Introduction

This user manual describes how to get started with the X-CUBE-SBSFU STM32Cube Expansion Package.

The X-CUBE-SBSFU Secure Boot and Secure Firmware Update 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 updates and access to confidential on-device data.

The Secure Boot (Root of Trust services) is an immutable code, always executed after a system reset, that checks STM32 static protections, activates STM32 runtime protections and then verifies the authenticity and integrity of user application code before every execution in order to ensure that invalid or malicious code cannot be run.

The Secure Firmware Update application receives the firmware image via a UART interface with the Ymodem protocol, checks its authenticity, and checks the integrity of the code before installing it. The firmware update is done on the complete firmware image, or only on a portion of the firmware image. Examples are provided for single firmware image configuration in order to maximize firmware image size, and for dual firmware image configuration in order to ensure safe image installation and enable over-the-air firmware update capability commonly used in IoT devices. Examples can be configured to use asymmetric or symmetric cryptographic schemes with or without firmware encryption.

The secure key management services provide cryptographic services to the user application through the PKCS #11 APIs (KEY ID-based APIs) that are executed inside a protected and isolated environment. User application keys are stored in the protected and isolated environment for their secured update: authenticity check, data decryption and data integrity check.

STSAFE-A100 is a tamper-resistant secure element (HW Common Criteria EAL5+ certified) used to host X509 certificates and keys, and perform verifications that are used for firmware image authentication during Secure Boot and Secure Firmware Update procedures.

X-CUBE-SBSFU is built on top of STM32Cube software technology, making the portability across different STM32 microcontrollers easy. It is provided as reference code to demonstrate best use of STM32 security protections.

X-CUBE-SBSFU is classified ECCN 5D002.
Contents

1 General information ................................................................. 9
  1.1 Terms and definitions .................................................. 9
  1.2 References ............................................................... 11

2 STM32Cube overview ........................................................... 12

3 Secure Boot and Secure Firmware Update (SBSFU) ................. 14
  3.1 Product security introduction ...................................... 14
  3.2 Secure Boot .............................................................. 14
  3.3 Secure Firmware Update .............................................. 15
  3.4 Cryptography operations .............................................. 16

4 Key management services ..................................................... 18

5 Protection measures and security strategy ............................ 20
  5.1 STM32L4 Series and STM32L0 Series ............................. 21
  5.2 STM32F4 Series, STM32F7 Series and STM32L1 Series ....... 24
  5.3 STM32G0 Series, STM32G4 Series and STM32H7 Series ....... 26
  5.4 STM32WB Series ......................................................... 30
  5.5 STM32L4 Series combined with STSAFE-A100 .................... 32

6 Package description ............................................................. 35
  6.1 General description .................................................... 35
  6.2 Architecture ............................................................. 36
    6.2.1 STM32CubeHAL .................................................. 37
    6.2.2 Board support package (BSP) .................................. 37
    6.2.3 Cryptographic Library ........................................... 37
    6.2.4 Secure Engine (SE) middleware ............................... 37
    6.2.5 Key management services (KMS) middleware ............... 37
    6.2.6 STSAFE-A middleware ........................................... 38
    6.2.7 Secure Boot and Secure Firmware Upgrade (SBSFU) application ... 39
    6.2.8 User application .................................................. 40
  6.3 Folder structure .......................................................... 41
6.4 APIs ................................................................. 42
6.5 Application compilation process with IAR™ toolchain ............... 42

7 Hardware and software environment setup .............................. 44
7.1 Hardware setup ..................................................... 44
7.2 Software setup ................................................................ 44
7.2.1 Development toolchains and compilers .......................... 44
7.2.2 Software tools for programming STM32 microcontrollers .... 44
7.2.3 Terminal emulator .................................................. 45
7.2.4 X-CUBE-SBSFU firmware image preparation tool ............ 45

8 Step-by-step execution ....................................................... 46
8.1 STM32 board preparation .............................................. 47
8.2 Application compilation ................................................ 49
8.3 Tera Term connection .................................................. 50
8.3.1 ST-LINK disable ..................................................... 50
8.3.2 Tera Term launch .................................................... 50
8.3.3 Tera Term configuration ......................................... 51
8.3.4 Welcome screen display ......................................... 52
8.4 SBSFU application execution .......................................... 52
8.4.1 Download request .................................................. 52
8.4.2 Send firmware ....................................................... 52
8.4.3 File transfer completion .......................................... 54
8.4.4 System restart ....................................................... 55
8.5 User application execution ............................................ 55
8.5.1 Download a new firmware image ............................... 55
8.5.2 Test protections ..................................................... 57
8.5.3 Test Secure Engine user code .................................. 57

9 Understanding the last execution status message at boot-up ...... 58

Appendix A Secure Engine protected environment ....................... 60
A.1 Firewall-based Secure Engine Isolation ................................ 61
A.1.1 SE core call gate mechanism .................................... 61
A.1.2 SE interface .......................................................... 62
A.2 MPU-based Secure Engine Isolation .................................. 63
Appendix A

A.2.1 Principle ........................................... 63
A.2.2 Constraints ........................................... 66

Appendix B  Dual-image handling ............................... 67

B.1 Elements and roles ........................................... 67
B.2 Mapping definition ........................................... 69

Appendix C  Single-image handling ............................... 70

C.1 Elements and roles ........................................... 70
C.2 Mapping definition ........................................... 70

Appendix D  Cryptographic schemes handling ....................... 71

D.1 Cryptographic schemes contained in this package .......... 71
D.2 Asymmetric verification and symmetric encryption schemes .. 72
D.3 Symmetric verification and encryption scheme ............... 73
D.4 X509 certificate-based asymmetric scheme without firmware encryption ... 74
D.5 Secure Boot and Secure Firmware Update flow .......... 76

Appendix E  Firmware image preparation tool ....................... 78

E.1 Tool location ........................................... 78
E.2 Inputs ........................................... 78
E.3 Outputs ........................................... 79
E.4 IDE integration. ........................................... 79
E.5 Partial Image ........................................... 80

Appendix F  KMS ........................................... 81

F.1 Key update process description .............................. 81
F.2 SBSFU static keys generation ................................ 82
F.3 UserApp menu ........................................... 83

Appendix G  SBSFU with STM32 and STSAFE-A100 ............... 84

G.1 Introduction to STSAFE-A100 ................................ 84
G.2 Certificate generation ....................................... 86
G.3 STSAFE-A100 provisioning ................................... 86
G.4 STM32 and firmware image provisioning ..................... 87
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>G.5</td>
<td>STSAFE-A100 ordering.</td>
<td>87</td>
</tr>
<tr>
<td>Appendix H</td>
<td>STM32WB Series specificities</td>
<td>88</td>
</tr>
<tr>
<td>H.1</td>
<td>Compilation process</td>
<td>88</td>
</tr>
<tr>
<td>H.2</td>
<td>Key provisioning</td>
<td>88</td>
</tr>
<tr>
<td>Appendix I</td>
<td>STM32H7 Series specificities</td>
<td>89</td>
</tr>
<tr>
<td>I.1</td>
<td>JTAG connection for STM32H753 devices</td>
<td>89</td>
</tr>
<tr>
<td>I.2</td>
<td>JTAG connection for STM32H7B3 devices</td>
<td>89</td>
</tr>
<tr>
<td>I.3</td>
<td>External Flash on STM32H7B3 devices</td>
<td>90</td>
</tr>
<tr>
<td>Revision history</td>
<td></td>
<td>92</td>
</tr>
</tbody>
</table>
List of tables

Table 1. List of acronyms .......................................................... 9
Table 2. List of terms ................................................................. 10
Table 3. Cryptographic scheme comparison .................................. 17
Table 4. MPU regions in the STM32F4 Series, STM32F7 Series and STM32L1 Series ........... 26
Table 5. MPU regions in the STM32G0 Series, STM32G4 Series and STM32H7 Series .......... 28
Table 6. Error messages at boot-up ............................................ 58
Table 7. MPU regions for Secure Engine isolation .......................... 64
Table 8. Cryptographic scheme list ............................................ 71
Table 9. Document revision history .......................................... 92
# List of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.</td>
<td>Secure Boot Root of Trust</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2.</td>
<td>Typical in-field device update scenario</td>
<td>15</td>
</tr>
<tr>
<td>Figure 3.</td>
<td>KMS functions overview</td>
<td>19</td>
</tr>
<tr>
<td>Figure 4.</td>
<td>SBSFU security IPs vs. STM32 Series (1 of 2)</td>
<td>20</td>
</tr>
<tr>
<td>Figure 5.</td>
<td>SBSFU security IPs vs. STM32 Series (2 of 2)</td>
<td>21</td>
</tr>
<tr>
<td>Figure 6.</td>
<td>STM32L4 and STM32L0 protection overview during SBSFU execution</td>
<td>22</td>
</tr>
<tr>
<td>Figure 7.</td>
<td>STM32F4, STM32F7 and STM32L1 protection overview during SBSFU execution</td>
<td>24</td>
</tr>
<tr>
<td>Figure 8.</td>
<td>STM32G0, STM32G4 and STM32H7 protection overview during SBSFU execution</td>
<td>26</td>
</tr>
<tr>
<td>Figure 9.</td>
<td>STM32G0, STM32G4, and STM32H7 protection overview during user application execution</td>
<td>29</td>
</tr>
<tr>
<td>Figure 10.</td>
<td>STM32WB protection overview during SBSFU execution</td>
<td>30</td>
</tr>
<tr>
<td>Figure 11.</td>
<td>STM32L4 / STSAFE-A100 protection overview during SBSFU execution</td>
<td>32</td>
</tr>
<tr>
<td>Figure 12.</td>
<td>Software architecture overview</td>
<td>36</td>
</tr>
<tr>
<td>Figure 13.</td>
<td>Project folder structure (1 of 2)</td>
<td>41</td>
</tr>
<tr>
<td>Figure 14.</td>
<td>Project folder structure (2 of 2)</td>
<td>42</td>
</tr>
<tr>
<td>Figure 15.</td>
<td>Application compilation steps</td>
<td>43</td>
</tr>
<tr>
<td>Figure 16.</td>
<td>Firmware image preparation tool IDE integration</td>
<td>45</td>
</tr>
<tr>
<td>Figure 17.</td>
<td>Step-by-step execution</td>
<td>46</td>
</tr>
<tr>
<td>Figure 18.</td>
<td>STM32 board preparation</td>
<td>47</td>
</tr>
<tr>
<td>Figure 19.</td>
<td>STM32CubeProgrammer connection menu</td>
<td>48</td>
</tr>
<tr>
<td>Figure 20.</td>
<td>STM32CubeProgrammer Option bytes screen</td>
<td>49</td>
</tr>
<tr>
<td>Figure 21.</td>
<td>STM32CubeProgrammer erasing</td>
<td>49</td>
</tr>
<tr>
<td>Figure 22.</td>
<td>Tera Term connection screen</td>
<td>50</td>
</tr>
<tr>
<td>Figure 23.</td>
<td>Tera Term setup screen</td>
<td>51</td>
</tr>
<tr>
<td>Figure 24.</td>
<td>SBSFU welcome screen display</td>
<td>52</td>
</tr>
<tr>
<td>Figure 25.</td>
<td>SBSFU encrypted firmware transfer start</td>
<td>53</td>
</tr>
<tr>
<td>Figure 26.</td>
<td>SBSFU encrypted firmware transfer in progress</td>
<td>53</td>
</tr>
<tr>
<td>Figure 27.</td>
<td>SBSFU reboot after encrypted firmware transfer</td>
<td>54</td>
</tr>
<tr>
<td>Figure 28.</td>
<td>User application execution</td>
<td>55</td>
</tr>
<tr>
<td>Figure 29.</td>
<td>Encrypted firmware download via user application</td>
<td>56</td>
</tr>
<tr>
<td>Figure 30.</td>
<td>User application test protection menu</td>
<td>57</td>
</tr>
<tr>
<td>Figure 31.</td>
<td>Firewall call gate mechanism</td>
<td>61</td>
</tr>
<tr>
<td>Figure 32.</td>
<td>Secure Engine call-gate mechanism</td>
<td>62</td>
</tr>
<tr>
<td>Figure 33.</td>
<td>Secure Engine interface</td>
<td>63</td>
</tr>
<tr>
<td>Figure 34.</td>
<td>SBSFU running in unprivileged level of software execution for standard operations</td>
<td>64</td>
</tr>
<tr>
<td>Figure 35.</td>
<td>SBSFU requesting a Secure Engine service</td>
<td>65</td>
</tr>
<tr>
<td>Figure 36.</td>
<td>Exiting a Secure Engine service</td>
<td>66</td>
</tr>
<tr>
<td>Figure 37.</td>
<td>Internal user Flash mapping (example of the NUCLEO-L476RG with 512-byte headers)</td>
<td>68</td>
</tr>
<tr>
<td>Figure 38.</td>
<td>User application vector table (example of the STM32L4 Series)</td>
<td>69</td>
</tr>
<tr>
<td>Figure 39.</td>
<td>Asymmetric verification and symmetric encryption</td>
<td>72</td>
</tr>
<tr>
<td>Figure 40.</td>
<td>Symmetric verification and encryption</td>
<td>73</td>
</tr>
<tr>
<td>Figure 41.</td>
<td>X509 asymmetric verification</td>
<td>74</td>
</tr>
<tr>
<td>Figure 42.</td>
<td>Certificate chain</td>
<td>75</td>
</tr>
<tr>
<td>Figure 43.</td>
<td>SBSFU dual-image boot flows</td>
<td>76</td>
</tr>
<tr>
<td>Figure 44.</td>
<td>SBSFU single-image boot flows</td>
<td>77</td>
</tr>
<tr>
<td>Figure 45.</td>
<td>Encrypted object creation</td>
<td>81</td>
</tr>
<tr>
<td>Figure 46.</td>
<td>Secure update procedure</td>
<td>82</td>
</tr>
<tr>
<td>Figure 47.</td>
<td>KMS key storage</td>
<td>83</td>
</tr>
</tbody>
</table>
List of figures

Figure 48. KMS menu ......................................................... 83
Figure 49. Certificate chain overview ........................................ 84
Figure 50. Pairing key and certificate provisioning overview .................. 85
Figure 51. Batch files using openssl .......................................... 86
Figure 52. Provisioning in STM32 and firmware image ....................... 87
Figure 53. Compile with Loader integration .................................. 88
Figure 54. JTAG connection capability on STM32H753 devices ............. 89
Figure 55. JTAG connection capability on STM32H7B3 devices ............... 90
Figure 56. STM32H7B3: MPU isolation + secure user memory, with external Flash .... 90
Figure 57. Memory mapping for STM32H7B3 devices with external Flash .......... 91
1 General information

The X-CUBE-SBSFU Expansion Package comes with examples running on the STM32L4 Series, STM32F4 Series, STM32F7 Series, STM32G0 Series, STM32G4 Series, STM32H7 Series, STM32L0 Series, STM32L1 Series, and STM32WB Series. An example combining STM32 microcontroller and STSAFE-A100 is also provided for the STM32L4 Series.

X-CUBE-SBSFU is provided as reference code for standalone STM32 system solution examples demonstrating best use of STM32 protections to protect assets against unauthorized external and internal access. X-CUBE-SBSFU proposes also a system solution example combining STM32 and STSAFE-A100, which demonstrates HW Secure Element protections for secure authentication services and secure data storage.

X-CUBE-SBSFU is a starting point for OEMs to develop their own SBSFU as a function of their product security requirement levels.

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

1.1 Terms and definitions

Table 1 presents the definition of acronyms that are relevant for a better understanding of this document.

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAD</td>
<td>Additional authenticated data</td>
</tr>
<tr>
<td>AES</td>
<td>Advanced encryption standard</td>
</tr>
<tr>
<td>CBC</td>
<td>AES cipher block chaining</td>
</tr>
<tr>
<td>CKS</td>
<td>Customer key storage</td>
</tr>
<tr>
<td>CTR</td>
<td>AES counter-based cipher mode</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct memory access</td>
</tr>
<tr>
<td>DSA</td>
<td>Digital signature algorithm</td>
</tr>
<tr>
<td>ECC</td>
<td>Elliptic curve cryptography</td>
</tr>
<tr>
<td>ECCN</td>
<td>Export control classification number</td>
</tr>
<tr>
<td>ECDSA</td>
<td>Elliptic curve digital signature algorithm</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite-state machine</td>
</tr>
<tr>
<td>GCM</td>
<td>AES Galois/counter mode</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical user interface</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware abstraction layer</td>
</tr>
</tbody>
</table>

a. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.
Table 1. List of acronyms (continued)

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDE</td>
<td>Integrated development environment</td>
</tr>
<tr>
<td>IV</td>
<td>Initialization vector</td>
</tr>
<tr>
<td>IWDG</td>
<td>Independent watch dog</td>
</tr>
<tr>
<td>FW</td>
<td>Firmware</td>
</tr>
<tr>
<td>FWALL</td>
<td>Firewall</td>
</tr>
<tr>
<td>KMS</td>
<td>Key management services</td>
</tr>
<tr>
<td>MAC</td>
<td>Message authentication code</td>
</tr>
<tr>
<td>MCU</td>
<td>Microcontroller unit</td>
</tr>
<tr>
<td>MPU</td>
<td>Memory protection unit</td>
</tr>
<tr>
<td>NONCE</td>
<td>Number used only once</td>
</tr>
<tr>
<td>OTFDEC</td>
<td>On-the-fly decryption</td>
</tr>
<tr>
<td>PCROP</td>
<td>Proprietary code read out protection</td>
</tr>
<tr>
<td>PEM</td>
<td>Privacy enhanced mail</td>
</tr>
<tr>
<td>RDP</td>
<td>Read 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>SM</td>
<td>State machine</td>
</tr>
<tr>
<td>UART</td>
<td>Universal asynchronous receiver/transmitter</td>
</tr>
<tr>
<td>UUID</td>
<td>Universally unique identifier</td>
</tr>
<tr>
<td>WRP</td>
<td>Write protection</td>
</tr>
</tbody>
</table>

*Table 2* presents the definition of terms that are relevant for a better understanding of this document.

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>
1.2 References

STMicroelectronics related documents

Public documents are available online from STMicroelectronics web site at www.st.com. Contact STMicroelectronics when more information is needed.

1. *Integration guide for the X-CUBE-SBSFU STM32Cube Expansion Package* (AN5056)
2. *Introduction to STM32 microcontrollers security* application note (AN5156)
3. *STM32CubeProgrammer software description* user manual (UM2237)
4. *Authentication, state-of-the-art security for peripherals and IoT devices* data sheet (DS12911)

Other documents

2 STM32Cube overview

What is STM32Cube?

STM32Cube is an STMicroelectronics original initiative to significantly improve designer’s productivity by reducing development effort, time and cost. STM32Cube covers the whole STM32 portfolio.

STM32Cube includes:
- A set of user-friendly software development tools to cover project development from the conception to the realization, among which:
  - STM32CubeMX, a graphical software configuration tool that allows the automatic generation of C initialization code using graphical wizards
  - STM32CubeIDE, an all-in-one development tool with peripheral configuration, code generation, code compilation, and debug features
  - STM32CubeProgrammer (STM32CubeProg), a programming tool available in graphical and command-line versions
  - STM32CubeMonitor-Power (STM32CubeMonPwr), a monitoring tool to measure and help in the optimization of the power consumption of the MCU
- STM32Cube MCU & MPU Packages, comprehensive embedded-software platforms specific to each microcontroller and microprocessor series (such as STM32CubeL4 for the STM32L4 Series), which include:
  - STM32Cube hardware abstraction layer (HAL), ensuring maximized portability across the STM32 portfolio
  - STM32Cube low-layer APIs, ensuring the best performance and footprints with a high degree of user control over the HW
  - A consistent set of middleware components such as FAT file system, RTOS, USB Host and Device, TCP/IP, Touch library, and Graphics
  - All embedded software utilities with full sets of peripheral and applicative examples
- STM32Cube Expansion Packages, which contain embedded software components that complement the functionalities of the STM32Cube MCU & MPU Packages with:
  - Middleware extensions and applicative layers
  - Examples running on some specific STMicroelectronics development boards

How does this software complement STM32Cube?

The proposed software is based on the STM32CubeHAL, the hardware abstraction layer for the STM32 microcontroller. The package extends STM32Cube by providing middleware components:
- Secure Engine for managing all critical data and operations, such as cryptography operations accessing firmware encryption key and others
- Key management services offering cryptographic services via PKCS #11 APIs
- STSAFE-A for managing HW Secure Element features
The package includes different sample applications to provide a complete SBSFU solution:

- **SE_CoreBin application**: provides a binary including all the "trusted" code.
- **Secure Boot and Secure Firmware Upgrade (SBSFU) application**:  
  - Secure Boot (Root of Trust)
  - Local download via UART Virtual COM
  - FW installation management
- **User application**:  
  - Downloads a new firmware in dual-image mode of operation
  - Provides examples testing protection mechanisms
  - Provides examples using KMS APIs

The sample applications are delivered in dual-image and single-image modes of operation and can be configured in different cryptographic scheme.

This user manual describes the typical use of the package:

- Based on the NUCLEO-L476RG board
- With sample applications operating in dual-image mode and configured with asymmetric authentication and symmetric FW encryption

More information about the configuration options and the single-image mode of operation are provided in the appendices of this document.

**Note:**  
*The KMS feature is available on the STM32L4 Series with example provided on the B-L475E-IOT01A board.  
The STSAFE-A feature is available on the STM32L4 Series with example provided on the B-L475E-IOT01A board.*
3 Secure Boot and Secure Firmware Update (SBSFU)

3.1 Product security introduction

A device deployed in the field operates in an untrusted environment and it is therefore subject to threats and attacks. To mitigate the risk of attack, the goal is to allow only authentic firmware to run on the device. In fact, allowing the update of firmware images to fix bugs, or introduce new features or countermeasures, is commonplace for connected devices, but it is prone to attacks if not executed in a secure way.

Consequences may be damaging such as firmware cloning, malicious software download or device corruption. Security solutions have to be designed in order to protect sensitive data (potentially even the firmware itself) and critical operations.

Typical countermeasures are based on cryptography (with associated secret key) and on memory protections:

- Cryptography ensures integrity (the assurance that data has not been corrupted), authentication (the assurance that a certain entity is who it claims to be) and confidentiality (the assurance that only authorized users can read sensitive data) during firmware transfer.

- Memory protection mechanisms prevent external attacks (for example by accessing the device physically through JTAG) and internal attacks from other embedded processes.

The following chapters describe solutions implementing confidentiality, integrity and authentication services to address the most common threats for an IoT end-node device.

3.2 Secure Boot

Secure Boot (SB) asserts the integrity and authenticity of the user application image that is executed: cryptographic checks are used in order to prevent any unauthorized or maliciously modified software from running. The Secure Boot process implements a Root of Trust (refer to Figure 1): starting from this trusted component (1), every other component is authenticated (2) before its execution (3).

Integrity is verified so as to be sure that the image that is going to be executed has not been corrupted or maliciously modified.

Authenticity check aims to verify that the firmware image is coming from a trusted and known source in order to prevent unauthorized entities to install and execute code.
3.3 Secure Firmware Update

Secure Firmware Update (SFU) provides a secure implementation of in-field firmware updates, enabling the download of new firmware images to a device in a secure way.

As shown in Figure 2, two entities are typically involved in a firmware update process:

- **Server**
  - OEM manufacturer server / web service
  - Stores the new version of device firmware
  - Communicates with the device and sends the new image version in an encrypted form if it is available

- **Device**
  - Deployed in the field
  - Embeds a code running firmware update process.
  - Communicates with the server and receives a new firmware image.
  - Authenticates, decrypts and installs the new firmware image and executes it.
Firmware update runs through the following steps:

1. If a firmware update is needed, a new encrypted firmware image is created and stored in the server.
2. The new encrypted firmware image is sent to the device deployed in the field through an untrusted channel.
3. The new image is downloaded, checked and installed.

Firmware update can be done on the complete firmware image, or only on a portion of the firmware image (only for dual-image configuration).

Firmware update is vulnerable to the threats presented in Section 3.1: Product security introduction: cryptography is used to ensure confidentiality, integrity and authentication.

**Confidentiality** is implemented so as to protect the firmware image, which may be a key asset for the manufacturer. The firmware image sent over the untrusted channel is encrypted so that only devices having access to the encryption key can decrypt the firmware package.

**Integrity** is verified so as to be sure that the received image is not corrupted.

**Authenticity** check aims to verify that the firmware image is coming from a trusted and known source, in order to prevent unauthorized entities to install and execute code.

### 3.4 Cryptography operations

The X-CUBE-SBSFU STM32Cube Expansion Package is delivered with four cryptographic schemes using both asymmetric and symmetric cryptography.

The default cryptographic scheme demonstrates ECDSA asymmetric cryptography for firmware verification and AES-CBC symmetric cryptography for firmware decryption. Thanks to asymmetric cryptography, the firmware verification can be performed with public-key operations so that no secret information is required in the device.

The alternative cryptographic schemes provided in the X-CUBE-SBSFU Expansion Package are:

- ECDSA asymmetric cryptography for firmware verification with AES-CBC or AES-CTR symmetric cryptography for firmware encryption
- ECDSA asymmetric cryptography for firmware verification without firmware encryption
- X509 certificate-based ECDSA asymmetric cryptography for firmware verification without firmware encryption
- AES-GCM symmetric cryptography for both firmware verification and encryption.

*Table 3* presents the various security features associated with each of the cryptographic schemes.
### Table 3. Cryptographic scheme comparison

<table>
<thead>
<tr>
<th>Features</th>
<th>Asymmetric with AES encryption</th>
<th>Asymmetric without encryption</th>
<th>X509 certificate-based asymmetric without encryption</th>
<th>Symmetric (AES-GCM)(^{(1)})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality</td>
<td>AES-CBC encryption, or AES-CTR encryption for STM32 MCUs supporting OTFDEC processing (FW binary)</td>
<td>None: the user FW is in clear format.</td>
<td></td>
<td>AES-GCM encryption (FW binary)</td>
</tr>
<tr>
<td>Integrity</td>
<td>SHA256 (FW header and FW binary)</td>
<td></td>
<td></td>
<td>AES-GCM Tag (FW header and FW binary)</td>
</tr>
<tr>
<td>Authentication</td>
<td>– SHA256 of the FW header is ECDSA signed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>– SHA256 of the FW binary stored in FW header</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cryptographic keys in device</td>
<td>Private AES-CBC key (secret)</td>
<td>Public ECDSA key</td>
<td>Public ECDSA key in X509 certificate chain (stored in STSAFE-A100)</td>
<td>Private AES-GCM key (secret)</td>
</tr>
<tr>
<td></td>
<td>Public ECDSA key</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. For the symmetric cryptographic scheme, it is highly recommended to configure a unique symmetric key for each product.
4 Key management services

Key management services (KMS) middleware provides cryptographic services through the standard PKCS #11 APIs (specified by OASIS) allowing to abstract the key value to the caller (using object ID and not directly the key value). KMS is executed inside a protected/isolated environment in order to ensure that key value cannot be accessed by an unauthorized code running outside the protected/isolated environment.

KMS also offers the possibility to use cryptographic services with keys that are managed securely outside the STM32 microcontroller, such as by an STSAFE-A100 Secure Element for example (rooting based on token ID).

KMS only supports a subset of PKCS #11 APIs:
- Object management functions: creation / update / deletion
- AES encryption functions
- AES decryption functions
- Digesting functions
- RSA and ECDSA Signing/Verifying functions
- Key management functions: key generation/derivation

KMS manages three types of keys:
- Static Embedded keys:
  - Predefined keys embedded within the code. Such keys can't be modified.
- Updatable keys with Static ID:
  - Keys IDs are predefined in the system
  - Key value can be updated in a NVM storage via a secure procedure using static embedded root keys (authenticity check, data integrity check and data decryption)
  - Key cannot be deleted
- Updatable keys with dynamic ID:
  - Key IDs are defined when creating the keys
  - Key value is created using internal functions. Typically, the `DeriveKey()` function creates dynamic objects.
  - Key can be deleted
For more details regarding the OASIS PKCS #11 standard, refer to [5].
5 Protection measures and security strategy

Cryptography ensures integrity, authentication and confidentiality. However, the use of cryptography alone is not enough: a set of measures and system-level strategy are needed for protecting critical operations and sensitive data (such as a secret key), and the execution flow, in order to resist possible attacks.

Secure software coding techniques such as doubling critical tests, doubling critical actions, checking parameters values, and testing a flow control mechanism, are implemented to resist basic fault-injection attacks.

The security strategy is based on the following concepts:

- Ensure single-entry point at reset: force code execution to start with Secure Boot code
- Make SBSFU code and SBSFU secrets immutable: no possibility to modify or alter them once security is fully activated
- Create a protected enclave isolated from SBSFU application and from User applications to store secrets such as keys, and to run critical operations such as cryptographic algorithms
- Limit surface execution to SBSFU code during SBSFU application execution
- Remove JTAG access to the device
- Monitor the system: intrusion detection and SBSFU execution time

*Figure 4 and Figure 5* give a high-level view of the security mechanisms activated on each STM32 Series.

*Figure 4. SBSFU security IPs vs. STM32 Series (1 of 2)*
5.1 STM32L4 Series and STM32L0 Series

Figure 6 illustrates how the system, the code, and the data are protected in the X-CUBE-SBSFU application example for the STM32L4 Series and STM32L0 Series.
Figure 6. STM32L4 and STM32L0 protection overview during SBSFU execution
Protections against outer attacks

Outer attacks refer to attacks triggered by external tools such as debuggers or probes, trying to access the device. In the SBSFU application example, RDP, tamper, DAP and IWDG protections are used to protect product against outer attacks:

- **RDP** (Read Protection): Read Protection Level 2 is mandatory to achieve the highest level of protection and to implement a Root of Trust:
  - External access via the JTAG HW interface to RAM and Flash is forbidden. This prevents attacks aiming to change SBSFU code and therefore mining the Root of Trust.
  - Option bytes cannot be changed. This means that other protections such as WRP and PCROP cannot be changed anymore.

  **Caution** - RDP level 1 is not proposed for the following reasons:
  1. Secure Boot / Root of Trust (single entry point and immutable code) cannot be ensured, because Option bytes (WRP) can be modified in RDP L1.
  2. Device internal flash can be fully reprogrammed (after flash mass erase via RDP L0 regression) with a new FW without any security.
  3. Secrets in RAM memory protected by firewall can be accessed by attaching the debugger via the JTAG HW interface on a system reset.

In case JTAG HW interface access is not possible at customer product, and in case the customer uses a trusted and reliable user application code, then the above-highlighted risks are not valid.

- **Tamper**: the anti-tamper protection is used to detect physical tampering actions on the device and to take related counter measures. In case of tampering detection, the SBSFU application example forces a reboot.

- **DAP** (Debug Access Port): the DAP protection consists in de-activating the DAP (Debug Access Port). Once de-activated, JTAG pins are no longer connected to the STM32 internal bus. DAP is automatically disabled with RDP Level 2.

- **IWDG** (Independent Watchdog): IWDG is a free-running down-counter. Once running, it cannot be stopped. It must be refreshed periodically before it causes a reset. This mechanism allows the control of SBSFU execution duration.

Protections against inner attacks

Inner attacks refer to attacks triggered by code running in the STM32. Attacks may be due to either malicious firmware exploiting bugs or security breaches, or unwanted operations. In the SBSFU application example, WRP, firewall, PCROP, and MPU protections preserve the product from inner attacks:

- **FWALL** (firewall): the firewall is configured to protect the code, volatile data and non-volatile data. Protected code is accessible through a single entry point (the call gate mechanism is described in Appendix A). Any attempt to jump and try to execute any of the functions included in the code section without passing through the entry point generates a system reset.

  In the KMS example, keys and cryptographic services are executed inside the isolated environment under firewall protection.
• **PCROP**

Proprietary code readout protection: a section of Flash is defined as execute-only applying PCROP protection on it: it is not possible to access this section in reading nor writing. Being an execute-only area, a key is protected with PCROP only if it is "embedded" in a piece of code: executing this code moves the key to a specific pointer in RAM. Placed behind the firewall, its execution is not possible from outside.

• **WRP** (write protection): write protection is used to protect trusted code from external attacks or even internal modifications such as unwanted writings/erase operations on critical code/data.

• **MPU** (memory protection unit): the MPU is used to make an embedded system more robust by splitting the memory map for Flash and SRAMs into regions having their own access rights. In the SBSFU application example, MPU is configured in order to ensure that no other code is executed from any memories during SBSFU code execution. When leaving the SBSFU application, the MPU configuration is updated to authorize also the execution of user application code.

---

For the STM32L0 Series, read protection is tightly coupled with write protection: when activated, any read-protected sector is also write-protected. For this reason, read protection cannot be activated.

### 5.2 STM32F4 Series, STM32F7 Series and STM32L1 Series

Figure 7 illustrates how the system, the code, and the data are protected in the X-CUBE-SBSFU application example for the STM32F4 Series, STM32F7 Series, and STM32L1 Series.

---

**Figure 7. STM32F4, STM32F7 and STM32L1 protection overview during SBSFU execution**

---

Legend

- MPU-RX privileged + WRP
- MPU-RX + WRP
- MPU-RW privileged
- MPU-RW

R: Read
W: Write
X: eXecute
Protections against outer attacks

Outer attacks refer to attacks triggered by external tools such as debuggers or probes, trying to access the device. In the SBSFU application example, RDP, tamper, DAP and IWDG protections are used to protect product against outer attacks:

- **RDP** (Read Protection): Read Protection Level 2 is mandatory to achieve the highest level of protection and to implement a Root of Trust:
  - External access via the JTAG HW interface to RAM and Flash is forbidden. This prevents attacks aiming to change SBSFU code and therefore mining the Root of Trust.
  - Option bytes cannot be changed. This means that other protections such as WRP and PCROP cannot be changed anymore.

  **Caution** - RDP level 1 is not proposed for the following reasons:
  1. Secure Boot / Root of Trust (single entry point and immutable code) cannot be ensured, because Option bytes (WRP) can be modified in RDP L1.
  2. Device internal flash can be fully reprogrammed (after flash mass erase via RDP L0 regression) with a new FW without any security.
  3. Secrets in RAM memory protected by firewall can be accessed by attaching the debugger via the JTAG HW interface on a system reset.

  In case JTAG HW interface access is not possible at customer product, and in case the customer uses a trusted and reliable user application code, then the above-highlighted risks are not valid.

- **Tamper**: the anti-tamper protection is used to detect physical tampering actions on the device and to take related counter measures. In case of tampering detection, the SBSFU application example forces a reboot.

- **DAP** (Debug Access Port): the DAP protection consists in de-activating the DAP (Debug Access Port). Once de-activated, JTAG pins are no longer connected to the STM32 internal bus. DAP is automatically disabled with RDP Level 2.

- **IWDG** (Independent Watchdog): IWDG is a free-running down-counter. Once running, it cannot be stopped. It must be refreshed periodically before it causes a reset. This mechanism allows the control of SBSFU execution duration.

Protections against inner attacks

Inner attacks refer to attacks triggered by code running in the STM32. Attacks may be due to either malicious firmware exploiting bugs or security breaches, or unwanted operations. In the SBSFU application example, WRP and MPU protections preserve the product from inner attacks:

- **WRP** (write protection): write protection is used to protect trusted code from external attacks or even internal modifications such as unwanted writing or erase operations on critical code or data.

- **MPU** (memory protection unit): the protected environment managing all critical data and operations (Secure Engine) is isolated from the other software components by leveraging the MPU. The Secure Engine code and data can be accessed only through privileged level of software execution. Therefore, a software running in non-privileged level cannot call the Secure Engine services nor access the critical data. This strict access control to Secure Engine services and resources is implemented by defining specific MPU regions as described in Table 4.
Besides, the MPU also ensures that only authorized code is granted execution permission when the Secure Boot and Secure Firmware Update processes are running. This is the reason why the MPU configuration is updated before launching the user application to authorize its execution. Nevertheless, the Secure Engine isolation settings and supervisor call mechanisms still apply when running the user application (not only when running the SBSFU code).

### 5.3 STM32G0 Series, STM32G4 Series and STM32H7 Series

_Figure 8_ illustrates how the system, the code, and the data are protected in the X-CUBE-SBSFU application example for the STM32G0 Series, STM32G4 Series and STM32H7 Series.

For the specificities of STM32H7B3 devices, refer to appendix _I.3 External Flash on STM32H7B3 devices._

_Figure 8. STM32G0, STM32G4 and STM32H7 protection overview during SBSFU execution_

---

<table>
<thead>
<tr>
<th>Region content</th>
<th>Privileged permission</th>
<th>Unprivileged permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure Engine code &amp; constants</td>
<td>Read Only (execution allowed)</td>
<td>No access</td>
</tr>
<tr>
<td>Secure Engine stack &amp; VDATA</td>
<td>Read Write (not executable)</td>
<td>No access</td>
</tr>
</tbody>
</table>

---

**Table 4. MPU regions in the STM32F4 Series, STM32F7 Series and STM32L1 Series**
Protections against outer attacks

Outer attacks refer to attacks triggered by external tools such as debuggers or probes, trying to access the device. In the SBSFU application example, RDP, tamper, DAP and IWDG protections are used to protect product against outer attacks:

- **RDP** (Read Protection): Read Protection Level 2 is mandatory to achieve the highest level of protection and to implement a Root of Trust:
  - External access via the JTAG HW interface to RAM and Flash is forbidden. This prevents attacks aiming to change SBSFU code and therefore mining the Root of Trust.
  - Option bytes cannot be changed. This means that other protections such as WRP and PCROP cannot be changed anymore.

  **Caution** - RDP level 1 is not proposed for the following reasons:
  1. Secure Boot / Root of Trust (single entry point and immutable code) cannot be ensured, because Option bytes (WRP) can be modified in RDP L1.
  2. Device internal flash can be fully reprogrammed (after flash mass erase via RDP L0 regression) with a new FW without any security.
  3. Secrets in RAM memory protected by firewall can be accessed by attaching the debugger via the JTAG HW interface on a system reset\(^{(1)}\).

In case JTAG HW interface access is not possible at customer product, and in case the customer uses a trusted and reliable user application code, then the above-highlighted risks are not valid.

- **Tamper**: the anti-tamper protection is used to detect physical tampering actions on the device and to take related counter measures. In case of tampering detection, the SBSFU application example forces a reboot.

- **DAP** (Debug Access Port): the DAP protection consists in de-activating the DAP (Debug Access Port). Once de-activated, JTAG pins are no longer connected to the STM32 internal bus. DAP is automatically disabled with RDP Level 2.

- **IWDG** (Independent Watchdog): IWDG is a free-running down-counter. Once running, it cannot be stopped. It must be refreshed periodically before it causes a reset. This mechanism allows the control of SBSFU execution duration.

1. *Not possible on the STM32H7 Series. Refer to Appendix I or more details.*

Protections against inner attacks

Inner attacks refer to attacks triggered by code running in the STM32. Attacks may be due to either malicious firmware exploiting bugs or security breaches, or unwanted operations. In the SBSFU application example, PCROP, WRP and MPU protections preserve the product from inner attacks:
• **PCROP** (proprietary code readout protection): a section of Flash is defined as execute-only applying PCROP protection on it: it is not possible to access this section in reading nor writing. Being an execute-only area, a key is protected with PCROP only if it is "embedded" in a piece of code: executing this code moves the key to a specific pointer in RAM. Placed behind the firewall, its execution is not possible from outside.

• **WRP** (write protection): write protection is used to protect trusted code from external attacks or even internal modifications such as unwanted writings/erase operations on critical code/data.

• **MPU** (memory protection unit): the protected environment managing all critical data and operations (Secure Engine) is isolated from the other software components by leveraging the Memory Protection Unit (MPU). The Secure Engine code and data can be accessed only through privileged level of software execution. Therefore, a software running in non-privileged level cannot call the Secure Engine services nor access the critical data. This strict access control to Secure Engine services and resources is implemented by defining specific MPU regions described in Table 5.

### Table 5. MPU regions in the STM32G0 Series, STM32G4 Series and STM32H7 Series

<table>
<thead>
<tr>
<th>Region content</th>
<th>Privileged permission</th>
<th>Unprivileged permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure Engine code &amp; constants</td>
<td>Read Only (execution allowed)</td>
<td>No access</td>
</tr>
<tr>
<td>Secure Engine stack &amp; VDATA</td>
<td>Read Write (not executable)</td>
<td>No access</td>
</tr>
</tbody>
</table>

Besides, the MPU also ensures that only authorized code is granted execution permission when the Secure Boot and Secure Firmware Update processes are running.

Before launching the user application, the MPU protection is disabled but the secure user memory protection is activated.

• **Secure user memory**: when the secure user memory protection is activated, any access to securable memory area (fetch, read, programming, erase) is rejected, generating a bus error. All the code and secrets located inside the secure user memory (protected environment) is fully hidden. Secure Engine stack and data are cleared when launching the user application as not under secure user memory protection.

*Figure 9* illustrates the closure of the secure user memory when starting the user application.
Figure 9. STM32G0, STM32G4, and STM32H7 protection overview during user application execution
5.4 STM32WB Series

Figure 10 illustrates how the system, the code, and the data are protected in the X-CUBE-SBSFU application example for the STM32WB Series.

Figure 10. STM32WB protection overview during SBSFU execution
Protection measures and security strategy

Protections against outer attacks
Outer attacks refer to attacks triggered by external tools such as debuggers or probes, trying to access the device. In the SBSFU application example, RDP, tamper, DAP and IWDG protections are used to protect product against outer attacks:

- **RDP** (Read Protection): Read Protection Level 2 is mandatory to achieve the highest level of protection and to implement a Root of Trust:
  - External access via the JTAG HW interface to RAM and Flash is forbidden. This prevents attacks aiming to change SBSFU code and therefore mining the Root of Trust.
  - Option bytes cannot be changed. This means that other protections such as WRP and PCROP cannot be changed anymore.

**Caution** - RDP level 1 is not proposed for the following reasons:
1. Secure Boot / Root of Trust (single entry point and immutable code) cannot be ensured, because Option bytes (WRP) can be modified in RDP L1.
2. Device internal flash can be fully reprogrammed (after flash mass erase via RDP L0 regression) with a new FW without any security.
3. Secrets in RAM memory protected by firewall can be accessed by attaching the debugger via the JTAG HW interface on a system reset.

In case JTAG HW interface access is not possible at customer product, and in case the customer uses a trusted and reliable user application code, then the above-highlighted risks are not valid.

- **Tamper**: the anti-tamper protection is used to detect physical tampering actions on the device and to take related counter measures. In case of tampering detection, the SBSFU application example forces a reboot.

- **DAP** (Debug Access Port): the DAP protection consists in de-activating the DAP (Debug Access Port). Once de-activated, JTAG pins are no longer connected to the STM32 internal bus. DAP is automatically disabled with RDP Level 2.

- **IWDG** (Independent Watchdog): IWDG is a free-running down-counter. Once running, it cannot be stopped. It must be refreshed periodically before it causes a reset. This mechanism allows the control of SBSFU execution duration.

Protections against inner attacks
Inner attacks refer to attacks triggered by code running in the STM32. Attacks may be due to either malicious firmware exploiting bugs or security breaches, or unwanted operations. In the SBSFU application example, CKS, WRP and MPU protections preserve the product from inner attacks:

- **CKS** (customer key storage): the SBSFU symmetric key is isolated in the Cortex®-M0+ core secure Flash and therefore cannot be accessed from the Cortex®-M4 core. Before each AES cryptographic decryption/encryption, the Cortex®-M4 core requests the Cortex®-M0+ code to load the key into the AES HW accelerator key register (only accessible from the Cortex®-M0+ core).

- **WRP** (write protection): write protection is used to protect trusted code from external attacks or even internal modifications such as unwanted writings/erase operations on critical code/data. Moreover, WRP allows to protect the SBSFU public key.

- **MPU** (memory protection unit): the MPU is used to make an embedded system more robust by splitting the memory map for Flash and SRAMs into regions having their own access rights. In the SBSFU application example, the MPU is configured in order to ensure that no other code is executed from any memory during SBSFU code
execution. When leaving the SBSFU application, the MPU configuration is updated to authorize also the execution of user application code.

5.5 STM32L4 Series combined with STSAFE-A100

Figure 10 illustrates how the system, the code, and the data are protected in the X-CUBE-SBSFU application example featuring the STM32L4 Series combined with STSAFE-A100.

Figure 11. STM32L4 / STSAFE-A100 protection overview during SBSFU execution

* Includes STSAFE-A middleware and IPC driver.
STM32 microcontroller protections against outer attacks

Outer attacks refer to attacks triggered by external tools such as debuggers or probes, trying to access the device. In the SBSFU application example, RDP, tamper, DAP and IWDG protections are used to protect product against outer attacks:

- **RDP** (Read Protection): Read Protection Level 2 is mandatory to achieve the highest level of protection and to implement a Root of Trust:
  - External access via the JTAG HW interface to RAM and Flash is forbidden. This prevents attacks aiming to change SBSFU code and therefore mining the Root of Trust.
  - Option bytes cannot be changed. This means that other protections such as WRP and PCROP cannot be changed anymore.

  **Caution** - RDP level 1 is not proposed for the following reasons:
  1. Secure Boot / Root of Trust (single entry point and immutable code) cannot be ensured, because Option bytes (WRP) can be modified in RDP L1.
  2. Device internal flash can be fully reprogrammed (after flash mass erase via RDP L0 regression) with a new FW without any security.
  3. Secrets in RAM memory protected by firewall can be accessed by attaching the debugger via the JTAG HW interface on a system reset.

In case JTAG HW interface access is not possible at customer product, and in case the customer uses a trusted and reliable user application code, then the above-highlighted risks are not valid.

- **Tamper**: the anti-tamper protection is used to detect physical tampering actions on the device and to take related counter measures. In case of tampering detection, the SBSFU application example forces a reboot.

- **DAP** (Debug Access Port): the DAP protection consists in de-activating the DAP (Debug Access Port). Once de-activated, JTAG pins are no longer connected to the STM32 internal bus. DAP is automatically disabled with RDP Level 2.

- **IWDG** (Independent Watchdog): IWDG is a free-running down-counter. Once running, it cannot be stopped. It must be refreshed periodically before it causes a reset. This mechanism allows the control of SBSFU execution duration.

STM32 microcontroller protections against inner attacks

Inner attacks refer to attacks triggered by code running in the STM32. Attacks may be due to either malicious firmware exploiting bugs or security breaches, or unwanted operations. In the SBSFU application example, WRP, firewall, PCROP, and MPU protections preserve the product from inner attacks:

- **FWALL** (firewall): the firewall is configured to protect the code, volatile data and non-volatile data. Protected code is accessible through a single entry point (the call gate mechanism is described in Appendix A). Any attempt to jump and try to execute any of the functions included in the code section without passing through the entry point generates a system reset.
• **PCROP** (proprietary code readout protection): a section of Flash is defined as execute-only applying PCROP protection on it: it is not possible to access this section in reading nor writing. Being an execute-only area, a key is protected with PCROP only if it is "embedded" in a piece of code: executing this code moves the key to a specific pointer in RAM. Placed behind the firewall, its execution is not possible from outside.

• **WRP** (write protection): write protection is used to protect trusted code from external attacks or even internal modifications such as unwanted writings/erase operations on critical code/data.

• **MPU** (memory protection unit): the MPU is used to make an embedded system more robust by splitting the memory map for Flash and SRAMs into regions having their own access rights. In the SBSFU application example, MPU is configured in order to ensure that no other code is executed from any memories during SBSFU code execution. When leaving the SBSFU application, the MPU configuration is updated to authorize also the execution of user application code.

**STSAFE-A Secure Element protections**

The STSAFE-A100 is a highly secure solution with a secure operating system running on the latest generation of secure microcontrollers:

• **Security features**: The chip is CC EAL5+ AVA_VAN5 Common Criteria certified and provides the following protections.
  – Active shield
  – Monitoring of environmental parameters
  – Protection mechanism against faults
  – Unique serial number on each die
  – Protection against side-channel attacks

• **Secure operating system**: STSAFE-A100 runs a secure operating system offering protection against logical and physical attacks.

• **Secure channel and device binding**: STSAFE-A100 allows a secure channel to be set up with the STM32 in order to prevent eavesdropping of sensitive information on the I²C line and to ensure pairing of a specific STM32 with a specific STSAFE-A100 (to prevent cloning).

The secure channel is based on symmetric cryptography: two AES 128-bit keys (the so-called host pairing keys) are used to implement services such as command authorization, command data encryption, response data encryption and response authentication.
6 Package description

This section details the X-CUBE-SBSFU package content and the way to use it.

6.1 General description

X-CUBE-SBSFU is a software package for STM32 microcontrollers. It provides a complete solution to build Secure Boot and Secure Firmware Update applications:

- Support of symmetric and asymmetric cryptography approaches with the AES-GCM, AES-CBC, and ECDSA algorithms for decryption, verification, or both with the use of X-CUBE-CRYPTOLIB.
- Support of X509 certificate chain verification of firmware image and firmware updates.(a)
- Two modes of operation:
  - The dual-image mode, which enables safe image programming, with resume capability in case of an interruption of the installation procedure
  - The single-image mode, which maximizes the user application size
- Integration of security peripherals and mechanisms in order to implement a SBSFU Root of Trust. RDP, WRP, PCROP, firewall, MPU, secure user memory, tamper, and IWDG are combined to achieve the highest security level(b).
- Use of a Secure Engine (SE) module as part of the middleware in order to provide a protected environment managing all critical data and operations such as secure key storage, cryptographic operations and others.
- Integration of secure key management services (KMS) offering symmetric and asymmetric cryptographic services via the PKCS #11 11 APIs and offering secure key storage, update services.
- Integration of the STSAFE-A100 secure element to provide the system with a tamper resistant (CC EAL5+ AVA VAN5 Common Criteria certified) Root of Trust, to offload the host MCU of ECDSA cryptographic operations and simplify key provisioning during manufacturing.
- Availability of the user application example source code.
- Availability of the firmware image preparation tool, provided both as executable and source code.

X-CUBE-SBSFU is ported on the STM32F4 Series, STM32F7 Series, STM32G0 Series, STM32G4 Series, STM32H7 Series, STM32L0 Series, STM32L1 Series, STM32L4 Series, and STM32WB Series. X-CUBE-SBSFU is also ported on the STM32L4 Series combined with STSAFE-A100, the X-NUCLEO-SAFE1A expansion board is supported for STSAFE-A100.

The package includes sample applications that the developer can use to start experimenting with the code.

---

(a) Specific to the STSAFE-A100.

(b) The availability of security IPs depends on the STM32 Series.
The package is provided as a zip archive containing source-code.

The following integrated development environments are supported:
- STM32CubeIDE and System Workbench for STM32 (SW4STM32)
- IAR Embedded Workbench® for Arm® (EWARM)
- Keil® Microcontroller Development Kit (MDK-ARM)

Note: The KMS feature is available on the STM32L4 Series with example provided on the B-L475E-IOT01A board.
The STSAFE-A100 feature is available on the STM32L4 Series with example provided on the B-L475E-IOT01A board

6.2 Architecture

This section describes the software components of the X-CUBE-SBSFU package illustrated in Figure 12.

Figure 12. Software architecture overview
6.2.1 STM32CubeHAL

The HAL driver layer provides a generic multi instance simple set of APIs (application programming interfaces) to interact with the upper layers (application, libraries and stacks). It is composed of generic and extension APIs. It is directly built around a generic architecture and allows the layers that are built upon, such as the middleware layer, implementing their functionalities without dependencies on the specific hardware configuration for a given microcontroller unit (MCU).

This structure improves the library code reusability and guarantees an easy portability onto other devices.

6.2.2 Board support package (BSP)

The software package needs to support the peripherals on the STM32 boards apart from the MCU. This software is included in the board support package (BSP). This is a limited set of APIs which provides a programming interface for certain board specific peripherals such as the LED and the User button.

6.2.3 Cryptographic Library

Two different cryptographic middleware are supported:

- X-CUBE-CRYPTOLIB supports symmetric and asymmetric key approaches (AES-GCM, AES-CBC, ECDSA) as well as hash computation (SHA256) for decryption and verification. SW cryptographic functions are used to avoid storing secret keys in HW Crypto IP registers that are not protected.

- mbed TLS cryptographic services delivered as open source code. Similarly as for X-CUBE-CRYPTOLIB, symmetric and asymmetric key approaches (AES-GCM, AES-CBC, ECDSA) as well as hash computation (SHA256) for decryption and verification are supported. Examples are provided for the 32L496GDISCOVERY, B-L475E-IOT01A, 32F413HDISCOVERY, and 32F769IDISCOVERY boards under folder 2_Images_OSC.

6.2.4 Secure Engine (SE) middleware

The Secure Engine middleware provides a protected environment to manage all critical data and operations (such as cryptography operations accessing firmware encryption key, and others). Protected code and data are accessible through a single entry point (call gate mechanism) and it is therefore not possible to run or access any SE code or data without passing through it, otherwise a system reset is generated (refer to Appendix A to get details about call gate mechanism).

Note: Secure Engine critical operations can be extended with other functions depending on user application needs. Only trusted code is to be added to the Secure Engine environment because it has access to the secrets.

6.2.5 Key management services (KMS) middleware

The secure key management services provide cryptographic services to the user application through the PKCS #11 APIs (KEY ID based APIs) that are executed inside the secure enclave. User Application keys are stored in the secure enclave and can be updated in a secure way (authenticity check, decryption, and integrity check before update).
6.2.6 **STSAFE-A middleware**

STSAFE-A middleware provides a complete set of APIs to access all the STSAFE-A100 device features from STM32 microcontrollers.

It integrates both low-level communication drivers to interface with the STSAFE-A100 hardware, and a higher-level processing exporting a set of command APIs to easily access the device features from the STM32 microcontroller.
6.2.7 Secure Boot and Secure Firmware Upgrade (SBSFU) application

Secure Boot (Root of Trust)
- Checks and applies the security mechanisms of STM32 platform to protect critical operations and secrets from attacks
- Authenticates and verifies the user application before each execution

Local download via UART Virtual COM
- Detects firmware download requests
- Downloads in STM32 Flash memory the new encrypted firmware image (header + encrypted firmware) via the UART Virtual COM using Ymodem protocol and the Tera Term tool (see Note)

FW installation management
- Detects new FW version to install
  - From local download service via the UART interface
  - Downloaded via user application (dual-image variant only)
- Secures FW upgrade:
  - Authentication and integrity check
  - FW decryption
  - FW installation
  - Anti-rollback mechanisms to avoid re-installation of previous firmware version
- Supports single image for maximizing the user application size
- Supports dual image for safe image programming
  - Resume firmware installation: in case of power off during the installation process, installation is resumed at next power on.
  - Multiple firmware images management: handles two firmware images (UserApp1 image and UserApp2 image) stored in internal STM32 Flash. A SWAP area is used in order to limit memory overhead needed during firmware installation (refer to Appendix B to get details about multiple images management).
  - Partial update: flexibility to update the complete firmware image or a portion of it.

Note
For the STM32WB Series, an example of standalone loader is provided. Refer to Appendix H for details.
6.2.8 User application

- Provides an example for downloading the user application via Ymodem protocol over a UART (Over The Air download mechanisms, such as BLE, Wi-Fi® or others, can be implemented in the user application but are not provided as examples in the X-CUBE-SBSFU application example).
- Provides examples testing the protections mechanisms.
- Provides an example for using some of the functionalities exported by SE such as getting information about the current firmware image.
- Provides examples using KMS exported services through a standard PKCS #11 interface: AES-GCM/CBC encryption/decryption, RSA signature/verification, key provisioning, and key derivation.
- Provides examples to demonstrate STSAFE-A exported services through a standard PKCS #11 interface: ECDSA signature generation using a device-unique private key, device certificate reading, signature verification, ECDSA key pair generation, ECDH Diffie-Hellman key derivation. These services are typically used in the context of a TLS exchange and are candidate building blocks for the development of an IoT node connected with a cloud-based service.
6.3 Folder structure

A top-level view of the folder structure is shown in Figure 13 and Figure 14.

Figure 13. Project folder structure (1 of 2)
6.4 APIs

Detailed technical information about the APIs is provided in a compiled HTML file located in the `STM32_Secure_Engine`, `STM32_Key_Management_Services`, and `STSAFE_A1xx` folders of the software package where all the functions and parameters are described.

6.5 Application compilation process with IAR™ toolchain

*Figure 15* outlines the steps needed in order to build the application and to demonstrate Secure Boot and Secure Firmware Update:

- **Step 1: Core binaries preparation**
  
  This step is needed to create the Secure Engine core binary including all the "trusted" code and keys mapped in the firewall code section. The SE Callgate function is specified as the entry point for the binary. The binary is linked with the SBSFU code in step 2.
- **Step 2: SBSFU**
  This step compiles the SBSFU source code implementing the state machine and configuring the protections. In addition, it links the code with the SECore binary generated at step 1 in order to generate a single SBSFU binary including the SE trusted code. It also generates a file including symbols for the user application to call the SE interface methods, a set of user-friendly APIs wrapping the single SE call gate API.

- **Step 3: user application example**
  It generates:
  - The user application binary file that is uploaded to the device using the Secure Firmware Update process (UserApp.sfb).
  - A binary file concatenating the SBSFU binary, the user application binary in clear format, and the corresponding FW header.

These three elements are placed properly for both the SBSFU and user application to run when the binary file is flashed into the device with a flasher tool. Hence, no FW installation procedure is required for SBSFU to start and boot the user application. This is a convenient way to test the user application with a single flashing stage.

_Note:_ Refer to Appendix H for specificities of the STM32WB Series.

**Figure 15. Application compilation steps**
7 Hardware and software environment setup

This section describes the hardware and software setup procedures.

7.1 Hardware setup

To set up the hardware environment, one of the supported boards introduced in Section 6.1: General description on page 35 must be connected to a personal computer via a USB cable. This connection with the PC allows the user:

- Flashing the board
- Interacting with the board via a UART console
- Debugging when the protections are disabled

7.2 Software setup

This section lists the minimum requirements for the developer to setup the SDK, run the sample scenario, and customize applications.

7.2.1 Development toolchains and compilers

Select one of the Integrated Development Environments supported by the STM32Cube Expansion Package.

Take into account the system requirements and setup information provided by the selected IDE provider.

7.2.2 Software tools for programming STM32 microcontrollers

**ST-LINK utility**

STM32 ST-LINK Utility (STSW-LINK004) is a full-featured software interface for programming STM32 microcontrollers. It provides an easy-to-use and efficient environment for reading, writing and verifying a memory device.

Refer to the STSW-LINK004 STM32 ST-LINK Utility software on www.st.com.

**Caution:** Make sure to use an up-to-date version of ST-LINK (V2.J29 or later).

**STM32CubeProgrammer**

STM32CubeProgrammer (STM32CubePro) is an all-in-one multi-OS software tool for programming STM32 microcontrollers. It provides an easy-to-use and efficient environment for reading, writing and verifying device memory through both the debug interface (JTAG and SWD) and the bootloader interface (UART and USB).

STM32CubeProgrammer offers a wide range of features to program STM32 microcontroller internal memories (such as Flash, RAM, and OTP) as well as external memories. STM32CubeProgrammer also allows option programming and upload, programming content verification, and microcontroller programming automation through scripting.

STM32CubeProgrammer is delivered in GUI (graphical user interface) and CLI (command-line interface) versions.
Refer to the STM32CubeProgrammer software (STM32CubeProg) on www.st.com.

7.2.3 Terminal emulator

A terminal emulator software is needed to run the demonstration.

The example in this document is based on Tera Term, an open source free software terminal emulator that can be downloaded from the https://osdn.net/projects/ttssh2/ webpage. Any other similar tool can be used instead (Ymodem protocol support is required).

7.2.4 X-CUBE-SBSFU firmware image preparation tool

The X-CUBE-SBSFU Expansion Package for STM32Cube is delivered with the *prepareimage* tool handling the cryptographic keys and firmware image preparation.

The *prepareimage* tool is delivered in two formats:

- Windows® executable: the standard Windows® command interpreter is required
- Python™ scripts: a Python™ interpreter as well as the elements listed in Middlewares\ST\STM32_Secure_Engine\Utilities\KeysAndImages\readme.txt are required

The Windows® executable is fully integrated in the supported IDEs and compilation process as shown in Figure 16.

![Figure 16. Firmware image preparation tool IDE integration](image)

More information about the preparation tool are provided in Appendix E: Firmware image preparation tool on page 78.
8  Step-by-step execution

The following steps describe a dual-image SBSFU scenario executed on NUCLEO-L476RG board with the default cryptographic scheme, further illustrated in Figure 17:

1. Download SBSFU application
2. SBSFU is running : download UserApp #A
3. UserApp #A is installed
4. UserApp #A is running, download UserApp #B
5. UserApp #B is installed then running

The UserApp#A and UserApp#B binaries are generated on the basis of the user application example project. Defining the application as #A or #B is done by changing the value of the UserAppId variable declared in the main.c of the application.

Figure 17. Step-by-step execution
8.1 **STM32 board preparation**

The target Option bytes setting is the following for the NUCLEO-L476RG board:

- RDP Level 0 is set
- Write protection is disabled on all Flash pages
- PCROP protection is disabled\(^{(a)}\)
- BFB2 bit disabled
- Chip is erased\(^{(b)}\)

Option bytes setting can differ from one STM32 Series to another as illustrated in Figure 18.

---

*a. Automatically done when switching from RDP level 1 to RDP level 0 except on STM32H7 Series.*

*b. Automatically done when switching from RDP level 1 to RDP level 0.*

---

**Figure 18. STM32 board preparation**

![Diagram showing the relationship between different settings and chip models](image)
Option bytes setting is verified by means of the STM32CubeProgrammer through the following four steps:

1. **Connection**: Menu Target / Connect with Under reset mode selected (refer to *Figure 19*)

*Figure 19. STM32CubeProgrammer connection menu*
2. Option bytes settings: Menu Target / Option bytes (refer to Figure 20)

Figure 20. STM32CubeProgrammer Option bytes screen

The STM32CubeProgrammer Option bytes screen is specific to the STM32 microcontroller series.

3. Erase chip: Menu Target / Erase Chip

Figure 21. STM32CubeProgrammer erasing

4. Disconnect: Menu Target / Disconnect

8.2 Application compilation

With the selected toolchain (IAR, Keil, or System Workbench) rebuild all the projects as explained in Section 6.5: Application compilation process with IAR™ toolchain on page 42.
Download the SB SFU project software to the target without starting a debug session (Security protections managed by SBSFU forbid JTAG connection as it is interpreted as an external attack).

8.3 Tera Term connection

Tera Term connection is achieved by applying in sequence the steps described from Section 8.3.1 to Section 8.3.4.

8.3.1 ST-LINK disable

The security mechanisms managed by SBSFU forbid JTAG connection (interpreted as an external attack). The ST-LINK must be disabled to establish a Tera Term connection. The following procedure applies from ST-LINK firmware version V2J29 onwards:

a) Power cycle the board after flashing SBSFU (unplug/plug the USB cable).
b) The SBSFU application starts and configures the security mechanisms in development mode. In product mode, security mechanisms are only checked to be at the correct values.
c) Power cycle the board a second time (unplug/plug the USB cable): the SBSFU application starts with the configured securities turned on and the Tera Term connection is possible.

8.3.2 Tera Term launch

The Tera Term launch requires that the port is selected as COMxx: STMicroelectronics STLink Virtual COM port.

Figure 22 illustrates an example based on the selection of port COM54.

Figure 22. Tera Term connection screen

a. Make sure the ST-LINK debugger/programmer embedded on the board runs the proper firmware version (V2J29 or higher). If this is not the case, please upgrade this firmware first.
8.3.3 Tera Term configuration

The Tera Term configuration is performed through the General and Serial port setup menus. Figure 22 illustrates the General setup and Serial port setup menus.

Figure 23. Tera Term setup screen

A configuration is saved using Menu Setup / Save Setup.

Caution: After each plug / unplug of the USB cable, the Tera Term Serial port setup menu may have to be validated again to restart the connection. Press the Reset button to display the welcome screen.
8.3.4 Welcome screen display

The welcome screen is displayed on Tera Term as illustrated in Figure 24.

Figure 24. SBSFU welcome screen display

```
(C) COPYRIGHT 2017 STMicroelectronics
Secure Boot and Secure Firmware Update

=(SBOOT) SECURE ENGINE INITIALIZATION SUCCESSFUL
=(SBOOT) STATE: CHECK STATUS ON RESET
    INFO: A Restart has been triggered by a Hardware reset!
    Consecutive Boot error counter = 0
    INF: Last execution detected error was:No error. Success.
=(SBOOT) STATE: CHECK NEW FIRMWARE TO DOWNLOAD
=(SBOOT) STATE: CHECK USER FW STATUS
No valid File found in the active slot nor new encrypted File found in the UserApp download area
Waiting for the local download to start...
=(SBOOT) STATE: DOWNLOAD NEW USER FIRMWARE
File> Transfer> YMODEM> Send ....
```

8.4 SBSFU application execution

At each reboot, the application checks if the user has requested a new firmware download by keeping the User button pressed.

If there is no download request, the application checks the status of the user firmware:

- Since the board was erased, no firmware is available.
- The application cannot jump to firmware and goes back to check if there is a download request.

8.4.1 Download request

When no user firmware is present, SBSFU automatically waits for the download procedure to start. Otherwise, the download request is obtained by holding the User button on the STM32 Nucleo board.

8.4.2 Send firmware

For sending the firmware (*.sfb), use the File > Transfer > YMODEM > Send menu in Tera Term as shown in Figure 25.
Once the UserApp.sfb file is selected, the Ymodem transfer starts. Transfer progress is reported as shown in Figure 26.

The progress gauge stalls for a short time at the beginning of the procedure while SBSFU verifies the firmware header validity and erases the Flash slot where the firmware image is downloaded.
### 8.4.3 File transfer completion

After the file transfer is completed, the system forces a reboot as shown in Figure 27.

**Figure 27. SBSFU reboot after encrypted firmware transfer**

```
--- (BOOT) STATE: DOWNLOAD NEW USER FIRMWARE
    File: Transfer> MODEM> Send...
    (BOOT) STATE: REBOOT STATE MACHINE
    ========== End of Execution ==========

--- (BOOT) System Security check successfully passed. Starting...
    (FIRM) Slot 0: 800000 / Slot 1: 800000 / Swap 0: 80000

--- (C) COPYRIGHT 2017 STMicroelectronics
    =
    =
    =

--- Secure Boot and Secure Firmware Update
--- (C) COPYRIGHT 2017 STMicroelectronics
    =
    =
---

--- (BOOT) SECURE ENGINE INITIALIZATION SUCCESSFUL
--- (BOOT) STATE: CHECK STATUS ON RESET
    INFO: A reboot has been triggered by a Software reset!
    Consecutive Boot on error counter = 0
    INFO: Last execution detected error was No error, Success.
--- (BOOT) STATE: CHECK NEW FIRMWARE TO DOWNLOAD
--- (BOOT) STATE: CHECK USER FW STATUS
    New Fu Encrypted, to be decrypted
--- (BOOT) STATE: INSTALL NEW USER FIRMWARE
--- (BOOT) STATE: VERIFY USER FW SIGNATURE
--- (BOOT) STATE: EXECUTE USER FIRMWARE

--- (C) COPYRIGHT 2017 STMicroelectronics
    =
    =
--- User App #1
---

--- Main Menu
---

    Download a new Fu Image 1
    Test Protections 2
    Test SE User Code 3
```

The system status that is printed as shown in Figure 27 consequently provides the following information:

- There is no firmware to download.
- The firmware is detected as encrypted. The user firmware is decrypted.
- If the decryption is OK, the user firmware is installed.
- If the installation is OK, the user firmware signature is verified.
- If the verification is OK, the user firmware is executed.
8.4.4 System restart
Pressing the Reset button forces the system to restart: the user application is started by SBSFU.

Note: Holding the User button during reset, triggers the forced download state instead of the user application execution.

8.5 User application execution
The user application is executed according to the selection illustrated in Figure 28 and further described from Section 8.5.1 to Section 8.5.3.

Figure 28. User application execution

8.5.1 Download a new firmware image
The download of a new firmware image is performed through the same steps as those presented for SBSFU in Section 8.4 on page 52:
1. Send firmware
   - In Tera Term, click on File>Transfer>YMODEM>Send
   - Select UserApp.sfb (compiled as UserApp#B)
2. The system reboots
3. The Secure Boot state machine handles the new image
   - Firmware header is verified
   - Firmware is decrypted
   - Firmware is installed
   - Firmware signature is verified
   - Firmware is executed
Figure 29. Encrypted firmware download via user application

```
=======================================
| New Fu Download                      |
| --- Send Firmware                     |
| --- Erasing download area ...         |
| --- Programming Completed Successfully!|
| --- Bytes: 17872                       |
| --- Image correctly downloaded - reboot|

= [SBOOT] System Security check successfully passed. Starting...
= [F-MING] Slot #0: 80000000 / Slot #1 c: 00000000 / Swap c: 50000000

=======================================
| (C) COPYRIGHT 2017 STMicroelectronics |
| Secure Boot and Secure Firmware Update|

=======================================
| (SBOOT) SECURE ENGINE INITIALIZATION SUCCESSFUL |
| (SBOOT) STATE: CHECK STATUS ON RESET           |
| INFO: A Reboot has been triggered by a Software reset! |
| INFO: Last execution detected error was:No error. Success. |
| (SBOOT) STATE: CHECK NEW FIRMWARE TO DOWNLOAD |
| (SBOOT) STATE: CHECK USER Fu STATUS            |
| New Fu Encrypted, to be decrypted              |
| (SBOOT) STATE: INSTALL NEW USER FIRMWARE       |
| (SBOOT) STATE: VERIFY USER Fu SIGNATURE        |
| (SBOOT) STATE: EXECUTE USER FIRMWARE           |

=======================================
| (C) COPYRIGHT 2017 STMicroelectronics.cs     |
| User App #8                                  |

=======================================
| Main Menu                                  |
| Download a new Fu Image                    | 1 |
| Test Protections                           | 2 |
| Test SE User Code                          | 3 |
```
8.5.2 Test protections

An example of the test protection menu is shown in Figure 30. The actual menu depends on the STM32 Series.

![Figure 30. User application test protection menu](image)

The test protection menu is printed at each test attempt of a prohibited operation or error injection as a function of the test run:

- Firewall tests (#1, #2)
  - Causes a reset trying to access protected code or data (either in RAM or Flash)
- PCROP test (#3)
  - Causes an error trying to access the PCROP region protecting the keys
- WRP test (#4)
  - Causes an error trying to erase write protected code
- IWDG test (#5)
  - Causes a reset simulating a deadlock by not refreshing the watchdog
- TAMPER test (#6)
  - Causes a reset if a tamper event is detected
  - In order to generate a tamper event, the user must connect PA0 (CN7.28) to GND (It may be enough to put a finger close to CN7.28).
- CORRUPT IMAGE test (#7)
  - Causes a signature verification failure at next boot.

Returning to the previous menu is obtained by pressing the x key.

8.5.3 Test Secure Engine user code

The version and size of the current user firmware are retrieved by means of a Secure Engine service, and printed in the console.
## Understanding the last execution status message at boot-up

Table 6 lists the main error messages together with their explanation.

<table>
<thead>
<tr>
<th>Error message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>No error. Success.</td>
<td>No problem encountered.</td>
</tr>
<tr>
<td>Firewall error.</td>
<td>A firewall exception occurred: some code or data protected by the firewall has been addressed out of the Secure Engine context.</td>
</tr>
<tr>
<td>Watchdog error.</td>
<td>Watchdog expiry: a processing is too long and the watchdog has not been reloaded in due time.</td>
</tr>
<tr>
<td>Memory fault.</td>
<td>Memory fault reported by the MPU fault handler.</td>
</tr>
<tr>
<td>Hard fault.</td>
<td>Arm® Cortex®-M hard fault exception.</td>
</tr>
<tr>
<td>Tampering fault.</td>
<td>TAMPER-detection report.</td>
</tr>
<tr>
<td>Check protections error.</td>
<td>Not used in the example code. This can be used to log errors when doing the periodic verification of applied protection mechanisms.</td>
</tr>
<tr>
<td>Check status on reset error.</td>
<td>Error encountered while checking the status at boot up (generic error).</td>
</tr>
<tr>
<td>Check new user FW to download error.</td>
<td>Error encountered while checking if there is a local download request (generic error).</td>
</tr>
<tr>
<td>Download new user FW error.</td>
<td>Error encountered while performing a local download (generic error).</td>
</tr>
<tr>
<td>Verify user FW status error.</td>
<td>Error encountered while verifying the status of the user firmware. This error is reached when the Flash state does not allow determining the firmware status (generic error).</td>
</tr>
<tr>
<td>Decrypt user FW error.</td>
<td>Not used in the current example code. This can be used to log generic errors related to the decrypt of a firmware. In the example, a more specific error is used: “Decrypt failure.”</td>
</tr>
<tr>
<td>Install user FW error.</td>
<td>Error encountered during the installation of a new firmware (generic error).</td>
</tr>
<tr>
<td>Verify user FW signature.</td>
<td>Error encountered while verifying the signature of the active firmware. In the example code, the signature is already checked during the firmware status check so this error is not supposed to be reported.</td>
</tr>
<tr>
<td>Resume FW installation error.</td>
<td>Error encountered during firmware installation procedure.</td>
</tr>
<tr>
<td>Execute user FW error.</td>
<td>Error encountered while trying to launch the active firmware (generic error).</td>
</tr>
<tr>
<td>SE lock cannot be set.</td>
<td>Error encountered while trying to configure Secure Engine in “Firmware execution” mode (unprivileged mode) before starting the active firmware.</td>
</tr>
<tr>
<td>Inconsistent FW size.</td>
<td>This error means that during a local download procedure the size indicated in the header does not match the size of the downloaded file.</td>
</tr>
<tr>
<td>FW too big.</td>
<td>This error means that during a local download procedure the header indicated a firmware size bigger than the capacity of the Slot #1.</td>
</tr>
<tr>
<td>Ymodem com failure.</td>
<td>During a local download procedure, the download operation did not complete successfully (Ymodem protocol issue).</td>
</tr>
</tbody>
</table>
Table 6. Error messages at boot-up (continued)

<table>
<thead>
<tr>
<th>Error message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>File not correctly received.</td>
<td>During a local download procedure, the binary file has not been received properly. At the moment this detection is minimalist.</td>
</tr>
<tr>
<td>Header authentication failed.</td>
<td>During a local download procedure, the header could not be authenticated successfully. This error is reached only if the header stored in RAM is altered (otherwise the download is bypassed without triggering a critical failure).</td>
</tr>
<tr>
<td>Decrypt failure.</td>
<td>Error encountered while decrypting the content of Slot #1. This error reports a decryption or an authentication issue as the final stage of the decryption is a check of the signature.</td>
</tr>
<tr>
<td>Signature check failure.</td>
<td>Error encountered while verifying the signature of the decrypted firmware during an installation procedure. In the example code, this error should not be reached as a signature issue would be captured at decrypt stage (reporting “Decrypt failure.”).</td>
</tr>
<tr>
<td>Incorrect binary format (not encrypted).</td>
<td>Error encountered during an installation procedure: the binary present in Slot #1 is not encrypted. Maybe the clear version of the firmware instead of the encrypted version was downloaded?</td>
</tr>
<tr>
<td>Flash error.</td>
<td>Flash error encountered during an installation procedure.</td>
</tr>
<tr>
<td>FWIMG pattern issue.</td>
<td>Error encountered during an installation procedure: internal issue while writing some SBSFU patterns.</td>
</tr>
<tr>
<td>Error while swapping the images in Slot #0 and Slot #1.</td>
<td>Error encountered during an installation procedure: failure while swapping the images (previous firmware and decrypted firmware).</td>
</tr>
<tr>
<td>Firmware version rejected by anti-rollback check.</td>
<td>Error encountered during an installation procedure: the firmware version cannot be accepted (newer firmware already installed or lower version than min. allowed version).</td>
</tr>
<tr>
<td>Unknown error.</td>
<td>Undocumented error (unexpected exception or unexpected state machine issue).</td>
</tr>
</tbody>
</table>
Appendix A  Secure Engine protected environment

The Secure Engine (SE) concept defines a protected enclave exporting a set of secure functions executed in a trusted environment.

The following functionalities are provided by SE to the SBSFU application example:

- Secure Engine initialization function
- Secure cryptographic functions
  - AES-GCM and AES-CBC decryption
  - SHA256 hash and ECDSA verification
  - Sensitive data (secret key, AES context) never leaves the protected environment and cannot be accessed from unprotected code
- Secure read/write access to firmware image information
  - Read and write operation on a protected Flash area
  - Access to this area is allowed only to protected code
- Secure service to lock some functions in Secure Engine
  - One way lock mechanism: once locked, no way to unlock it except via a system reset
  - Once locked, functions execution is no more possible via call gate mechanism
  - Functionalities that are locked via the lock mechanism in Secure Engine example:
    - Secure Engine initialization function
    - Secure Encryption functions with OEM key
    - Secure read/write access to firmware image information
    - Secure service to lock some functions in Secure Engine

Note: Functionalities exported by SE can be extended depending on final user applications needs

In the KMS variant, SE functionalities are extended with the secure key management services providing cryptographic services to the user application through the PKCS #11 APIs (KEY ID based APIs) are executed inside the Secure Engine protected environment. User Application keys are stored in this protected/isolated environment.

In the STSAFE-A variant, KMS is extended with STSAFE-A middleware, so as to provide access to keys and services provided by the Secure Element through a standard interface. Communication with the STSAFE-A100 is secured with symmetric keys stored in the protected/isolated environment.

In order to deal with the firewall callgate mechanism and to provide the user with a set of secure APIs, SE is designed with a two-level architecture, composed of SE Core and SE Interface.

The call gate concept and the two-level architecture apply also when using the MPU to protect the Secure Engine, as described in Section A.2: MPU-based Secure Engine Isolation.
A.1 Firewall-based Secure Engine Isolation

A.1.1 SE core call gate mechanism

The firewall is opened or closed using a specific "call gate" mechanism: a single entry point (placed at the 2nd word of the Code segment base address) must be used to open the gate and to execute the code protected by the firewall. If the protected code is accessed without passing through the call gate mechanism then a system reset is generated.

As the only way to respect the call gate sequence is to pass through the single call gate entry point, therefore, if the application requires to have multiple functions protected by the firewall and called from unprotected code outside it (e.g. encrypt and decrypt functions), a way to select which of the internal functions to execute is needed. A solution is to use a parameter to specify which function to execute, for instance CallGate(F1_ID), CallGate(F2_ID), and so on. According to the parameter, the right function is internally called.

**Figure 31. Firewall call gate mechanism**

Caution: The code section must include all the code executed when the firewall is open. For instance, if the call sequence is callgate->f1()->f1a()->f1b(), all the three functions f1(), f1a() and f1b() must be included in the code section.

*Figure 32* shows the steps to perform cryptographic operations (that require access to the key) in order to respect the call gate mechanism.

For the cryptographic functions:
1. The SBSFU code calls the call gate function in order to open the firewall and to execute protected code
2. The call gate function checks parameters and securities and then calls the requested Crypto function
3. The SE Crypto functions calls an internal ReadKey function that moves the keys into the protected section of SRAM1 and then use them in the cryptographic operations.

Figure 32. Secure Engine call-gate mechanism

A.1.2 SE interface

Code protected by the firewall must be non-interruptible and it is up to the user code to disable interrupts before opening the firewall.

SE interface provides a user-friendly wrapper handling the entrance and exit to a protected enclave where the actual SE call gate function is executed as illustrated in Figure 33.
SE interface mechanism simplifies the control access to the call gate independent from user implementation. SE interface APIs are shared with the user application, which therefore executes sensitive operations (if not locked via the Secure Engine lock service) in a secure way using the services provided by SE.

A.2 MPU-based Secure Engine Isolation

A.2.1 Principle

The MPU-based Secure Engine isolation relies on the concept of privileged and unprivileged levels of software execution. The software must run in unprivileged level of execution by default (when SBSFU or the User Application is running), except for very specific actions like platform initialization or interrupt handling. This is described in Figure 34.
When the software runs in unprivileged mode, any attempt to access the Secure Engine code or data results in an MPU fault: this ensures the isolation of the critical assets.

This isolation of the Secure Engine is implemented thanks to specific MPU regions as shown in Table 7.

### Table 7. MPU regions for Secure Engine isolation

<table>
<thead>
<tr>
<th>Region content</th>
<th>Privileged permission</th>
<th>Unprivileged permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure Engine code &amp; constants</td>
<td>Read Only (execution allowed)</td>
<td>No access</td>
</tr>
<tr>
<td>Secure Engine stack &amp; VDATA</td>
<td>Read Write (not executable)</td>
<td>No access</td>
</tr>
</tbody>
</table>

To run a Secure Engine service, the caller must first enter the privileged level of software execution through a controlled access point. This is done using the concept of SE interface (refer to Section A.1.2: SE interface, keeping in mind that MPU protection replaces the Firewall protection). It abstracts the request to get the privileged level of software execution: this request consists in triggering a supervisor (SVC) call.

In the SBSFU example delivered in the X-CUBE-SBSFU Expansion Package, the SBSFU application implements an SVC handler to catch this SVC call and process it with another SE interface service to enter the Secure Engine via its call gate as shown in Figure 35.

**Note:** The SVC handler must be trusted because it is a key element of the Secure Engine access control.
The SE call gate mechanism uses the concepts described in Section A.1.1: SE core call gate mechanism to provide a unique entry point to the Secure Engine services. The difference with Section A.1.1 is that the MPU protection replaces the Firewall protection; the constraints for the placement of the call gate code are only the MPU region constraints (the call gate must be located in the privileged code region).

When the Secure Engine processing ends, the call gate concept provides a single exit point and the SVC call return sequence applies. This return sequence brings the software back to the unprivileged level of execution. Then, any further direct access to the Secure Engine code and data generates an MPU fault.

The Secure Engine service exit is described in Figure 36.
A.2.2 Constraints

The 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 SVC handler is the controlled access point to privileged level of execution (this must be trusted code). Additionally, any piece of code running in privileged mode must be trusted also (interrupt routines, initialization code, and others) so that the controlled access point is not bypassed. It is key to partition the software very carefully and avoid granting privileged level of execution when not required (the software must run in unprivileged mode as much as possible).

The MPU controls the Cortex®-M access to the memory. Any peripheral acting as a master on the bus might access the Secure Engine code and data without triggering an MPU fault (for instance a DMA peripheral). It is therefore required to make sure that only trusted code can program these peripherals. For instance, in the X-CUBE-SBSFU example for 32F413HDISCOVERY, an MPU region covers the DMA registers to make sure it is not possible to program these peripherals in unprivileged mode of execution: only the privileged code can configure the DMAs.

Last but not least, for the STM32F4 Series and the STM32F7 Series, the Secure Engine protection is ensured only as long as the required MPU settings are maintained. If User Application code is not fully trusted or if a bug can be exploited by a hacker, the MPU configuration must be maintained during User Application execution.

This latter constraint does not exist for the STM32 Series with secure user memory. Before launching the user application, the MPU protection is disabled and the secure user memory protection is activated. When secured, any access to securable memory area (fetch, read, programming, erase) is rejected, generating a bus error. All the code and secrets located inside secure user memory (protected environment) is fully hidden. Secure user memory closure is done through a service provided by the bootloader code (BL_EXIT_STICKY) for the STM32G0 Series and STM32G4 Series or by RSS (exitSecureArea()) for the STM32H7 Series.
Appendix B  Dual-image handling

Some SBSFU application examples handle two firmware images stored in internal Flash.

B.1 Elements and roles

- **Slot #0:**
  - This slot contains the active firmware (firmware header + firmware). This is the user application that is launched at boot time by SBSFU (after verifying its validity).

- **Slot #1:**
  - This slot is used to store the downloaded firmware (firmware header + encrypted firmware) to be installed at next reboot.
  - In case of partial image, the size of this slot can be lower than the size of Slot #0 (which contains full image). Slot #1 can be sized according to the maximum possible partial image.

- **Swap region:**
  - This is a Flash area used to swap the content of Slot #0 and Slot #1.
  - Nevertheless, this area is not a buffer used for each and every swap of Flash sector. It is used to move a first sector, hence creating a shift in Flash allowing swapping the two slots sector by sector.

*Figure 37* represents the mapping on NUCLEO-L476RG. The mapping order for slots and swap elements depends on the STM32 Series:

- For The STM32 Series with secure user memory, the Slot #0 header must be mapped just after SBSFU code in order to be protected by the secure user memory.

- For the STM32L4 Series, the firewall code and data (Slot #0 header) 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).
Figure 37. Internal user Flash mapping (example of the NUCLEO-L476RG with 512-byte headers)

- intvect_start
-SE_Code_region_ROM_start
-SE_Code_region_ROM_end / SE_IF_region_ROM_start
-SE_IF_region_ROM_end / SB_region_ROM_start
-SB_region_ROM_end / region_SLOT_1_start
-region_SLOT_1_start + SFU_IMG_IMAGE_OFFSET

Slot #1

- Download image header (512)
- Download image
- Active image header (512)
- Active image

Slot #0

- Swap area
- Free for user data

-region_SLOT_1_end / region_SLOT_0_start
-region_SLOT_0_start + SFU_IMG_IMAGE_OFFSET
-region_SLOT_0_end / region_SWAP_start
-region_SWAP_end
B.2 Mapping definition

The mapping definition is located in the `Linker_Common` folder for each example. *Figure 38* shows how to find information such as slot size and SBSFU code size in the NUCLEO-L476RG example.

To start the application, SBSFU initializes the SP register with the user application stack pointer value, then jumps to the user application reset vector (refer to *Figure 38*).

*Figure 38. User application vector table (example of the STM32L4 Series)*

Source code: `sfu_fwiimg_services.c` & `system_stm32l4xx.c`

1. End of SBSFU execution : `SFU_IMG_ExecuteUserFW(void)`  
   /* Initialize user application Stack Pointer */  
   __set_MSP(__IO uint32_t *)(SFU_IMG_SLOT_0_REGION_BEGIN + SFU_IMG_IMAGE_OFFSET);  
   /* Jump to user application */  
   jump_address = ((__IO uint32_t*)((uint32_t)SFU_IMG_SLOT_0_REGION_BEGIN + SFU_IMG_IMAGE_OFFSET + 4));

2. Start of USER application: `SystemInit(void)`  
   SCB->VTOR = (uint32_t)__ICFEDIT_intvec_start__;  
   /* __ICFEDIT_region_SLOT_0_start__ + 512 */
Appendix C  Single-image handling

Some SBSFU application examples handle one single firmware image stored in internal Flash.

This mode of operation allows maximizing the user firmware size by:

- Reducing the SBSFU footprint in Flash
- Allocating more Flash space for the user application

These benefits come at the cost of some features:

- Safe firmware image programming cannot be ensured: as soon as an installation is interrupted (power off), the firmware update process must be restarted from the beginning (including the download phase).
- The user application cannot download a new firmware image: the local download procedure is the only way to update the active user code.

C.1 Elements and roles

Slot #0:

- This slot contains the active firmware (firmware header + firmware). This is the user application that is launched at boot time by SBSFU (after verifying its validity).
- This slot is directly updated when a new firmware image is downloaded and installed (after firmware header verification)

C.2 Mapping definition

The mapping definition is located in the Linker_Common folder for each example. Figure 38 shows how to find information such as slot size and SBSFU code size in the NUCLEO-L432KC example.

To start the application, SBSFU initializes the SP register with the user application stack pointer value, then jumps to the user application reset vector (refer to Figure 38: User application vector table (example of the STM32L4 Series)).
Appendix D  Cryptographic schemes handling

Four cryptographic schemes are provided as example to illustrate the cryptographic operations. The default cryptographic scheme uses both symmetric (AES-CBC) and asymmetric (ECDSA) cryptography. So, it handles a private key (AES128 private key) as well as a public key (ECC key).

Two alternate schemes are provided. They are selected by means of a SECoreBin compiler switch (named "SECBOOT_CRYPTO_SCHEME").

The X509 certificate-based asymmetric scheme is configured in STSAFE-A variant.

D.1 Cryptographic schemes contained in this package

Table 8 shows the cryptographic scheme selected with the SECBOOT_CRYPTO_SCHEME compiler switch.

<table>
<thead>
<tr>
<th>SECBOOT_CRYPTO_SCHEME value</th>
<th>Authentication</th>
<th>Confidentiality</th>
<th>Integrity</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECBOOT_ECCDSA_WITH_AES128_CBC_SHA256 (default)</td>
<td>ECDSA</td>
<td>AES128-CBC</td>
<td>SHA256</td>
</tr>
<tr>
<td>SECBOOT_ECCDSA_WITH_AES128_CTR_SHA256(1)</td>
<td>ECDSA</td>
<td>AES128-CTR</td>
<td>SHA256</td>
</tr>
<tr>
<td>SECBOOT_ECCDSA_WITHOUT_ENCRYPT_SHA256</td>
<td>ECDSA</td>
<td>None(2)</td>
<td>SHA256</td>
</tr>
<tr>
<td>SECBOOT_AES128_GCM_AES128_GCM_AES128_GCM(3)</td>
<td>AES-GCM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SECBOOT_X509_ECCDSA_WITHOUT_ENCRYPT_SHA256</td>
<td>ECDSA</td>
<td>None(2)</td>
<td>SHA256</td>
</tr>
</tbody>
</table>

1. This cryptographic scheme is selected for products with external Flash and OTFDEC support.
2. The SBSFU project must also be configured to deal with a clear firmware image by setting the SFU_IMAGE PROGRAMMING_TYPE compiler switch to the value SFU_CLEAR_IMAGE.
3. For the symmetric cryptographic scheme, it is highly recommended to configure a unique symmetric key for each product.
D.2 Asymmetric verification and symmetric encryption schemes

These schemes (SECBOOT_ECCDSA_WITH_AES128_CBC_SHA256, SECBOOT_ECCDSA_WITH_AES128_CTR_SHA256, SECBOOT_ECCDSA_WITHOUT_ENCRYPT_SHA256) are implemented for firmware decryption and verification as illustrated in *Figure 39*.

*Figure 39. Asymmetric verification and symmetric encryption*
D.3 Symmetric verification and encryption scheme

This scheme (SECBOOT_AES128_GCM_AES128_GCM_AES128_GCM) is implemented for firmware decryption and verification as illustrated in Figure 40.

Figure 40. Symmetric verification and encryption
D.4 X509 certificate-based asymmetric scheme without firmware encryption

This scheme (SECBOOT_X509_ECDSA_WITHOUT_ENCRYPT_SHA256) is implemented for firmware verification as illustrated in Figure 41.

Figure 41. X509 asymmetric verification

The X509 certificate-based asymmetric scheme makes use of a chain of X509 certificates to deliver the public key used to verify the firmware header signature.

In the example provided, two certificates (the Root CA and OEM CA (first intermediate CA)) are embedded inside STSAFE-A100, while the second intermediate CA and leaf certificate (firmware signing certificate) are delivered as part of the firmware header.
In order to use the public key contained in the leaf certificate, the certificate chain is first verified by the SBSFU code to ensure that the delivered firmware signing public key is authentic. Once the certificate chain is verified, the firmware signing public key is used to verify the firmware header signature.
D.5 Secure Boot and Secure Firmware Update flow

Figure 43 and Figure 44 indicate how the cryptographic operations (asymmetric cryptographic scheme with FW encryption) are integrated in the SBSFU execution boot flows.

Figure 43. SBSFU dual-image boot flows
Figure 44. SBSFU single-image boot flows

Upgrade flow

1. Header
   - SFU Magic
   - Security protocol version
   - FW Version
   - FW Size
   - Partial FW Offset (1)
   - Partial FW Size (1)
   - FW Tag
   - SHA256 computed on clear full FW
   - FW Tag
   - SHA256 computed on clear partial FW (1)
   - Init Vector
   - Reserved
   - Header TAG
   - SHA256 signed with ECDSA

2. Firmware image
   - Encrypted firmware
   - With AES CBC

3. Upgrade flow
   - FW image download (header part)
   - Verify image header TAG
   - FW image download (encrypted FW part)
   - Decrypt and Flash FW on the fly
   - Reboot
   - Check/Apply security protections
   - FW update available
   - Verify image header TAG
   - Verify full FW Tag
   - Execute upgraded FW

Normal boot flow

1. Check/Apply security protections
2. Verify image header TAG
3. Verify full FW Tag
4. Check FW status valid/not valid
5. Execute FW

Security check and decrypt done in SBSFU context after reboot

(1) Not used in single image boot flow
Appendix E  Firmware image preparation tool

The X-CUBE-SBSFU STM32Cube Expansion Package is delivered with the `prepareimage` firmware image preparation tool allowing:

- Taking into account the selected cryptographic scheme and keys
- Encrypting the firmware image when required
- Generating partial firmware image, by extracting binary differences between two full images
- Generating the firmware header with all the data required for the authentication and integrity checks

The `prepareimage` tool is delivered in two formats:

- Windows® executable: the standard Windows® command interpreter is required
- Python™ scripts: a Python™ interpreter as well as the elements listed in Middlewares\ST\STM32_Secure_Engine\Utilities\KeysAndImages\readme.txt are required

The Windows® executable enables a quick and easy use of the package with all three predefined cryptographic schemes. The Python™ scripts, delivered as source code, offer the possibility to define additional cryptographic schemes in a flexible manner.

Note: Refer to Appendix F and Appendix G for KMS and STSAFE-A specificities.

E.1  Tool location

The Python™ scripts as well as the Windows® executable are located in the Secure Engine component, in folder Middlewares\ST\STM32_Secure_Engine\Utilities\KeysAndImages.

E.2  Inputs

The package is delivered with some default keys and cryptography settings in folder Applications\2_Images\2_Images SECoreBin\Binary.

Each of the following files can be used as such, or modified to take the user settings into account:

- `ECCKEY.txt`: private ECC key in PEM format. It is used to sign the firmware header. This key is not embedded in the `SECoreBin`, only the corresponding public key is generated by the tools in file `se_key.s`
- `nonce.bin`: this is either a nonce (when AES-GCM is used) or an IV (when AES-CBC is used). This value is added automatically by the tools to the firmware header.
- `OEM_KEY_COMPANY1_key_AES_CBC.bin`: symmetric AES-CBC key. This key is used for the AES-CBC encryption and decryption operations, and is embedded in file `se_key.s`. This file is exclusive with `OEM_KEY_COMPANY1_key_AES_GCM.bin`
- `OEM_KEY_COMPANY1_key_AES_GCM.bin`: symmetric AES-GCM key. This key is used for all AES-GCM operations and is embedded in file `se_key.s`. This file is exclusive with `OEM_KEY_COMPANY1_key_AES_CBC.bin`
The tool uses the appropriate set of files based on the cryptographic scheme selected by means of SECBOOT_CRYPTO_SCHEME in file Applications\2_Images\2_Images SECoreBin\Inc\se_crypto_config.h.

E.3 Outputs

The tool generates:

- The se_key.s file compiled in the SECoreBin project: this file contains the keys (private symmetric key and public ECC key when applicable) embedded in the device and the code to access them. When running the tool from the IDE, this file is located in Applications\2_Images\2_Images SECoreBin\EWARM\.

- A .sfb file packing the user firmware header and the encrypted user firmware image (when the selected cryptographic scheme enables user firmware encryption). When running the tool from the IDE, this file is generated in Applications\2_Images\2_Images UserApp\Binary\.

- A .bin file concatenating the SBSFU binary, UserApp binary, and active FW image header. Flashing this file into the device with a flasher tool makes the UserApp installation process simple, since the FW header and FW image are already correctly installed. It is not needed to use the SBSFU application for installing the UserApp. For STM32 devices with OTFDEC support and external Flash, two separate binary files are generated:
  - A first binary concatenating SBSFU binary and active FW image header (SBSFU_UserApp_Header.bin) to be programmed into the internal Flash memory.
  - A second binary (UserApp.sfb) to be Programmed into the external memory at the Slot #0 start address.

Caution:

- Before programming the .bin file into the device, a mass erase must be performed. In order to detect any malicious software, SBSFU verifies at start up that there is no additional code after UserApp in Slot #0.
- When Slot #0 is located in external Flash, Slot #0 must be erased before programming the .bin file.

- Two log files, output.txt, located in Applications\2_Images \2_Images SECoreBin\EWARM\ and in Applications\2_Images\2_Images UserApp\EWARM to trace the executions of prebuild.bat and postbuild.bat.

E.4 IDE integration

The prepareimage tool is integrated with the IDEs as Windows® batch files for:

- Pre-build actions for the SECoreBin application: at this stage, the cryptographic keys are managed
- Post-build actions for the UserApp application: at this stage, the firmware image is built
When compiling with the IDE, the keys and firmware image are handled. No extra action is required from the user. At the end of the compilation steps:

- The required keys are embedded in the `SECoreBin` binary
- The firmware image to be installed is generated in the proper format, with the appropriate firmware header, as a `.sfb` file. This `.sfb` file can be transferred over the Ymodem protocol for installation by SBSFU.
- The `.bin` file that can be flashed for the test of UserApp (`SBFU_UserApp.bin`).

The batch files integrating the tool in the IDE are located in folder `Applications\2_Images\2_Images SECoreBin\EWARM`:

- `prebuild.bat`: invoking the tool to perform the pre-build actions when compiling the `SECoreBin` project
- `postbuild.bat`: invoking the tool to perform the post-build actions when compiling the `UserApp` project

These batch files allow seamless switching from the Windows® executable variant to the Python™ script variant of the `prepareimage` tool. The procedure is described in the files themselves.

### E.5 Partial Image

A partial image contains only the binary portion of the new firmware image to install, versus the active firmware image.

Partial-image usage presents various benefits:

- Smaller firmware image to download (reduced download duration)
- Faster firmware image installation
- Memory mapping optimization, with possibly smaller Slot #1 (maximum size of partial image) and bigger Slot #0 (maximum size of full image)

This feature is available on dual-image variant only (not available on single-image variant).

For the partial-image feature, the header structure includes two instances of the FW tag:

- Partial FW tag: tag of the partial image only. This tag is checked by SBSFU during the image installation phase.
- FW tag: tag of the full image (after the installation of the partial image). This tag is checked by SBSFU during the image boot phase.

The `prepareimage` tool can be used to generate partial firmware image, starting from the active image and the new firmware image to install:

1. First perform a binary comparison between the two full images in clear
2. Then apply the usual image preparation procedure

Refer to the detailed procedure `Example for partial update` described in the `readme.txt` file of the `prepareimage` tool (`Middlewares\ST\STM32_Secure_Engine\Utilities\KeysAndImages\readme.txt`).
F.1 Key update process description

PKCS #11 APIs manage keys through objects containing different type of information:

- Object header: giving information about the object itself, such as attribute size, number of attributes and object ID
- Object attributes: such as type, size and value

Static embedded keys are embedded in the code and cannot be modified. On the contrary, updatable keys can be modified in a NVM storage located inside the protected/isolated environment:

- An updatable key with dynamic ID can be created via a secure object creation procedure running inside the protected/isolated environment ensuring that the key remains inside the protected/isolated environment (key value is stored in the NVM storage and only the object ID is returned to the application).
- An updatable key with static ID can be updated in the NVM storage via a secure update procedure using static embedded root keys (authenticity check, data integrity check and data decryption). It means that the key must be provided to KMS in a specific format in order to ensure the key authenticity, the key integrity and the key confidentiality. KMS example is provided with a tool allowing to automatically generate the encrypted object based on ECDSA asymmetric cryptography for data authenticity/integrity verification and based on AES-CBC symmetric cryptography for data confidentiality. Once an encrypted object is downloaded into the device, SBSFU application detects it at the next system reset and SBSFU application processes it via the KMS secure update procedure (ImportBlob() function).

*Figure 45 and Figure 46 illustrate the key creation and update procedure.*
F.2 SBSFU static keys generation

With KMS middleware integration, SBSFU keys are no more stored in a section under PCROP protection but inside the KMS code running in the secure enclave as shown in Figure 47. During SECoreBin compilation stage, prebuild.bat updates SBSFU static embedded keys inside kms_platf_objects_config.h located in Applications\2_Images\2_Images SECoreBin\EWARM| (instead of the se_key.s file update done in the standard variant).
F.3 UserApp menu

A specific menu is added providing examples using KMS services exported services through a standard PKCS #11 interface: AES-GCM/CBC encryption/decryption, RSA signature/verification, key provisioning, key derivation.
Appendix G SBSFU with STM32 and STSAFE-A100

G.1 Introduction to STSAFE-A100

STSAFE-A100 is a tamper-resistant secure element (HW Common Criteria EAL5+ certified) used to host X509 certificates and keys, and perform verifications that are used for firmware image authentication during Secure Boot and Secure Firmware Update procedures.

STSAFE-A100 is connected to STM32 using the I²C HW interface. Paring keys must be provisioned inside STSAFE-A100 and STM32 to secure the system:

- Host_Mac_Key: a symmetric key used to pair a specific STM32 with a specific STSAFE-A100 in order to prevent product cloning
- Host_Cipher_Key: a symmetric key used to encrypt I²C communication between STM32 and STSAFE-A100 in order to establish a secure communication channel

To combine an STSAFE-A100 with an STM32 for an SBSFU application, cryptographic scheme X509 certificate-based asymmetric scheme without firmware encryption is used (refer to Appendix D Cryptographic schemes handling for more details). This cryptographic scheme is based on a four-certificate chain principle:

- Root CA Cert: root certificate to be provisioned once inside the STSAFE-A100
- OEM CA Cert: first intermediate certificate from the OEM to be provisioned once inside the STSAFE-A100
- OEM Divisional CA Cert: second intermediate certificate from the OEM to be inserted inside the header of each new firmware image
- Firmware Signing Cert: firmware signing certificate from the OEM to be inserted inside the header of each new firmware image

Figure 49. Certificate chain overview
As explained above, the STM32, STSAFE-A100, and firmware image must be provisioned with some keys and or certificates:

- **STM32**: pairing keys must be inserted inside the SBSFU application code (inside the part that is executed inside the protected environment) to be able to communicate securely with an STSAFE-A100 component.

- **STSAFE-A100**:
  - Pairing keys must be provisioned inside the STSAFE-A100 to be able to communicate securely with an STM32 component.
  - Root CA Cert and OEM CA Cert must be provisioned inside the STSAFE-A100 to be able to verify OEM Divisional CA Cert and Firmware Signing Cert that are received as part of the header of the new firmware image to be installed on the STM32.

- **Firmware image**: OEM Divisional CA Cert and Firmware Signing Cert must be inserted in the header of the new firmware image to be installed on the STM32.

**Figure 50. Pairing key and certificate provisioning overview**
G.2 Certificate generation

Generation of the certificates is done using *openssl* via a set of batch files provided in the STSAFE-A100 project example variant as shown in Figure 51:

- **GEN_SBSFU_SAMPLE_ROOT_CA_ECC_NIST_P256.bat**: generates Root CA public and private ECC keys and a self signed root certificate.
- **GEN_SBSFU_SAMPLE_INTER1_CA_ECC_NIST_P256.bat**: generates First Intermediate CA (OEM CA) ECC key pair and certificate signed by the RootCA.
- **GEN_SBSFU_SAMPLE_INTER2_CA_ECC_NIST_P256.bat**: generates Second Intermediate CA (OEM Divisional CA) ECC key pair and certificate signed by the OEM CA.
- **GEN_SBSFU_SAMPLE_FW_SIGNING_ECC_NIST_P256.bat**: generates the Firmware Signing ECC key pair and certificate signed by the OEM Divisional CA.

Figure 51. Batch files using *openssl*

G.3 STSAFE-A100 provisioning

To provision STSAFE-A100 with pairing keys and certificates that are used in the context of the SBSFU application example, a provisioning tools application project is provided as example in the X-CUBE-SBSFU package (*2_Images_STSAFE\STSAFE_Provisioning*).
STSAFE_PAIRING_keys.bin must be updated in order to provision the same keys into the STM32.

Refer to file readme in this project to get details on the procedure and the steps executed by this project.

G.4 STM32 and firmware image provisioning

To provision the STM32 with the pairing keys and insert the certificates in the firmware image headers, the prepare image tool concept of the X-CUBE-SBSFU Expansion Package is used (refer to Appendix E Firmware image preparation tool for more details):

- IDE pre-build script is used to insert pairing keys inside the SBSFU code
- IDE post-build script is used to insert certificated inside the firmware image header

G.5 STSAFE-A100 ordering

To use the STSAFE-A100 in the context of the SBSFU application example, the STSAFE-A100 must be personalized at STMicroelectronics manufacturing stage with some specific data (such as private keys for instance) corresponding to a profile called "SPL2".

The sales reference for ordering the STSAFE-A100 device is STSAFA100S8SPL02.
Appendix H  STM32WB Series specificities

H.1  Compilation process

A specific project (Loader) implementing the firmware download feature (over the Ymodem protocol) is provided in order to make easier the integration of over-the-air download capability (Bluetooth Low Energy protocol for STM32WB Series).

Figure 53 outlines the additional step 0 in order to compile then integrate the Loader into SBSFU during the compilation process.

Figure 53. Compile with Loader integration

H.2  Key provisioning

Before the first execution of the SBSFU example, the symmetric key used for AES cryptographic functions must be provisioned into the Cortex® M0+. Follow the instruction list from the SECoreBin readme.txt file.
Appendix I  STM32H7 Series specificities

I.1 JTAG connection for STM32H753 devices

*Figure 54* shows that, when a secure memory is configured, the JTAG pins are disabled during SBSFU execution. Connecting the STM32CubeProgrammer tool (STM32CubeProg) is not possible. Connection becomes possible once UserApp is started (the hot plug mode must be selected in the ST-LINK configuration panel).

There is no way to connect the STM32CubeProgrammer tool (STM32CubeProg) if the user application cannot be executed nor started because of any potential problem during the SBSFU application execution. To mitigate this risk, the switch SFU_SECURE_USER_PROTECT_ENABLE is enabled only after the development phase with the activation of the SFU_FINAL_SECURE_LOCK_ENABLE switch.

![Figure 54. JTAG connection capability on STM32H753 devices](image)

I.2 JTAG connection for STM32H7B3 devices

*Figure 55* shows that, when a secure memory is configured, the JTAG pins are disabled during SBSFU execution. Connecting the STM32CubeProgrammer tool (STM32CubeProg) is not possible. Connection becomes possible once UserApp is started (the hot plug mode must be selected in the ST-LINK configuration panel) only if SFU_DAP_PROTECT_ENABLE is deactivated.

To mitigate this risk, the switch SFU_SECURE_USER_PROTECT_ENABLE is enabled only after the development phase with the activation of the SFU_FINAL_SECURE_LOCK_ENABLE switch.
I.3 External Flash on STM32H7B3 devices

The STM32H7B3xxx microcontrollers offer the possibility to store the firmware images in an external memory in case the user application requires more memory size than what it is available in internal Flash memory.

OTFDEC write-only key registers are configured by Secure Engine protected enclave before starting user application as illustrated in Figure 56.

During installation process, Slot #0 firmware remains encrypted to ensure the confidentiality. On-the-fly decryption is performed by OTFDEC peripheral during the execution of the user application from the external Flash memory.
A specific cryptographic scheme (SECBOOT_ECCDSA_WITH_AES128_CTR_SHA256) is provided to illustrate the cryptographic operations required for this configuration: This cryptographic scheme uses AES-CTR encryption and asymmetric (ECDSA) cryptography.

To offer the same level of security as other STM32 microcontrollers, the header of Slot #0 must not be accessible during user application execution. For this reason, the header of Slot #0 is stored into the internal Flash in order to be protected through the secure user memory.

To illustrate this capability, an application called 2 images ExtFlash is provided for the STM32H7B3I-DK board in the X-CUBE-SBSFU Expansion Package. **Figure 57** shows the typical memory mapping for such configuration.

**Figure 57. Memory mapping for STM32H7B3 devices with external Flash**

```c
/* Slot 0 in external memory : 4 Mbyte */
#define exported_symbol __CFEDIT_region_SLOT_0_header__ 0x00020000;
#define exported_symbol __CFEDIT_region SLOT_0_start__ 0x00000000;
#define exported_symbol __CFEDIT_region SLOT_0_end__ 0x00000000;
```

*: To preserve the coherency between all applications (with or without external Flash and OTFDEC), the same memory structure is applied for Slot #0 (header and image are contiguous). For STM32H7B3, the header located in the external Flash is not used.
## Table 9. Document revision history

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-Dec-2017</td>
<td>1</td>
<td>Initial release.</td>
</tr>
<tr>
<td>20-Dec-2017</td>
<td>2</td>
<td>Removed references to the integration guide in Chapter 7: Understanding the last execution status message at boot-up and B.2 Mapping definition. Updated Table 4: Error messages at boot-up and Section 5.2.2: Software tools for programming STM32 microcontrollers.</td>
</tr>
<tr>
<td>20-Apr-2018</td>
<td>3</td>
<td>Document scope extended to asymmetric and symmetric cryptography schemes. Added single-image mode. Extended support of the STM32L4 Series:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Updated all chapters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Updated Appendix A Secure Engine protected environment and Appendix B Dual-image handling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added Appendix C Single-image handling, Appendix D Cryptographic schemes handling, and Appendix E Firmware image preparation tool</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Removed the MSC appendix</td>
</tr>
<tr>
<td>18-Dec-2018</td>
<td>4</td>
<td>Product scope extended to the STM32F4 Series, STM32F7 Series, and STM32G0 Series:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Updated Section 5: Protection measures and security strategy, Chapter 8: Step-by-step execution, and Section B.1: Elements and Roles</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added Section A.2: MPU-based Secure Engine Isolation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Secure library offer extended to mbedTLS:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Updated Section 6.2.3: Cryptographic Library, and Section 6.3: Folder structure</td>
</tr>
</tbody>
</table>
### Table 9. Document revision history (continued)

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>22-Jul-2019</td>
<td>5</td>
<td>Updated the entire document:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Product scope extended to the STM32G4 Series, STM32H7 Series, STM32L0 Series, STM32L1 Series, and STM32WB Series</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Added the integration of the STSAFE-A100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Added secure key management services with PKCS #11 APIs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Added Appendix F KMS, Appendix G SBSFU with STM32 and STSAFE-A100, Appendix H STM32WB Series specificities, Appendix I STM32H7 Series specificities</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Removed Appendix SBSFU application state machine</td>
</tr>
<tr>
<td>4-Feb-2020</td>
<td>6</td>
<td>Added AES-CTR encryption of external Flash for the microcontrollers supporting OTFDEC processing:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Added I.2 JTAG connection for STM32H7B3 devices and I.3 External Flash on STM32H7B3 devices</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Updated E.3 Outputs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Updated Figure 5: SBSFU security IPs vs. STM32 Series (2 of 2), Figure 14: Project folder structure (2 of 2), Figure 39: Asymmetric verification and symmetric encryption and Figure 43: SBSFU dual-image boot flows</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Updated Table 3: Cryptographic scheme comparison and Table 8: Cryptographic scheme list</td>
</tr>
</tbody>
</table>
IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, please refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2020 STMicroelectronics – All rights reserved