life.augmented

STM32 Security MOOC
Security in practice

Purpose

- In this MOOC we have already seen
  - The crypto concepts
  - The STM32 MCU security features
- Now, it is time to understand how to use them in real life.
- We will see how to combine these concepts to achieve the level of security you need

life.augmented

Payoff

- After following this MOOC, you should be able to
  - Have an overview of the concepts involved in MCU security
  - Evaluate possible solutions to fill your security needs
  - Understand what security involves in term of environment and tools
  - Have a rough idea of the cost of security

Agenda

1 Short introduction on MCU security

2 From simple use case to more advanced

3
- Firmware protection : to achieve confidentiality of the firmware

4
- Ensure firmware is not corrupted

5
- Ensure firmware is the one you want to run

6 Secure boot

7 Secure firmware update

life.augmented

What is security on MCU?

Security

Definition

Security is the degree of resistance to, or **protection** from, harm.

It applies to any vulnerable and valuable **asset**, such as a person, dwelling, community, nation, or organization.

Wikipedia

Security

Assets

- Security is about protecting assets
- Information, capability, advantage, feature, financial or technical resource that may be damaged, lost or disrupted
  - Consumers Personal Information: Financial, Health, Location, Passwords, Accounts
  - A Product or Solution: Processes, Services, Intellectual Property, Firmware, Brand
  - Health and Safety: Medical devices, Manufacturing Processes and Equipment, Transport & Vehicles
  - The Work Place: Production Equipment, Environmental and Access Controls
- Digital (software sources), physical (a car or a server) or commercial (brand)
- Damage to an asset may affect the normal operation of the system (Denial of Services) as well as that of individuals and organizations involved with the system

Security

CIA security properties

- Properties to be protected depend on applications & assets
- **C**onfidentiality
  - Information is not made available or disclosed to unauthorized individuals, entities, or processes
- **I**ntegrity
  - Maintain and assure the accuracy and completeness of data over its entire life-cycle. Data cannot be modified in an unauthorized or undetected manner.
- **A**vailability
  - Information available when needed. Computing systems used to store and process information, security controls used to protect it, and communication channels used to access it function correctly.

Security

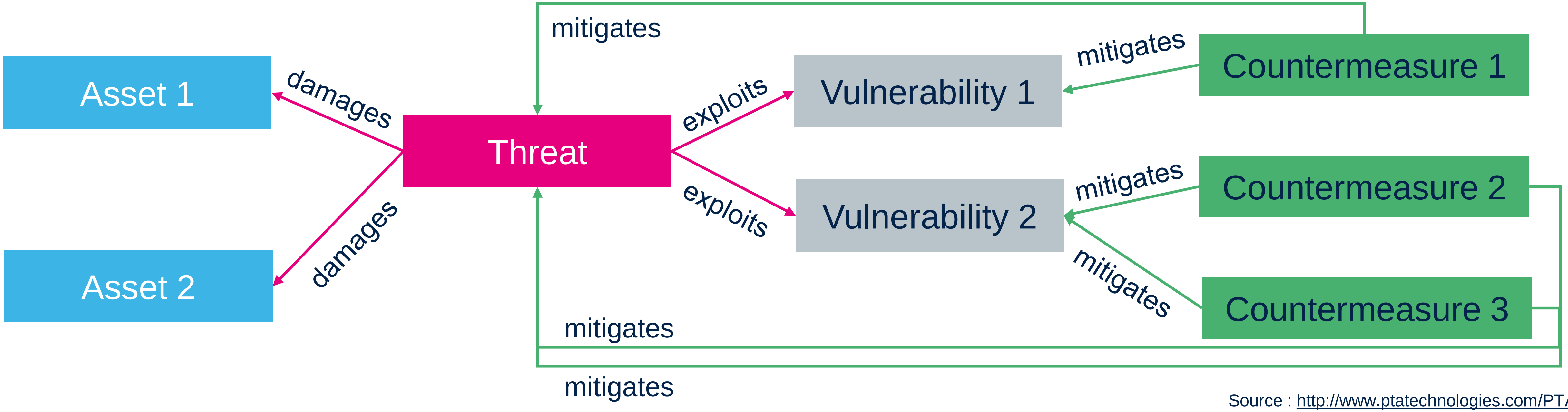
Threats & Vulnerabilities

- **Threats**
  - A threat is a specific scenario or a sequence of actions that exploits a set of vulnerabilities and may cause damage to one or more of the system's Assets
- **Vulnerabilities**
  - Is a weakness, limitation or a defect in one or more of the system's elements that can be exploited to disrupt the normal operation of the system
  - They may be in specific modules of the system, its architecture, its users and operators, and/or in its associated regulations, operational and business procedures

Security

Security Analysis

- Threats exploit Vulnerabilities and damage Assets.
- Countermeasures (also known as security functions) mitigate Vulnerabilities and therefore might mitigate Threats

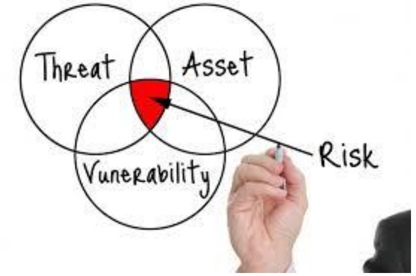


Source : http://www.ptatechnologies.com/PTA5.htm

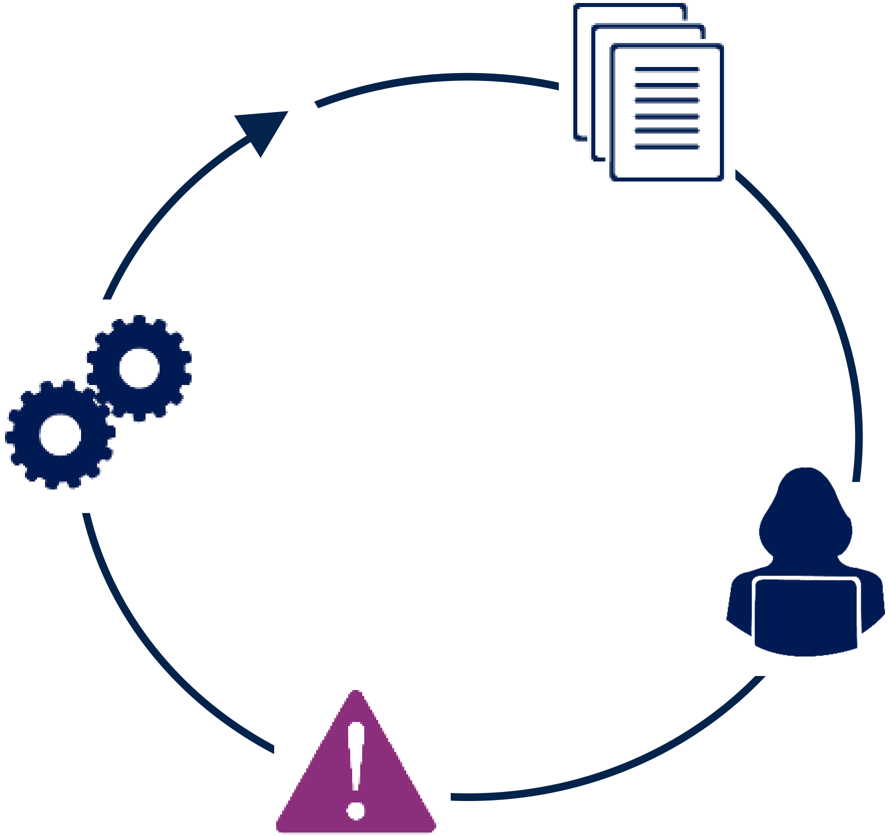
life.augmented

Security

Building a fortified solution is all about risk management



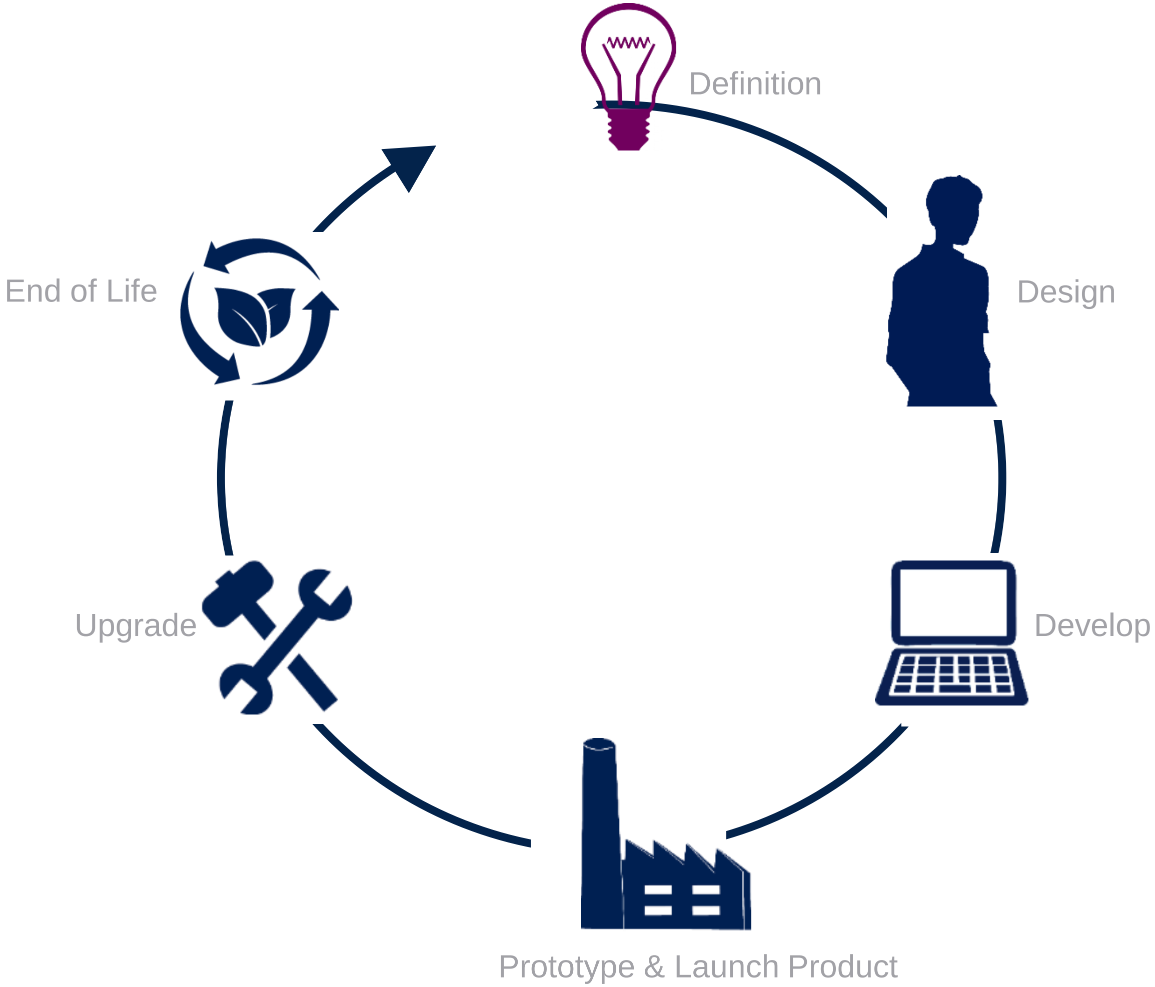
- Understand the value of the Assets you are going to protect, taking into account all stake holders
- Understand your Threats and Vulnerabilities
- Develop a security strategy to reduce Risk, using right level of security for the value of the Assets being protected
- Make use of the integrity and cryptographic tools available



Security

Product security should be factored-in from day one

- Define the product and identify the Assets you plan to protect
- Design the product based on the appropriate level of security
- Develop methods and processes to protect those Assets over the entire life-cycle
- Develop secure processes to handle firmware throughout the products life-cycle



Security Strategies

Security Strategies to address CIA properties relies on

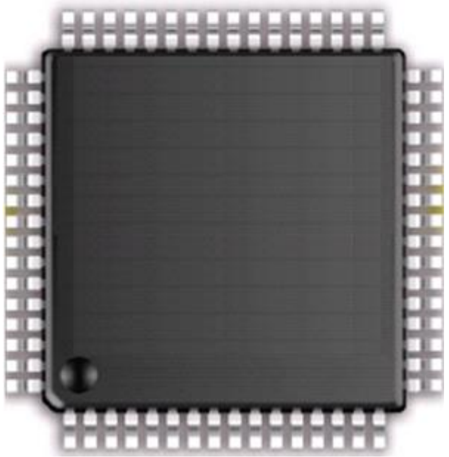
- Authentication of both parties
- Encryption of exchanges
- Integrity of exchanges
- Robustness of stakeholders
  - Connectivity
  - Cloud server
  - End nodes
  - Gateways…

Security

Device requirements

On device side

- Secure boot
- Secure firmware update
- Tamper detection
- Runtime protection
- Key Storage



Cryptography Tools

Cryptography relies on

- Standard algorithms
  - Digital signature (ECC, RSA…)
  - Data encryption (3DES, AES-CBC, AES-CCM…)
  - Data authenticity (HMAC, CMAC…)
  - Data integrity (SHA2, SHA3…)
- Secure Key Management
  - Generation
  - Storage
  - Protection during usage

Protecting the firmware

- We will show here 3 ways to help protecting the firmware
  - Protect the firmware confidentiality
    - I don’t want my firmware to be read by anybody
  - Protect the firmware integrity
    - I want to be able to detect any modification made on the firmware
  - Protect the firmware authenticity
    - I don’t want a firmware from someone else to be installed on my device

Firmware confidentiality

Firmware confidentiality

- One of the basic security concerns is to protect the firmware content from reading to avoid stealing the firmware, coping the product, etc …
- STM32 provides 2 levels of protections for this purpose:
  - First level is RDP (ReaDout Protection). RDP1 is mostly used
  - Second level is PCROP (Proprietary Code ReadOut Protection)

Protection though RDP

- This mechanism is the simplest to setup
- Once firmware is loaded in FLASH, either flasher or embedded firmware can activate this protection
- 2 protection levels
  - Both levels will prevent the reading of internal FLASH from outside the chipset
  - RDP level 1: allows “regression” to clean device
  - RDP level 2: no possible regression to clean device

Protection RDP level 1 pros/cons

- Pros
  - This level can be tested easily
  - The regression from Level 1 to Level 0 involves automatic FLASH erase.
  - This feature is used for basic firmware update through either JTAG/SWD or system bootloader
    - Start from a running device in RDP1
    - Connect to the board and perform a RDP1 to RDP0 regression
    - Download new firmware version
- Cons
  - This update mechanism is not secure because firmware is downloaded in clear. Only way to keep security is to perform update in secure area

Protection RDP level2 pros/cons

- Pros
  - Level of security is higher compared to RDP Level 1.
  - This level prevents regression
  - It is as simple as RDP Level 1 to setup
- Cons
  - No JTAG or system bootloader connection is possible
  - Update is still possible but has to be managed by the firmware itself

RDP1 vs RDP2

- With RDP1, JTAG and bootloader can be connected. This increases the surface of attack. But relatively easy to do firmware maintenance.
- With RDP2, no access to the chip through debug port. So, attacks to retrieve the content of the FLASH require very high expertise and very expensive equipment. Maintenance (update) requires more code to be developed.
- STM32 is not part of secure chip family. This is generic purpose MCU. So, the level of protection cannot be compared to a secure element for instance.

Further protection with PCROP

- Security can be compared with a castle with levels of protections.
- The PCROP brings the second level of protection in case first level has not been forced
- PCROP bring protection against FLASH reading (and writing). Even at RDPL0, code is protected. Only possible action is to execute it.
- PCROP is setup through option bytes and protect a region of the FLASH
- When RDP1 or RDP2 is activated, PCROP protects against reading of code instructions (only instruction fetch is possible when accessing the PCROP protected area).

life.augmented

PCROP implementation

- Using PCROP brings some constraints on the code development:
  - The code should be compiled with specific option removing any data access from the generated assembly code (this is compiler configuration). Code generated is a bit bigger than without this option
  - The code to be protected should be isolated, from the rest of the code, in an area of the FLASH (this is linker configuration)
  - The PCROP region has to be activated after the firmware is flashed
- For more details, application notes are available here:
  - https://www.st.com/en/embedded-software/x-cube-pcrop.html#resource

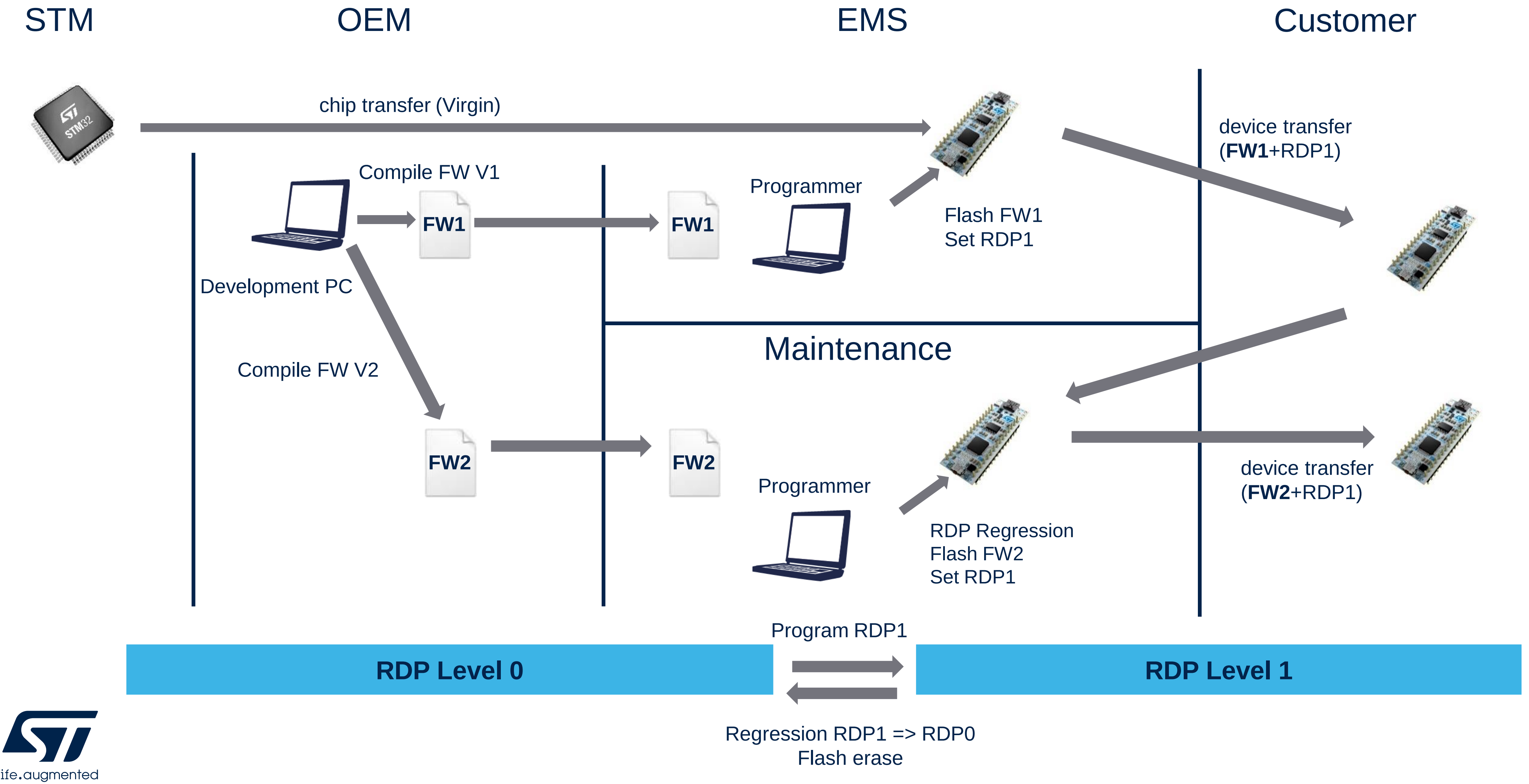
PCROP and updates

- If using RDP1 and PCROP, regression to RDP0 can
  - Erase PCROP zones at the same time
  - Preserve PCROP zones
  - It depends on the PCROP configuration in option bytes
- This allows for instance to preserve the PCROP content and perform only update of the remaining parts
- This mechanism could be used when you have some calibration data to keep. In that case, data have to be converted into functions which is possible thanks to a simple script.
- In combination with RDP2, PCROP areas becomes immutable.

Firmware confidentiality

- We addressed here only basic firmware protection
- This level is enough in many cases and does not require important investment
- One main point not addressed here is the ability to update firmware while keeping it protected at the same time.
- With these mechanisms we can only ensure firmware is protected during execution on the field
- If an update is needed, it has to be done in a secured way (dedicated people for instance)

Firmware confidentiality life cycle



STM

OEM

EMS

Customer

chip transfer (Virgin)

Compile FW V1

FW1

FW1

Development PC

Programmer

Flash FW1
Set RDP1

device transfer
(**FW1**+RDP1)

Maintenance

Compile FW V2

FW2

FW2

Programmer

RDP Regression
Flash FW2
Set RDP1

device transfer
(**FW2**+RDP1)

Program RDP1

RDP Level 0

RDP Level 1

Regression RDP1 => RDP0
Flash erase

Hands-on

- The goal of this hands-on is to show the different steps to
  - Develop the code and create the firmware binary
  - Program the binary
  - Set the RDP level 1
  - Use the working device
  - Update the device using RDP regression
  - Use the new working device

life.augmented

Firmware integrity

Firmware integrity

- Integrity means : ensure the firmware is not corrupted
- Why a firmware could be corrupted ?
  - Software bug involving FLASH write at the wrong place
  - Electrical shock leading to FLASH bit flip
  - Attacker wanting to change the behavior of the firmware
- What are the means to check the integrity of a firmware ?
  - Checksum
  - HASH
- Principle is compute the HASH of the firmware and add this HASH to the firmware binary
- On firmware execution, the same computation will be performed and result will be compared to the one written. If different, then firmware is corrupted !

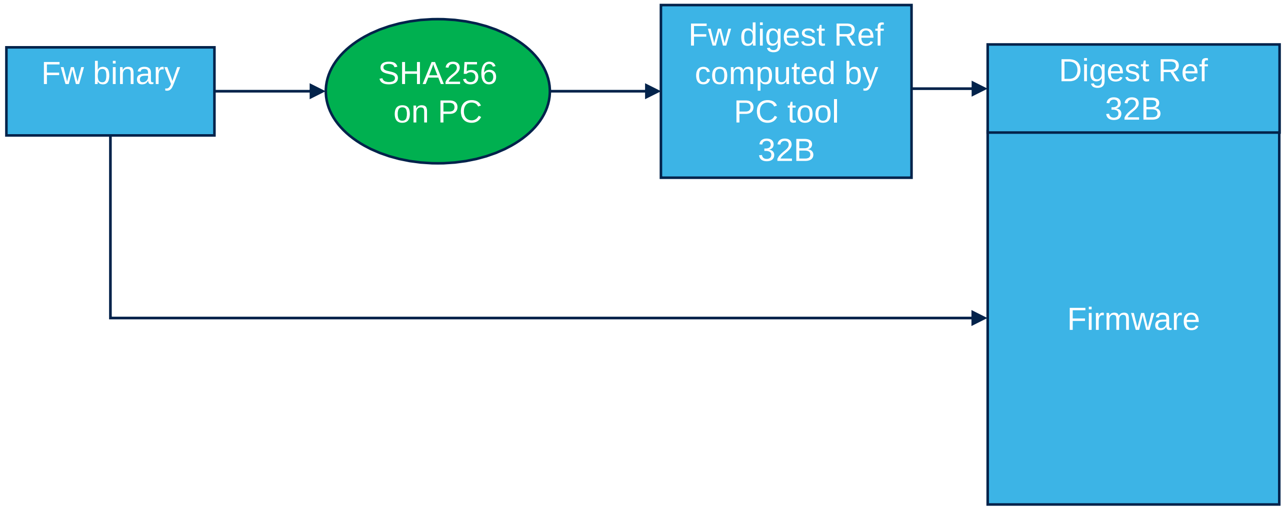
life.augmented

Checking firmware integrity with CRC

- CRC is Cyclic Redundancy Check
- Computation can be done either by software or using hardware dedicated peripheral
- CRC is used usually to secure DATA transmission
- CRC is not secure in a way this is easy to forge data to obtain specific CRC
- On the other hand, CRC computation is faster

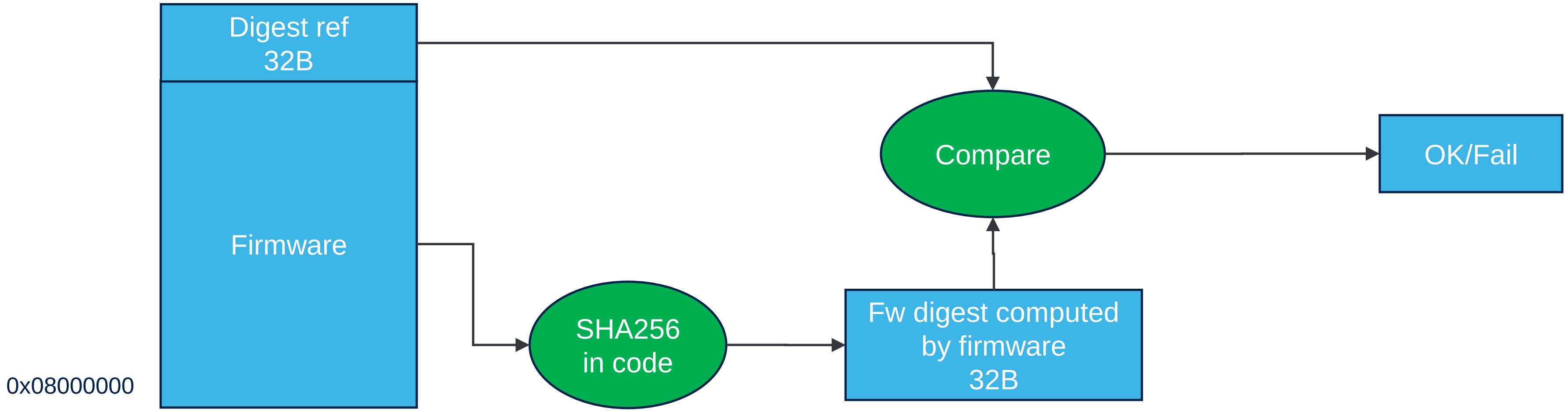
Integrity flow example with SHA256

Production



Final binary to be flashed

Firmware executed on the chip



Protection of the integrity check

- The integrity check relies on the digest value.
- This value should be protected.
- Basic protection could be WRP or adding a MAC of the data
- If firmware content is secret, RDP should also be activated
  - Now the digest computation has a side effect: the whole firmware is read as input to CRC or HASH algorithm.
  - RDP1 does not protect the RAM so it could be possible to read the firmware just by connect under reset at successive timings and read the content of the RAM (assumption is that the whole flash content need to be read to RAM for digest computation, if that’s not the case, then there is no such risk).
  - So, RDP2 or secure memory is mandatory in this case

Firmware integrity hands-on

- The goal of this hands-on is to show the different steps to
  - Develop the code and create the firmware binary
  - Compute the HASH
  - Concatenate firmware and HASH in a single binary
  - Program the binary
  - Run the device and check firmware digest checking works fine
  - Check the impact of modification of 1 bit in firmware
  - Add protections such as RDP level 1, WRP on STM32G0

Firmware authenticity

Firmware authenticity

- Firmware authenticity: ensure the firmware is the one you want
- For some application it is important that embedded firmware cannot be replaced by a non authentic firmware
- The way to make sure a firmware is authentic is to use a signature
- Only the OEM can sign the firmware
- The device can check if signature is correct

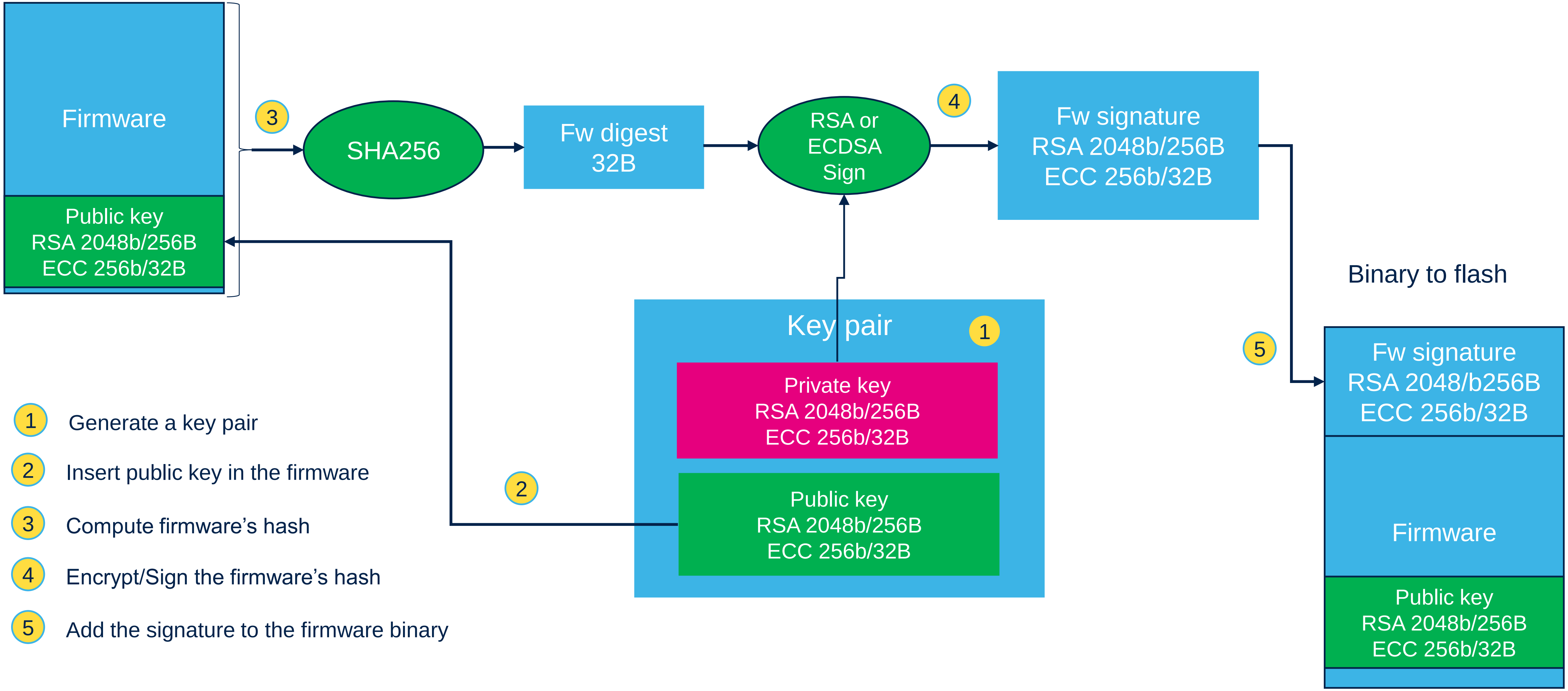
Firmware authentication

- Authentication uses asymmetric cryptography (RSA or ECC)
  - Signature creation is performed with the private key
  - Signature verification is performed with the public key
- Private key remains at OEM’s secure place
- Public key is in the chip
- Public key is not secret but must be write protected (immutable)
- Less robust authentication could use symmetric cryptography (AES GCM for instance, using the generated TAG as a signature)
  - This is less robust because same key is used for signature creation and verification

Firmware authentication : signature

- What should be signed ?
  - As the main purpose is to make sure that firmware is authentic, the signature is done on the firmware digest.
  - This way, if signature is verified, integrity and authenticity of the firmware is checked
- The basic principle of a signature is a “signing” of a hash with a private key
- The signature verification consists in “verifying” this signature with the public key and the computed firmware digest

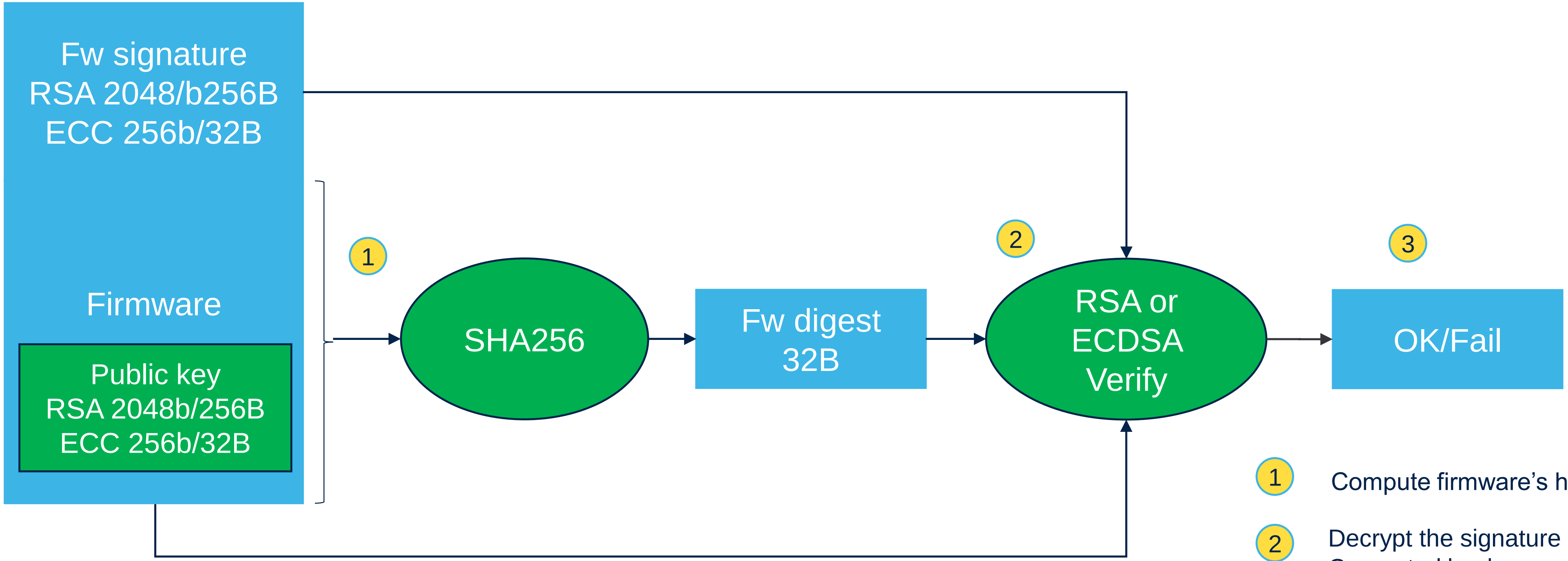
Signature generation flow



1. Generate a key pair
2. Insert public key in the firmware
3. Compute firmware’s hash
4. Encrypt/Sign the firmware’s hash
5. Add the signature to the firmware binary

life.augmented

Signature verification flow



1 Compute firmware’s hash

2 Decrypt the signature and compare with Computed hash

3 Take actions depending on result

life.augmented

Protection of the authentication check

- The protection related to integrity check applies in the same way
- The public key must be protected so it cannot be modified in any way
- The authentication used in this way does not bring much more compared to integrity check : when no firmware update is required, the integrity check is enough as long as firmware cannot be replaced by another one.

Limitation of this setup

- Now we know how to authenticate a code.
- What we have is a firmware that is authenticating itself
- The problem is that part of code doing the authentication (signature check) is not yet authenticated while it is running
- One solution is to use the secure boot that will do this authentication before launching the code.

life.augmented

Firmware authentication hands-on

- The goal of this hands-on is to show the different steps to
  - Develop the code and create the firmware binary
  - Generate a key pair to be used for authentication
  - Insert the public part of the key inside the firmware
  - Compute the HASH of the firmware
  - Sign the HASH using the private key and concatenate this signature with the firmware binary
  - Program the binary
  - Run the device and check firmware signature checking works fine

Sum-up

- We have addressed the 3 main security topics:
  - Confidentiality : Code and data cannot be read
  - Integrity : Any code modification is detected
  - Authenticity : Code running on target was signed by you
- The authenticity automatically involves integrity
- The Confidentiality and Integrity checks are use cases that can be addressed with just one application running
- The Authenticity has to be addressed through secure boot

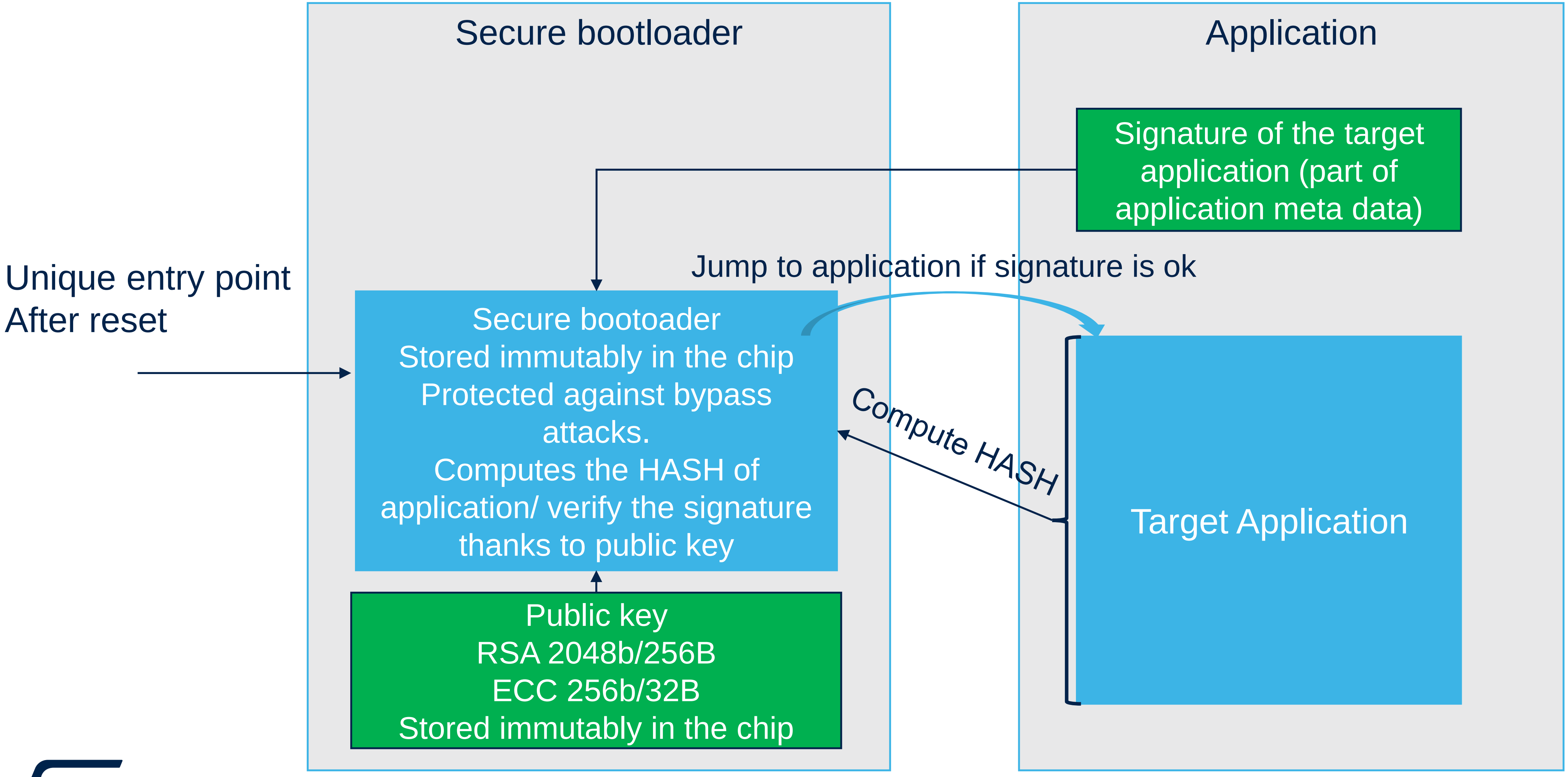
Secure bootloader

life.augmented

What is a secure bootloader ?

- A secure boot is the first code running after reset, it mustn’t be bypassed
- Its main role is to verify the application firmware before launching it
- The secure boot has to be robust to attacks
- Most commonly, secure bootloader relies on a public key, stored in an immutable way, to verify the signature of the target application
- Principle is the same as the authentication example we made except that authenticated code does not include secure boot itself

Secure boot principle



Secure bootloader

Application

Signature of the target application (part of application meta data)

Jump to application if signature is ok

Unique entry point
After reset

Secure bootoader
Stored immutably in the chip
Protected against bypass attacks.
Computes the HASH of application/ verify the signature thanks to public key

Compute HASH

Target Application

Public key
RSA 2048b/256B
ECC 256b/32B
Stored immutably in the chip

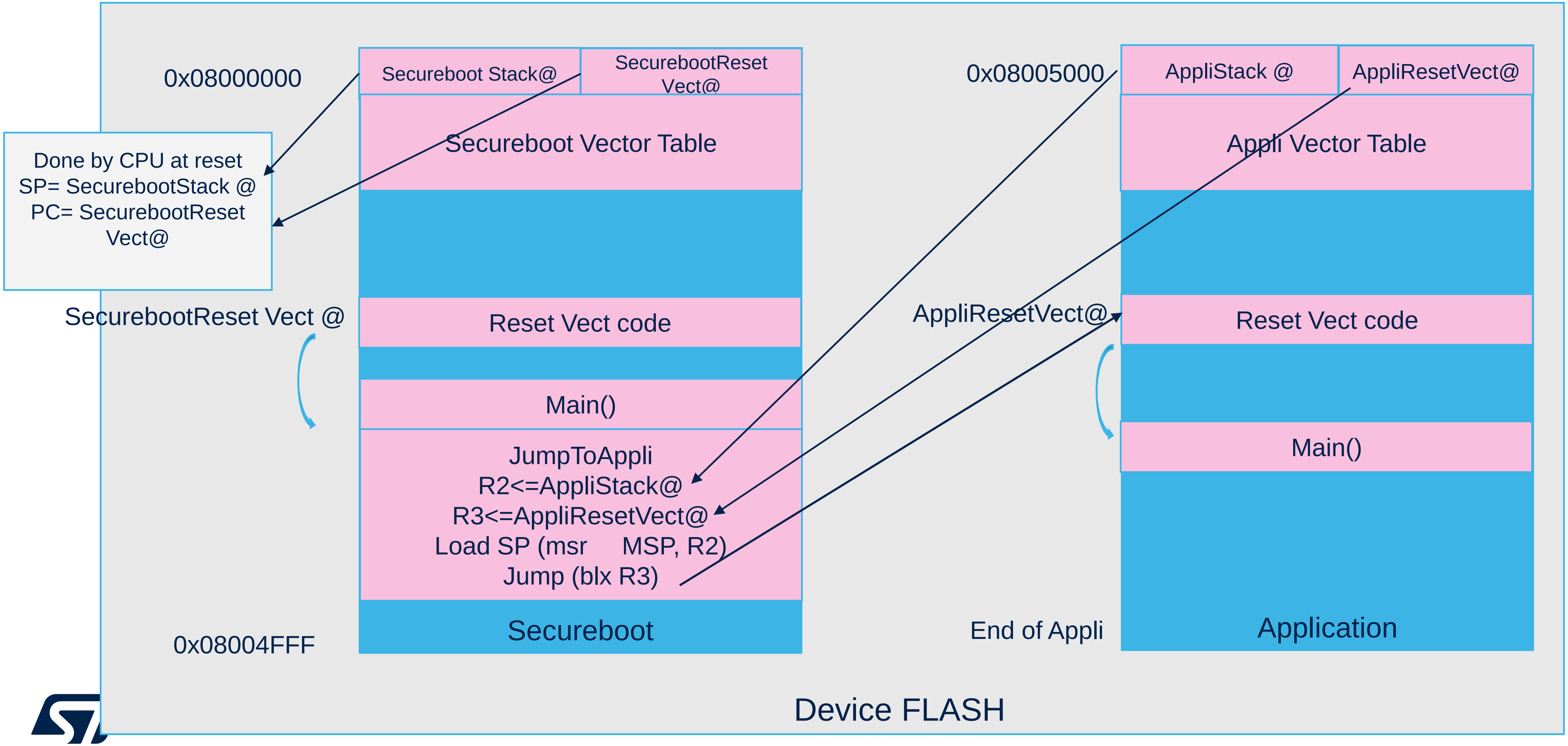
Step 1
create simple secure bootloader

life.augmented

Separate boot from application

- The secure boot is the first application to run
  - It is usually at beginning of flash address
  - It is the code always running after reset
  - This is a “standalone” application: It does not need to be compiled or linked with the target user application
- Need to define a flash mapping
  - FLASH is N KB size
  - The first KB of FLASH are reserved for secure boot code
  - Then the rest is allocated to application, configuration data, etc
- RAM usage
  - The secure boot will have some data in RAM during its execution
  - This RAM has to be emptied before jumping to application to avoid leaving unwanted data to be available when application runs

Secure boot launching application



Simple bootloader hands-on

- The goal of this hands-on is to show the different steps to
  - Create application and a simple bootloader
  - Define a FLASH memory map for both bootloader and application
    - Reserve certain size for bootloader
    - Change the application location in FLASH (linker file and vector table offset)
  - Have the jumper code in bootloader to launch the application
  - Program both bootloader and application to the FLASH and run from bootloader to application

Step 2
trusting the secure bootloader

life.augmented

Trusting the secure boot

- Conditions for secure boot to work properly:
- Ensure unique boot entry: the MCU boot mechanism has to make sure that secure boot is executed
- Secure boot code and public key must not change: this is the immutability condition
- If these 2 conditions are fulfilled you can trust the secure boot

Securing boot entry on STM32

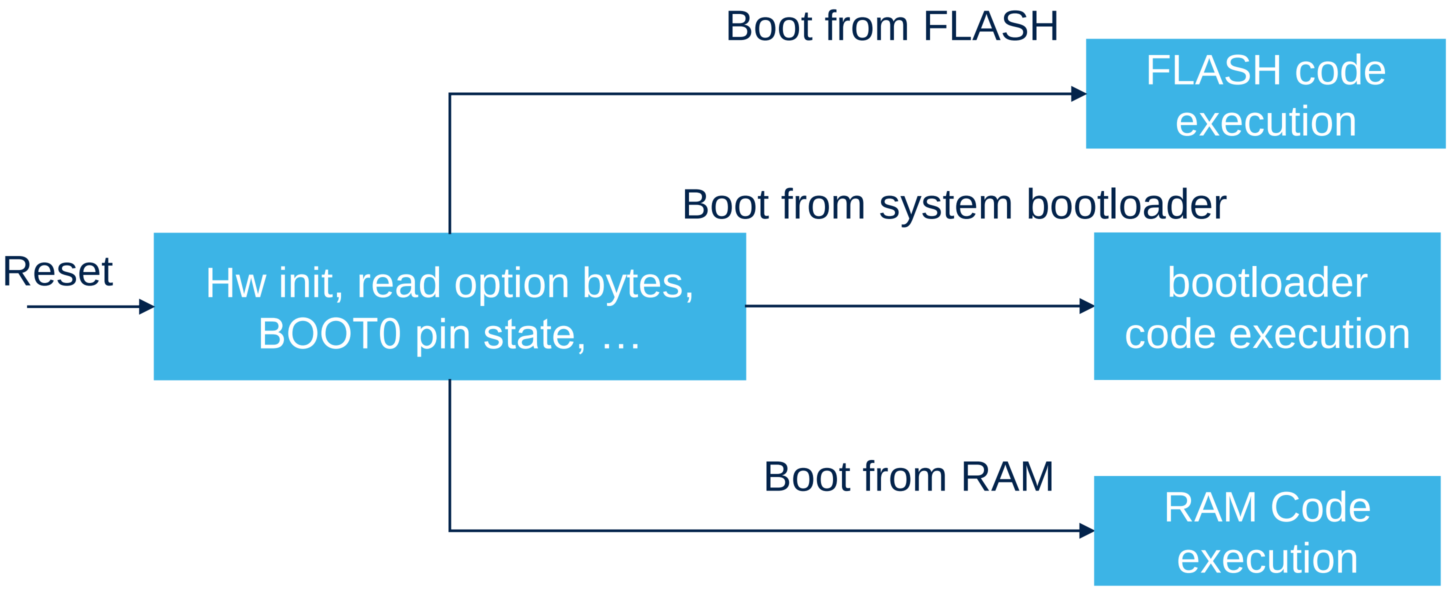
life.augmented

Securing the STM32 boot entry

- The secure bootloader relies on the first code to be executed after boot.
- On STM32, it is mandatory to setup the option bytes so that Boot from FLASH is selected.
- On some STM32, this requires RDP Level 2 to be activated
- On recent STM32, other security mechanisms ensure the booting on FLASH with only RDP Level 1

STM32 boot principle

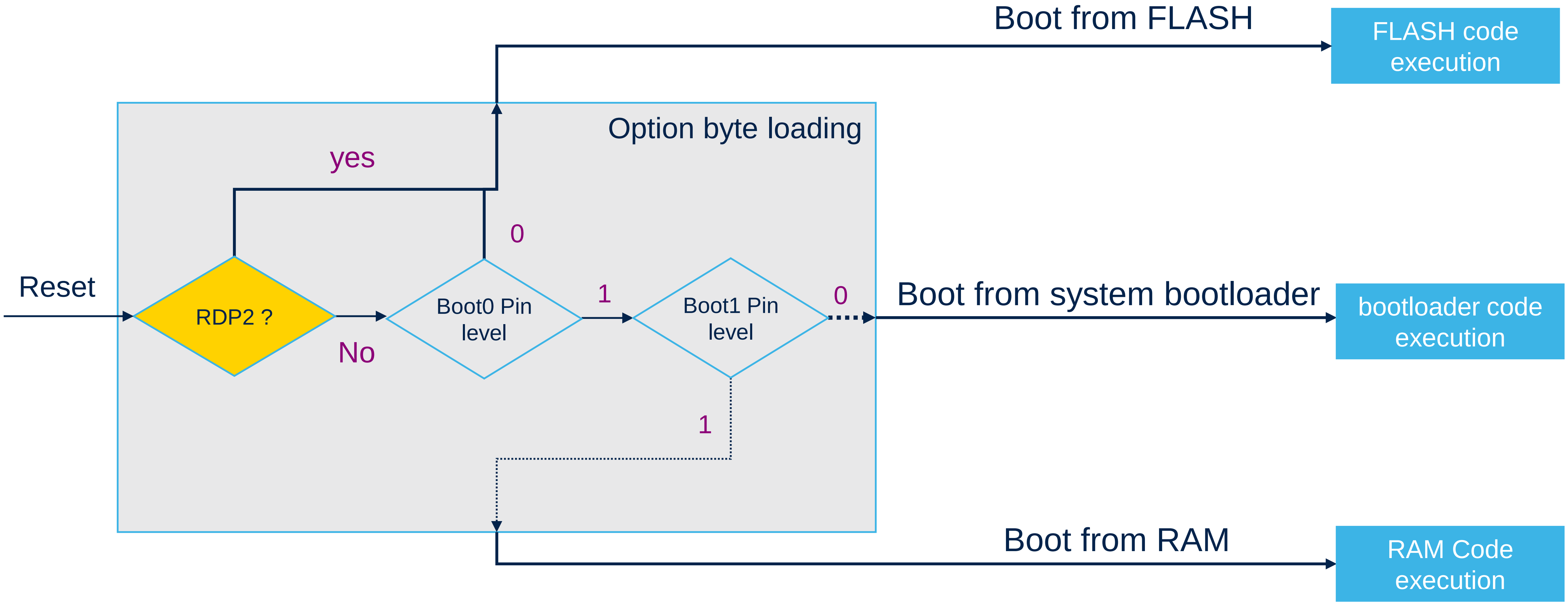
- All STM32 are able to boot at least on 3 different regions
  - User FLASH
  - System bootloader
  - Internal RAM
- Selection can be done
  - With option bytes
  - Possibly with Boot0 pin


Reset
Hw init, read option bytes, BOOT0 pin state, …
Boot from FLASH
FLASH code execution
Boot from system bootloader
bootloader code execution
Boot from RAM
RAM Code execution

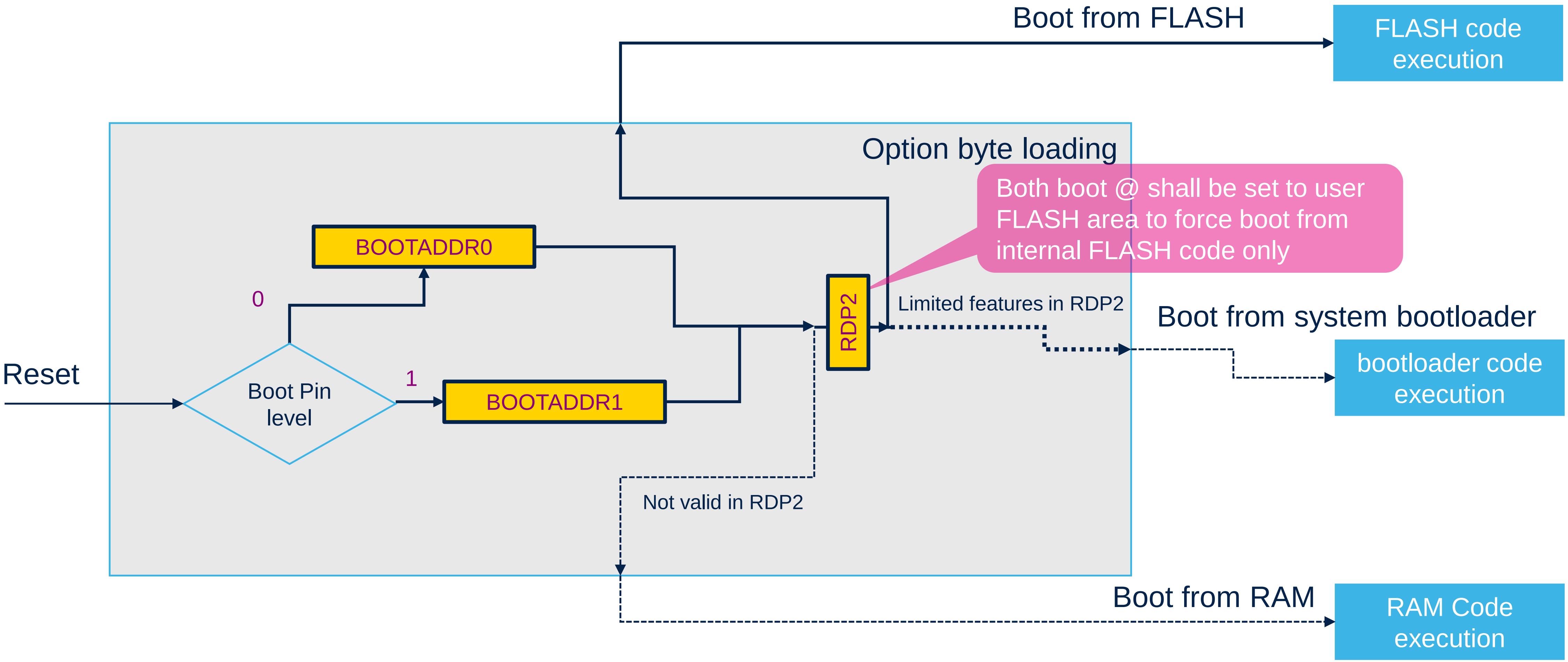
STM32F series Boot

- RDP Level2 ensures that MCU boots on user flash
- STM32F1 is the exception. RDPLevel 1 supported, so secure boot is not possible on this chip
- STM32F7 introduces BOOTADDR0/BOOTADDR1 : These option bytes can be both set to the same user flash address.
  - This improves the entry point robustness using RDP2

STM32F series boot principle



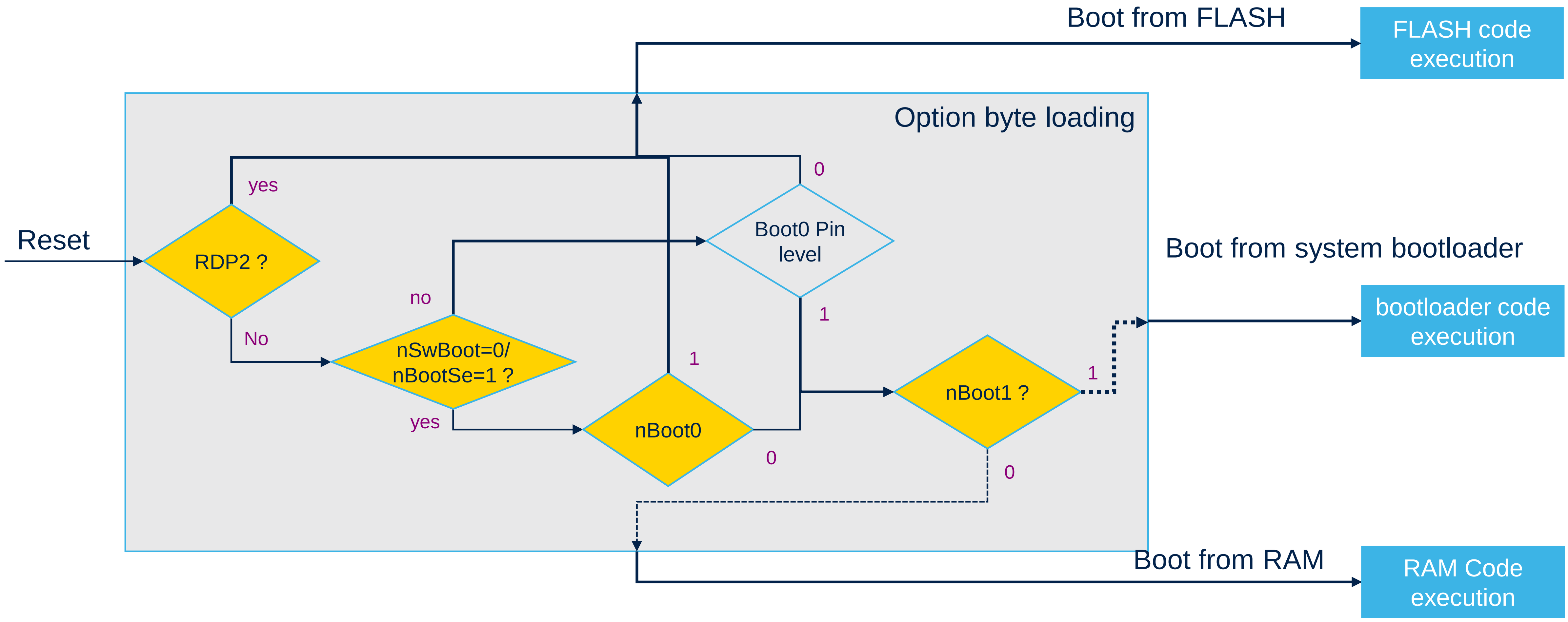
STM32F7 series boot principle



STM32L series boot

- STM32L1 : Same as STM32F Family
- L0 and L4 Series introduce software configuration of boot entry
- STM32L0: nBOOT_SEL=1 forces software configuration for boot.
- STM32L4 : nSWBOOT0=0 forces software configuration for boot.
- Associated with this selection, a nBoot0 option bit is used instead of BOOT0 pin.
- STL32L5 : Will be detailed further

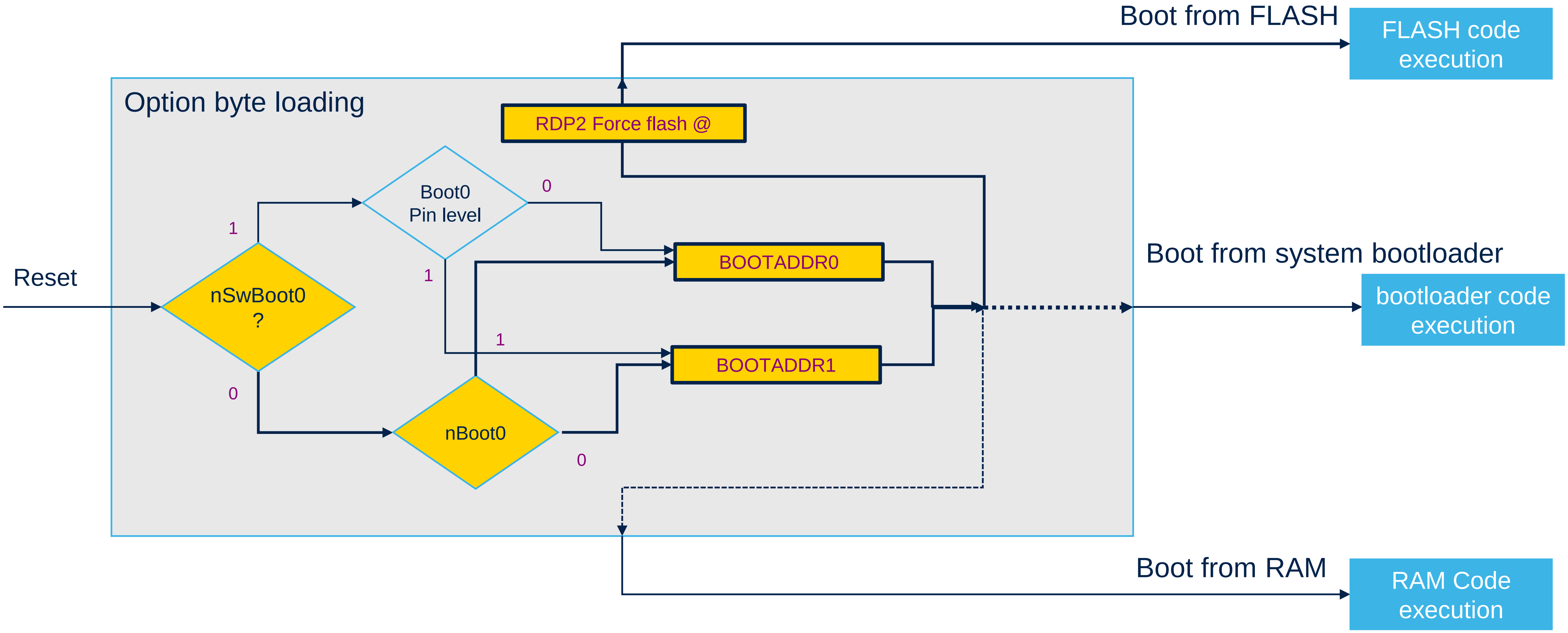
STM32L0/L4 series boot principle



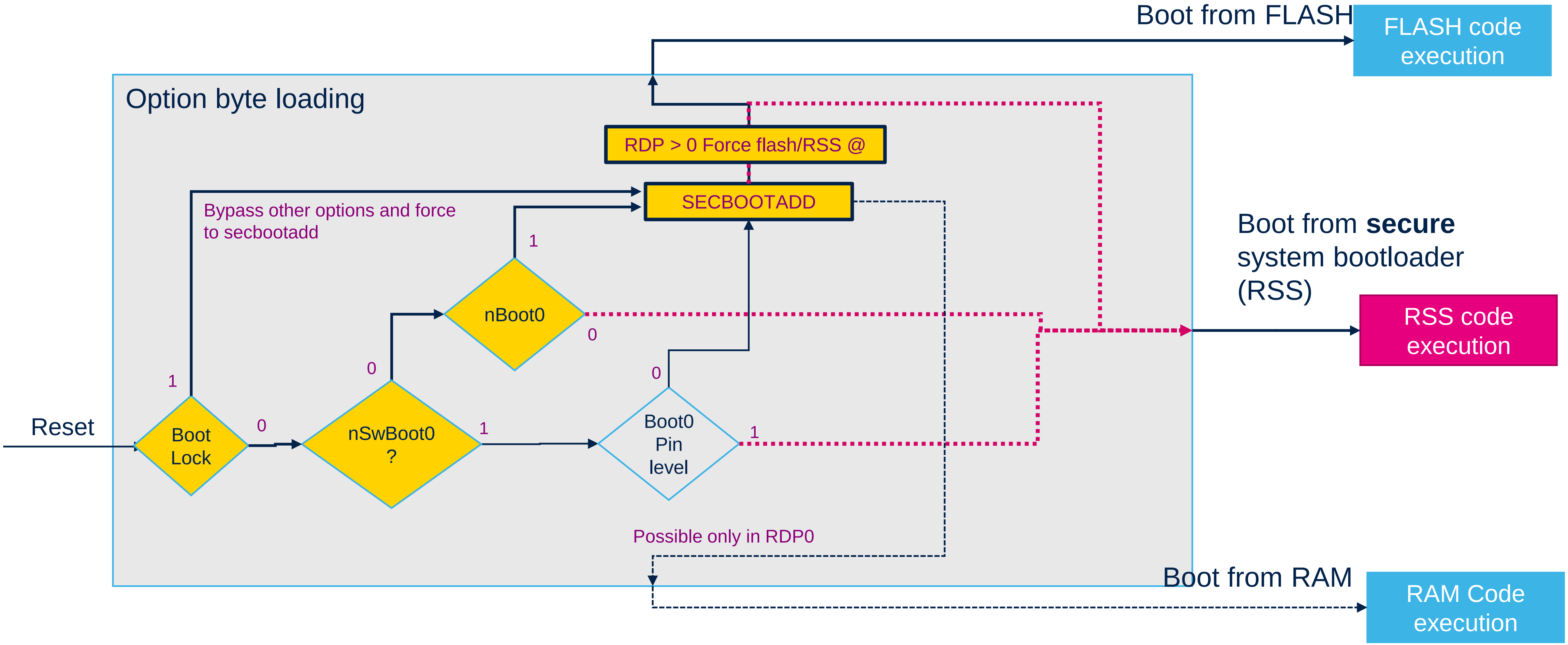
STM32L5

- STM32L5 as described in MOOC part 2 provides more configuration option for securing the boot entry
- The configuration depends on TZEN option bit setting.
- With TZEN=0, the boot options are a mix between STM32L4 and STM32F7: Use of nSWBOOT0 and BOOTADD0/1
- With TZEN=1, the boot options enable only 2 possible targets: SECBOOTADD0 or the secure system bootloader (RSS).
- A Bootlock option byte allows bypassing Boot0

STM32L5 TZEN=0 boot principle



STM32L5 TZEN=1 boot principle



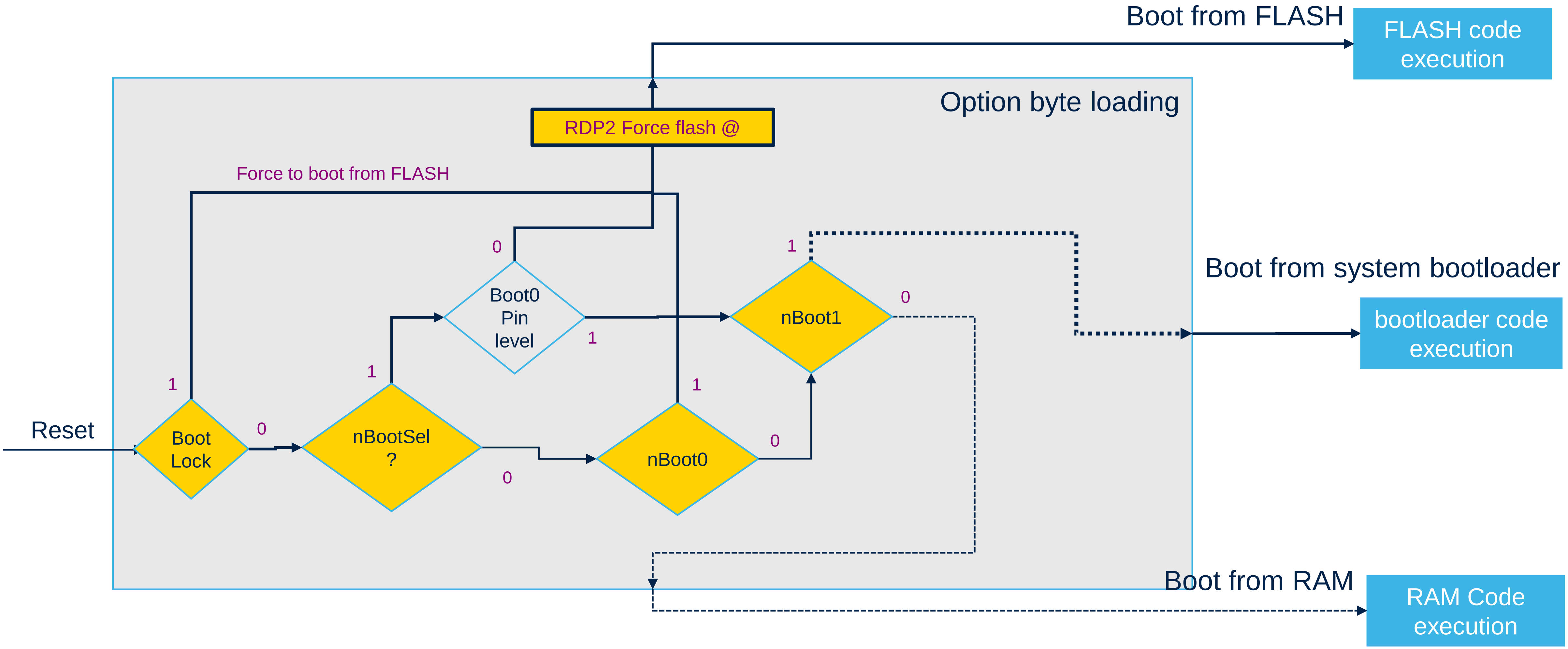
STM32WB

- STM32WB boot is the same L4

STM32G0 / STM32G4

- STM32G family boot is similar to STM32L0
  - nBOOT_SEL=0 forces software configuration for boot
  - nBoot0 pin level or nBoot0 config (depending on nBOOT_SEL) choose boot from user FLASH or other sources depending on nBoot1 config
  - nBoot1 could further decide to boot from sytem bootloader or RAM
- STM32G family also introduces the BOOTLOCK option which will bypass all the other boot settings in option byte or boot pin and force the boot from user FLASH, regardless of RDP level
  - BOOTLOCK can only be deactivated when RDP is in level 0 or with a RDP1 to 0 regression
  - BOOTLOCK=1 + RDP1 makes the boot from FLASH immutable
- RDP2 also forces boot from FLASH

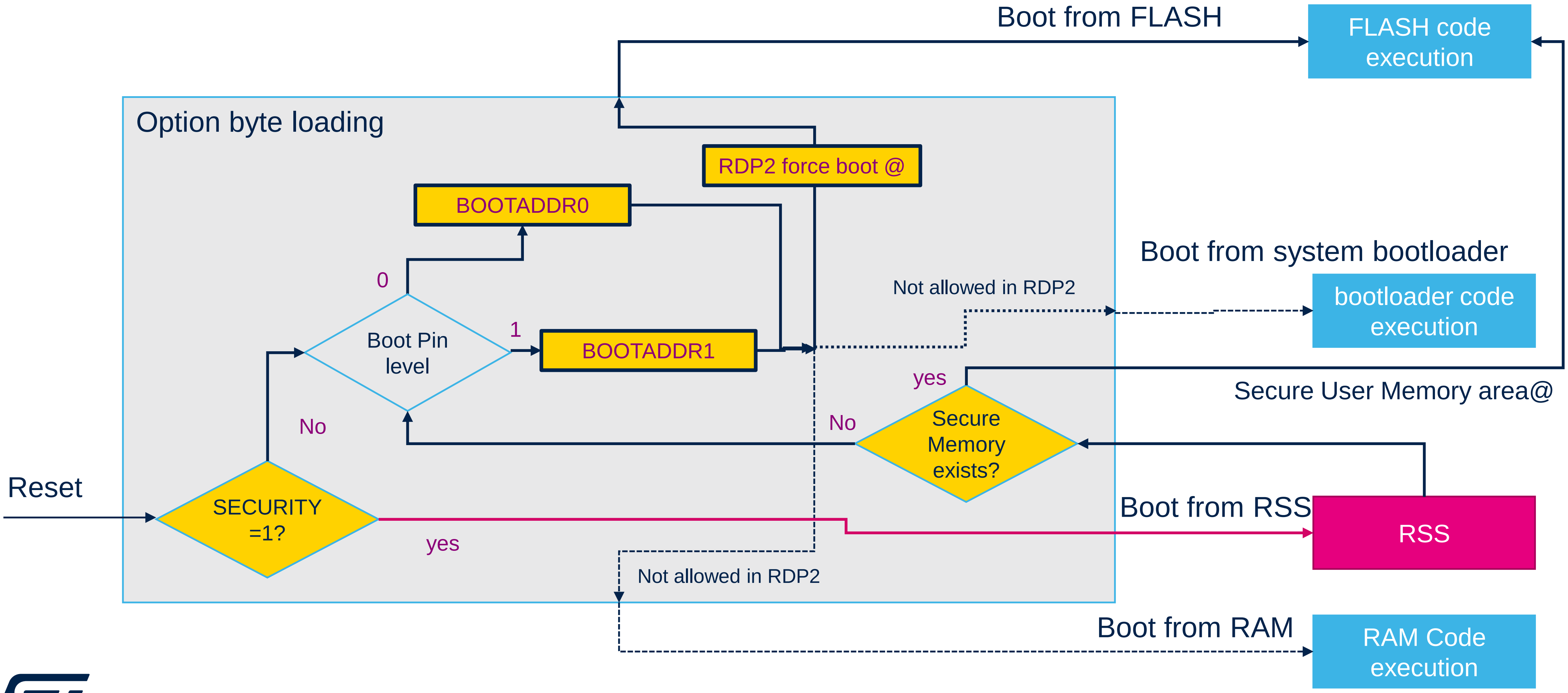
STM32G0/G4 boot principle



STM32H7

- Similar to STM32F7, STM32H7 family also has two boot addresses defined in BOOTADDR0 and BOOTADDR1 option
  - BOOTADDR0/1 could be configured to boot from user FLASH, system bootloader or RAM
  - Selection of the two address is done by BOOT pin
- On STM32H7 which has additional security features, additional RSS (Root Secure Service) area is embedded
  - When security is activated (by SECURITY option bit), boot is forced to RSS instead of normal system bootloader
  - Secure User Memory area can be defined when SECURITY is activated
  - RSS will jump to code in Secure User Memory as long as Secure User Memory is defined.

STM32H7 series boot principle



Ensuring immutability on STM32

- The first step for immutability is the use of Write Protect mechanism.
  - This mechanism always apply on a per flash sector basis.
  - The WRP mechanism is activated thanks to option bytes
  - Out of RDP2, option bytes can be changed
- RDP2 is also required to make sure option bytes will not change
- On chips including secure memory, the immutability is ensured as soon as RDP1 is activated.

Ensuring immutability with secure memory

- STM32G0, STM32G4, STM32H7 and STM32L5 have the ability to define a secure memory.
- When running code inside secure memory it is possible to erase FLASH inside this secure memory.
- When running code outside secure memory, FLASH erase is no more possible.
- The WRP also applies inside secure memory area
- So, to get immutability, this is same as other STM32: WRP + RDP2.

Sum up secure boot protection

- Ensure single entry point : RDP2, boot_lock, secure memory
- Ensure immutability: RDP2+WRP, partially with RDP1 + secure memory
- The association of Single entry point + immutability makes the secure boot as if it was ROM Code.
- This is what we call Root of trust: we trust that the code executing after reset is always the same.

Secure boot protection variations

- Depending on the environment constraints it is not always necessary to have the maximum protection (RDP2 for instance)
- But in that case, a detailed analysis has to be done to understand the weakness of a setup.
- For instance, on STM32H7, using secure memory + RDP1 + WRP inside the secure memory can be enough as long as code inside this area is not able to change this configuration.
- Such setup could be useful to still have the ability to change option bytes if this is necessary.

Immutable bootloader hands-on

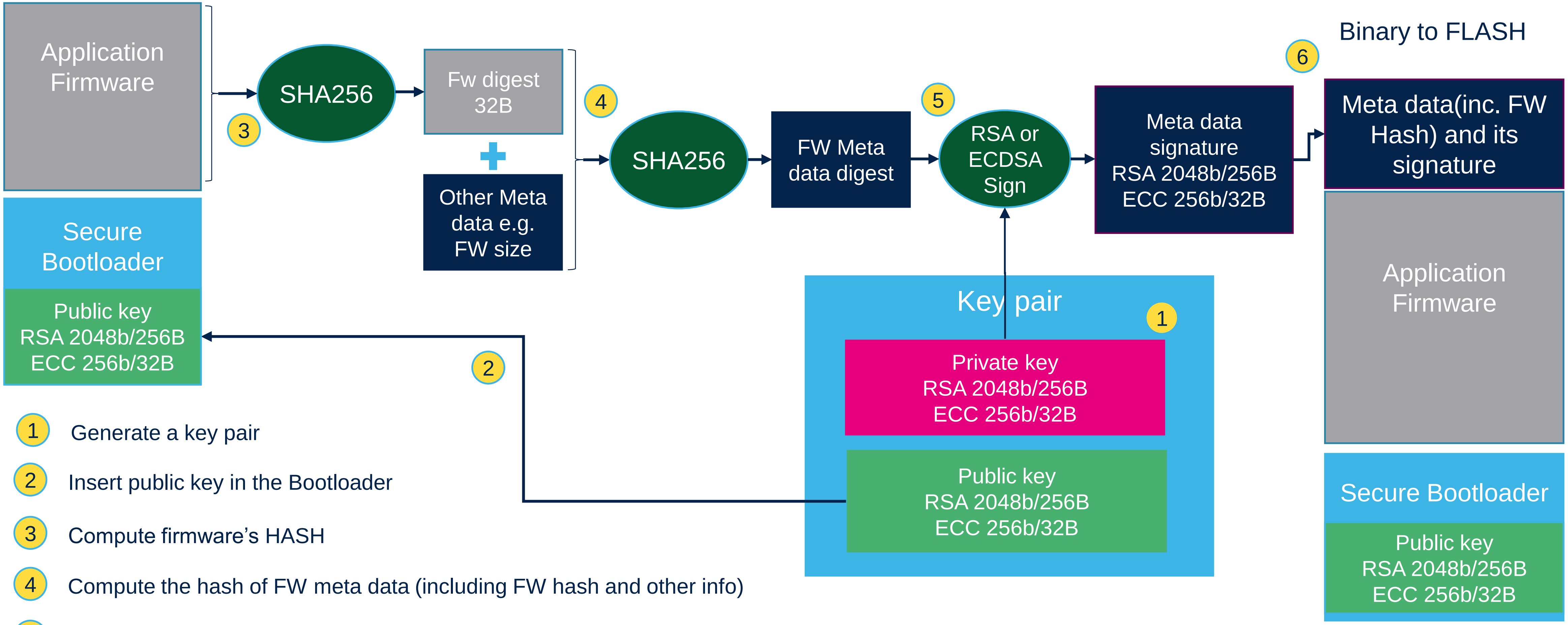
- The goal of this hands-on is to show the different steps to
  - Add immutability protection to the simple bootloader
  - Build the binary of both bootloader and application
  - Program bootloader an application to FLASH
  - Run the device and test the immutability of the bootloader from application

Step 3
authenticate target application from bootloader

Application authentication by bootloader

- Similar to the concept as described in Firmware Authenticity
  - Secure Bootloader shall authenticate the target application by verifying its signature
- The part of code to be signed is only the application.
  - Bootloader code is not part of data for signature generation or verification
- Usually a meta data (or header area) is attached to the application to allow bootloader to get the information of the data to be verified
- The public key will reside in the bootloader memory area (immutable)
- Bootloader will jump to the target application only if the signature verification is successful

Signature generation flow



1 Generate a key pair

2 Insert public key in the Bootloader

3 Compute firmware’s HASH

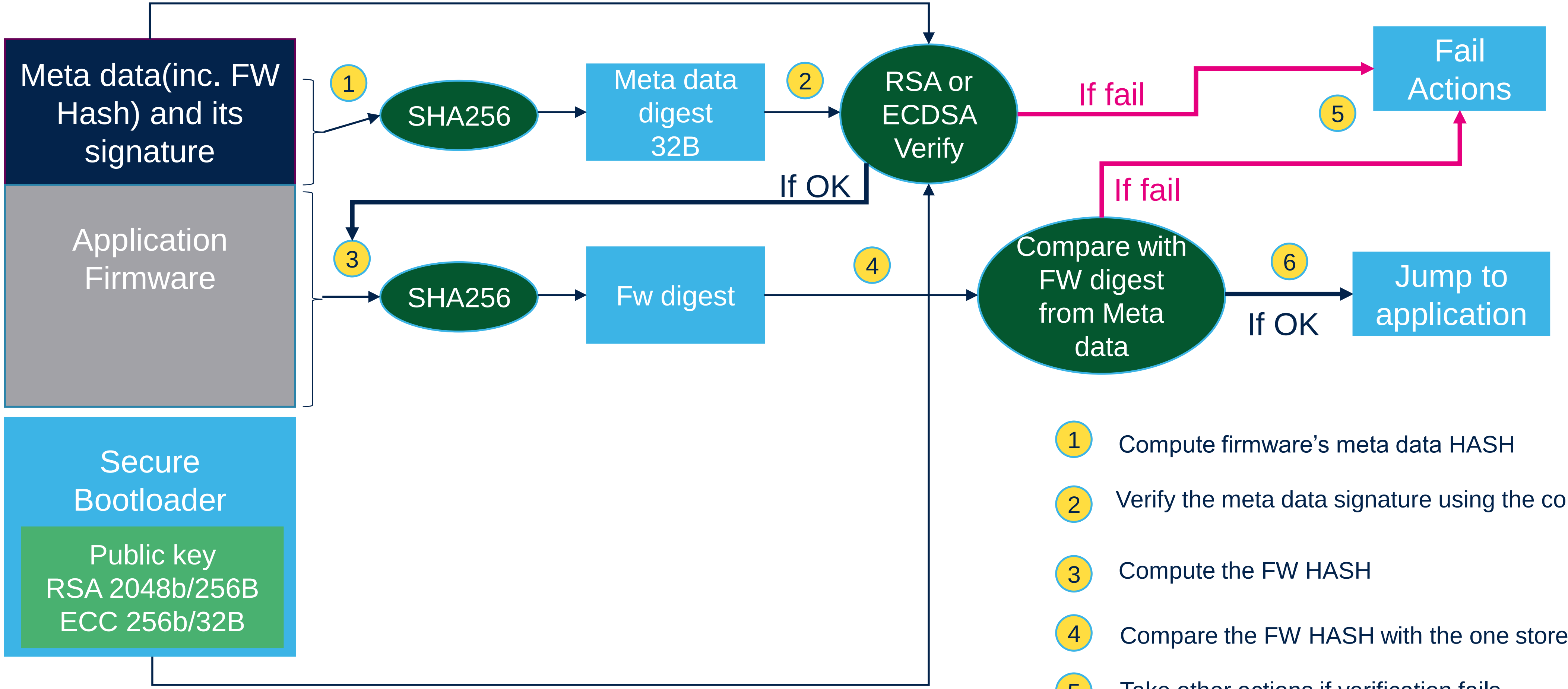
4 Compute the hash of FW meta data (including FW hash and other info)

5 Sign the firmware’s HASH

6 Add the signature (and other descriptive data) to the app firmware binary

life.augmented

Signature verification flow



1 Compute firmware’s meta data HASH

2 Verify the meta data signature using the computed HASH

3 Compute the FW HASH

4 Compare the FW HASH with the one stored in meta data

5 Take other actions if verification fails

6 Jump to application FW if verification is OK

Bootloader with authentication hands-on

- The goal of this hands-on is to show the different steps to
  - Add application authentication before jumping to the application in the immutable bootloader
  - Generate the meta data/header for the application (includes info such as size, version, signature, etc.)
  - Build the binary of both bootloader and application with meta data
  - Run the device and check the application signature verification inside bootloader

Secure bootloader sum-up

- We have addressed 3 main topics of Secure Bootloader:
  - Basic bootloader
    - Separation of bootloader from application
    - Launch application from bootloader
  - Make the bootloader trustworthy
    - Ensure the secure bootloader single entry point after reset
    - Ensure the immutability of the secure bootloader code
  - Authenticate the application from bootloader
    - Compute the FW meta data HASH and verify the meta data signature
    - Compute the application FW HASH and compare with the hash value stored in meta data area
    - Jump to application only if authentication is OK

life.augmented

Secure FW update

life.augmented

Why secure firmware update?

- Firmware update is a feature common in IoT products
  - The reason of firmware update could be bug fixes, adding new product functionalities, etc.
  - It is also a necessary feature to keep improving security protection along with product lifecycle in case any new vulnerability is discovered
- However adding firmware update capability also add new surfaces to attacks
  - Malware injecting through update procedure becomes possible
  - Firmware content could also be jeopardized or leaked during the update
  - Device firmware could be downgraded to an older version where know vulnerability still exists
- A Secure Firmware update shall mitigate those risks brought by the update feature itself

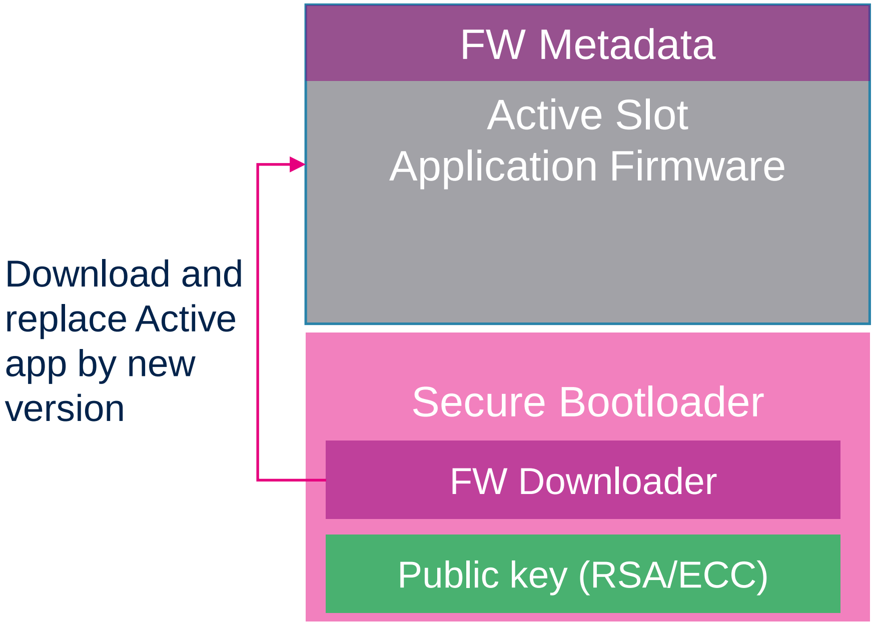
What should be protected during firmware update?

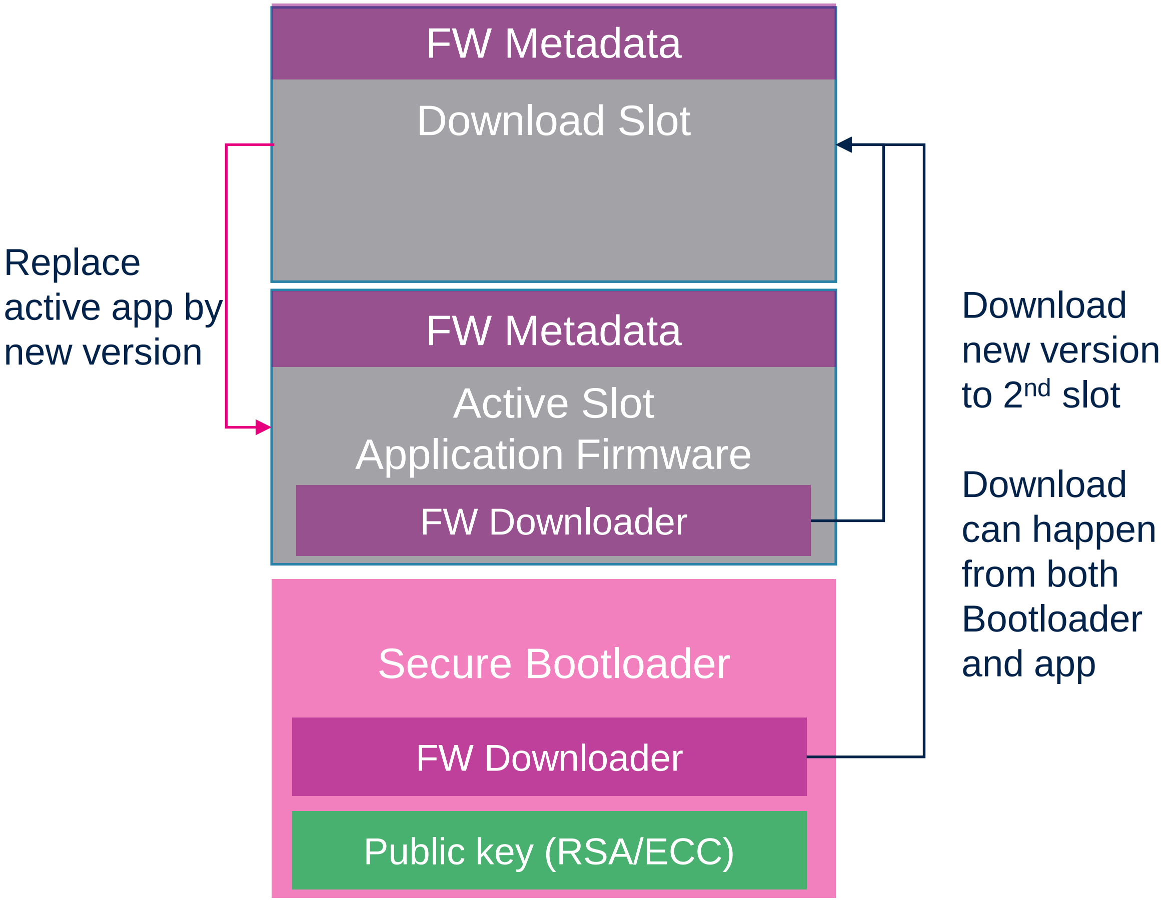
- Firmware Integrity & Authenticity
  - Secure firmware update shall always verify the integrity and authenticity of the new firmware to be downloaded and installed in the device
- Firmware Confidentiality
  - Firmware content shall not be expose to unauthorized party because of firmware update
  - Firmware content can not be transmitted in clear on the communication channel for upgrade
- Avoid firmware version downgrading
  - Secure firmware update shall have anti-rollback capability to avoid device firmware being replaced by an older version firmware which also has valid authentication data

Firmware download for update

- A Secure Firmware Update starts with the capability of updating the existing application in the device
- Device need to be able to download the data of the new firmware through a communication port
  - Data might be transmitted through a wired or wireless communication channel
  - Depending on the application requirement, different communication port or method could be used
- Firmware download can be a feature included in the Bootloader
- Firmware download in Application is also possible
  - linked to some constrains of the system, such as the memory space, …

Typical FLASH layout





- Single Slot for application image
- Update happens from Bootloader only

- Dual Slot for application image
- Download of new version can happen from Bootloader or application

Ensuring the firmware authenticity

- Firmware authentication is similar to the same concept explained in Secure Bootloader
  - The goal is to verify the authenticity of the new version of firmware before installing it.
  - Integrity of firmware will also be covered if the firmware is authenticated
- New version firmware shall come along with a set of meta data which includes necessary information for authentication, such as
  - Firmware HASH
  - Signature of the meta data
  - Other information describing the firmware (e.g. size, version, …)
- Verification procedure will verify the signature of the meta data itself as well as the digest of firmware content

Protect the firmware confidentiality

- To achieve the firmware confidentiality during firmware update, one option is to establish a secure communication channel
  - e.g. An IoT device can establish a secure communication channel with server using TLS. All the firmware data transmitted will be in encrypted format only
  - One concern of this approach is that you need to trust the firmware update server where your firmware is originally uploaded in clear. How the firmware is stored on the server is out of your control
- Another option is to have the firmware encrypted with predefined symmetric key before uploading for update service
  - Firmware will always be stored on the updating server in encrypted format and stays encrypted during downloading regardless of the communication channel encryption state
  - The symmetric key to decrypt the firmware need to be provisioned in the device beforehand
  - The symmetric key need to be protected for its confidentiality and integrity

Firmware encryption key protection on STM32

- RDP level 1/2 which is available on most of the STM32 family will protect the symmetric key from debug port access
- WRP can avoid the key data to be modified accidentally or intentionally (together with RDP2, key integrity can be assured)
- Product families with PCROP feature can protect the key from being read through debug port or read by code as data (but key content can still be retrieved by executing the code inside PCROP area)
- Isolation mechanism can be applied to further protect the key from being accessed by application (to avoid key exposure due to application vulnerabilities)

Firmware encryption key protection on STM32

- Protect the Encryption key on STM32 by isolation (from application)
  - L0/L4: Firewall can be activated after power on to protect the key data and its usage to prevent any access by the code outside Firewall area, including Bootloader code and application code.
    - It is still possible for the application to call the firmware decryption service from Firewall protected area even after execution already jumped from bootloader to application code
  - G0/G4/H7: Secure Bootloader including the firmware encryption key can be put inside the Secure User Memory area, which will not be accessible anymore after bootloader jumping to application.
    - Limitation is that the firmware decryption using the symmetric key can only happen when bootloader code is still running (not possible after jumping to application)

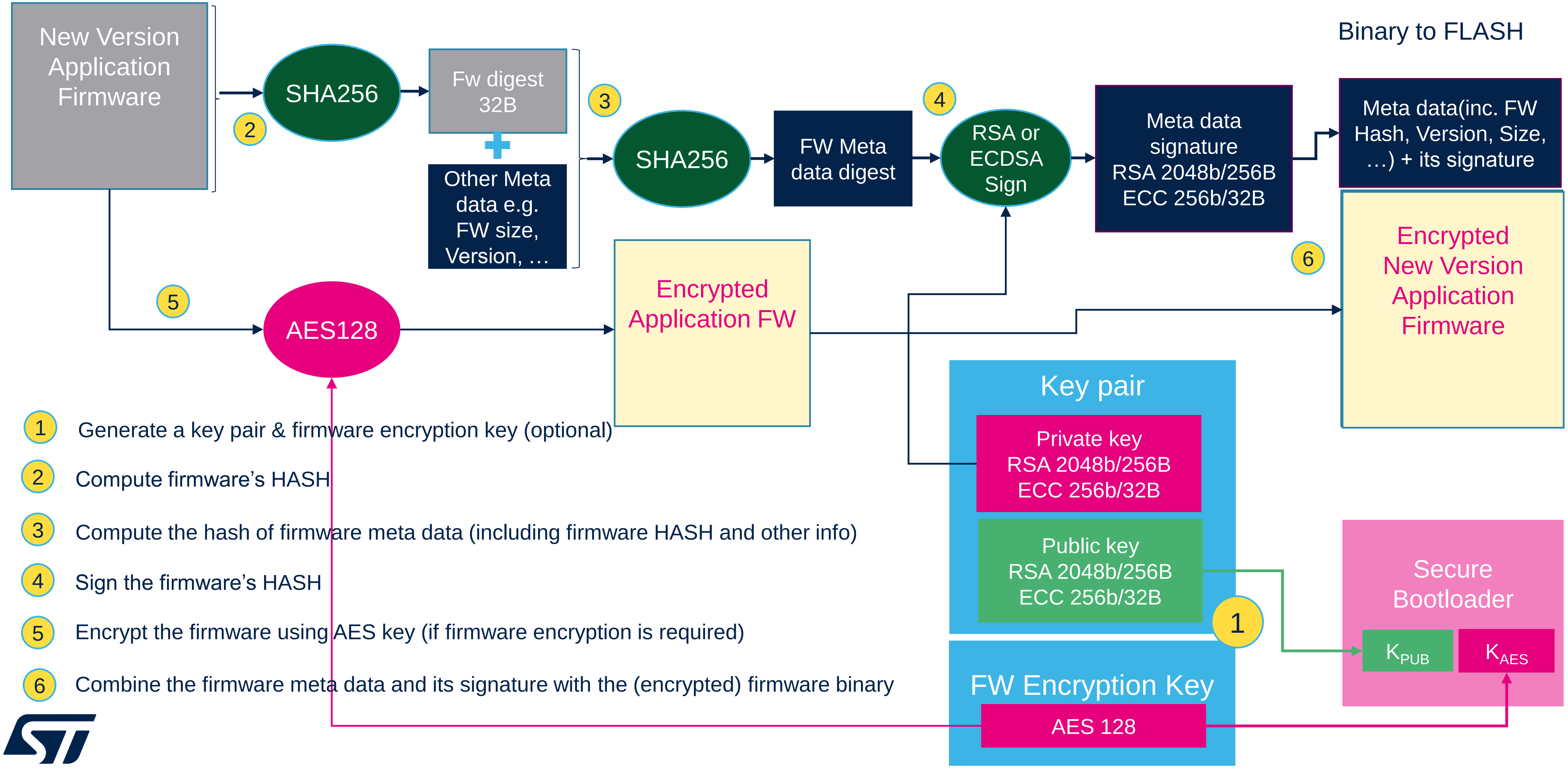
Firmware encryption key protection on STM32

- Protect the Encryption key on STM32 by isolation (from application)
  - **STM32L5**: Same Secure User Memory area and protection feature can be applied on L5 to achieve the same goal. In addition, TrustZone feature can also be used to do the isolation by putting the key inside Secured area and the application in Non Secured area.
  - **STM32WB**: There is the CKS (Customer Key Storage) feature thanks to the service provided from the code running on M0+ core and the isolation mechanism between M0+ and M4. Encryption key can be provisioned to the chip and be protected by CKS.
    - Decryption using the same key can be invoked by both bootloader and application without exposing the key content.

Avoid firmware downgrade

- Anti-roll back is also an important aspect of the secure firmware update process
- Firmware update shall check the firmware version and only allow newer version to be installed to the device
- Firmware version info shall be part of the meta data provided along with firmware binary, and need to be verified as well
- Info of the version of active application firmware shall be kept on the device for anti-rollback check during the firmware update procedure
- Integrity of version info shall be guaranteed

New version firmware preparation flow

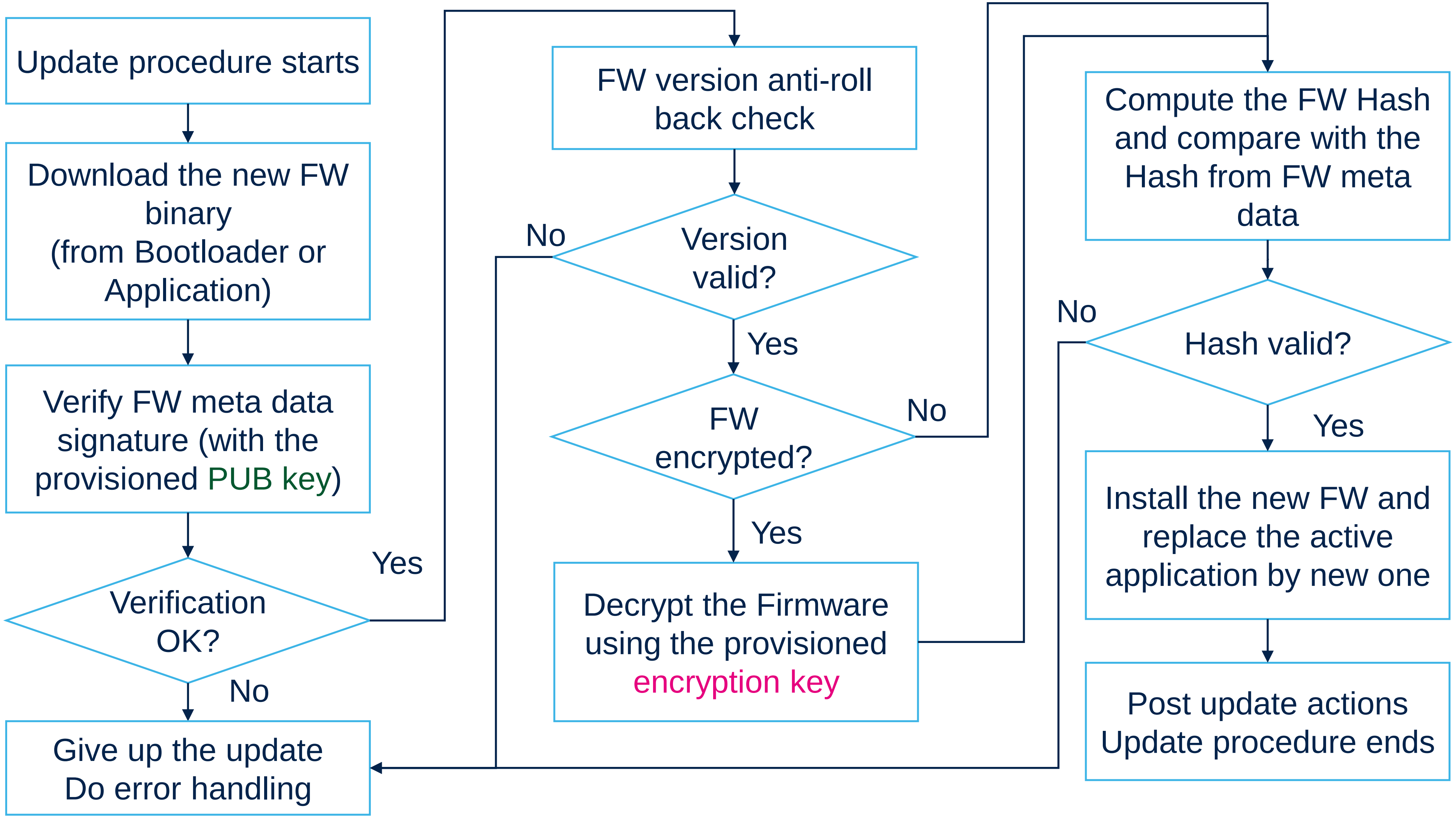


New Version Application Firmware
2
SHA256
Fw digest 32B
+
Other Meta data e.g. FW size, Version, …
3
SHA256
FW Meta data digest
4
RSA or ECDSA Sign
Meta data signature RSA 2048b/256B ECC 256b/32B
Binary to FLASH
Meta data(inc. FW Hash, Version, Size, …) + its signature
6
Encrypted New Version Application Firmware
5
AES128
Encrypted Application FW
Key pair
Private key RSA 2048b/256B ECC 256b/32B
Public key RSA 2048b/256B ECC 256b/32B
1
FW Encryption Key
AES 128
Secure Bootloader
K_{PUB}
K_{AES}

1. Generate a key pair & firmware encryption key (optional)
2. Compute firmware’s HASH
3. Compute the hash of firmware meta data (including firmware HASH and other info)
4. Sign the firmware’s HASH
5. Encrypt the firmware using AES key (if firmware encryption is required)
6. Combine the firmware meta data and its signature with the (encrypted) firmware binary

life.augmented

Secure firmware update basic flow



Update procedure starts
Download the new FW binary (from Bootloader or Application)
Verify FW meta data signature (with the provisioned PUB key)
Verification OK?
Yes
No
Give up the update Do error handling
FW version anti-roll back check
Version valid?
No
Yes
FW encrypted?
No
Yes
Decrypt the Firmware using the provisioned encryption key
Compute the FW Hash and compare with the Hash from FW meta data
Hash valid?
No
Yes
Install the new FW and replace the active application by new one
Post update actions Update procedure ends

Secure firmware update sum-up

- We have addressed several main topics of Secure Firmware Update:
  - New firmware binary download capability through a communication channel
    - Download of new firmware binary can happen from bootloader or from application
  - Ensure the authenticity of the new firmware to be installed
    - Verify the firmware meta data signature
      - Public key for signature verification need to be provisioned to the device
      - Public key integrity shall be protected
    - Verify the firmware HASH
  - Ensure the confidentiality of the firmware during download
    - One way is to rely on establishment of secure communication channel
    - Another way is to have the firmware encrypted in the first place
      - firmware decryption step is needed during firmware update in such case
      - firmware encryption key need to be provisioned in the device and be protected
  - Avoid firmware downgrade
    - firmware version shall be included in the firmware meta data to be verified
    - Version validation shall happen before new firmware can be installed

Thank you

life.augmented