. For convenience, we first extend its size to 0x2F000 (with fill 0x5A). Then concatenate it with the CSF data. The combined image is again extended to a fixed length (0x31000), which is used as the IVT image size parameter.

The new memory layout is shown in the following image.

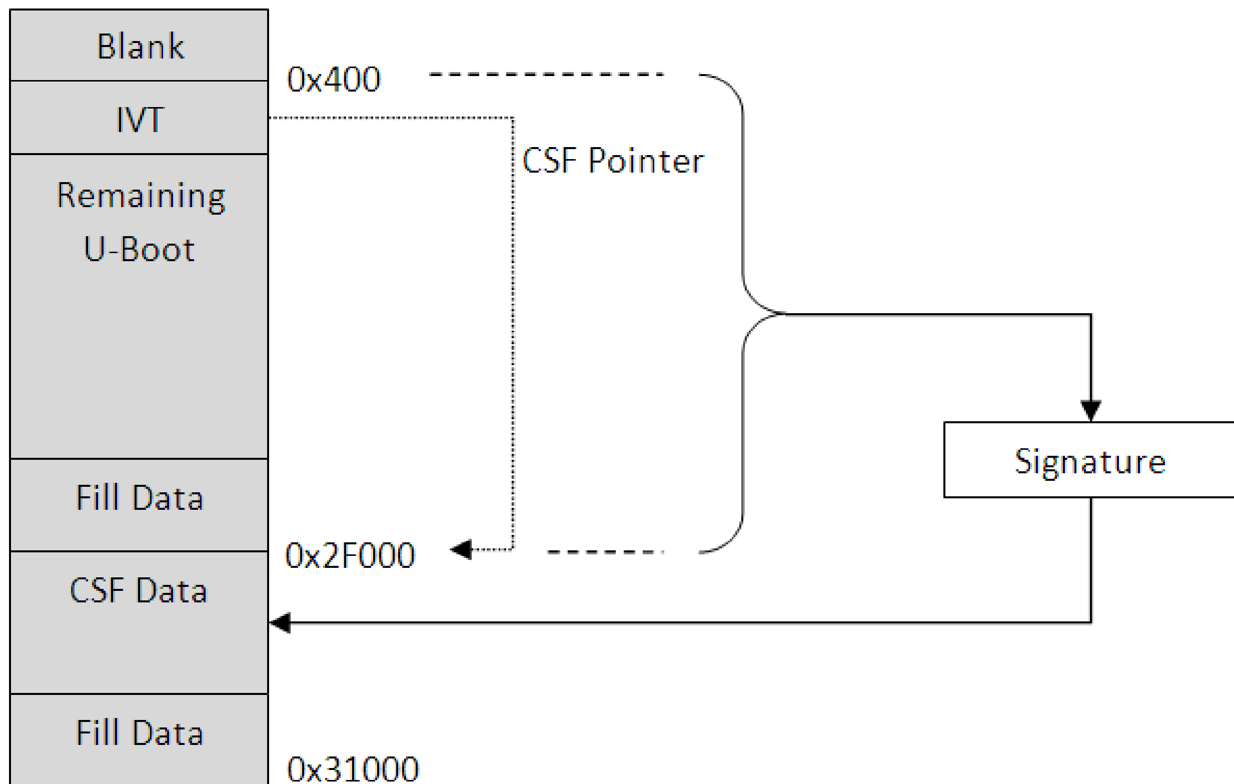


**Figure 4. HAB U-Boot Layout**

HAB APIs are ROM implemented. The entry table is located in a fixed location in the ROM. We export them so that we can have some information about the secure boot process and utilize it in a subsequent boot phase. For convenience, some wrapper APIs are implemented based on the HAB APIs.

- `get_hab_status` is used to dump information of authentication result.

- `authenticate_image` is used by U-Uoot to authenticate uImage.

For security hardware to work, CAAM related clocks (CG0[4~6]) must be open. In the original U-Boot, these clocks are closed by default. `hab_caam_clock_enable` and `hab_caam_clock_disable` are created to open and close them.

The generation of CSF data is not in the scope of this document. CST tool will be used for this purpose. The tool is introduced in "HAB Code-Signing Tool User's Guide", which is included in the CST tool package.

The second stage is the authentication of uImage by U-Boot. `authenticate_image` is called by U-Boot to verify uImage when executing `bootm`. At this time, the uImage, together with its CSF data, should have been located in DDR. The new uImage layout is shown in Figure 5:

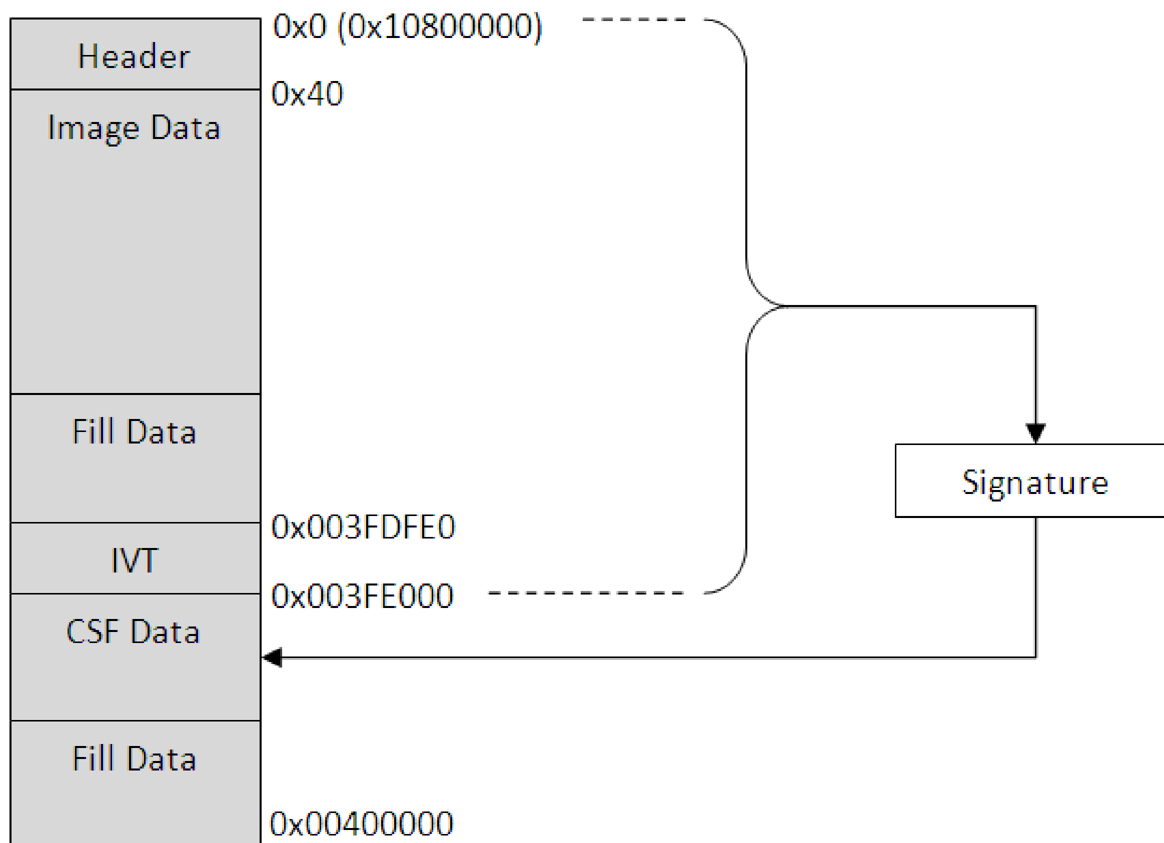


**Figure 5. HAB uImage Layout**

The uImage will be read from boot medium to DDR location 0x10800000. The CSF data pointer in IVT should consider this extra offset.

# 4 Test Procedure

The following procedures suppose you have a PC running Ubuntu and a MX6Q_ARM2 board with MX6Q TO1.1 chip mounted. For a different environment, the steps may differ.

1. Retrieve the CST Tool.

   http://www.freescale.com/webapp/sps/download/mod_download.jsp?colCode=IMX_CST_TOOL&appType=moderatedWithoutFAE&fpsp=1&WT_TYPE=Initialization/Boot/Device%20Driver%20Code%20Generation&WT_VENDOR=FREESCALE&WT_FILE_FORMAT=zip&WT_ASSET=Downloads&sr=15&Parent_nodeId=1255017883505753547804&Parent_pageType=product

   **NOTE**

   - It may take several days before Freescale sends you the download link.

2. Unzip "BLN_CST_MAIN_01.00.00.zip".
3. chmod u+x linux/* keys/*
4. fromdos keys/*.sh
5. cd keys

- Create a text file called serial, which contains 8 digits.
- Create a text file called key_pass.txt, which contains two lines of identical text, such as "cst_test".
- ./hab4_pki_tree.sh
- For question prompt, enter "n", "2048", "10", "4" one by one.
- This script will generate private key and public key pairs in the working directory.

6. cd ../crts

```
../linux/srktool -h 4 -t SRK_1_2_3_4_table.bin -e SRK_1_2_3_4_fuse.bin -d sha256 -c
./SRK1_sha256_2048_65537_v3_ca_crt.pem,./SRK2_sha256_2048_65537_v3_ca_crt.pem,./
SRK3_sha256_2
048_65537_v3_ca_crt.pem,./SRK4_sha256_2048_65537_v3_ca_crt.pem -f 1
```

- This command will generate root public key file "SRK_1_2_3_4_table.bin" and its corresponding hash "SRK_1_2_3_4_fuse.bin". The content of the latter will be later on burned to chip eFuse.

**NOTE**

Don't leave space between the pem file names. Otherwise the generated SRK table and fuse file will not be correct.

7. Create a "u-boot" directory in "BLN_CST_MAIN_01.00.00", make an "u-boot.csf" file in it

------------- file content begin -------------

```
[Header]
    Version = 4.0
    Security Configuration = Open
    Hash Algorithm = sha256
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS
[Install SRK]
    File = "../crts/SRK_1_2_3_4_table.bin"
    Source index = 0
[Install CSFK]
    File = "../crts/CSF1_1_sha256_2048_65537_v3_usr_crt.pem"
[Authenticate CSF]
[Install Key]
    Verification index = 0
    Target index = 2
    File = "../crts/IMG1_1_sha256_2048_65537_v3_usr_crt.pem"
# Sign padded u-boot starting at the IVT through to the end with
# length = 0x2F000 (padded u-boot length) - 0x400 (IVT offset) = 0x2EC00
# This covers the essential parts: IVT, boot data and DCD.
# Blocks have the following definition:
#    Image block start address on i.MX, Offset from start of image file,
#    Length of block in bytes, image data file
[Authenticate Data]
    Verification index = 2
    Blocks = 0x27800400 0x400 0x2EC00 "u-boot-pad.bin"
```

------------- file content end -------------

ROM code copies U-Boot to DDR location 0x27800000, which is added with 0x400 (the initiated blank area size in U-Boot image) to form the authenticated start address 0x27800400. 0x400 and 0x2EC00 is the start address and the length of data to be authenticated in the binary file.

8. Create a secure U-Boot image generation script "habimagegen" in "u-boot" directory

------------- file content begin -------------

```
#! /bin/bash
echo "extend u-boot to 0x2F000..."
objcopy -I binary -O binary --pad-to 0x2f000 --gap-fill=0x5A u-boot.bin u-boot-pad.bin

echo "generate csf data..."
../linux/cst --o u-boot_csf.bin < u-boot.csf
```

```
echo "merge image and csf data..."
cat u-boot-pad.bin u-boot_csf.bin > u-boot-signed.bin

echo "extend final image to 0x31000..."
objcopy -I binary -O binary --pad-to 0x31000 --gap-fill=0x5A u-boot-signed.bin
u-boot-signed-pad.bin

echo "u-boot-signed-pad.bin is ready"
```

------------- file content end -------------

9. chmod u+x habimagegen
10. Build secure U-Boot. The building process is the same as generating normal U-Boot.

**NOTE**

Make sure the macro "CONFIG_SECURE_BOOT" is defined in "./include/configs/mx6q_arm2.h"

11. Copy "u-boot.bin" to "BLN_CST_MAIN_01.00.00/u-boot" directory.
12. ./habimagegen

This will create certified U-Boot image "u-boot-signed-pad.bin"

**NOTE**

Step 7 to step 12 generate secure U-Boot image using a static HAB DATA allocation method. It is only used to demonstrate the signature process. For practical usage, a dynamic HAB DATA allocation method should be employed. Please refer to Dynamically Allocate HAB Data for details.

13. Create an "uImage" directory in "BLN_CST_MAIN_01.00.00"

Make "habUimagegen" and "genIVT" script in it. An "uImage.csf" should also be created.

File genIVT

------------- file content begin -------------

```
#! /usr/bin/perl -w
use strict;
open(my $out, '>:raw', 'ivt.bin') or die "Unable to open: $!";
print $out pack("V", 0x402000D1); # Signature
print $out pack("V", 0x10801000); # Jump Location
print $out pack("V", 0x0);       # Reserved
print $out pack("V", 0x0);       # DCD pointer
print $out pack("V", 0x0);       # Boot Data
print $out pack("V", 0x10BFDFE0); # Self Pointer
print $out pack("V", 0x10BFE000); # CSF Pointer
print $out pack("V", 0x0);       # Reserved
close($out);
```

------------- file content end -------------

0x10BFDFE0 is the IVT self address after uImage is copied to DDR. 0x10BFE000 is where the CSF data begins. 0x10801000 is jump location. However, U-boot has its own mechanism to jump into the kernel so this jump location is not actually being used. The HAB ROM code requires that it must be an address inside the authentication range, in our case, between 0x10800000 and 0x10BFE000.

File habUimagegen

------------- file content begin -------------

```
#! /bin/bash
echo "extend uImage to 0x3FDFE0..."
objcopy -I binary -O binary --pad-to 0x3fdfe0 --gap-fill=0x5A uImage uImage-pad.bin

echo "generate IVT"
./genIVT
```

```
echo "attach IVT..."
cat uImage-pad.bin ivt.bin > uImage-pad-ivt.bin

echo "generate csf data..."
../linux/cst --o uImage_csf.bin < uImage.csf

echo "merge image and csf data..."
cat uImage-pad-ivt.bin uImage_csf.bin > uImage-signed.bin

echo "extend final image to 0x400000..."
objcopy -I binary -O binary --pad-to 0x400000 --gap-fill=0x5A uImage-signed.bin
uImage-signed-pad.bin
```

------------- file content end -------------

File uImage.csf

------------- file content begin -------------

```
[Header]
    Version = 4.0
    Security Configuration = Open
    Hash Algorithm = sha256
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS
[Install SRK]
    File = "../crts/SRK_1_2_3_4_table.bin"
    Source index = 0
[Install CSFK]
    File = "../crts/CSF1_1_sha256_2048_65537_v3_usr_crt.pem"
[Authenticate CSF]
[Install Key]
    Verification index = 0
    Target index = 2
    File = "../crts/IMG1_1_sha256_2048_65537_v3_usr_crt.pem"
# Sign padded uImage start at address 0x10800000
# length = 0x3FE0000
# This covers the essential parts: original uImage and the attached IVT
# Blocks have the following definition:
#     Image block start address on i.MX, Offset from start of image file,
#     Length of block in bytes, image data file [Authenticate Data]
    Verification index = 2
    Blocks = 0x10800000 0x0 0x003FE000 "uImage-pad-ivt.bin"
```

------------- file content end -------------

0x10800000 is where the uImage starts in DDR. 0x0 and 0x003FE000 is the start address and length to be authenticated in the binary.

14. chmod u+x habUimagegen genIVT
15. Build uImage as you normally would and copy it to "BLN_CST_MAIN_01.00.00/uImage" directory.
16. ./habUimagegen

    This will create certified uImage "uImage-signed-pad.bin"

**NOTE**

Step 13 to step 16 generate secure uImage using a static HAB DATA allocation method. It is only used to demonstrate the signature process. For practical usage, a dynamic HAB DATA allocation method should be employed. Please refer to Dynamically Allocate HAB Data for details.

17. Burn the "u-boot-signed-pad.bin" and "uImage-signed-pad.bin" to SD card.

    The kernel should start up successfully by now.

    At this point the whole HAB chain is not actually enabled, because the default chip secure config is "open" which ignores any authentication error and continues the boot process. We will enable it in the next step.

18. Burn the chip fuse to enable HAB boot

**NOTE**

!!! CAUTION!!! Make sure you perform all the steps correctly. Otherwise you may damage a chip permanently. It is strongly recommended that a socket board is used, so in the worst case scenario, we can change a chip without totally damaging the board.

- Burn the SRK table. Take this fuse file as example (**Remember, you may have totally different fuse value**).
  - Dump the content of the eFuse file

    ```
    >
    hexdump -e '/4 "0x"' -e '/4  "%X""\n"' SRK_1_2_3_4_fuse.bin
    >
    0xFDF28547
    0x270D6AC6
    0xEE44AD7B
    0x58B0724
    0x49DA1948
    0xB4374A3F
    0xFFEFED48
    0x4247C04F
    ```

    They will be burned to chip with kernel utilities.

  - Normal boot into kernel.
  - cd /sys/fsl_otp
  - Use the 8 dwords generated in last step and burn them to SRK fuse one by one.

    ```
    >
    echo 0xfdf28547 > HW_OCOTP_SRK0
    echo 0x270d6ac6 > HW_OCOTP_SRK1
    echo 0xee44ad7b > HW_OCOTP_SRK2
    echo 0x058b0724 > HW_OCOTP_SRK3
    echo 0x49da1948 > HW_OCOTP_SRK4
    echo 0xb4374a3f > HW_OCOTP_SRK5
    echo 0xffefed48 > HW_OCOTP_SRK6
    echo 0x4247c04f > HW_OCOTP_SRK7
    ```

  - Verify that the correct value has been burned.

    ```
    >
    ls | grep "SRK.$" | xargs cat
    ```

- Burn OTPMK. These fuse values are necessary to enable the hardware secure logic in the chip.

  ```
  >
  echo 0x975b69a7 > HW_OCOTP_OTPMK0
  echo 0xafae0b5d > HW_OCOTP_OTPMK1
  echo 0x6f780499 > HW_OCOTP_OTPMK2
  echo 0x3dda7a47 > HW_OCOTP_OTPMK3
  echo 0x76fcba3c > HW_OCOTP_OTPMK4
  echo 0x6d5c9ef6 > HW_OCOTP_OTPMK5
  echo 0xb166b40a > HW_OCOTP_OTPMK6
  echo 0x8f449c5d > HW_OCOTP_OTPMK7
  ```

- Turn on chip HAB

  **NOTE**

  !!! CAUTION !!! Make sure you perform all the steps correctly. After this step, there will be no way back, and a chip may permanently be bricked.

  - Turn on RNG_TRIM

    ```
    >
    echo 0x00040000 > HW_OCOTP_MEM0
    ```

  - Put SEC_CONFIG to close (turn on chip security).

```
>
echo 0x2 > HW_OCOTP_CFG5
```

19. Boot the board again, the security patch should work now.

    Successful authentication will give the following dump message in U-Boot load:

    ```
    ------------------------

    Boot Device: SD
    HAB Configuration: 0xcc, HAB State: 0x99
    No HAB Events Found!
    I2C:   ready

    ------------------------
    ```

    in uImage load

    ```
    ------------------------

    Verifying Checksum ... OK

    Authenticate uImage from DDR location 0x10800000...

    HAB Configuration: 0xcc, HAB State: 0x99
    No HAB Events Found!

    Loading Kernel Image ... OK

    ------------------------
    ```

    "No HAB Events Found!" means no problem is found in authentication and the image can be trusted.

20. Do a corruption experiment to verify that HAB actually works.
    - Corrupt u-boot-signed-pad.bin

      Make a backup first. Use hex editor tools, such as ghex2, to open the u-boot-signed-pad.bin, and go to some byte before 0x2F000. The value should be "0x5A". Change it to something else and save it. Now the U-Boot has been changed, but the certificate does not change accordingly. A mismatch will occur during the authentication.

    - Flash the corrupted u-boot-signed-pad.bin to SD card, and try to boot the board again. This time, ROM code will meet an authentication failure and will not jump into U-Boot. So there will be nothing shown on the terminal.
    - Change back to the original u-boot-signed-pad.bin.
    - Do the same thing to uImage-signed-pad.bin corrupt value before location "0x003F_DFE0".
    - Flash the corrupted uImage to SD card, and try to boot it. U-Boot will find an authentication fail and break the boot process. Some dump information like the following will show in the terminal.

      ```
      ------------------------

      Verifying Checksum ... OK
      Authenticate uImage from DDR location 0x10800000...
      HAB Configuration: 0xcc, HAB State: 0x99
      --------- HAB Event 1 -----------------
      event data:
              0xdb 0x00 0x1c 0x41 0x33 0x18 0xc0 0x00
              0xca 0x00 0x14 0x00 0x02 0xc5 0x00 0x00
              0x00 0x00 0x0d 0x34 0x10 0x80 0x00 0x00
              0x00 0x3f 0xe0 0x00
      Authenticate UImage Fail, Please check
      MX6Q ARM2 U-Boot >

      ------------------------
      ```

      An event data is dumped out. This means an authentication failure.

    - Change back to uncorrupted uImage. The boot will work again.

      After step 1 through step 20, we can verify that the secure boot works fine.

# 5 Dynamically Allocate HAB Data

The implementation described earlier has some limitations. If you take U-Boot as example, an important assumption is that the original U-Boot size is less than 0x2F000. On this premise, we can statically allocate **HAB Data** to 0x2F000 without causing any problem. However, when this assumption doesn't hold, the static way is not a good idea.

The **HAB Data** address is determined by the linker script file, `u-boot.lds`. Using the static way, this file has the following settings:

```
...
. = ALIGN(4);
_end_of_copy = .;
. = TEXT_BASE + 0x2F000;
__hab_data = .;
. = . + 0x2000;
__hab_data_end = .;
__bss_start = .;
...
```

The location counter is first put to a fixed location `TEXT_BASE + 0x2F000`, from where the `__hab_data` begins. When the original U-Boot size is larger than 0x2F000, this will cause overlap between **HAB Data** and the original U-Boot sections, causing the linkage fail.

To avoid this issue, we turn to use a dynamic way of putting the **HAB Data**. With this method, the `u-boot.lds` is changed to the following:

```
...
. = ALIGN(4);
_end_of_copy = .;
. = ALIGN(0x1000)
__hab_data = .;
. = . + 0x2000;
__hab_data_end = .;
__bss_start = .;
...
```

Instead of statically placing the location counter, an `ALIGN` statement is used to put location counter to the next 0x1000 alignment address after all U-Bboot sections, preventing the overlap introduced by static address assignment.

With this change, the U-Boot layout can be revised as shown in the figure below.

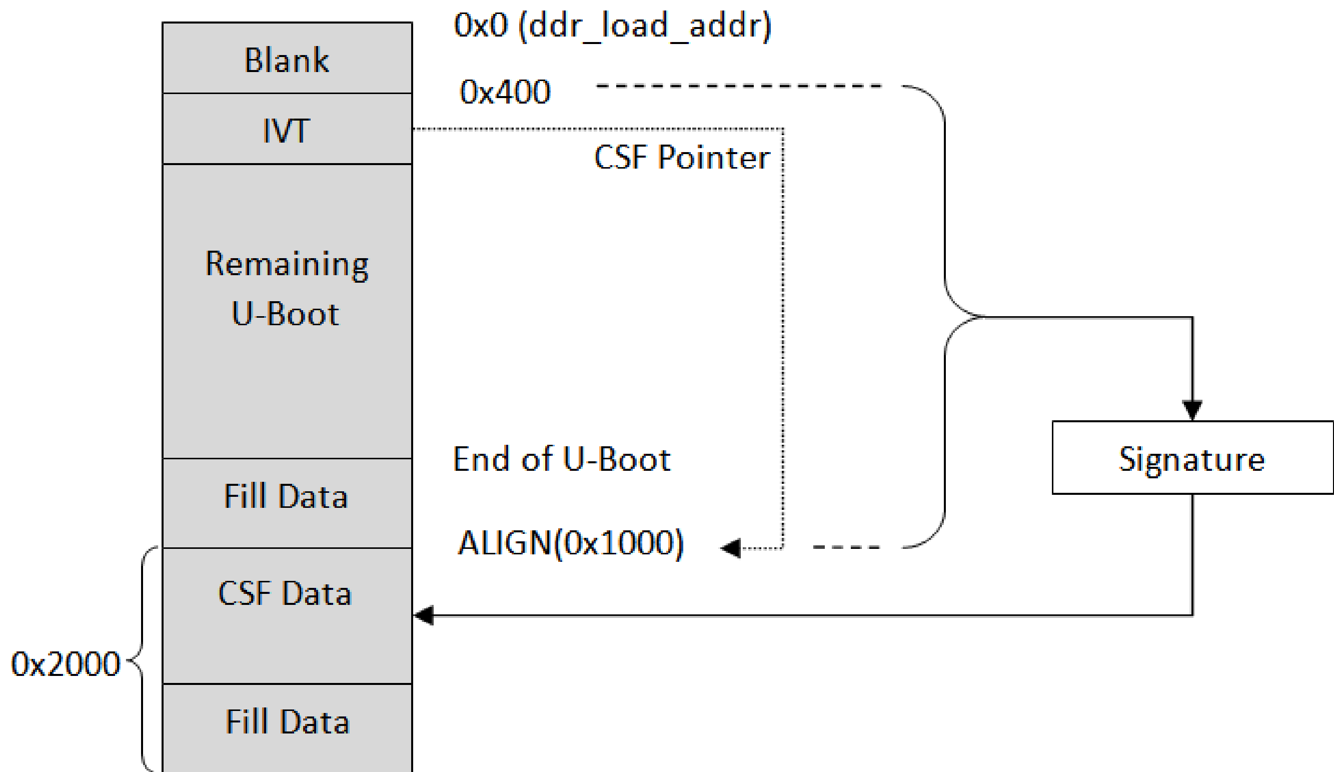


**Figure 6. HAB U-Boot Layout - Dynamic Way**

Similarly, uImage layout can be revised as shown in the figure below.

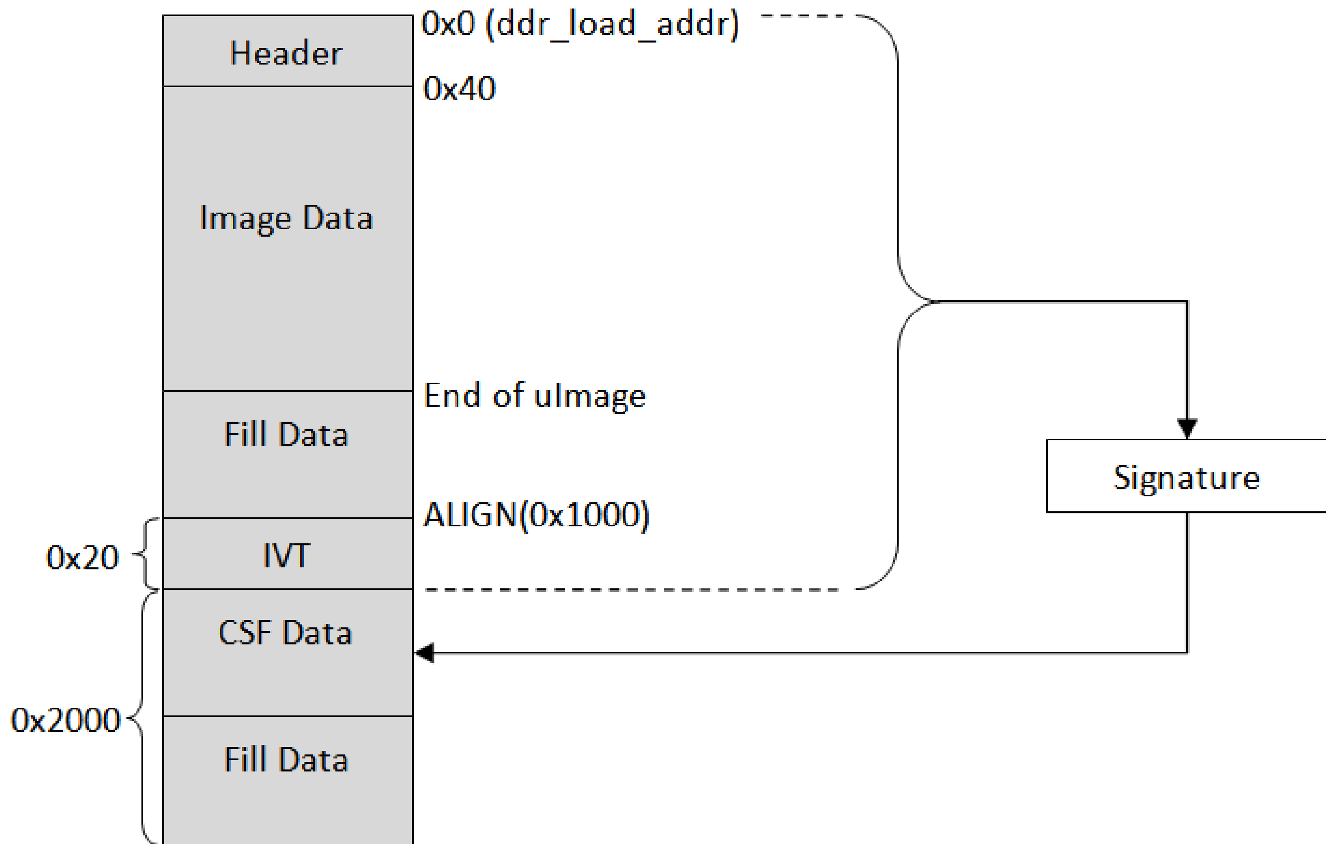


**Figure 7. HAB uImage Layout – Dynamic Way**

With these layout changes, all the CSF configuration file and signature scripts should be changed accordingly. We have made this procedure automatic with a set of file templates and scripts. They make the generation of secure U-Boot and uImage quite simple. All you need to provide is the ddr location where the U-Boot and uImage is located. All other signature works are performed automatically.

1. Generation of secure U-Boot

   ```
   > ./mk_secure_uboot 0x27800000
   ```

   0x27800000 is the ddr location where the U-Boot image is copied to by ROM code. u-boot-signed-pad.bin will be generated, which is the U-Boot image with signature.

2. Generation of secure uImage

   ```
   > ./mk_secure_uimage 0x10800000
   ```

   0x10800000 is the ddr location where the uImage is copied to by U-Boot code. uImage-signed-pad.bin will be generated, which is the uImage with signature.

Please contact Freescale to get these scripts.

*How to Reach Us:*

**Home Page:**
freescale.com

**Web Support:**
freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer’s technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale and the Freescale logo are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. All other product or service names are the property of their respective owners. ARM is the registered trademark of ARM Limited. ARM Cortex™-A9 is a trademark of ARM Limited.

© Freescale Semiconductor, Inc., 2013. All rights reserved.

Document Number: IMX6HABUG
Rev. L3.0.35_1.1.0
01/2013