Introduction

The Secure Boot and Secure Firmware Update (SBSFU) solution allows the update of the STM32 microcontroller built-in program with new firmware versions, adding new features, and correcting potential issues. The update process is performed in a secure way to prevent unauthorized update and access to confidential on-device data such as secret code and firmware encryption key.

In addition, Secure Boot (Root of Trust services) checks and activates the STM32 security mechanisms, and verifies the authenticity and integrity of user application code before every execution to ensure that invalid or malicious code cannot be run.

The X-CUBE-SBSFU user manual (UM2262) explains how to get started with X-CUBE-SBSFU and details SBSFU functionalities.

This application note describes how to adapt X-CUBE-SBSFU and integrate it with the user’s application; It answers such questions as:

- How to port X-CUBE-SBSFU onto another board?
- How to tune the X-CUBE-SBSFU configuration to fit the user’s needs?
- How to generate a new firmware encryption key?
- How to debug X-CUBE-SBSFU?
- How to adapt SBSFU?
- How to adapt the user’s application?

Note: Throughout this application note, the IAR™ EWARM IDE is used as an example to provide guidelines for project configuration.

Throughout this document, Secure Boot and Secure Firmware Update applications are referred to as SBSFU.
Contents

1 General information .................................................. 6
2 Related documents ....................................................... 7
3 Porting X-CUBE-SBSFU onto another board ......................... 8
  3.1 Hardware adaptation ............................................... 8
  3.2 Memory mapping definition ....................................... 9
  3.2.1 SBSFU region definition parameters .......................... 12
  3.2.2 Firmware image slot definition parameters .................. 13
  3.2.3 Project specific linker files ................................... 14
4 SBSFU configuration .................................................... 16
  4.1 Features to be configured ........................................ 16
  4.2 Cryptographic scheme selection .................................. 17
  4.3 Security configuration ............................................. 17
  4.4 Development or production mode configuration ................. 19
5 Generating cryptographic key .......................................... 21
  5.1 Generating a new firmware AES encryption key .................. 21
  5.2 Generating a new public/private ECDSA pair of keys for firmware verification ....................................... 21
6 Tips for debugging ..................................................... 23
  6.1 Compiler optimizations level ..................................... 23
  6.2 Memory mapping adaptation ...................................... 23
  6.3 Debugging SECoreBin .............................................. 25
7 Adapting SBSFU .......................................................... 26
  7.1 Implementing a new cryptographic scheme for SBSFU .......... 26
  7.2 Optimizing memory mapping ...................................... 27
8 Adapting the user application ......................................... 29
  8.1 How to make an application SBSFU compatible .................. 29
  8.2 Use of the Flash memory to store user data .................... 31
8.3 Changing the firmware download function in the user application . . . . . . 32

9 Revision history  ......................................................... 33
## List of tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1.</td>
<td>List of acronyms</td>
<td>6</td>
</tr>
<tr>
<td>Table 2.</td>
<td>List of terms</td>
<td>6</td>
</tr>
<tr>
<td>Table 3.</td>
<td>SBSFU code size reduction</td>
<td>27</td>
</tr>
<tr>
<td>Table 4.</td>
<td>Document revision history</td>
<td>33</td>
</tr>
</tbody>
</table>
List of figures

Figure 1. SBSFU project structure ......................................................... 8
Figure 2. Memory mapping example (NUCLEO-L476RG) .............................. 9
Figure 3. Linker file architecture .......................................................... 10
Figure 4. Mapping constraints with MPU isolation (NUCLEO-G071RB example) .... 11
Figure 5. SBSFU regions (mapping_sbsfu.icf from NUCLEO-L476RG) ............... 12
Figure 6. Firmware image slot definitions (mapping_fwimg.icf from NUCLEO-L476RG) 13
Figure 7. SECOREBin specific linker file .................................................. 14
Figure 8. SBSFU specific linker file ......................................................... 14
Figure 9. UserApp specific linker file (NUCLEO-L476RG example) .................... 15
Figure 10. SBSFU configuration .............................................................. 16
Figure 11. Switching the cryptographic scheme ......................................... 17
Figure 12. STM32L4 Series security configuration (app_sfu.h) ....................... 18
Figure 13. STM32F4 Series and STM32F7 Series security configuration (app_sfu.h) 18
Figure 14. STM32G0 Series security configuration (app_sfu.h) ....................... 19
Figure 15. Option Bytes management ...................................................... 20
Figure 16. New firmware encryption key ................................................... 21
Figure 17. New private/public keys ......................................................... 22
Figure 18. Compiler optimizations .......................................................... 23
Figure 19. Memory mapping adaptations ................................................... 24
Figure 20. Checking the WRP protection .................................................... 24
Figure 21. Debugging inside SECOREBin ................................................... 25
Figure 22. User’s own cryptographic scheme implementation .......................... 26
Figure 23. Example of memory mapping optimization on NUCLEO-G071RB – 2 images 28
Figure 24. Example of memory mapping optimization on NUCLEO-G071RB – 1 image 28
Figure 25. Vector table position update (NUCLEO-L476RG example) ............... 29
Figure 26. IDE adaptations ................................................................. 30
Figure 27. Free Flash pages (example of NUCLEO-L476RG) ......................... 31
Figure 28. UserApp firmware download overview ....................................... 32
1 General information

Table 1 and Table 2 present the definitions of acronyms and terms that are relevant for a better understanding of this document.

Table 1. List of acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Advanced encryption standard</td>
</tr>
<tr>
<td>DAP</td>
<td>Debug access port</td>
</tr>
<tr>
<td>ECDSA</td>
<td>Elliptic curve digital signature algorithm</td>
</tr>
<tr>
<td>GCM</td>
<td>AES Galois/counter mode</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware abstraction layer</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated development environment</td>
</tr>
<tr>
<td>FWALL</td>
<td>Firewall</td>
</tr>
<tr>
<td>PEM</td>
<td>Privacy enhanced mail</td>
</tr>
<tr>
<td>PCROP</td>
<td>Proprietary code read out protection</td>
</tr>
<tr>
<td>RDP</td>
<td>Readout device protection</td>
</tr>
<tr>
<td>SB</td>
<td>Secure Boot</td>
</tr>
<tr>
<td>SE</td>
<td>Secure Engine</td>
</tr>
<tr>
<td>SFU</td>
<td>Secure Firmware Update</td>
</tr>
<tr>
<td>SBSFU</td>
<td>Secure Boot and Secure Firmware Update</td>
</tr>
<tr>
<td>UART</td>
<td>Universal asynchronous receiver/transmitter</td>
</tr>
<tr>
<td>WRP</td>
<td>Write protection</td>
</tr>
</tbody>
</table>

Table 2. List of terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firmware image</td>
<td>A binary image (executable) run by the device as user application.</td>
</tr>
<tr>
<td>Firmware header</td>
<td>Bundle of meta-data describing the firmware image to be installed. It contains firmware information and cryptographic information.</td>
</tr>
<tr>
<td>mbedTLS</td>
<td>mbed implementation of the TLS and SSL protocols and the respective cryptographic algorithms.</td>
</tr>
<tr>
<td>sfb file</td>
<td>Binary file packing the firmware header and the firmware image.</td>
</tr>
</tbody>
</table>

The X-CUBE-SBSFU Secure Boot and Secure Firmware Update Expansion Package runs on STM32 32-bit microcontrollers based on the Arm®(a) Cortex®-M processor.

---

a. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and or elsewhere.
2 Related documents

1. Getting started with the X-CUBE-SBSFU STM32Cube Expansion Package user manual (UM2262)
2. Getting started with STM32CubeL4 for STM32L4 Series and STM32L4+ Series user manual (UM1860)
3. Getting started with STM32CubeF4 MCU Package for STM32F4 Series user manual (UM1730)
4. Getting started with STM32CubeF7 MCU Package for STM32F7 Series user manual (UM1891)
5. Getting started with STM32CubeG0 for STM32G0 Series user manual (UM2303)
7. Development checklist for STM32Cube Expansion Packages user manual (UM2312)
8. STM32CubeProgrammer software description user manual (UM2237)
9. STM32F3 Series, STM32F4 Series, STM32L4 Series and STM32L4+ Series Cortex®-M4 programming manual (PM0214)
10. STM32F7 Series and STM32H7 Series Cortex®-M7 processor programming manual (PM0253)
11. STM32L0 Series and STM32G0 Series Cortex®-M0+ programming manual (PM0223)
3 Porting X-CUBE-SBSFU onto another board

X-CUBE-SBSFU supplements the STM32Cube™ software technology, making portability across different STM32 microcontrollers easy. It comes with a set of examples implemented on given STM32 boards that are useful starting points to port the X-CUBE-SBSFU onto another STM32 board. The NUCLEO-L476RG and NUCLEO-L432KC boards are used as examples in this document.

3.1 Hardware adaptation

A few changes are needed in order to adapt X-CUBE-SBSFU to another board:

1. GPIO configuration for UART communication with the host PC (in file sfu_low_level.h)
2. Flash configuration: NUCLEO-L432KC gives an example of single-bank Flash interface whereas NUCLEO-L476RG is dual-bank based (in file sfu_low_level.c)
3. Button configuration: NUCLEO-L476RG gives an example based on the push button whereas NUCLEO-L432KC simulates a virtual button with a GPIO (in file app_hw.h)
4. Tamper GPIO pin configuration (in file sfu_low_level_security.h)
5. DAP - Debug port configuration (in file sfu_low_level_security.h)

Figure 1 presents the SBSFU project structure together with the location of the files where porting changes are expected.

Figure 1. SBSFU project structure

Flash modification must also be reported into SECoreBin and UserApp low-level interfaces
3.2 Memory mapping definition

As already highlighted in the X-CUBE-SBSFU user manual (refer to [1]), a key aspect is the placement of all elements inside the Flash memory of the device:

- Secure Engine: protected environment to manage all critical data and operations.
- SBSFU: Secure Boot and Secure Firmware Update
- Slot #0: this slot contains active firmware (firmware header + firmware)
- Slot #1: after a download procedure, this slot stores downloaded firmware (firmware header + encrypted firmware) to be installed at next reboot
- Swap area: Flash area used to swap the content of Slot #0 and Slot #1 during the installation process

Figure 2 presents the Flash memory mapping illustrated by the NUCLEO-L476RG example.

Figure 2. Memory mapping example (NUCLEO-L476RG)
The linker file definitions shared between the three X-CUBE-SBSFU projects (SECoreBin, SBSFU, UserApp) are grouped in the Linker_Common folder as presented in Figure 3:

- `mapping_fwimg.icf`: contains firmware image definitions such as Slot #0, Slot #1, and Swap area
- `mapping_sbsfu.icf`: contains SBSFU definitions such as SE_Code_region, SE_Key_region, and SE_IF_region
- `mapping_export.h`: export the symbols from `mapping_sbsfu.icf` and `mapping_fwimg.icf` to the SBSFU applications

Each region can be extended when adding more code is needed, or shifted to another address as long as the resulting security settings satisfy security requirements.

**Figure 3. Linker file architecture**

Secure Engine isolation based on FWALL protection: STM32L4 Series

The entire security peripheral configuration is automatically computed based on the SBSFU linker symbols exported with file `mapping_export.h`: RDP protection, WRP protected area configuration, PCROP protected area, FWALL configuration, MPU configuration.

SecureEngine isolation based on MPU protection: STM32F4 Series, STM32F7 Series, and STM32G0 Series

The security peripheral configuration (RDP, WRP, PCROP, secure user memory if available for the series) is automatically computed based on the SBSFU linker symbols except for MPU configuration:

- MPU SE code region size depends on `__ICFEDIT_SE_Code_region_ROM_end__`
- MPU SBSFU code region size depends on `__ICFEDIT_SB_Code_region_ROM_end__`

The mapping constraints with MPU isolation are illustrated in Figure 4.
Figure 4. Mapping constraints with MPU isolation (NUCLEO-G071RB example)
3.2.1 SBSFU region definition parameters

Figure 5 presents the parameters in file mapping_sbsfu.icf that are used for the configuration of the SBSFU regions.

Figure 5. SBSFU regions (mapping_sbsfu.icf from NUCLEO-L476RG)

The compliance with SBSFU constraints requires that the following conditions are met:

- Slots areas must be aligned on the Flash sector size, which is 2048 bytes (0x800) for devices in the STM32L4 Series.
- The minimum size of SWAP is 4 Kbytes and at least equal to the size of the largest sector.
- The sizes SLOT0 and SLOT1 must be multiples of the SWAP size.
- The sizes SLOT0 and SLOT1 must be equal.
### 3.2.2 Firmware image slot definition parameters

*Figure 6* presents the parameters in file `mapping_fwimg.icf` that are used for the configuration of the image regions.

**Figure 6. Firmware image slot definitions (mapping_fwimg.icf from NUCLEO-L476RG)**

For STM32L4 dual-bank Flash memory devices, firewall specific constraints are:

- Firewall code segment must be in bank1, firewall non-volatile data (including header of Slot #0) segment must be in bank2.

- Firewall code and non-volatile data segments must be located at the same offset from the base address in each bank (ensuring that secrets are always protected even if the banks are swapped).

For the STM32G0 Series, one constraint exists: the Slot #0 header must be mapped just after the SBSFU code in order to be protected by the secured memory.

The `SFU_IMAGE_OFFSET` value depends on the STM32 microcontroller series:

- For the STM32L4 Series and STM32F4 Series, the default value is used: 512 bytes.
- For the STM32F7 Series: 1024 bytes. (with the Cortex®-M7, the vector table must be aligned on 1024 bytes).
- For the STM32G0 Series: 2048 bytes. The secure user memory end address is aligned on the Flash sector size.

```c
/* Slots Regions must be aligned on the Flash sector size */

/* swap region 8 Kbytes */
define exported symbol __ICFEDIT_region_SWAP_start__ = 0x0000 F200;
define exported symbol __ICFEDIT_region_SWAP_end__ = 0x0000 F4FF;

/* slot 0 region 0x70000 = 57 sections of 8 Kbytes (456 Kbytes) */
define exported symbol __ICFEDIT_region_SLOT_0_start__ = 0x0000 0000;
define exported symbol __ICFEDIT_region_SLOT_0_end__ = 0x0000 F7FF;

/* slot 1 region 0x70000 = 57 sections of 8 Kbytes (456 Kbytes) */
define exported symbol __ICFEDIT_region_SLOT_1_start__ = 0x0000 E000;
define exported symbol __ICFEDIT_region_SLOT_1_end__ = 0x0000 FFFF;
```
3.2.3 Project specific linker files

SECoreBin places critical code and critical data such as the secrets as illustrated in Figure 7.

Figure 7. SECoreBin specific linker file

```c
/* place SECoreBin: */
do not initialize { section .noinit, section BOOTINFO_DATA);
define block SE_VECTOR with alignment = 512 {readonly section .intvec };

/**
 ** link app into SECoreBin region
 **
 ** Placement of critical code and critical data:

place in SE_Entry_Secure_ROM_Region {readonly section .SE_Key_Cnct小心翼};
place in SE_Key_ROM_region {readonly section .SE_Key_DataEventArgs};

place at address mem: _ICEDIT_SE_CallGate_region_ROM_start__ {readonly section .SE_CallGate_Code };
place in SE_ROM_region {readonly, block SE_VECTOR};
place in SE_SRAM1_region {rewrite, section BOOTINFO_DATA};
```

The SBSFU linker file is in charge of SBSFU application placement that includes SECoreBin binary as shown in Figure 8.

Figure 8. SBSFU specific linker file

```c
/**
 ** link app into SBSFU region
 **
 ** Placement of critical code and critical data:

place at address mem: _ICEDIT intvec start__ {readonly section .intvec }
place at address mem: _ICEDIT_SE_CallGate_region_ROM_start__ {readonly section SE_CORE_Bin};
place in SE_ROM_region {section .SE_ROM_Code};
place in SB_ROM_region {readonly };~
place in SB_SRAM1_region {readwrite, block CSTACK, block HEAP};
```
UserApp must be configured to run in Slot #0 (SLOT_0 start address + SFU_IMG_IMAGE_OFFSET) as illustrated in Figure 9 where SFU_IMG_IMAGE_OFFSET is 512 bytes for the STM32L4 Series.

Figure 9. UserApp specific linker file (NUCLEO-L476RG example)

```c
/*-Specials-* /
#define exported symbol __ICFEDIT_intvec_start__ = __ICFEDIT_region_SLOT_0_start__ + 512;
/*-Memory Regions-* /
#define symbol __ICFEDIT_region_ROM_start__ = __ICFEDIT_intvec_start__;
#define symbol __ICFEDIT_region_ROM_end__ = __ICFEDIT_region_SLOT_0_end__;
#define symbol __ICFEDIT_region_RAM_start__ = __ICFEDIT_SE_region_SRAM1_end__ + 1;
```

1. Depends on the STM32 microcontroller series.
4 SBSFU configuration

4.1 Features to be configured

X-CUBE-SBSFU supports:

- 2 modes of operation: dual and single image
- 3 cryptographic schemes using symmetric and asymmetric cryptographic operations
- 2 cryptographic middleware:
  - STMicroelectronics middleware: X-CUBE-CRYPTOLIB library integrated into the 1_Images and 2_Images variants.
  - Third-party middleware: mbedTLS (open-source code) cryptographic services. Examples are provided for the 32L496GDISCOVERY and B-L475E-IOT01A boards in the 2_Images_OSC variant.

The configuration possibilities go beyond these options through compilation switches:

- Local loader can be removed to reduce the memory footprint (dual image only)
- Verbose switch can be activated to make the debug easier
- Debug mode can be disabled (no more `printf` on the terminal during SBSFU execution) to reduce the memory footprint
- Security IPs can be turned off to make the debug easier

*Figure 10* presents the SBSFU configuration solutions with the related files and compilation switches.

*Figure 10. SBSFU configuration*
4.2 Cryptographic scheme selection

X-CUBE-SBSFU is delivered with three cryptographic schemes using both asymmetric and symmetric cryptography:

- ECDSA asymmetric cryptography for firmware verification and AES-CBC symmetric cryptography for firmware decryption
- ECDSA asymmetric cryptography for firmware verification without firmware encryption.
- AES-GCM symmetric cryptography for both firmware verification and decryption

The selection among these schemes is done by means of the `SECBOOT_CRYPTO_SCHEME` compilation switch as depicted in Figure 11.

![Figure 11. Switching the cryptographic scheme](image)

4.3 Security configuration

The SBSFU example is delivered with STM32 security protection configuration allowing to protect secrets against both outer and inner attacks.

STM32 security peripherals can be deactivated independently as per user’s decision in order to achieve a different protection level (for example, for STM32L4 Series devices, Firewall and PCROP allow the activation of protections against inner attacks). Any STM32 security configuration modification requires a security protection evaluation at system product level to ensure that protections are well set according to product constraints and specifications.

During the development phase, the disabling of all IPs may be required for making debug easier.

*Figure 12* shows the various security configuration solutions available in file `app_sfu.h` for the STM32L4 Series.
Figure 12. STM32L4 Series security configuration (app_sfu.h)

Figure 13 shows the various security configuration solutions available in file app_sfu.h for the STM32F4 Series and STM32F7 Series.

Figure 13. STM32F4 Series and STM32F7 Series security configuration (app_sfu.h)

Figure 14 shows the various security configuration solutions available in file app_sfu.h for the STM32G0 Series.
### 4.4 Development or production mode configuration

The first step before any code modification is often to configure the SBSFU project in development mode in order to enable IDE debug facilities and add SBSFU debug traces:

1. Deactivate all security protections: `SFU_xxx_PROTECT_ENABLE`
2. Deactivate `SFU_FINAL_SECURE_LOCK_ENABLE`
3. Activate `SFU_FWIMG_BLOCK_ON_ABNORMAL_ERRORS_MODE`
4. Activate `SECBOOT_OB_DEV_MODE`
5. Optionally, activate the verbose mode: `SFU_VERBOSE_DEBUG_MODE` (for details about the impact on mapping, refer to Chapter 5: Generating cryptographic key)

At the end of the development phase, the SBSFU project must be configured in production mode for the final release:

1. Activate all required security protections: `SFU_xxx_PROTECT_ENABLE`
2. Deactivate verbose mode: `SFU_VERBOSE_DEBUG_MODE`
3. Deactivate `SFU_FWIMG_BLOCK_ON_ABNORMAL_ERRORS_MODE`
4. Deactivate `SECBOOT_OB_DEV_MODE`
5. Activate `SFU_FINAL_SECURE_LOCK_ENABLE` to configure RDP level 2

Read Protection Level 2 is mandatory to achieve the highest level of protection and to implement a Root of Trust. It is user's responsibility to activate it in the final SW to be programmed during the product manufacturing stage.

In production mode, the Secure Boot checks the Option Byte values (RDP, WRP, PCROP) and blocks execution in case a wrong configuration is detected. Depending on the platform, few other Option Bytes must be configured such as:

- BFB2 disabled for STM32L4 Series devices with dual-bank Flash
- nDBANK disabled for the STM32F7 Series
Caution: Option Bytes must be configured to the production mode values by means of STM32CubeProgrammer (STM32CubeProg), just after programming the software during the production stage. If this is not done, the device remains unsecured. Refer to [8] for the way to use STM32CubeProgrammer.

*Figure 15* shows how Option Bytes are managed at SBSFU startup:

*Figure 15. Option Bytes management*
5 Generating cryptographic key

5.1 Generating a new firmware AES encryption key

Key generation and firmware encryption are performed automatically during the compilation process with the `prebuild.bat` and `postbuild.bat` scripts (refer to [1] for a detailed description of the build process).

*Figure 16* shows the few steps to modify the firmware encryption key:

1. Change the key value in file `OEM_KEY_COMPANY1_keys_AES_xxx.bin`
2. Compile SECoreBin: `prebuild.bat` is executed and `se_key.s` is generated
3. Compile UserApp: `postbuild.bat` is executed and UserApp is encrypted

![Figure 16. New firmware encryption key](image)

5.2 Generating a new public/private ECDSA pair of keys for firmware verification

As for the AES encryption key, the public key (`SE_ReadKey_Pub()`) is automatically modified when the private key (`ECCKEY.txt`) is changed.

*Figure 17* shows the few steps to modify the private and public keys for ECDSA asymmetric cryptography firmware verification:
Generating cryptographic key

1. Change the key value in file ECCKEY.txt
2. Compile SECoreBin: prebuild.bat is executed and se_key.s is generated
3. Compile UserApp: postbuild.bat is executed and UserApp is encrypted

Figure 17. New private/public keys
6 Tips for debugging

6.1 Compiler optimizations level

Projects are delivered with the highest level of compiler optimizations turned on for size aspects. Such optimizations can make the debug complex. Changing the compiler optimization level possibly impacts memory mapping.

Figure 18. Compiler optimizations

6.2 Memory mapping adaptation

When changing the compiler optimizations level or activating the development mode with verbose compilation switch, the user can adapt the SBSFU memory mapping, for instance reducing firmware image slots to avoid overlap.

Figure 19 depicts the 3 steps of the memory adaptation based on an example:

1. Identify the gap by analyzing the linker message: 0x1d9 bytes
2. Identify the concerned region by consulting the project.map file: __ICFEDIT_SB_region_ROM_start__
3. Apply the modification in file mapping_sbsfu.icf: 0x300 bytes
The impact of memory mapping adaptation on security peripheral configurations must be checked despite the fact that it is automatically computed. For example, check the WRP configuration using STM32CubeProgrammer (STM32CubeProg) as shown in Figure 20.

**Firmware image slot definitions may be reduced to avoid overlap**
6.3 Debugging SECoreBin

To debug inside SECoreBin, SBSFU projects option must be changed to load SECoreBin symbols. This is performed in the debugger menu as presented in Figure 21:

- Browse to select file Project.out
- Set Offset to 0
- Check the Debug info only box

![Figure 21. Debugging inside SECoreBin](image)
7 Adapting SBSFU

7.1 Implementing a new cryptographic scheme for SBSFU

X-CUBE-SBSFU comes with some predefined cryptographic schemes (refer to Section 4.2: Cryptographic scheme selection on page 17). It is also possible to extend the package with the user’s own cryptographic scheme.

In order to implement a new cryptographic scheme for SBSFU, follow the steps illustrated in Figure 22 and described below.

Figure 22. User’s own cryptographic scheme implementation

Updating the code running on device side:

1. **Step 1**: define a new value for `SECBOOT_CRYPTO_SCHEME`.
2. **Step 2**: look carefully at the signatures of the APIs that the bootloader requires. The cryptographic services must have the same signatures to avoid updating the SBSFU code.
3. **Step 3**: define a new `SE_FwRawHeaderTypeDef` structure and respect the constraints to remain compatible with the existing SBSFU code.
4. **Step 4**: implement the code of the cryptographic services in `se_crypto_bootloader.c`. 
Updating the tools running on host side to prepare the keys and the firmware image:

5. **Step 5**: update the preparation tools to support the new cryptographic scheme (`prepareimage.py`; `translate_key.py`; `keys.py`).

6. **Step 6**: update the IDE integration to generate the appropriate keys and firmware image.
   - A new batch file is required to call the preparation tools with the appropriate commands; `prebuild.bat` copies this batch file to create `postbuild.bat`.
   - `prebuild.bat` must be updated to take into account the new cryptographic scheme and generate the proper keys and `postbuild.bat`.

### 7.2 Optimizing memory mapping

Several options exist to reduce SBSFU code size in order to maximize the size of the user application slot. Some of these options are summarized in Table 3.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description / Consequence</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select 1-image variant</td>
<td>Download a new firmware image from the user application is no more possible.</td>
<td>Slot size is doubled vs. 2-image projects</td>
</tr>
<tr>
<td>Select AES-GCM symmetric cryptographic scheme</td>
<td>Shared symmetric key secret stored in the device.</td>
<td>~ 9 Kbytes</td>
</tr>
<tr>
<td>Disable SFU_DEBUG_MODE</td>
<td>No more information displayed on terminal during SBSFU execution</td>
<td>~ 9 Kbytes</td>
</tr>
<tr>
<td>Disable SECBOOT_USE_LOCAL_LOADER</td>
<td>No more local loader inside the SBSFU application. This is not compatible with 1-image variant.</td>
<td>~3 Kbytes</td>
</tr>
<tr>
<td>Implement a hardware decryption</td>
<td>Select STM32 devices integrating cryptographic hardware IP.</td>
<td>Depends on user’s implementation</td>
</tr>
<tr>
<td>If all the code running on STM32 is fully trusted and robust then you can remove Secure Engine internal isolation based on MPU for STM32F4/F7/G0</td>
<td>Removes alignment constraints with MPU regions.</td>
<td>Up to 12 Kbytes depending on products</td>
</tr>
</tbody>
</table>

The total gain depends on the mapping constraints described in Section 3.2: Memory mapping definition on page 9.

As an example, Figure 23 highlights the mapping modifications to be done. Starting from 2 images with symmetric crypto scheme, the SFU_DEBUG_MODE and SECBOOT_USE_LOCAL_LOADER switches are disabled, resulting in a 16-Kbyte increase of the user application size.
Figure 23. Example of memory mapping optimization on NUCLEO-G071RB – 2 images

Figure 24 shows the same optimization done on a 1-image variant: user application size is increased by 24 Kbytes

Figure 24. Example of memory mapping optimization on NUCLEO-G071RB – 1 image
8  Adapting the user application

8.1  How to make an application SBSFU compatible

First of all, the mapping of the user application must be modified in order to allow the application to run in Slot #0:

- Code section starting by the vector table must be configured to run from Slot #0, just after the image header: \_ICFEDIT\_region\_SLOT\_0\_start\_\_ + 512
  
  (SFU\_IMG\_OFFSET = 512 for the STM32L4 Series)

- Data section must start after the Secure Engine protected area:
  \_ICFEDIT\_SE\_region\_SRAM1\_end\_ + 1

Refer to Section 3.2: Memory mapping definition on page 9 for more details on memory constraints.

Then, during system initialization, VTOR must be set to the new location of vector table as shown in Figure 25.

Figure 25. Vector table position update (NUCLEO-L476RG example)

Finally, as done in the UserApp example, the IDE configuration must be updated to:

1. Generate a UserApp.bin file
2. Include search path for linker common files
3. Call postbuild.bat to generate UserApp.sfb and SBFU\_UserApp.bin
4. Integrate se\_interface\_appli.o to access Secure Engine runtime services if any
As explained in [1], there are some additional constraints depending on the STM32 series:

- STM32F4 Series and STM32F7 Series: MPU-based Secure Engine isolation relies fully on the fact that privileged level of software execution is required to access the Secure Engine services. The user application must take this constraint into account and trust any piece of code running in privileged mode.
- STM32G0 Series: when secured, any access to securable memory area (fetch, read, programming, erase) is rejected, generating a bus error. As a consequence, there is no Secure Engine runtime services available for the user application.

Note: IWDG is started during SBSFU execution. It must be refreshed within a 6-second period.
8.2 Use of the Flash memory to store user data

The storage of user data in Flash pages (or Flash sectors) is possible with some restrictions:

- Out of the SBSFU code area
- Not in the images slots (Slot #0, Slot #1)
- Not in the Swap area

Figure 27 provides a memory-mapping example based on the NUCLEO-L476RG where the Flash is available from page 489 to page 511 for the user to store data, install a file system or emulate an EEPROM.

Figure 27. Free Flash pages (example of NUCLEO-L476RG)
8.3 Changing the firmware download function in the user application

This possibility is available only in the dual-image mode of operation.

A sample code based on the YMODEM protocol over UART is available in X-CUBE-SBSFU UserApp project. The download procedure is located in file `fw_update_app.c` as illustrated in Figure 28.

Figure 28. UserApp firmware download overview

```c
HAL_StatusTypeDef FW_UPDATE_Run(void)
{
    HAL_StatusTypeDef ret = HAL_ERROR;
    uint32_t fw_header_input[4], fw_header_len;
    STM_FlashTypeDef Flash = STM_FLASH_DEFAULT;
    STM_FlashTypeDef ff_image_dbl_areas;
    /* Print Firmware Update welcome message */
    FW_UPDATE_PrintWelcome();
    /* Get Info about the download area */
    if (STM_APP_GetDownloadAreaInfo(&ff_image_dbl_areas) != HAL_ERROR)
    {
        /* Download new firmware image */
        ret = FW_UPDATE_DownloadNewFirmware(ff_image_dbl_areas);
        if (HAL_OK == ret)
        {
            /* Read header in slot i */
            memcpy(&fw_header_input[4], &ff_image_dbl_areas.DownloadAddr,
            /* Ask for installation at next reset */
            void) STM_APP_InitNextReset(&fw_header_input);
            /* System Reboot */
            printf("Image correctly downloaded - reboot\n\n\n");
            HAL_Delay(10000);
            NVIC_SystemReset();
            if (ret == HAL_OK)
            {
                printf("Operation failed!\n\n");
            }
            return ret;
        }
    }
}
```
Table 4. Document revision history

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>20-Dec-2017</td>
<td>1</td>
<td>Initial release.</td>
</tr>
<tr>
<td>31-Aug-2018</td>
<td>2</td>
<td>Document structure and content entirely updated:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Refocused on the integration topics presented in <em>Introduction</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Adapted to the asymmetric and symmetric cryptography schemes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Adapted to the single-image and dual-image modes</td>
</tr>
<tr>
<td>18-Dec-2018</td>
<td>3</td>
<td>Product scope extended to the STM32F4 Series, STM32F7 Series, and STM32G0 Series:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Updated <em>Chapter 1: General information</em>, <em>Chapter 2: Related documents</em>, <em>Section 3.2: Memory mapping definition</em>, <em>Section 4.3: Security configuration</em>, <em>Section 4.4: Development or production mode configuration</em>, and <em>Section 8.1: How to make an application SBSFU compatible</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Added <em>Chapter 7: Adapting SBSFU</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Secure library offer extended to mbedTLS:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Updated <em>Section 4.1: Features to be configured</em></td>
</tr>
</tbody>
</table>