

École Polytechnique Fédérale de Lausanne

Embedded secure boot: threat modeling and implementation with MCUboot

by Alex Ferragni

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Damian Vizár
External Expert

Ismail Ben Salah
Co-Supervisor

Daniele Antonioli
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

August 19, 2021

"Je sais pas si c’est une bonne idée, mais…"

Acknowledgments

I would like to express my very great appreciation to Ismail, without who none of this would have been possible.

I would also like to extend my deepest gratitude to Damian, whose guidance was invaluable.

Many thanks to Mathias and HexHive for accepting to supervize this project and for providing the thesis template.

I am very grateful Daniele, who offered precious feedback.

Special thanks to Edoardo Franzi and CSEM, for providing me a great internship oportunity in a delightful group.

I would like to thanky my family, who continuously supported me.

Finally, I would like to thank my friends who helped me in their own way.

Neuchâtel, August 19, 2021 Alex Ferragni

Abstract

This project focuses on the security of embedded systems that are connected over the Internet. More specifically, we focus on developing a secure bootloader, which is a software responsible for preparing the state of a device when it boots and should ensure the its functionality is not altered. We want to allow automatic software updates while preventing attackers from tampering with the system. We are interested in the Nucleo-H753ZI, which is a development board that includes all tools and hardware necessary to implement a secure bootloader. Many secure bootloaders already exist, but most of them rely on specific hardware and are not compatible with some devices such as the Nucleo-H753ZI. Additionally, their threat model is often incomplete or insufficient for a practical use. The attacks against which a specific bootloader is designed are not always clear, so it is complicated to find an existing solution that fits particular security requirements. In particular, we want a bootloader which provides confidentiality, integrity, availability of the installed image and that can update it. Therefore, this work focuses on exploring available secure boot solutions to find a suitable one and define a complete threat model along with it if necessary. The most suitable choice in our case was an open-source secure boot library (MCUboot), which is mostly hardware agnostic, but requires a bootloader to be built around it. Therefore, we present an implementation of a secure bootloader based on this library. Moreover, we define an abstract threat model compatible with most bootloaders, including MCUboot, which faithfully captures real-world intuition of a secure bootloader, as it can describe the most common real-world attackers against bootloaders.

Contents

Chapter 1

Introduction

This work takes place in the context of the Internet of things (**IoT**). It represents physical objects can be controlled by embedded devices and their respective software. Such devices can communicate with each other over the Internet. In such a setting, it can be desirable to ensure that those devices can fully operate without physical intervention. It is an imperative in most cases to allow such devices to update their software, still without physically accessing them. Since it is tricky to let a software update itself on-the-fly, a bootloader is usually included in such systems. The bootloader will ease the update process, and the software could directly download updates over the Internet without manual intervention. However, allowing such flexibility introduces new threats, namely threats that might want to take control of the device. In applications where the functionality of a device is critical (for example, in the medical domain), a more secure approach is necessary. In this work, we are interested in exploring the secure bootloader solutions available on a Nucleo-H753ZI development board.

Securing the software of a device can be a complicated task. We need to make sure that the device will only ever execute authorized and unmodified code. In addition to that, we want to ensure that the device remains functional without requiring manual repair. We also want to ensure that no unauthorized code can be installed. In this context, the range of possible threats is broad and the means to attack the device are diverse. Additionally, many kind of threats against an IoT device require specific hardware features in order to be mitigated. It follows that defining a precise threat model is important, and related work might be incompatible with a target device due to the hardware requirements.

After examining a selection of related work, we came to the conclusion that no existing solution was easily applicable on a Nucleo-H753ZI and had a fully satisfactory threat model. For each related work, at least one problem arised. The hardware mechanisms used to secure a bootloader are various, and so the related work was most often incompatible with out target device. Their design often relies on a component (or even architecture) that we cannot match, and their design was thus not even applicable in our case. Sometimes, the threat model was

not satisfactory because it was either too weak or incomplete. Finally, some works are not open-source, meaning we could not reuse them or even draw inspiration from them. We found no related work that combined an extensive and strong enough threat model, compatible hardware prerequisites and a usable open-source implementation.

It was therefore necessary to propose our own secure boot implementation and define a threat model, based on the related work. Since implementing a secure bootloader from scratch would be a very complex task, it is necessary to start from an existing implementation whose design is compatible with our device. We chose to use an open-source secure bootloader library, **MCUboot**, as most of the underlying assumptions can be met and the hardware was abstracted. This is however only a library, and a full bootloader still needed to be built around it. The library does not officially support our device, so a new port also had to be implemented. Some changes to the library itself were also be necessary, as it was not designed for our target device. Our bootloader serves as a wrapper around MCUboot and facilitates the implementation of a secure bootloader on another device. It was also desirable to include some utility tools to the bootloader (to help developing, testing and monitoring) since MCUboot did not include any.

This allows us to have a fully functional implementation on the Nucleo-H753ZI, but a threat model was still necessary. Unfortunately, MCUboot does not provide a usable threat model. MCUboot being only a library, a threat model cannot be fully defined at its level; it will depend on the complete bootloader. MCUboot only defines a few security assumptions on which its design rely. To define a threat model, our approach was to review the literature and define the categories of attacker models that are used (explicitly or implicitly) depending on the underlying hardware security. It was then possible to choose an appropriate basic threat model and fully extend it to our needs and to match the secure bootloader design.

As a result, we implemented a secure bootloader on the Nucleo-H753ZI that uses the MCUboot library. The bootloader verifies the authenticity of an image using a digital signature algorithm, before executing it. The design supports the installation of encrypted updates and possesses the redundancy necessary to ensure that, in our threat model, it is always possible to boot from a valid image. The bootloader includes utility tools such as a basic input/output system (**BIOS**) and preparation scripts. Such tools proved to be useful for development, testing and monitoring purposes. Additionally, the bootloader is designed to regroup all hardware-specific functions in a single place, allowing for an easier porting process. The bootloader is designed for the threat model that we defined.

The main contributions of this work are the following:

- **A threat model for a secure bootloader**. The proposed secure bootloader implementation was designed for this threat model. As mentioned earlier, existing threat models are often not exhaustive, or mentioned attacks without specifying whether the design was supposed to mitigate them. Therefore, we propose a formal two-phase threat model that precisely defines the capabilities and goals of different attackers. This way, many practical attacks can

be mapped to this attack model. The capabilities are restricted to a set which corresponds to attacks that the system can reasonably be defended against, but broad enough to include as many attacks as possible. Along with the most common attacker in the domain of secure boot (the jailbreak attacker), two more attackers are included: a DoS attacker and an IP thief. Some related works were designed to ensure confidentiality, but the precise attacker model (in particular, the capabilities of an attacker that tries to extract secret information) is always missing. Similarly, a DoS attacker is rarely mentioned in the related literature, even though it is important to prevent DoS attacks against IoT devices. Hence, a DoS attacker is also included in the threat model.

- **A case study of the implementation of a secure bootloader based using MCUboot on a Nucleo-H753ZI**. Along with an implementation of a secure bootloader that is based on our threat model, we conduct a corresponding case study. We confront the implementation with our threat model, reasoning about its security. In particular, we illustrate this with some real-world attacks mapped to our model. Additionally, we present a test methodology used to assess the correctness of the system from within usable when no debugging features are available. Finally, we evaluate the porting process itself and provide documentation necessary to use the system or port it on another device. Those are important because the bootloader is intended to be easy to use. Should someone need to port this bootloader to another device, the provided documentation will guide them through this process.

Chapter 2

Background

In this section, we shall present a few concepts that will be used throughout this work.

2.1 Bootloader

A **bootloader** is a software that takes care of booting a device (i.e., taking care of its initialization). As such, it often comes as a small software that is always executed after a reset and that is responsible for preparing the state of the device and executing the *main* software. It can have different goals depending on the context.

The most common feature is the initialization of the device on which it is running. The state of the device might not be persisted after a system reset, and the bootloader will thus have to re-prepare the state of the device after each reset. This can be done, for example, by preparing and loading the different drivers that will be used by the target program. This can also include the configuration of the underlying hardware, for example by writing to configuration registers.

Another usage for a bootloader directly comes from the fact that the bootloader is executed before the main software even starts. Indeed, a software might not be capable of modifying its own code while it is executing, which make it hard to apply changes to a software without external help. Since the bootloader is executed independently from the main software, it can more easily modify the latter by directly rewriting its software without risking a crash. The main software could then request changes by storing changes in non-volatile memory and reboot to let the bootloader apply them. Indeed, non-volatile memory is a category of memory whose content is persistent across restarts. Similarly, a bootloader can monitor the state of a main software during initialization, for example by verifying its integrity.

2.2 Integrated Hardware Security Features

Let us now introduce a few common hardware security features that we will use in this project.

Single Boot Entry Point

A simple mechanism that is necessary for any secure bootloader is the ability to force the hardware to always start executing the software from a given address. Most microcontrollers have a default boot address, which is the address from which the device will start fetching and executing code. For practical reasons, this is usually set to the start of the non-volatile memory. This way, a software installed at the start of the non-volatile memory will be executed upon boot. It is possible that a device offers no way to modify this default boot address. If this is the case, a single boot entry point is guaranteed by definition.

However, it may be possible to modify the boot address, and if this is the case a single boot entry point might be compromised. At any point, the boot address could be modified and the program at the previous boot address would no longer be executed upon boot. To still allow for a single boot entry point when this address in configurable, the hardware needs to provide a way to prevent further modifications of the boot address, permanently freezing this configuration. Otherwise, we cannot rely on a single boot entry point. The STM32H753 (i.e., the processor of the NUCLEO-H753ZI) proposes a unique boot entry point [1, p. 29]

Non-Volatile Memory Protections

Depending on the hardware, multiple non-volatile memory protections might be usable. We shall present here a few protections that will be necessary in this project.

A first security that is required for any secure bootloader is the ability to have read-only memory (**ROM**). The motivation is that we can trust that a software in read-only memory cannot be altered or corrupted. There are mainly two ways of storing code inside read-only memory. The first one is to embed the code directly during the creation of the read-only memory. For example, the content of the ROM could be sent to the manufacturer, which will write the ROM's content. Another more convenient way of programming ROM can be used if the hardware supports it. Instead of directly using ROM, one could use non-volatile memory which can then be permanently made read-only. This way, the code can be programmed once and then frozen to obtain read-only memory.

Another feature that will be required is the ability to safely store a secret in non-volatile memory. This requires hardware with a form of protection against unauthorized memory reads. There are multiple ways to achieve this, but the common guarantee we want is that the secure storage should only be readable by a given software, and should be inaccessible otherwise. The STM32H753 implements this by defining a secure area that becomes fully inaccessible

once it is locked [1, p. 29]. It is only possible to unlock the area by resetting the device. It then becomes possible to use the secret while it is accessible and lock it afterwards. This secure area might also be made immutable depending on the underlying hardware. If this is not the case, its immutability can still be simulated using other security features. For example, it might be possible to setup a secure area so that only a read-only piece of code can access it. In this case, if the immutable code itself cannot modify the content of the secure area, it effectively becomes immutable as well, as it cannot be accessed by any other piece of code.

Note that those features must not be bypassed by a debugging tool. A debugging tool (or **debugger**) is a device that can be used to directly access the internal state of a processor during its execution. It is typically used during development to analyze and control the execution of the processor. Non-volatile memory protection become meaningless if a debugger is capable of ignoring them. For example, it would be pointless to have a read-only piece of code executing if a debugger could simply modify the content of the processor's registers to execute other instructions. Similarly, a secure area should safely contain a secret and no debugger should be able to simply extract it. For that purpose, most microcontroller with integrated hardware security features provide a way to prevent a debugger from interfering with the security. This can for example be done by permanently disabling the debugging features or by reducing the debugger's abilities in order to not harm the underlying security. The STM32H753, for example, ignores all debugging events during the execution of the secure area [2, pp. 189–190]

Hardware Accelerated Cryptography

Lastly, cryptographic primitives might be implemented in either software or hardware. The advantage of using a software implementation is that this solution is portable, as no additional hardware support is required. the hardware might contain modules that can be used to enhance the cryptography. Alternatively, the cryptographic computations can be delegated to specialized component, if such components are available. Compared to a general-purpose processor, a component specialized in executing a set functions can be heavily optimized and thus offers faster execution times. However, the code becomes less portable as the hardware needs to possess such components. STM32H753 proposes a cryptographic processor and a hash processor for that purpose [2, pp. 1278–1279, 1247–1348]

Additionally, intermediate computations are in general less exposed than with a software implementation. Indeed, intermediate results have to be stored somewhere. In a processor, they might be stored in general-purpose registers or in volatile-memory, both of which might be accessible from the outside. On the other hand, a cryptographic module can have internal specialized registers, which are harder to access from the outside, meaning that intermediate cryptographic results are harder to extract. However, one should note that both inputs and outputs are not protected by the module as they are directly handled by the processor (and thus at least temporarily stored in general-purpose registers or memory). Thus, the added security only concerns intermediate results.

2.3 Secure Boot

Let us now introduce the notion of **secure boot**, which is a booting process that provides certain guarantees.

Principle

The goal of a secure bootloader is to ensure that only an authentic software is executed. In particular, a secure bootloader is intended to run before a *main* program and execute it afterwards only if the latter one is authentic. This is particularly relevant in a preferable context where the code itself could be updated, which means it could also be corrupted or replaced by an unauthorized code. This corresponds to a jailbreak attack, which consists in trying to persistently modify a software in order to execute attacker-controlled code. Thus, a secure bootloader is mainly designed to protect against jailbreak attacks, and it is preferable for a secure bootloader to handle software updates itself.

To check whether an image is authentic, the secure bootloader itself should always be authentic as well. Otherwise, the bootloader itself could be replaced and no guarantee can be provided. For that purpose, we use two hardware properties: a single boot entry point and read-only memory. We call a memory region that possesses both properties a **root of trust**. Therefore, the bootloader should be stored in ROM which is also the unique boot entry point (i.e., in a root of trust). Indeed, the hardware will then guarantee that the bootloader cannot be modified and will always be executed after a system reset. It is thus possible to safely use it to verify a software.

Authenticity Verification

In fact, the secure bootloader works by establishing a **chain of trust**. The chain of trust consists in successive verifications, where each trusted software verifies the next one, which in turn becomes trusted. By starting from a chain of trust, we can essentially attest that all softwares are authentic before executing them. In our case, the chain of trust will only be made of the secure bootloader and the target image, but we can imagine having multiple levels of bootloaders or images. Figure 2.1 depicts a signature-based root of trust.

The image verification itself could be either hash-based or signature-based. A hash-based verification would essentially hash the entire content of an image and compare it against a reference hash that is embedded in the bootloader (and thus, immutable). The advantage is that image forgery is in practice impossible as long as the underlying hash function is secure. However, the bootloader becomes incompatible with updates, as another authentic image will have a different hash. To be verified, the bootloader would either need to possess in advance this hash (which means the image can only be part of a fixed set of images, so no new versions can be authenticated) or have a way to add hashes in the bootloader. The latter solution is problematic

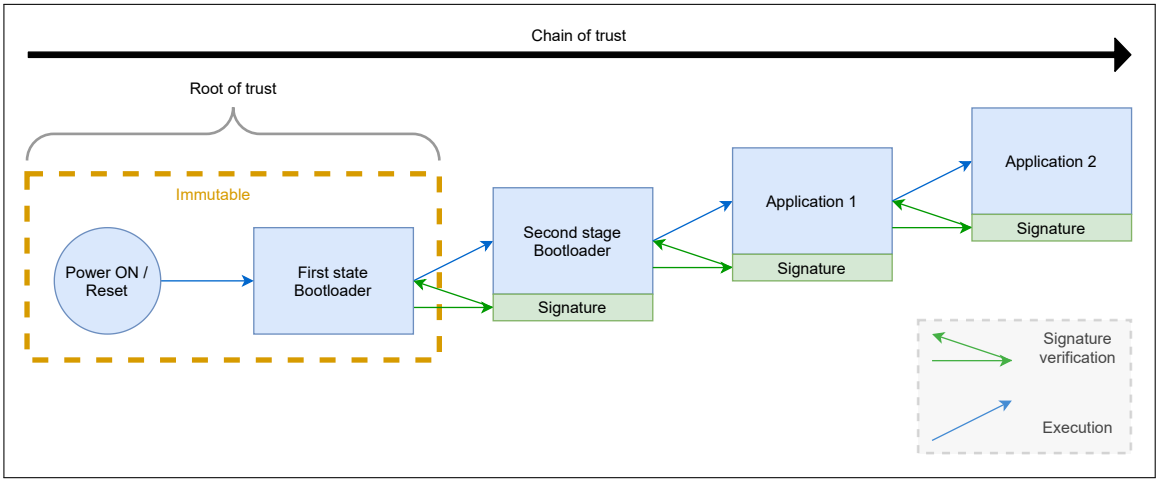

Figure 2.1: A signature-based chain of trust.
The first stage bootloader is always executed upon boot and is immutable, it is thus a root of trust. The chain of trust is extended to subsequent programs by performing verifications.

as allowing to add image hashes also adds an attack vector, as an undesirable hash might be added, which defeats the purpose of a secure bootloader. On the other hand, a signature-based verification can be used. The advantage is that is can easily handle updates, as any image signed using a private signature key can be verified by the bootloader using the corresponding public signature key, which can be embedded in the bootloader. However, if the private signature key is ever leaked, images can easily be forged and the secure bootloader is effectively broken. Additionally, there is no trivial way to revoke a key, as that would introduce additional problems.

2.4 MCUboot

Let us now describe the design of a generic secure bootloader that is based on the MCUboot library, starting with its design. In fact, MCUboot is configurable and allows for different models, but we choose to only describe the particular configuration that we will use later in the implementation.

2.4.1 Bootloader Design

To avoid confusion, we will use the term **input image** in this chapter to describe the software that we want to authentify with MCUboot. On the other hand, an **MCUboot image** is an image format which contains a software along with cryptographic data necessary to authentify or encrypt it. In the rest of the thesis, **image** will simply refer to an MCUboot image.

The bootloader is intended to be stored in read-only memory and to be the only boot entry

point. This way, it is an immutable root of trust. It can be thought of as being only updatable on a per-hardware revision. Since the bootloader itself is a root of trust, we can use it to extend the chain of trust to the image before safely executing it. Thus, when the system is reset, the bootloader will always read the content of the primary slot and verify its signature (by using a public key embedded in the bootloader). If the signature is verified, the bootloader will then pass control to the image since it is authentic. In case the signature fails, the execution will stop and the device will hang. This way, we can ensure that only authorized images are run after reset.

The update protocol will check whether a newer version in the secondary slot is ready to be installed. Upon reboot, the bootloader will read the content of the secondary slot (where a requested update should be stored) and decide whether the upgrade process will be initiated or not. To install the update, both images will be swapped in non-volatile memory (such that the most recent version is now installed). The previously installed version is kept in the non-volatile memory to ensure the system can rollback to it if necessary. The status of the update process is persisted in storage to ensure it can recover from a failure. The swapping process is done using a third storage slot (the scratch area) and some steps are persisted to internal storage in a log. This ensure that if at any point during the update the system were to fail (e.g., a power failure or a system reset), it could read the log to see that an upgrade process was ongoing and could resume it. This log is marked as completed when the update is finished.

2.4.2 Flash Areas

MCUboot defines three different areas of the (internal and external) flash[1] memory that the bootloader should use. They are defined as follows:

- **Primary slot**. This area will contain everything necessary to use the currently installed MCUboot image (i.e., its code and constants, but also its header, status information and cryptographic data). It is thus the code that MCUboot will execute (provided that its verification is successful).
- **Scratch area**. This area is simply an empty region that can be used by MCUboot during the update process to temporarily store part of the images while swapping them.
- **Secondary slot**. This area can be filled (by the bootloader, or by the image itself) with a newer version of the image to initiate the update process. If the newer image is authenticated by the bootloader, it will eventually be installed in the primary slot and run from there.

[1] Flash memory is a type of non-volatile memory. Its content can be programmed, but an entire flash sector has to be erased before any of the words it contains can be written to a second time

2.4.3 Boot Flow

This section will describe in more details the algorithm used by MCUboot. *Algorithm 1* shows the execution flow of the bootloader. As mentioned previously, the bootloader will check if an update is available in the secondary slot and apply it if it is validated. It will also revert back to the version available in secondary slot if the image requests it. Independently from the update, it will always verify the image in primary slot (by checking its signature) before booting it.

Algorithm 1 Bootloader Flow

```
1:  upon event Reset do
2:      // Update phase
3:      status ← Read update status
4:      image ← Read primary slot content
5:      upgrade ← Read secondary slot content
6:      Decrypt upgrade
7:      if status indicates update is not complete then
8:          // Resume update
9:          UPDATE()
10:     else if upgrade is authentic then
11:         if upgrade is marked for update then
12:             // Newer version should be installed
13:             Clear status
14:             UPDATE()
15:         else if image is not authentic then
16:             // Current version is corrupted, fallback to secondary slot
17:             Clear status
18:             UPDATE()
19:     // Boot phase
20:     if image is authentic then
21:         Jump to image
22:     else
23:         // No valid image exists
24:         Panic
```

The update protocol is designed to swap the content of primary and secondary slots while being failure-resistant (i.e. resisting system resets). It does so by persisting its progress to storage. More precisely, it will regularly append its progress to the upgrade status, which is located in the primary slot. This way, if the update process is called after a previous interrupted update, it will skip all steps that were already executed and continue where it has stopped, effectively resuming the update. *Algorithm 2* shows the update protocol.

To initiate an update, the image is responsible of preparing the new image in the secondary slot. *Algorithm 3* shows the function that the image should use to request an update. The image will then take care of downloading a new image from the update server, and storing it in the secondary slot. To indicate to the bootloader that an update is requested, the image in the

Algorithm 2 Update Protocol

```
1:  procedure UPDATE()
2:      n ← number of flash sectors in an image
3:      for i ← 0; i < n; i ← i+1 do
4:          // Upgrade -> Scratch
5:          if (i,0) is not in status then                    } Step 1
6:              data ← read sector i of secondary slot
7:              Decrypt data
8:              Store data in scratch area
9:              Write (i,0) in status
10:         // Image -> Upgrade
11:         if (i,1) is not in status then                    } Step 2
12:             data ← read sector i of primary slot
13:             Encrypt data
14:             Store data in sector i of secondary slot
15:             Write (i,1) in status
16:         // Scratch -> Image
17:         if (i,2) is not in status then                    } Step 3
18:             data ← read scratch area
19:             Store data in sector i of primary slot
20:             Write (i,2) in status
21:     Mark update as complete in status
```

secondary slot should then be marked for an update. Note that unlike *Algorithms 1* and *2*, *Algorithm 3* is contained in the primary slot and executed by the image, and not by MCUboot.

Algorithm 3 Image Update Primitive

```
1: procedure REQUEST_UPDATE
2:     // Called whenever the image wants to initiate an update
3:     image ← download image from update server
4:     Store image in secondary slot
5:     Mark secondary slot header for update
6:     Reset the system
```

2.4.4 Cryptographic Schemes

We say that an input image should be converted to an MCUboot image before being usable by MCUboot. Let us now describe how.

Setup

First, let us present the initialization of the system, which consist in sampling and storing cryptographic keys that will be required during the encryption/signature process as well as the decryption/verification process.

Two key pairs are required. An **ECDSA** [3, pp. 43–49] key pair is generated which will be used to authenticate the input images using asymmetric cryptography. It is composed of a public key k_{sign}^{pub} and a private key k_{sign}^{priv}. The private key will be useful to sign input images while the public counterpart will be used by MCUboot to verify signatures. An **ECIES** [3, pp. 50–54] key pair is also generated and will be used to safely provision the image encryption key to MCUboot using ECIES. It is composed of a public key k_{enc}^{pub} and a private key k_{enc}^{priv}. Figure 2.2 summarizes the keys generated during the initialization process.

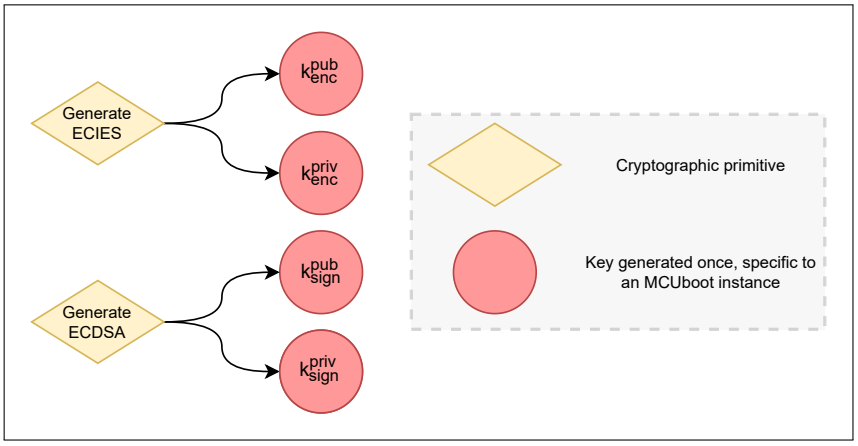

Figure 2.2: Initialization of cryptographic schemes

Image Signature

First of all, we need to prove the integrity and authenticity of the input image. To do that, the content of the input image as well as its header are hashed using SHA256, giving $hash_{img}$. This hash is then signed using ECDSA, with k_{sign}^{priv} as the key. This generates $signature$, which is put in the MCUboot image. Finally, to indicate which key was used to sign the input image, the public counterpart of k_{sign}^{priv} (k_{sign}^{pub}) is hashed with SHA256, giving $hash_{sign}^{pub}$. This hash is also put in the MCUboot image. Figure 2.3 summarizes the signature process.

Image Encryption

We will explain here in detail the process through which an input image can be converted to an MCUboot-compatible image, the latter one being encrypted and signed. This is the state in

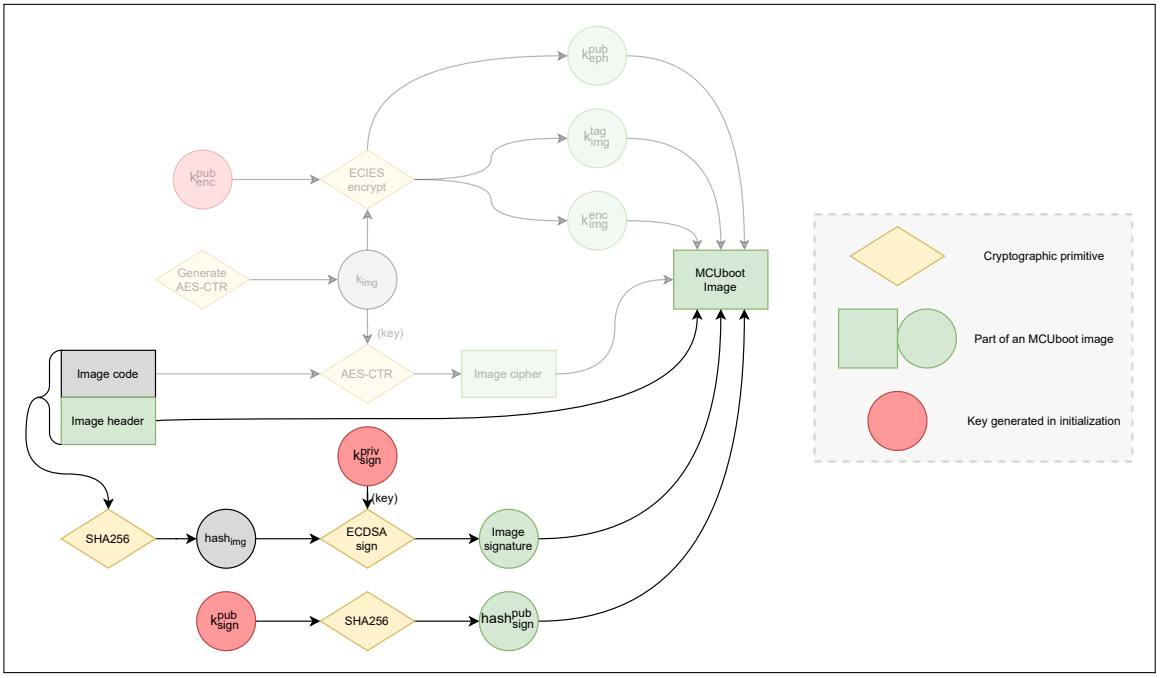

Figure 2.3: Signature process of an image

which a software should be stored in the secondary slot. The bootloader can only verify and apply updates that are in this state. The encryption process described in this Section is repeated for every new input image and summarized in Figure 2.4. In particular, keys generated during the encryption process are specific to an input image and will thus be generated again for every other input image. This process takes place in the update generating server.

Now that a signature has been created, the input image shall be encrypted. A new AES-CTR key is thus generated. This key will be used to encrypt the content of the input image. We call this key k_{img}. The content of the input image is thus encrypted using AES-CTR (with an initialization vector and a counter of 0) and k_{img}, which results in $image\ cipher$. The header of the input image is not encrypted. Both the unencrypted header and $image\ cipher$ are added to the resulting MCUboot image. To be used during the decryption process, the image encryption key should be safely provisioned. It is thus encrypted and signed using ECIES with k_{enc}^{pub}. This will generate an encrypted image key (k_{img}^{enc}) and its corresponding tag (k_{img}^{tag}), as well as an ephemeral public key (eph_pub) which will be used in the ECIES decryption process. Those three results are thus put in the MCUboot image.

Section 2.4.5 summarizes the structure of an encrypted and signed MCUboot image, with all cryptographic information that is included.

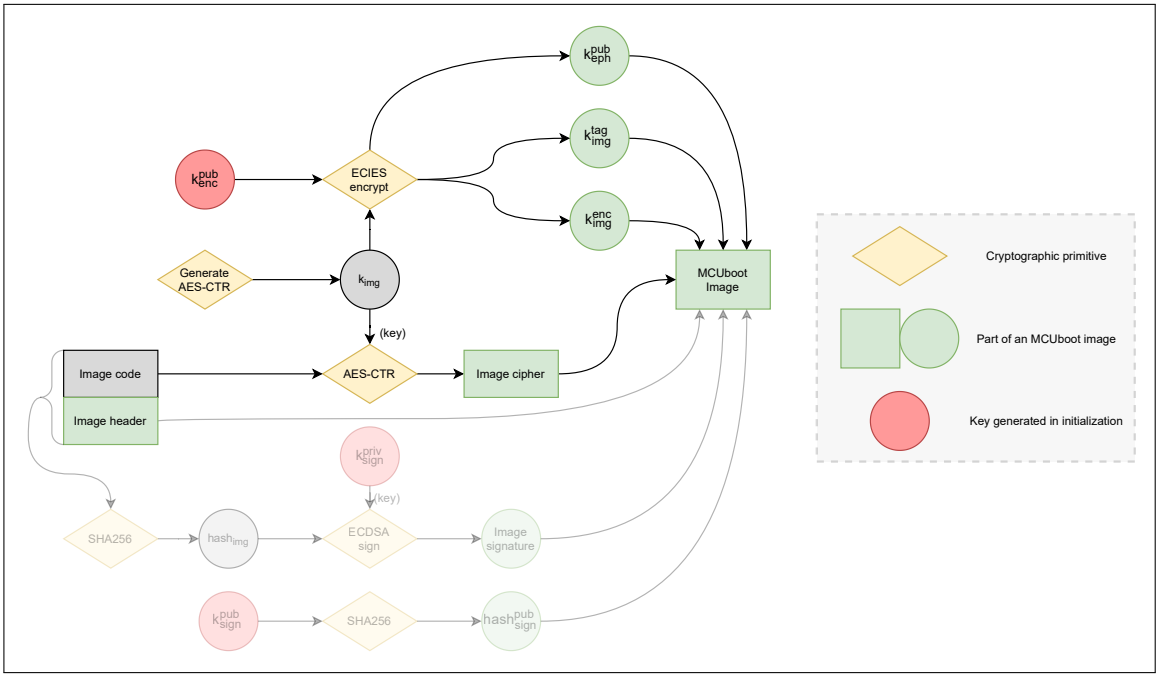

Figure 2.4: Encryption process of an image

Image Decryption

Let us now present the decryption process of an MCUboot image. This process takes place in the bootloader and occurs whenever an update (located in the secondary slot) should be applied and uses both k_{sign}^{pub} and k_{enc}^{priv}. Note that verifications have to be made at multiple steps during the decryption or signature verification process. Any failure to verify a signature or to compare a value will show that the MCUboot image is not authentic or has been altered and the decryption will be aborted immediately. Figure 2.5 summarizes the decryption process.

First, the bootloader needs to recover the key necessary to decrypt the MCUboot image (k_{img}). Since it was encrypted and signed using ECIES encryption, it will be recovered using ECIES decryption. The inputs of the ECIES algorithm are the key to be decrypted (k_{img}^{enc}), the public ephemeral key (k_{eph}^{pub}), the secret bootloader key (k_{enc}^{priv}) and the reference tag (k_{img}^{tag}). The algorithm will indicate whether the key is valid and will output the decrypted image key if this is the case. Using k_{img}, the image cipher can then simply be decrypted with AES-CTR (with an initialization vector and counter of 0), resulting in the input image plaintext.

Image Verification

Since the MCUboot image signature verification process requires the plaintext version of the input image code, it will take place after the decryption process. It will also take place after every

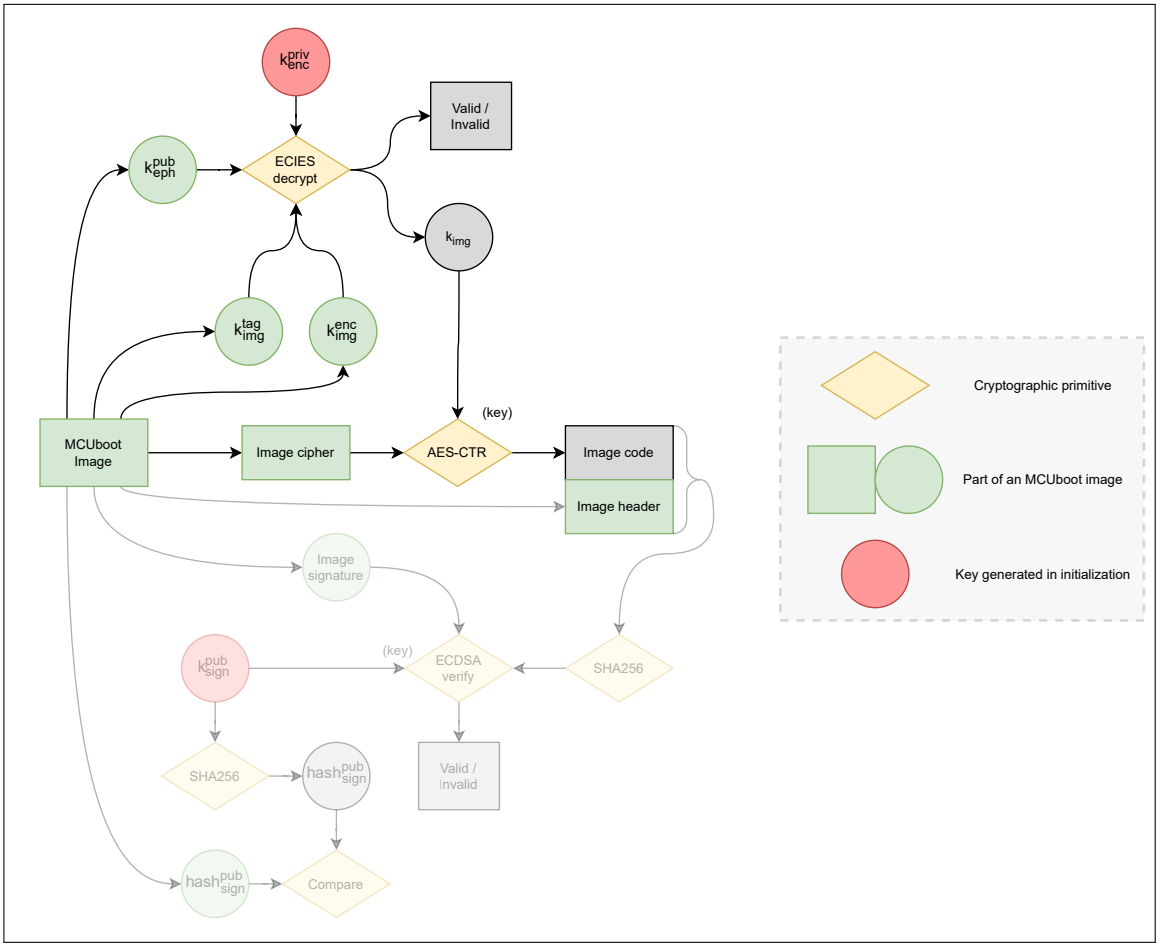

Figure 2.5: Decryption process of an image

reboot, before executing the MCUboot image, since it has to be verified every time. Figure 2.3 summarizes the signature verification process. To verify the MCUboot image, the decrypted content of the input image and its header are first hashed using SHA256 to obtain $hash_{img}$. Then, a verification of the key necessary to verify the MCUboot image is made by computing a hash of k_{sign}^{pub} with SHA256 and comparing it to $hash_{sign}^{pub}$. This step is not necessary if only a single signing key pair is used, but it can be used to choose a key if multiple public keys (corresponding to multiple signing entities) are embedded in the bootloader. Finally, the signature of the MCUboot image ($signature$) is then verified with ECDSA using k_{sign}^{pub} as the key, and using the previously computed image hash. Its result will indicate whether the MCUboot image is authentic and can be applied or executed.

2.4.5 Image Structure

Figure 2.7 depicts the complete structure of an MCUboot image.

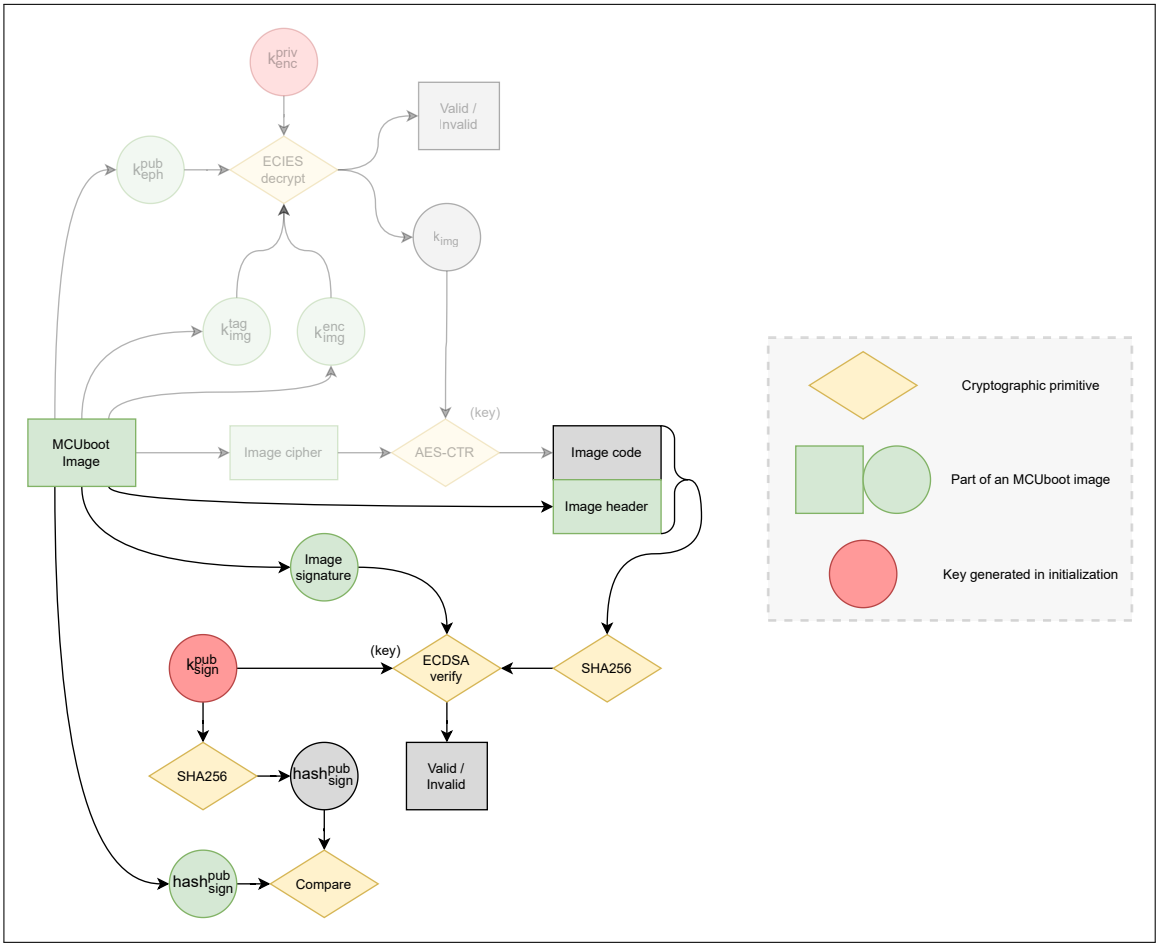

Figure 2.6: Signature verification process of an image

The MCUboot image is generally split into three parts: the header, the content and the trailer. The header is an area where information about the input image are stored. Such information include image size, version and cryptographic scheme. Since the header contains important data that must be readable before encryption, it is signed as part of the authentication process but is never encrypted. This way, data can simply be read by the bootloader but any tampering will be detected. The content area contains the actual software of the input image, which is thus encrypted and signed. Finally, the trailer area serves two purposes. Firstly, it contains information necessary to the bootloader to successfully decrypt and verify an MCUboot image. Secondly, it has an area reserved for status information of the MCUboot image. This area is initially empty, meaning that a freshly generated MCUboot image will have an empty status area. This area is available to store persistent status-related information on the MCUboot image and is used by the bootloader. This includes current MCUboot image swap status (as shown in Figure 2.7) as well as a flag that indicates whether the slot contains an update that should be applied (in the case of the secondary slot). The MCUboot image in primary slot itself should set this flag in the secondary slot to initiate an update when it is ready. Since it is supposed to be modified, the status area is neither encrypted nor signed.

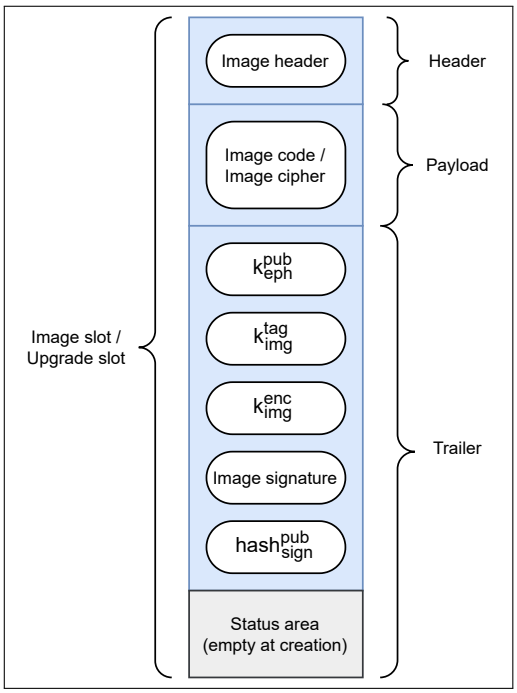

Figure 2.7: Structure of an MCUboot image. An update is stored in the secondary slot and contains the image cipher as a payload, while an installed image is stored in the primary slot and contains the input image code as a payload.

Chapter 3

Threat Model

In practical scenarios, we can distinguish three main classes of attackers against a secure bootloader. Firstly, an attacker might be interested in jailbreaking the device. **Jailbreaking** is defined here as the action of modifying the system to remove restrictions which prevent users from installing unintended software. This can be achieved by first exploiting a vulnerability in the software to gain a temporary access to the system and then trying to alter the system such that control is not lost even after a system reset, effectively taking permanent control of the device. Additionally, an attacker might simply want to perform a denial-of-service (**DoS**) attack using limited means. This could represent an attacker that wants to prevent it from functioning correctly, for example by interfering with an update protocol. An attacker might also want to steal the intellectual property (**IP**) of the image (if that is applicable). If the image is a proprietary software, an attacker might want to obtain knowledge of the image to reuse it for their own (potentially commercial) purposes.

The three different attackers will be referred to as **jailbreak attacker**, **DoS attacker** and **IP thief**. The attacker model will be inspired by those examples. Let us now introduce the model we will use in this work, starting with the system model.

3.1 System Model

The main target application is a constrained embedded system. Indeed, we are interested in securing the boot process of a constrained embedded device. More specifically, the device we are considering is made of a microcontroller located on a board with possibly some external components.

The system has different components that are both internal and external to the microcontroller. Inside the microcontroller, there is a CPU (that will execute the code), some RAM and

some non-volatile memory (split into segments that can each have their own properties) that are connected to each other with a bus. Additionally, outside of the microcontroller there can be some optional external non-volatile memory, an optional secure element and peripherals to communicate with a user or the update server. Those components are connected to pins of the microcontroller with wires that are exposed on the board. We will describe those components in more detail in *Section 3.1*. Figure 3.1 illustrates the layout of the system.

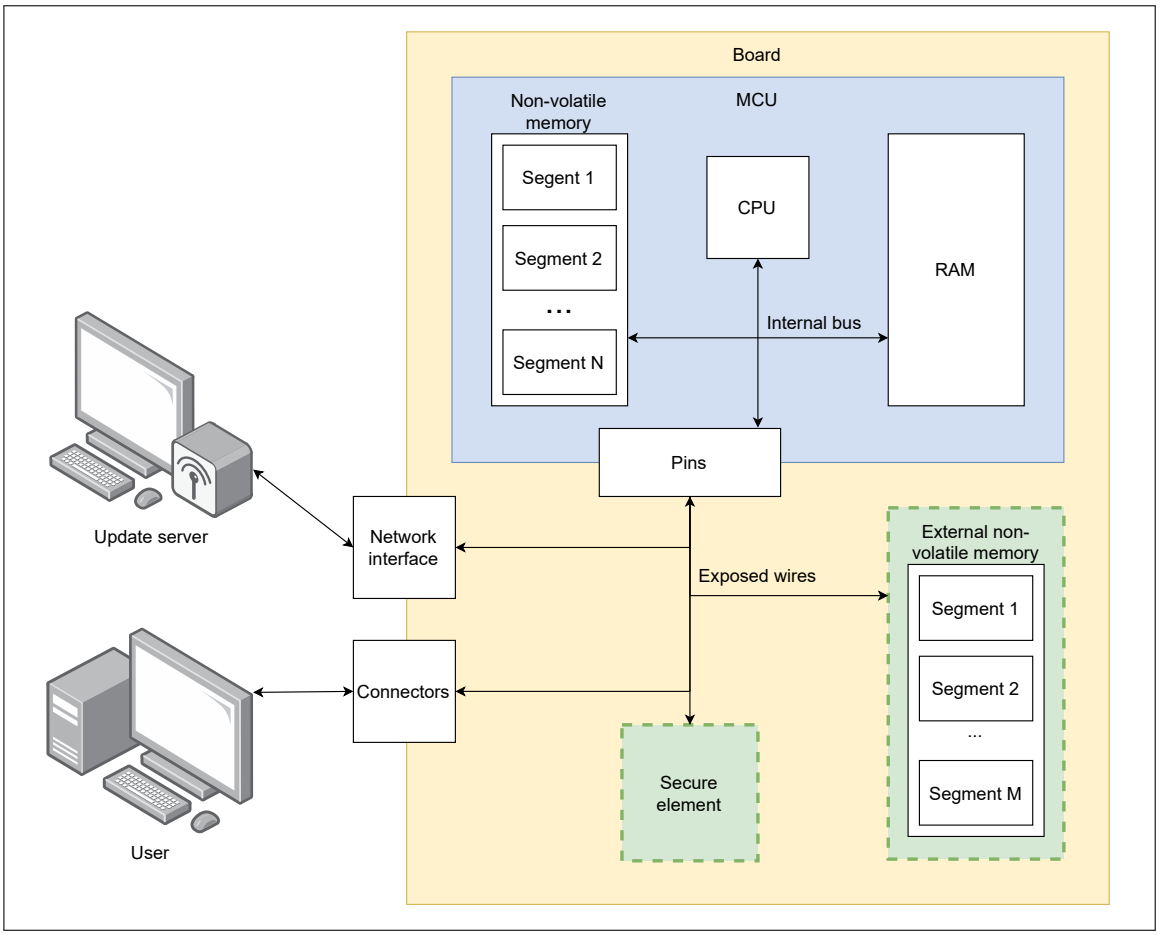

Figure 3.1: Physical layout of the system.
The system is composed of and update server, a user and a board (connected via network interface and connectors). The optionally contains a secure element and an external non-volatile memory. It also has a microcontroller, which contains a CPU, some internal non-volatile memory and RAM.

The model we use is suitable because it is minimal while still being sufficient enough to implement a complete secure boot. In particular, it does not require a trusted execution environment or a trusted recovery storage and can even be implemented without the secure element. Not including such elements in the model makes it less general, but we argue that this model covers most of the applications. It also avoids adding unnecessary complexity to the model. This is desirable for an embedded system, which has access to limited resources. This model is also general enough to fully implement a secure boot as well as a secure update protocol.

Internal Components

Let us now describe individually the components of the system and the way they interact with each other, starting with the content of the microcontroller. We define an **internal** component (described in this chapter) as the components which is located within the microcontroller, as opposed to **external** components (described in *Section 3.1*) which are physically separated from the microcontroller.

First of all, the processor is responsible for executing code (of both the bootloader and the image) and performing computations. It will fetch instructions directly from non-volatile memory and execute them. It also contains the necessary computational power to successfully execute the bootloader and the image (which includes some basic cryptographic support, such as computing a hash or encrypting some data). It also has direct access to memory (both volatile and non-volatile) and can freely read and write to them (as long as their properties allow it). The processor can simply be modeled as a state machine, each state corresponding to an instruction, and which follows the execution flow of the code. It also possesses internal registers, which completely define the current state of the processor. After a power-up, the processor is put in its initial state. This corresponds to the state where the content of every volatile memory in an undefined state, and where the processor's registers are put to an initial functioning state (corresponding to being ready to execute the first instruction of the bootloader). We define a system reset or simply **reset** as the action of putting the system back to this initial state.

Internal Memory

The processor then needs internal volatile memory. Random-access memory (**RAM**) is thus placed next to the processor and both of them are connected to each other via an internal bus. This memory can be used to store temporary data and its content will be put back to an initial state (i.e. lost) upon a system reset. The processor also needs access to non-volatile memory (referred to as **storage**) to store persistent data such as the code itself. Since we do not want the software to be lost when the device resets, the code will be located in storage. The processor will be directly connected to the storage by a bus (like the memory) so that it can interact with it.

The non-volatile memory is divided into segments, with each having its own fixed read/write/execute configuration. Segments have arbitrary but fixed sizes. A segment could for example be read-only or it could be freely readable and writable with a given granularity. A segment could also be such that it is only possible to write to a given word of memory only once, and a bigger subset (of given granularity) of the segment should programmatically erased before writing to this specific word again. A segment could also only be executable (i.e. it cannot be read or written to, only executed by the processor) or only accessible during the booting process (i.e. it becomes completely inaccessible once the bootloader has finished executing, and only becomes accessible again after a system reset).

All of the previously defined components internal to the microcontroller are then connected to those outside of it by pins located around the microcontroller, effectively allowing indirect communication between both worlds (it is indirect because such communication has to go through pins and is thus limited to an interface exposed by the microcontroller).

External Components

Part of the non-volatile memory can be external. More specifically, the system can optionally include an external storage which is exposed on the board. This storage (unlike the internal one) cannot be directly connected to the processor with a bus. It will however be connected to the pins of the microcontroller with some exposed wires (and thus the processor will have access to the external storage). The external storage may be useful for example if the internal one is not large enough to contain both the bootloader and the image.

Similarly, an external **secure element** can be included in the system to provide some additional security. It will then also be connected to the pins of the microcontroller. The secure element can be used to reinforce the security of a system, by providing more secure hardware which can be used for the most critical parts of the system (such as secret key storage and signature verification). An example of a secure element is illustrated on Figure 3.2.

Peripherals

The board itself can be accessed from the outside via different peripherals. They allow interaction between the system and a user that has physical access to the device. The most relevant example is the debug port. It can be used to interface with the device and have access to the internal state of the processor and even modify it. In particular, this would give arbitrary read and write access to both memory and storage, which is useful to debug the system. As such, this port is also used to program a software into the storage (either internal or external). Therefore, the user itself is part of the system model and can interact with the device using peripherals. The user should typically only need to manually connect to the device once, to install the first version of the image or setup the system. After that, the system should handle updates by itself without interventions of the user. Note that other external elements such as direct memory access (**DMA**) are not represented explicitly in the model since they can implicitly be modeled.

We also consider an update server connected by a network interface as part of the system. The update server is responsible for distributing newer versions of the image. The new software will be sent through the network interface and put in storage so that it can safely be installed on the device. The network interface could be Ethernet, Bluetooth, Wi-Fi, or more generally anything that allows remote communication with the server.

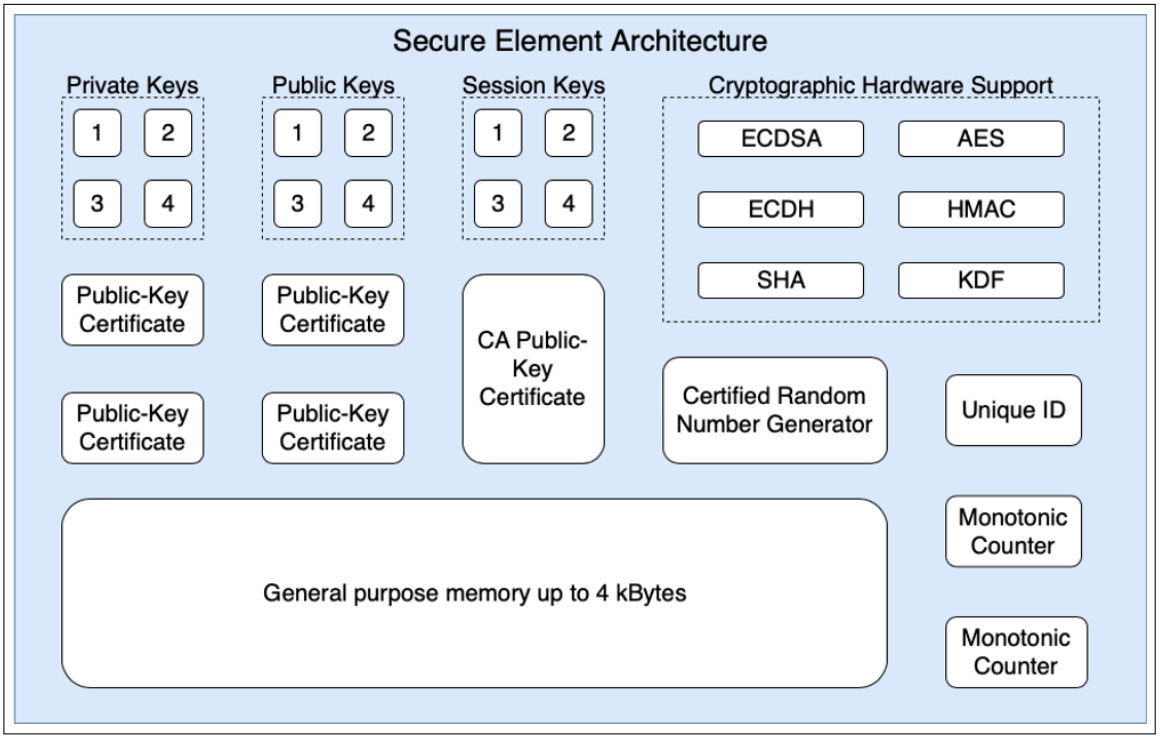

Figure 3.2: An example of a secure element and its functionalities [4].
A secure element (SE) is a chip that is hardened against physical attacks. Its purpose is to provide access to elements required form most security purposes while keeping them in a more secure area. This can include some non-volatile memory, slots to store public and private keys as well as certificates, a unique ID, a random number generator, a monotonic counter and even some specialized cryptographic hardware support.

Software

The software itself is composed of a bootloader and an image. The role of the bootloader is to initialize the system and perform verifications before passing control to the image. On the other side, the image represents the user payload. It can be a simple application or an entire operating system. It is assumed correct in the sense that we do not consider image crashes here. An authentic image will always execute as it was intended to in absence of tampering. The images themselves can simply be modeled by an ordered sequence of programs that listen for updates and always output 1. We call **version** an image with a given sequence number. As such, the first image version has sequence number 1, and the **update** of an version v is $v+1$.

3.2 Assets

To define the assets, we have to consider that we are mainly interested in an embedded system that should be left without user intervention. The System-on-Chip is expected to be setup only

once and should not require physical intervention to be maintained. It could even be left in an inaccessible state (for example, embedded in a device that is hardly reachable), so repairing the state of the system by manually accessing it can be costly or complicated. For this reason, the assets of the system are:

- *The integrity and authenticity* of the image. It is important because the embedded system has to maintain its primary function. A persistent modification or corruption of the image would prevent the device from functioning properly. Thus, we want to ensure that the running software is the original one, or any newer authorized version of it.
- *The execution flow* of the processor. Ensuring the control flow is the intended one is desirable since it guarantees that only authentic software is executed. Modifying the control flow may allow an attacker to take control of the device if they were able to force the device to execute their own code instead of the intended one.
- *The availability* of the device is also important since it has to keep functioning. Persistently corrupting the state of the device in a way that would prevent it from booting again would make it unable to function properly. This could for example happen in a scenario where an update is interrupted while it is rewriting the software.
- *The intellectual property* of the image is another aspect of the system that we want to protect. This part is irrelevant if the image itself is open-source, but it is important if the software is part of the product sold by a company. If an attacker were to extract the code of the image stored on the device (for example by reading it from storage and reverse-engineering it), they could reproduce the system at will without paying for the software. Additionally, keeping the software secret helps in protecting against software attacks, as they are harder to design without knowledge of the software. For those reasons, we consider the intellectual property of the software as an important asset of the system.

3.3 Security Assumptions

In this section, we will describe the security assumptions that we may safely rely on in this project.

Trusted Manufacturers And Components

First of all, we assume that the manufacturers are trusted. That includes any manufacturer that had a role in the production of the hardware on top of which the secure boot is implemented, whether it is the manufacturer of the processor, microcontroller, external storage, board or even secure element. This means that they are honest and will not introduce malicious elements or

backdoors in the system. Similarly, they are competent and thus capable of correctly manufacturing the components. We thus trust that the received hardware complies with the properties advertised by the manufacturers. In particular, the secure element is considered as resistant against any physical attacks it was designed for and can be seen as an impenetrable chip.

Moreover, we omit supply-chain attacks and chip replacements. Therefore, we trust that the device used is the expected one, and no components has been replaced. The device itself cannot be replaced by a malicious version (whether it is during production, shipment or during its usage) and no parts of the device (e.g., chips) can be replaced by a malicious or faulty one either. We also trust that the system was initially setup in a secure environment. More precisely, there is a setup phase during which the device can be initialized, and this phase must be attacker-free. It could for example be used to setup the physical layout of the system (e.g., external non-volatile memory and secure element), or to flash the bootloader itself and configure the microcontroller. Similarly, we trust that the software updates are produced, signed and encrypted securely, which means they are produced in an attacker-free environment. In particular, securing the update generation is out-of-scope for this project. Thus, the upload server cannot upload a malicious version of the software that is signed with an authentic key. That does not mean, however, that the communication channel used to upload it is trusted.

Trusted Chip Boundaries

We also assume that the boundaries of individual chips cannot be directly penetrated by the attacker. In other words, they are not capable of directly interfacing (e.g., by physically connecting a probe) with anything that is contained within the boundaries of a chip. That does not mean, however, that it is completely inaccessible. If a component exposes an interface to the outside of a chip, then it can still be accessed indirectly via this interface. For example,an attacker would not be able to directly read data from the internal storage without going through the connectors of the microcontroller or the processor first. More generally, we will assume that an attacker is limited to non-invasive attacks. The reason is that physical alterations of the system might compromise the boundaries of a chip directly. That means that an attacker is not capable of doing anything that would leave some physical property of the device irreversibly modified. In particular, we do not consider attacks such as decapping a chip, removing some component, invasive side-channel attacks or invasive fault injection attacks in the scope of this work.

Finite Attacker Power

Note that we also consider that the attacker has only a finite amount of time and computational power and knows the algorithms that are used. This means, in particular, that we assume the attacker cannot do brute-force attacks which would require unrealistic computational power and/or would take an unrealistic amount of time to perform. Moreover, the implemented mechanisms and algorithms are assumed to be publicly known. In other words, there shall be not security by obscurity. The software should thus be resistant to attacks even if the attacker is

aware of the design of the system. Note that this is not incompatible with the fact that we may want to keep the software confidential (for the intellectual property, when that is applicable): most of the security will take place at bootloader level, and the bootloader will be open-source in our case. Thus the security cannot rely on the assumption that the confidentiality is kept at all times.

3.4 Two-Phase Model

Definition

An attack will be modeled by a two-phase game. More precisely, we will consider phase 1 and phase 2. Phase 1 is a preparatory phase. During this stage, the attacker has access to all of their capabilities and can use them to prepare the state of the system for the next step or extract information about the system. The system will then transition to phase 2. During this phase, the attacker now has more limited capabilities and tries to achieve their goal. The two-stage model was introduced for two reasons. The first one is that we wanted to give an adversary a maximal degree of abilities in order to capture many relevant real-world attacks while keeping the model as simple as possible. The second reason is that powerful abilities during a unique phase allowed for trivial attacks, as we will see here.

Phase 1 represents the phase where the attacker has access all of their capabilities (as we will define in *Section 3.7*) and tries to prepare the system or gain knowledge of the system for the next phase. This corresponds to a stage where the attacker has a temporary unrestricted access to the system, for example an exploratory phase. The CPU might be influenced (for example by manipulating non-volatile memory) and it may even be executing attacker-controlled code during this phase. The attacker can spend as much time as they want in phase 1 and can freely chose when it ends. The attacker will typically try to make persistent changes to the system to influence the result of phase 2. When phase 1 is over, the system will keep its current state for phase 2 (although the system might go through some additional steps in-between depending on the attacker model). Then, during phase 2, the attacker will have limited capabilities and they will try to complete their attack. Again, phase 2 will end when the attacker decides it. This phase corresponds to the core of the attack, as the state of the system at the end of phase 2 will determine the result of the attack. As such, phase 2 can be seen as a challenge phase. Indeed, the system will decide whether the attacker's goals is achieved after phase 2 has ended.

Choice Of A Two-Phase Model

With a single-phase model, an attacker would keep the same capabilities throughout the attack, which means the choice of capabilities has to be a balance between being as powerful as possible while still being restricted enough to be realistically defendable. Unfortunately, even

if we only consider a jailbreak attacker, we cannot reach a satisfying solution. Take the case of a write to internal memory during a jailbreak attack. If an attacker can modify the content of internal memory during execution, a jailbreak attack is trivial since an attacker could modify instructions right before they are executed, so the model would be meaningless. On the other hand, if the attacker cannot ever write to internal memory, most attacks based on modifying the content of the image and letting the bootloader execute it would not be captured by this model. It would be equivalent to assuming that the image cannot ever be directly modified by an attacker, which leads to a jailbreak model that is too weak.

The solution we chose is to give more flexibility to the model by defining two phases, each one with specific capabilities. This way, we have a more natural way to represent the attacks in our model. To continue the previous example, the jailbreak attacker could now write to internal memory during phase 1 (so we cannot assume that the image is authentic when the board boots) but they are still limited in a way they cannot trivially execute a jailbreak attack during phase 2 (which is when they need to execute attacker-controlled code to succeed their attack). The choice of the two-phase model fits particularly well the binary nature of the software. Indeed, the software is composed of a bootloader and a user image. Any non-persistent change made to the system or temporary control gained (i.e., in phase 1) is lost after a system reset. Thus, the bootloader then regains control of the system, and the attacker might loose some capabilities (which corresponds to phase 2). In practice, that means that the attacker do not have access to capabilities that would prevent the system from defending their attacks (e.g., via trivial victories). Some examples of how real-world attack are translated into this model can be found in *Section 3.7*.

3.5 Attacker Goals

Let us now define the goals of the three different attackers.

Jailbreak Attacker

The goal of the jailbreak attacker is to execute an attacker-controlled image after phase 3. To model an attacker-controlled execution, we consider an image which outputs 0 (as opposed to a genuine image, which always outputs 1). As such, we can see the goal of the attacker as modifying the image so that it outputs 0 after phase 2. We distinguish three possible outcomes of phase 2 (and thus of the attack game), depending on the actions made by the attacker:

- *The bootloader executes an authorized image.* The image outputs 1. This means that the attacker was unable to affect the execution flow and the system safely executed an authorized image. This is the ideal scenario and marks a failure of the attacker.
- *The bootloader executes an attacker-controlled image.* The image outputs 0. Reaching this

state is the only goal of the attacker, as it means its attack was successful. In this case, we consider that the security of the bootloader was breached and the attack is successful.

- *The bootloader refuses to execute the image and hangs.* The bootloader detected that the integrity of the image was compromised and refuses to boot in order to avoid a security breach. This scenario marks a failure of the attacker as the security was not compromised.

Note that we do not consider arbitrary code execution by itself as a victory for the attacker. Whenever the image is executed, a software vulnerability may allow an attacker to modify the execution flow while the system is running. This is entirely dependent on the image and that is not what we are trying to prevent by adding a secure boot mechanism. What we want is to prevent an attacker from keeping this control even after a system reset. To maintain control, they would have to persist changes to the non-volatile memory where the image and the bootloader are stored, and that is what we want to detect. Preventing a modified image from booting allows us to ensure that an attacker would have to repeat an attack at each reboot of the system. Hence, we are interested in preventing persistent tampering of the image rather than temporary control flow hijack from within the image.

DoS Attacker

The goal of the DoS attacker is to put the system in an invalid state in phase 2. The system must be unable to boot to a genuine image even after having reset the system. Note that only booting to an attacker-controlled image also marks a successful attack, as we might no be able to revert the system to an authentic image. More precisely, we distinguish three different outcomes for phase 2:

- *The bootloader successfully boots to an authentic image.* This means that the system can still safely function. This marks a failure for the attacker.
- *The bootloader successfully boots to an attacker-controlled image.* In that case, the main purpose of the image could be compromised and it might not function correctly, or function at all. This marks a success for the attacker, as the system may be permanently stuck with a non-authentic image.
- *The bootloader refuses to execute the image and hangs.* The bootloader detected that the integrity of the image was compromised and refuses to boot in order to avoid a security breach. This means that the system is permanently stuck in this state, and the attacker achieved their goal.

IP Thief

Since the IP thief aims at gathering information about the content of a specific image, we model their goal as trying to recognize a genuine signed and encrypted image from random data that is also signed and encrypted. Indeed, distinguishing them would mean that the attacker was capable of extracting some information about the image. Such information can be gathered by the attacker during phase 2, or they could reuse information obtained during phase 1. In this scenario, we distinguish 2 outcomes:

- *The attacker can recognize the image better than by guessing randomly.* This means that the attacker successfully violated the intellectual property of the image, and their attack is considered as a success.
- *The attacker cannot recognize the image better than by guessing randomly.* This means that the attacker could not violate the intellectual property of the image and their attack is a failure.

3.6 Attack Surface

Now that the assumptions regarding the trusted parts of the system have been stated, we will describe in this section the different attack surfaces, which are the elements we cannot entirely trust since they might be entry points of an attack (and manipulated by an attacker as such).

First of all, anything that is not contained inside the boundaries of a chip is considered as **exposed**. More specifically, such components are considered as exposed on the device to an attacker and could be manipulated by them. We will define attacker capabilities more precisely in *Section 3.7*, but it essentially means that any medium by which information might that travels outside of a chip is potentially part of the attack surface. This includes any peripheral or connector on the board, but also every pin that connects a chip to its neighborhood and the potential exposed wires. We assume that any of these might be connected to a malicious device in order to compromise the system, depending on the attacker. A direct consequence of this is that any information that exits or enters a chip (including the microcontroller) is part of the attack surface as well, since it travels on an untrusted medium.

Additionally, the image might be an entry point for an attacker, and as such is considered as part of the attack surface. Even if the image is authentic, we cannot assume it is free of vulnerabilities. It might still contain some software vulnerabilities that could be used by an attacker to gain control of the device. As such, we may not trust that the execution of the image will not harm the state of the software. Similarly, the content of the non-volatile memory is also part of the attack surface. Modifying the content of the storage is a way to modify the software itself. That may result in execution of modified code upon restart. In the case of an external non-volatile memory, it could be directly attacked since its connectors are exposed to an attacker. Even if the storage in internal to the microcontroller, it could be indirectly attacked, for example

via a software vulnerability in the image, or using another component such as a DMA. For those reasons, we may not trust that the storage is in the state it should be, it might have been modified.

3.7 Attacker Model

Let us now describe in more details this threat model.

3.7.1 Attack Structure

In this section, we will describe precisely the different capabilities of the attacker. More specifically, we will show in which ways the attackers can attack the system during phases 1 and 2. The capabilities will be represented by actions which will be freely usable by the attacker during corresponding phase. *Actions 1* summarizes all capabilities considered in this work. Note that an attacker might not have access to all functionalities. Following sections will define which attacker has access to which capabilities during a single phase.

Capabilities described in the following sections might not be useful or relevant for some attackers in a given phase, but we prefer giving too many capabilities, as opposed to too few. As mentioned in *Section 3.4*, the choice of a two-phase model was motivated by the fact that we want to cover as many attacks as possible. This way, the model might be stronger than necessary and might fit attacks that were not initially anticipated, rather than being too restrictive and only fitting a sample of attacks.

All Capabilities

General

First of all, the attacker can control the precise time when they perform actions. We model that in the following way: the CPU will continually be executing instructions and the attacker can perform any number of actions in-between any two consecutive CPU instructions. This way, the attacker can freely interleave processor execution with other capabilities. Then, phases 1 and 2 end when the attacker decides it. To end a phase, the attacker needs to decide that they are done and stop interfering with the system, which will allows the attack game to carry on.

Additionally, the attacker is capable (at any point during both phases) of restoring the initial system state. This can be done with **factoryReset**(), which will restore the bootloader as well as the first genuine image in internal storage and trigger a system reset. Furthermore, it will erase the content of all non-volatile and volatile memory before doing so. This can effectively

Actions 1: All capabilities considered in this work

General

- **factoryReset**(): Restores the initial valid state of the system. Clears non-volatile memory content, restores the bootloader, generates a new authentic image and installs it, then triggers a system reset.
- **getUpdate**(): Increase current version number v to $v+1$ and returns image version $v+1$, as sent by the update server.
- **installImage**(*image*): If no update is ongoing, installs *image* in secondary slot, marks it for update and reset the system. If an update is ongoing, does nothing.

Arbitrary R/W to internal memory

- **readInternalNVM**(*address*): Returns the content of the internal non-volatile memory at address *address*. Can return $\perp$ if the configuration of the corresponding region disallows it.
- **writeInternalNVM**(*address, data*): Writes *data* at address *address* of internal non-volatile memory. Can return $\perp$ if the configuration of the corresponding region disallows it.
- **readRAM**(*address*): Returns the content of the internal RAM at address *address*. Can return $\perp$ if the configuration of the corresponding region disallows it.
- **writeRAM**(*address, data*): Writes *data* at address *address* of RAM. Can return $\perp$ if the configuration of the corresponding region disallows it.

Arbitrary R/W to external memory

- **readExternalNVM**(*address*): Returns the content of the external non-volatile memory at address *address*.
- **writeExternalNVM**(*address, data*): Writes *data* at address *address* of external non-volatile memory.

System reset

- **systemReset**(): Forces a system reset. Volatile memory and CPU state are reset to default values.

Communication attacks

- **readWirePCB**(*wire*): Reads the state of exposed wire *wire* and returns its value. Includes read capability to remote communications since they are received through an exposed interface.
- **writeWirePCB**(*wire, data*): Writes *data* on exposed wire *wire*. Includes write capability to remote communications since they are sent through an exposed interface.

restore the state of the system to its initial genuine state. This can for example be useful if an attack needs to corrupt the image at some point, but still be able to restore it afterwards. The attacker can also request for a genuine new software version by using **getUpdate**(). This action will request a new image to the update server and show its content (as it would be received by a genuine update process, so the image may still be signed and even encrypted). The system keeps track of the version number, so if the last requested version is v, using this action will provide version $v + 1$. To install another image, the attacker can then use **installImage**(*image*). This function will initiate the update process to install *image*, but only if no update is already ongoing. The bootloader might still try to verify the image before installing it, as with every new image.

Arbitrary R/W To Internal Memory

An attacker that can arbitrarily read internal non-volatile (resp. volatile) memory can do it using the two corresponding actions. **readInternalNVM**(*address*) and **readRAM**(*address*) can be used to read the content of internal memory (both volatile and non-volatile). Those actions are subject to the underlying memory restrictions, meaning that the read can fail if the corresponding memory region is execute-only or requires more privileges. Similarly, actions **writeInternalNVM**(*address, data*) and **writeRAM**(*address, data*) can be used to modify the content of both non-volatile and volatile internal memory. Again, the write access can be refused depending on the memory configuration.

Arbitrary R/W To External Memory

An attacker that has also access to external non-volatile memory can read and modify its content using **readExternalNVM**(*address*) and **writeExternalNVM**(*address, data*). Once more, the read or write requests can be refused by the system based on the memory region configuration.

System Reset

An attacker capable of triggering a system reset can do it by using **systemReset**(). This will force a system reset, which will reinitialize the content of volatile memory as well as the state of the CPU.

Communication Attacks

The ability to read an exposed wire's value is indicated by the presence of a **readWirePCB**(*wire*) action. If available, an attacker can use it to read the value of a wire (or connector) which is exposed on the board. This can represent the fact that the attacker can connect directly to any exposed interface (for example using lab equipment to snoop on the pcb, or with a malicious device). Since the network interface (through which all remote communication go) is part

of the board's connectors, this action allows the attacker to read any remote communication as well. Similarly, **writeWirePCB**(*wire, data*) can be used to overwrite the value of any exposed wire or interface of the board. This also allows to interact with the pins of the MCU, as writing values to pins might request actions to the MCU if the corresponding features are enabled. Altogether, the attacker can read individual packets (or bits) as they are transmitted through wires, as well as arbitrarily modifying them and adding or removing packets. That means that the attacker is capable of performing many communications attacks, including dropping packets, modifying their payloads, sniffing, or even more involved attacks such as performing a man-in-the-middle or a replay attack. Practically, since the attacker can control communications, it is equivalent to being capable of plugging malicious devices in any available connector, to initiate a communication. That includes, for example, plugging a malicious external storage module, or connecting via the debug port to try and reprogram the software directly.

Phase 1

During phase 1, all attackers correspond to non-invasive physical attackers which also have some control over the CPU. For example, they can control internal and external memory (both volatile and non-volatile), unless a hardware protection mechanism prevents it. They can also control any communication and trigger system resets. Figure 3.3 illustrates all attackers during phase 1. All attackers possess capabilities that correspond to those of a physical non-invasive attacker which also has access to a vulnerability in the image. Full attacker capabilities are shown in *Actions 2*.

Phase 2

Let us now describe shortly the different attackers during phase 2. Complete descriptions can be found in the respective attacker's chapter (i.e., *Sections 3.7.2, 3.7.3* and *3.7.4*)

Formally, the jailbreak attacker is a non-invasive physical attacker during phase 2. This means they have the ability to arbitrarily read and write to mutable segments of the external non-volatile memory, as well as a complete control over exposed communications (which include communication between the microcontroller and the other components). Figure 3.4 illustrates the jailbreak attacker during phase 2.

Similarly, the DoS attacker can be seen as a more limited non-invasive physical attacker during phase 2. They can request to install an arbitrary update and reset the system at will to permanently corrupt the state of the system. Figure 3.5 illustrates the DoS attacker during phase 2.

Finally, the IP thief is also a physical non-invasive attacker during phase 2. They are capable of manipulating exposed communications as well as the content of the external storage, and

Actions 2: Attacker capabilities during phase 1

General

- **factoryReset**()
- **getUpdate**()
- **installImage**(*image*)

Arbitrary R/W to internal memory

- **readInternalNVM**(*address*)
- **writeInternalNVM**(*address, data*)
- **readRAM**(*address*)
- **writeRAM**(*address, data*)

Arbitrary R/W to external memory

- **readExternalNVM**(*address*)
- **writeExternalNVM**(*address, data*)

System reset

- **systemReset**()

Communication attacks

- **readWirePCB**(*wire*)
- **writeWirePCB**(*wire, data*)

Capabilities are explained in *Section 3.7.1*

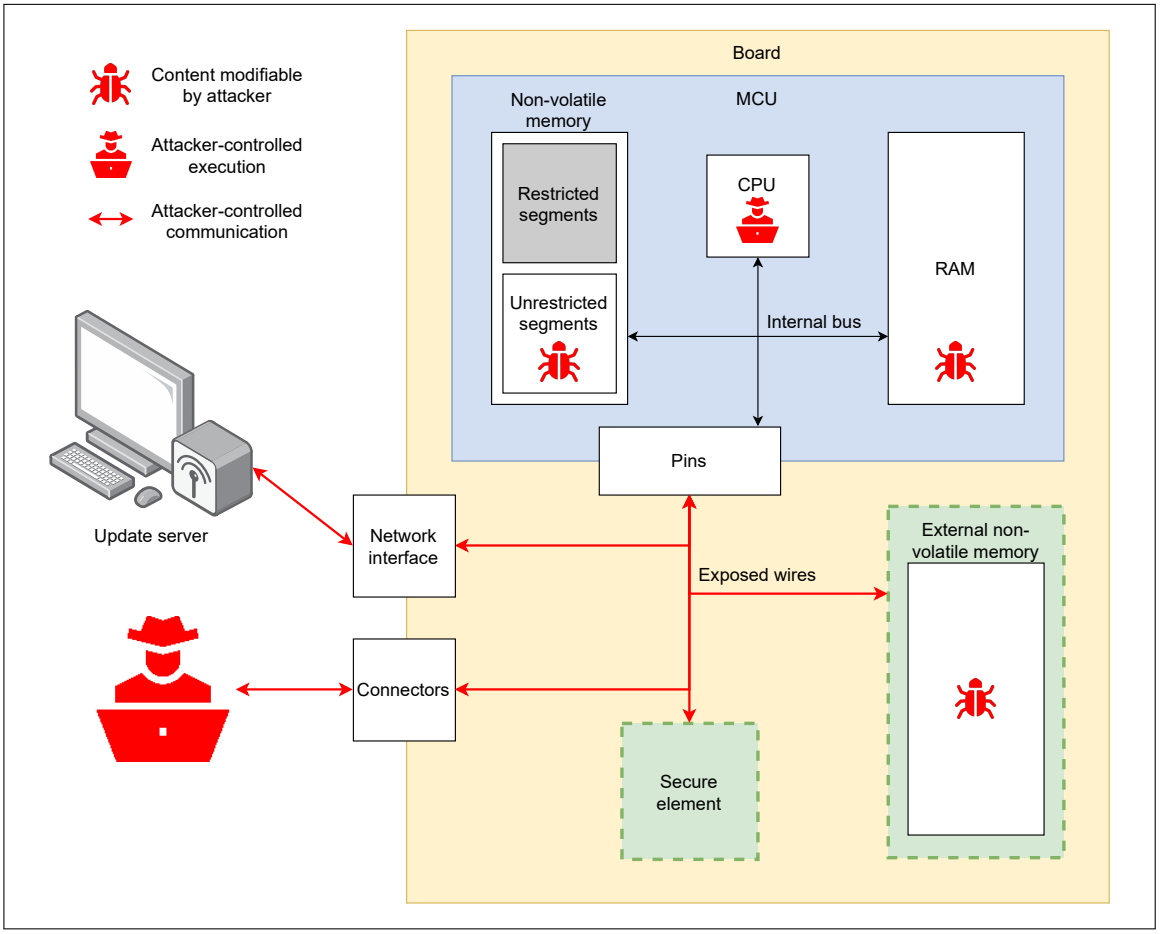

Figure 3.3: All attackers during phase 1

During phase 1, all attackers can fully manipulate exposed communications and connect malicious peripherals. Content of non-restricted storage (both internal and external) might have been replaced by attacker-controlled data. The CPU is under the attacker's full control, they can thus interact with internal storage and memory and indirectly execute arbitrary code.

triggering system resets. Their goal is to distinguish an image from random data during the phase 2, to prove their knowledge of the image. Figure 3.6 illustrates the IP thief during phase 2.

Attack Game

The attack game represents the entirety of an attack on the system. It includes the preparation of the system, both attack phases as well as the final outcome of the attack. It is defined on a per-attacker basis and will thus be presented in *Sections 3.7.2, 3.7.3* and *3.7.4* similarly to phase 2. We shall still define here some procedures that will be used to describe the different attack games.

Before preparing the state of the system, cryptographic keys and scheme will be sampled with

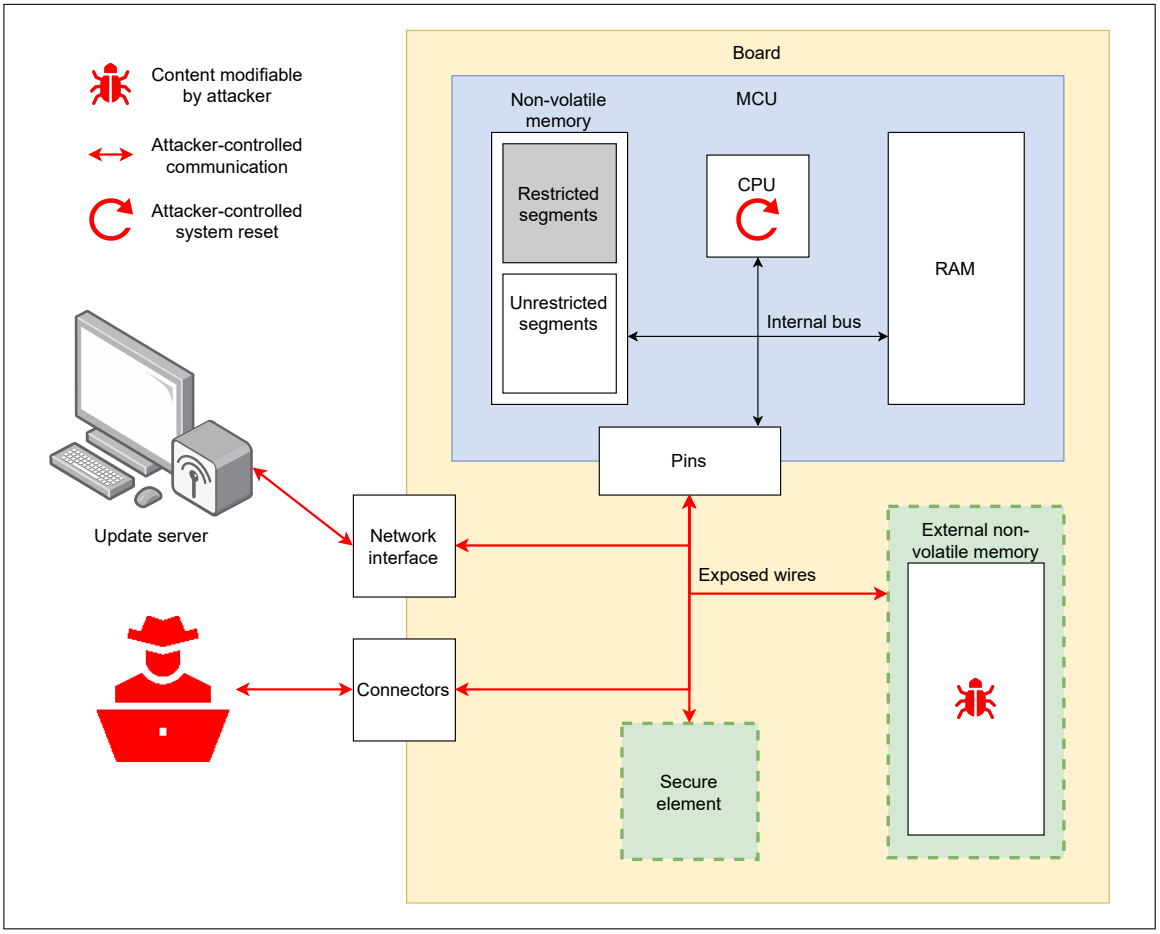

Figure 3.4: Jailbreak attacker during phase 2
The jailbreak attacker can fully manipulate exposed communications and connect malicious peripherals. Content of non-restricted external storage might have been replaced by attacker-controlled data.

SAMPLE_CRYPTO(). The exact cryptographic scheme can vary depending of the design of the secure boot, but this procedure should always create and return the cryptographic information (such as public and private keys) necessary for the design. For example, if signature verification is performed using asymmetric cryptography, a pair of public/private keys used for signature verification should be provided.

Then, to generate a valid image from a given software, CREATE_IMAGE(`version`, `software`, `keys`) should be used. This function must also be defined on a per-design basis, since it should be compatible with the system. It takes three parameters: `version` is the version of the image that will be created, `software` is the code that we want to encapsulate in an image, and `keys` contains the cryptographic information necessary to generate the image. In particular, the initial image will have its code defined by `base_I`. This corresponds to the code of a basic genuine image which has all necessary functionalities (depending on the needs of the design) and always returns 1. To initialize the state of the system, INIT_BOARD(`image`, `keys`, `bootloader`, `config`) will

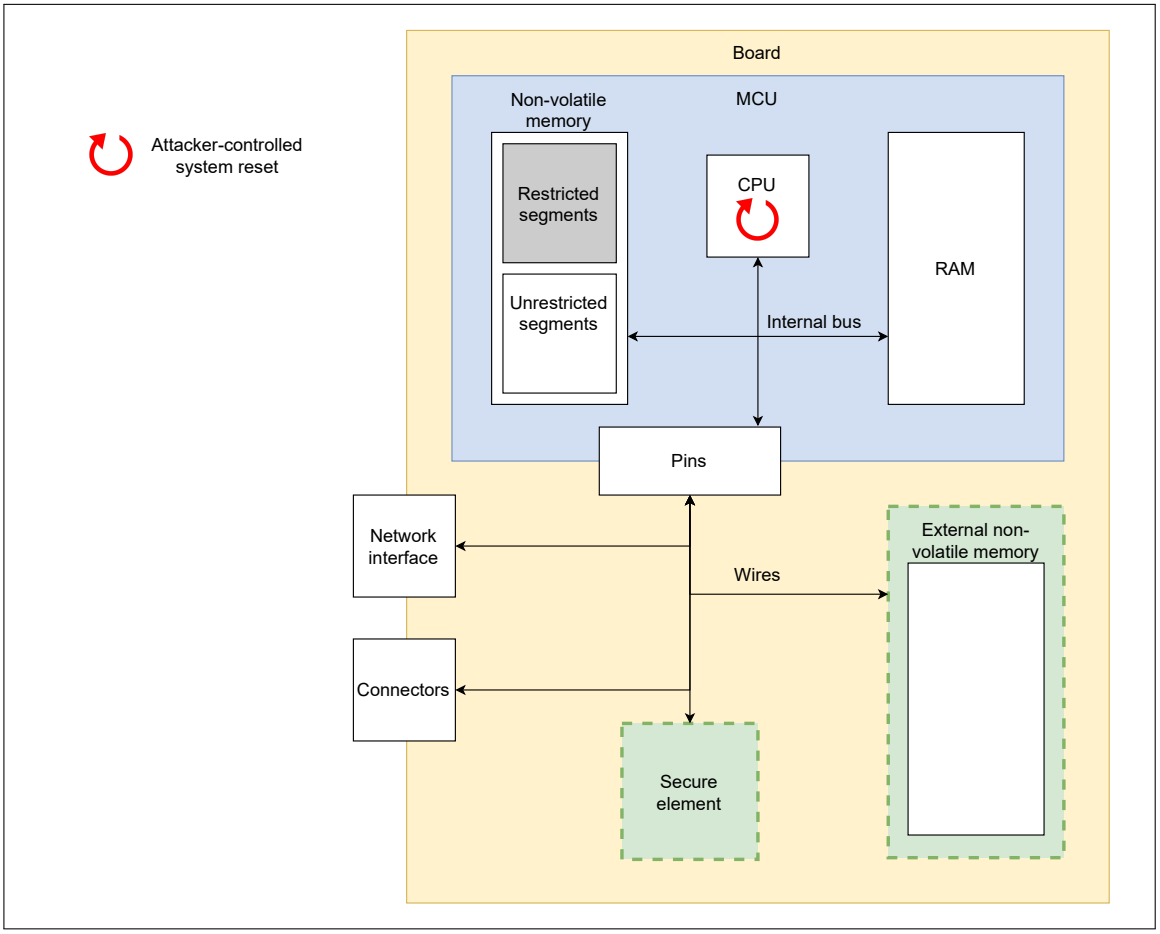

Figure 3.5: DoS attacker during phase 2
The DoS attacker can only trigger system resets at will and request update installation during the phase 2.

then be used. This procedure will install both `bootloader` and `image` on the board. The board will also be configured as required by the design, depending on `config`. If necessary, current cryptographic information (`keys`) will also be used. Again, this is dependent on the bootloader design. *Algorithm 4* summarizes the update request protocol.

Algorithm 4 Update request protocol

```
1: procedure GETUPDATE()
2:     v ← v + 1
3:     I ← CREATE_IMAGE(v, base_I, keys)
4:     return I
```

Finally, after phase 2, the system will have to decide whether the attack is successful using FINALIZE(). The exact behavior will depend on the attack model, but it will examine the final state of the system and determine is the security was compromised. The output of FINALIZE() is 1 if the attack is a success and 0 otherwise. We will define this procedure more precisely on a

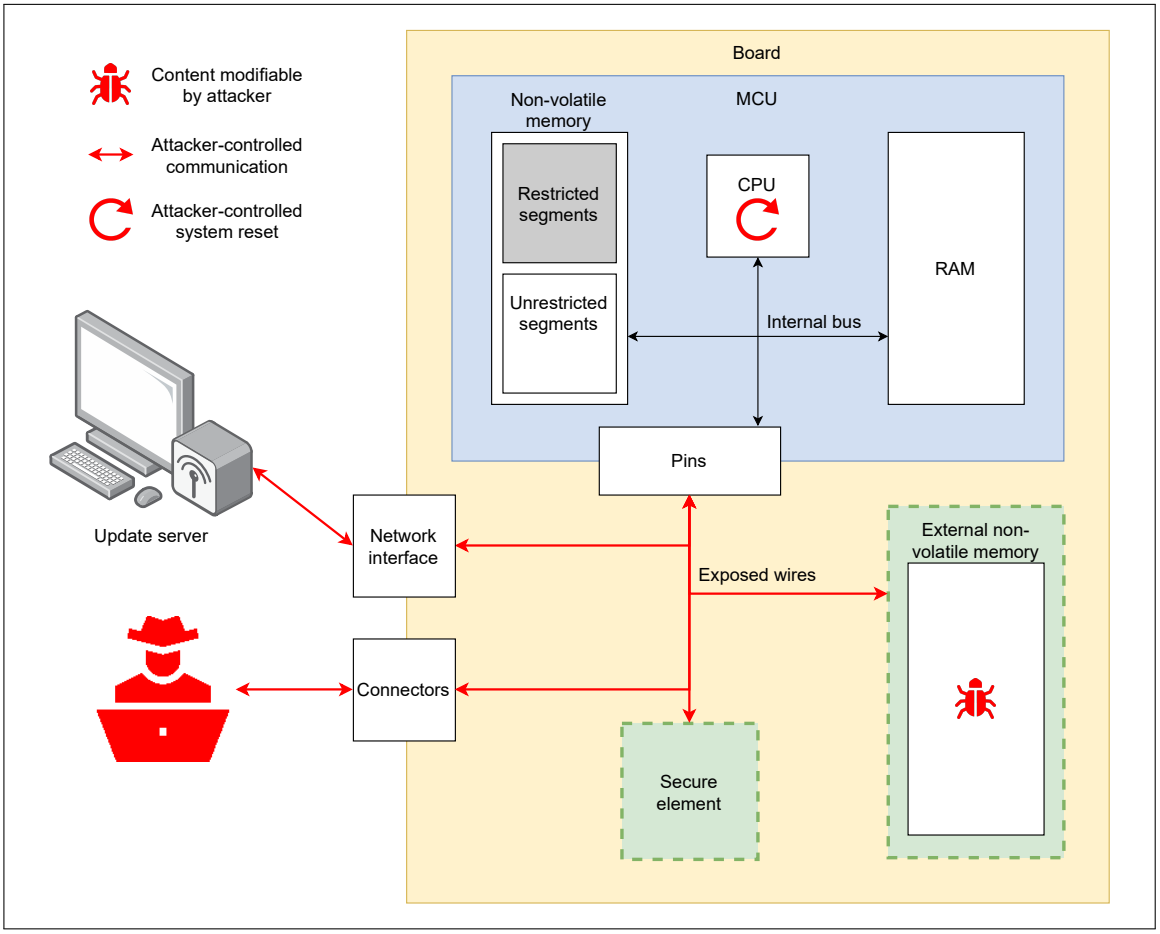

Figure 3.6: IP thief during phase 2
During phase 2, the IP thief has control of every external part of the system to distinguish a valid image from random data.

per-attack basis.

3.7.2 Jailbreak Attacker

Let us now list the jailbreak attacker's capabilities during phase 2 as well as the attack game. During phase 2, the jailbreak attacker becomes a physical non-invasive attacker but who lost access to internal memory. This can represent the fact that the system was reset, which means that the immutable Bootloader is executing again and any temporary privilege gained by exploiting vulnerabilities in the image is lost. *Actions 3* shows attacker capabilities.

Phase 2

The reason the attacker lost control of internal memory is that we want to exclude trivial

Actions 3: Jailbreak attacker capabilities during phase 2

General

- **factoryReset**()
- **getUpdate**()
- **installImage**(*image*)

Arbitrary R/W to external memory

- **readExternalNVM**(*address*)
- **writeExternalNVM**(*address, data*)

System reset

- **systemReset**()

Communication attacks

- **readWirePCB**(*wire*)
- **writeWirePCB**(*wire, data*)

Notice the attacker only lost access to internal memory compared to phase 1. Capabilities are explained in *Section 3.7.1*

jailbreak attacks. If the attacker were able to modify internal memory during the phase 2, they could simply write their own image right before the bootloader jumps to the authentic image (since this one has been verified). Thus, we need to restrict the jailbreak attacker so that they cannot arbitrarily read or write to internal memory during phase 2.

Attack Game

We will describe here the attack game of a jailbreak. *Algorithm 5* summarizes the attack procedure. First, the system in initialized and put in a valid state as described earlier. More precisely, the image version v is set to 1, the cryptographic keys are sampled, a first image is created and the board is initialized using the image, the keys and the bootloader. At this point, the system is fully functional and attacker-free. Then comes phase 1, during which the attacker has arbitrary access to capabilities described in *Section 3.7.1*. The attacker may keep information σ to use them during phase 2. When the attacker has completed their phase 1, a system reset is done to initiate phase 2 during which the attacker has now reduced capabilities (as shown in *Section 3.7.2*). Finally, when phase 2 is complete, the attacker becomes passive and the system decides the outcome of the attack using FINALIZE(). The system may already have decided whether to boot from an image during phase 2, but it can continue execution until its choice if it is not the case.

Algorithm 5 Jailbreak attack game

```
1: procedure JAILBREAK_ATTACK()
2:     // Initialization
3:     v ← 1
4:     keys ← SAMPLE_CRYPTO_KEYS()
5:     I ← CREATE_IMAGE(v, base_I, keys)
6:     INIT_BOARD(I, keys, bootloader, config)
7:     // Phase 1
8:     σ ← A^{phase_1}()
9:     RESET_SYSTEM()
10:    // Phase 2
11:    A^{phase_2}(σ)
12:    // Result
13:    b ← FINALIZE()
14:    return b
```

As described in *Section 3.5*, the attack is then successful only if an attacker-controlled image was executed. In our model, this is represented by the fact that an image might output 0 if it was modified. Thus, FINALIZE() will examine the state of the system and determine if the installed image is genuine and declare that the attack is a success if the image is executed despite the fact that it is not authentic. In this case, FINALIZE() will return 1. Otherwise, the attack is a failure and it will return 0.

Attack Examples

Let us now illustrate how some concrete jailbreak attacks can be translated to this model.

Imagine a first scenario where the user of the device itself wants to jailbreak the device and install its own software. The user has physical access to the device. Imagine that the currently installed (authentic) software has a flaw in its design and allows a user to arbitrarily read or write to memory (even internal). The user can thus try to install its software directly to the primary slot. In the attacker model, this corresponds to phase 1; the attacker has (in particular) arbitrary R/W access to internal memory (characterized by the access to **readInternalNVM**() and **writeInternalNVM**()). Control may temporarily be gained depending on the model, but the user wants to persistently modify the software, so let us consider the moment the device resets. Now, the bootloader will start executing and the attacker temporarily lost control; this corresponds to a phase 2 where the attacker lost R/W access to internal memory. The bootloader now needs to decide whether to boot from the modified image, determining whether the attack is successful. The same attack model could also be applied if the attacker were remote and exploited a vulnerability in the image that gives arbitrary R/W rights to internal memory. The attacker would then try to install a backdoor in the software to gain a persistent access.

Similar attacks can be imagined, but with different ways to install a malicious image. For

example, we could imagine an external attacker that has no physical access to the device, but controls the internet router. In this scenario, communications with the update server are over the internet, so the attacker can effectively control remote communications. In the attacker model, that is translated by an attacker in phase 1 that has arbitrary R/W access to the PCB wires responsible of the remote communication (characterized by the partial access to **writeWirePCB**() and **readWirePCB**()). The attacker can now wait for an update to be downloaded, intercept the packets containing the new software and replace them by attacker-controlled content. The device might now try to install an invalid image (which corresponds to phase 2), depending on the system.

A more involved example can be thought of. This time, an attacker with physical access to the device tries to take advantage of the update process to install its own image, using a time of check / time of use (TOCTOU) attack. The attacker is connected to the device and thus has arbitrary access to internal memory, and can also reset the system at will. Additionally, the attacker has knowledge of a valid update file. This can be represented by the access to **readInternalNVM**(), **writeInternalNVM**(), **systemReset**() and **getUpdate**(). During phase 1, the attacker could write its own image to the secondary slot and manually request an update by doing so. They will then reset the system, to start the update process. This moment corresponds to the transition to phase 2, in a scenario where the attacker capabilities are left unchanged. The attacker will wait for the system to validate the update. Before the system applies the validated update, the attacker will now write its own image to the secondary slot. At this point, the bootloader might still apply the attacker image and try to execute it, depending on the design.

3.7.3 DoS Attacker

Phase 2

During phase 2, the Denial-of-Service attacker can only trigger system resets, request updates and restore the system to its initial state (as shown in *Actions 4*). The objective is to reset the system at carefully chosen steps to permanently corrupt the state of the system.

Firstly, the attacker does not have access to internal memory to exclude a trivial attack. Indeed, the attacker could simply erase the content of all (internal + external) memory and the bootloader would be unable to find any valid image. Similarly, any way to arbitrarily influence external memory or communications during an update could lead to an easy system corruption. For this reason, the attacker has no direct control over memory (both internal and external) and communications.

The main attacker's capabilities is the ability to request updates (only while no other update is ongoing) and arbitrarily reset the system. The attacker can however still use **factoryReset**() as this will trigger a system reset after having restored the first version of the image. Even if it were

Actions 4: DoS attacker capabilities during phase 2

General

- **factoryReset**()
- **getUpdate**()
- **installImage**(*image*)

System reset

- **systemReset**()

Capabilities are explained in *Section 3.7.1*

called during an update, restoring the image and triggering a system reset effectively cancels the ongoing update and puts the system back in a valid state before giving the control back to the attacker (even if the content of the ongoing update may be lost).

Attack Game

Algorithm 6 summarizes the attack procedure of a DoS. The procedure is very similar to the jailbreak attack procedure (*Algorithm 5*) with a key difference we will explain here. This time, a full factory reset is done between phases 1 and 2. This is caused by the fact that the attacker may already lock the system during their exploratory steps (in phase 1) but we want phase 2 to start in a bootable state, as the attacker's victory would be trivial otherwise. The result of the attack is a success only if the bootloader decides to hang (as described in *Section 3.5*). Thus, FINALIZE() will check whether the system is unable to boot, which indicates a successful attack.

Algorithm 6 DoS attack game

```
1:  procedure DOS_ATTACK()
2:      // Initialization
3:      v ← 1
4:      keys ← SAMPLE_CRYPTO_KEYS()
5:      I ← CREATE_IMAGE(v, base_I, keys)
6:      INIT_BOARD(I, keys, bootloader, config)
7:      // Phase 1
8:      σ ← A^phase_1()
9:      FACTORY_RESET()
10:     // Phase2
11:     A^phase_2(σ)
12:     // Result
13:     b ← FINALIZE()
14:     return b
```

Attack Examples

Let us now illustrate how some concrete DoS attacks can be translated to this model.

We can imagine a remote attacker that is connected to the same network as the device. They cannot fully control the communication but can at least scramble the communication emitted or received by the device. Their goal is to brick the device by trying to corrupt the update image. Since phase 1 is an exploratory phase and is concluded by a factory reset, phase 1 would be empty in this case. The attacker will wait for an update to be downloaded. When an update is being sent over the network, the attacker will scramble the data that is sent so that the device receives corrupted or random data. In our model, this would be represented by an attacker in phase 2 that can request to install a particular image, which would be a scrambled version of a valid update. It is thus represented by the limited access to **getUpdate**() and **installImage**(). We would thus consider equivalent to a case where the attacker is only able to request a valid image, scramble its content and request it to be installed. The device will thus receive an invalid update and will decide whether to install and execute it.

Another scenario considers an attacker that has a temporary and limited access to the device. They only control the power source of the device and can arbitrarily turn it off or on. They manage to be present during a system update and try to brick the device by interrupting the update process during specific steps. In our model, this would again be represented by an empty phase 1. The attack would thus take place during phase 2. We can formally consider that turning a device off and then on again is equivalent, from the point of view of the device, to a system reset. We can thus represent the attack with the ability to get a single valid image, install it and arbitrarily reset the system. This is characterized by the access to **getUpdate**(), **installImage**() and **systemReset**(). The attacker will then arbitrarily try to turn the power source of the device off and on multiple times. Depending on the design, the state of the device might be corrupted.

3.7.4 IP Thief

Let us finally list the IP thief's capabilities.

Phase 2

During phase 2, the IP thief has only lost the access to internal memory. The reason is that it would allow a trivial victory, as the attacker could read the content of images in internal while they are not encrypted. *Actions 5* shows all of their capabilities.

Choice Of An Attack Game

The attack procedure for the IP theft is more complex than for other attackers. There is

Actions 5: IP thief capabilities during phase 2

General

- **factoryReset**()
- **getUpdate**()
- **installImage**(*image*)

Arbitrary R/W to external memory

- **readExternalNVM**(*address*)
- **writeExternalNVM**(*address, data*)

System reset

- **systemReset**()

Communication attacks

- **readWirePCB**(*wire*)
- **writeWirePCB**(*wire, data*)

Capabilities are explained in *Section 3.7.1*

additional preparation necessary to make sure the game is suitable for our model. Let us explain the choices made in more detail first. The model we want is similar to a simple ciphertext attack, that is that the attacker should be presented with a ciphertext that can contain either known data or random data, and they should prove their knowledge by distinguishing both cases. In our case, the entire image (encrypted and signed) represents the ciphertext.

A first simple solution would be to either give the attacker an encrypted image, or random data. This way, the ciphertext would contain either know value, or random data, as wanted. The problem with this approach comes from a key difference between our model and a simple ciphertext attack: in our case, an image should have a valid *behavior*. More precisely, it should be possible to install and execute the image with the bootloader. However, if the image is completely random, its signature will not be valid, and the attack could simply distinguish them by observing whether the image is recognized or refused.

A better alternative would be to randomly pick data, and generate a valid signature with it. The resulting image would then be a valid image whose underlying payload is random. This way, this random image would be accepted and installed by the bootloader. Nonetheless, the content of the image is still random and trying to execute its code will most likely result in crashes, which the adversary can observe. Again, a trivial attacker victory would be to install the received image and observe whether it boots successfully or crashes. It follows that the random image should not only have a valid signature, but should also behave like a valid image. Otherwise, the model

cannot fit our needs as it will automatically include a trivial attacker victory.

Another possible solution embeds a random value inside a valid image. The idea is to have a valid image with a flag (e.g., a variable) that is either a know one, or chosen randomly. The resulting image would then be signed and encrypted as usual. This way, both images behave the same way and will be verified by the bootloader; their functionalities are the same and they only differ by the value of the flag they contain. This approach is a starting solution, but the problem is that it is not general enough. It considers as secure any design that only encrypts the flag and leaves everything else in clear, even though that is clearly undesirable in practice. For this reason, we cannot use a random value embedded in the code of a valid image. Even if we randomize every part that is not functional (e.g., everything except the code of an image), the functional part of the image would not be covered, and so a design that encrypt everything except the functional part would be considered as secure by the model, even if that is still not desirable.

Therefore, we need to have an image whose payload is entirely random, and we need to handle the fact that it is unable to execute correctly. If the entirety of the payload is either random or valid, then the model will successfully cover information leak in every part of the image without making assumptions on its structure. The key idea is thus to take an image with a random payload, but modify the system to *simulate* a valid behavior. More precisely, the system should be modified to recognize the random image given to the attacker, and whenever this image is installed, the system should covertly act as if a valid image were installed instead. We will see this in more details after the attack game has been explained.

Attack Game

Let us now cover the attack game of an IP thief, as shown in *Algorithm 7*. The beginning of the game is the same as with the jailbreak and DoS attackers. The state of the system is initialized, and phase 1 takes place.

After phase 1, the system will generate a random bit `b` to choose a challenge input image `I`, which is known by the system but not the attacker. If `b` is 0, the content of `I` will simply be the next valid image, and the system will behave as usual. On the other hand, if `b` is 1, random data will be generated and used as the content of `I`. The system will needto perform additional steps during updates, to prevent trivial attacker victory, as we will explain in this section. Then, the input image `I` will be converted to an MCUboot image `M` as usual, and the latter will be given to the attacker, which will need to determine whether its content is valid or random

Since the random image very likely does not contain valid code, installing this image would immediately brick the device, and that would be observable by the attacker. To solve this issue, we configure the system to perform additional steps during updates, depending on the value of `b`. Conceptually, we will simulate the execution of a valid image (`base_I`) when the random image should be executed (or when the image needs to access constants stored in the primary slot). To do this, the system first needs to store whether the currently installed image is the random

Algorithm 7 IP theft game

```
1: procedure IP_THEFT()
2:     // Initialization
3:     v ← 1
4:     keys ← SAMPLE_CRYPTO_KEYS()
5:     I ← CREATE_IMAGE(v, base_I, keys)
6:     INIT_BOARD(I, keys, bootloader, config)
7:     // Phase 1
8:     σ ← A^{phase_1}()
9:     b ← ${0, 1}
10:    if b = 0 then
11:        M ← base_I
12:    else
13:        M ← ${0, 1}^{|base_I|}
14:    v ← v + 1
15:    I ← CREATE_IMAGE(v, M, keys)
16:    RESET_SYSTEM()
17:    // Phase 2
18:    b' ← A^{phase_2}(σ, I)
19:    // Result
20:    if b' = b then
21:        return 1
22:    else
23:        return 0
```

one, by setting a `isRandInstalled` flag during the installation of an image via INSTALLIMAGE, which is part of *Actions 5*. The CPU will thus read the primary slot as if the random image was replaced by `base_I` when it fetches its content via CPUFETCH, except when the bootloader is still executing. The bootloader still needs to read the random image in order to perform updates correctly. This will effectively simulate that a valid image is installed, preventing an attacker from easily discovering it.

Algorithm 8 Internal image replacement

```
1:  rand ← content of the random image
2:  img ← content of base_I
3:  isRandInstalled ← 0
4:  upon event INSTALLIMAGE(image) do
5:      if b = 1 and image = rand then
6:          // Random image will be installed, we need to replace it during execution
7:          isRandInstalled ← 1
8:      else
9:          isRandInstalled ← 0
10: upon event CPUFETCH(address) do
11:     if b = 1 and isRandInstalled = 1 and address is part of the primary slot and
12:         the bootloader is not executing anymore then
13:         // Random image is installed and primary image is executing, simulate valid image
14:         return content of img at address
15:     else
16:         // Fetch data normally
17:         [...]
```

This way, the behavior of the system is the same whether the image is valid or random, which forces the attacker to recognize the image without simply observing whether the image behaves normally or crashes. The attack model is thus more suitable to indicate whether an attacker can recognize an image without looking at internal memory. Finally, a FINALIZE() procedure is not necessary here, as we can simply check whether the attacker's choice b' corresponds to the random variable b. If the attacker cannot guess b with a probability greater than by guessing randomly, the attack is a failure.

Attack Examples

Finally, we will look at some examples of IP that can be translated to our attacker model.

A simple example considers an attacker which has introduced in the same network as the device. They are only capable of observing communication during an update. Formally, we have that the attacker can observe the content of an update (as sent by the update server during an update). To represent this, we consider an empty phase 1. this is due to the fact that phase 1 represents an exploratory phase, and the attacker cannot explore the system here. Before phase 2, an image (either valid or random) is generated and given to the attacker during phase 2. This

is formally similar to the scenario where the attacker observes an update, so we do not need to consider that the attacker has additional capabilities. Having seen the update, the attacker can now try to extract the software that it contains. Formally, we represent this with the fact that the attacker needs to tell whether this image is valid or random, which would indicate a capacity to deduce information from this image.

Another scenario considers an external attacker that wants to observe the content of a specific image. The attacker is external since they have a physical access to the device but can only control the external memory. The attacker manages to be present during the update process and tries to gain knowledge of the new software. Formally, we again consider an empty phase 1. An image, either valid or random, is generated before phase 2 and then given to the attacker. Since we want to represent the fact that they can observe the update process, we formally consider that the attacker themselves can request this update and control external memory. This is characterized by the access to **installImage**(), **readExternalNVM**() and **writeExternalNVM**(). The attacker will thus request that the received image is installed and will control external memory to try to extract a secret from it. If they manage to extract information from the target image (such as its software), this will formally be indicated by the fact that they will be able to tell whether it is a random image.

A last, more elaborate, example can be thought of. This time, the attacker wants to gain knowledge such that they can decrypt all images. To achieve this, they managed to exploit a vulnerability in the software and gained a temporary full control over the image. Formally, this corresponds to phase 1. The attacker can have many capabilities, but we consider at least an arbitrary R/W to memory (both internal and external). Formally, this is characterized by the access to **readExternalNVM**(), **writeExternalNVM**(), **readInternalNVM**() and **writeInternalNVM**().The attacker can then try to extract cryptographic secrets from the system during phase 1, while the faulty image is executing. Then, to represent whether the attacker could extract secrets, the second half of the attack game takes place. Again, an image (either random or valid) is generated and given to the attacker during phase 2. We represent the fact that the attacker wants to decrypt image by not giving them more capabilities during phase 2. Indeed, they had full control of the image during phase 1 and should simply be able to decrypt an image with that knowledge if the attack is successful. Therefore, the attacker will only try to decrypt the image during phase 2 and indicate whether it is random, which will prove or disprove its ability to decrypt any image.

3.7.5 Attacker Advantage

We define the *advantage* of an attacker as being the likeliness of success of an attack, given an attacker model and a bootloader design. More formally, $Adv^{attacker}_{bootloader}(A)$ denotes the advantage of an attacker A which follows an attacker model $attacker$ against a system design $bootloader$.

In general, we will define it as $Adv^{attack}_{bootloader}(A) = Pr(A^{attack\_game} = 1)$ In our case, $bootloader$ will be $MCUboot$ and $attacker$ can be either $Jailbreak$ (for a jailbreak attacker), DoS (for a Denial-

of-Service attacker) or IP_thief (for an IP thief). In particular, we define $Adv^{Jailbreak}_{bootloader}(A)$ as the probability of a successful jailbreak attack against a given bootloader. We defined earlier in *Section 3.7.2* that the attack is a success if the jailbreak game JAILBREAK_ATTACK() returns 1. Therefore:

$$Adv^{Jailbreak}_{bootloader}(A) = Pr(A^{Jailbreak\_attack} = 1)$$

Where $A^{Jailbreak\_attack}$ is the result of a jailbreak attack game JAILBREAK_ATTACK() done by an attacker A. Similarly, we define $Adv^{DoS}_{bootloader}(A)$ as the probability of a successful Denial-of-Service attack against some design. Again, as defined in *Section 3.7.3*, the attack is a success if DOS_ATTACK() returns 1. Thus:

$$Adv^{DoS}_{bootloader}(A) = Pr(A^{Dos\_attack} = 1)$$

Where A^{DoS_attack} is the result of a DoS attack game DOS_ATTACK() done by an attacker A. Lastly, the case of the IP thief is a bit different. Since there are two scenarios (where $\texttt{b} = 1$ and $\texttt{b} = 0$), we define $Adv^{IP\,thief}_{bootloader}(A)$ as the difference between two probabilities, such that the result is 1 if the attacker is always correct and 0 if they cannot do better than guessing randomly. We remind the reader that b corresponds to the randomness of the image that is given to the attacker, and b' is the final decision of the attacker given an attack game. We define:

$$Adv^{IP\_thief}_{bootloader}(A) = Pr(\texttt{b'} = 1|\texttt{b} = 1) - Pr(\texttt{b'} = 1|\texttt{b} = 0)$$

This way, if the adversary is always correct, $Pr(\texttt{b'} = 1 \mid \texttt{b} = 1) = 1$ and $Pr(\texttt{b'} = 1 \mid \texttt{b} = 0) = 0$ so $Adv^{IP\_thief}_{bootloader}(A) = 1$. Similarly, if the attacker chooses b' randomly with probability $\frac{1}{2}$, $Pr(\texttt{b'} = 1 \mid \texttt{b} = 1) = Pr(\texttt{b'} = 1 \mid \texttt{b} = 0) = \frac{1}{2}$ so $Adv^{IP\_thief}_{bootloader}(A) = 0$. In general, the attacker advantage is chosen such that its value ranges from 1 to 0. A value of 1 means the attacker will always win (and the system is thus not considered as secure against a given attack). On the other hand, a value of 0 means that the system is completely secure in this model and the attacker cannot successfully perform an attack. In the case of a IP thief, this means that b and b' are statistically independent.

3.8 Summary

In this chapter, we defined a two-phase attacker model to formally describe the security of the system. This model was chosen to fit as many real-world attacks as possible and because it fits

well the dual nature of the system (bootloader - image). Additionally, both phases are articulated following an attack game specific to each attacker model. Phase 1 corresponds to a preparation phase where the attacker explores the system while phase 2 is a challenge phase where the core of the attack takes place. Three categories of attackers have been defined. A jailbreak attacker wants to install persistent threats of a device, a DoS attacker tries to put the device in a permanently unusable state and an IP thief tries to extract the intellectual property of the images. Those attackers correspond to the most common threats against a bootloader in general. The capabilities of all attackers are precisely defined on a per-phase basis and practical attacks can be then represented in this model by mapping practical actions to available capabilities.

Chapter 4

Case Study: MCUboot

In this chapter, we will look at our implementation of a secure boot using MCUboot. The device on which the bootloader is implemented is a Nucleo-H753ZI development board. Figure 4.1 depicts the development board.

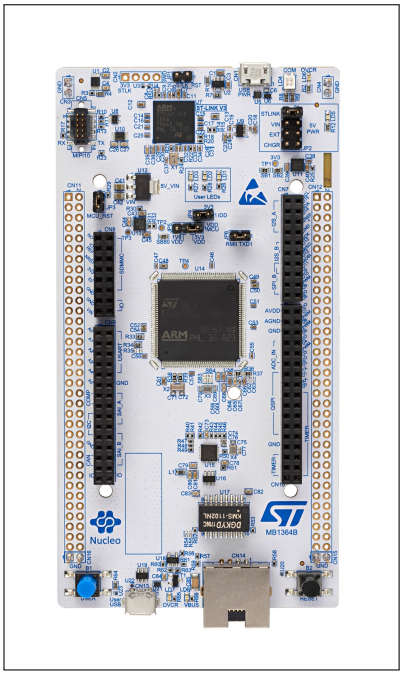

Figure 4.1: STM32 Nucleo-H753ZI development board with STM32H753ZI MCU

4.1 Application To System Model

First, let us show how the case study relates to the system and threat models.

Internal Memory

The Nucleo-H753ZI directly follows the physical layout of the system shown in Figure 3.1. The Nucleo-H753ZI board does not have external non-volatile memory. This is not a problem since the model states that an external non-volatile memory is optional, and not including external memory can only make the design more secure (with regards to the threat model). Regardless, the system is compatible with external memory, as long as anything placed in the external flash is encrypted, with the decryption key stored either in a protected region of internal flash, or in a secure element.

The Nucleo-H753ZI has a STM32H753 microcontroller, which in turn has a Cortex-M7 CPU. The internal flash is 2MB big and the internal flash is 64KB big. The flash is separated in 16 sectors of 128KB each. Based on the MCUboot areas, we define six segments of the (internal and external) non-volatile memory as follows:

- **Secure bootloader storage**. This region must be configured as being execute-only, which means that it cannot be written to or even read. Furthermore, it can only be executed during the boot process, and has to be part of the internal non-volatile memory. This way, it can be used as a secure storage area. It contains secrets of the bootloader such as the private secure element pairing keys. Being execute-only, the content of this segment cannot be extracted and can only be used by the bootloader (for example to establish a secure channel with the secure element).
- **Bootloader**. This region (also part of the internal non-volatile memory) is configured as being read-only, which means that its content is immutable. It is also set to be the only entry point of the CPU after a reset. Together, both properties allow us to consider this segment as the root of trust. Thus, it will contain the code and constants of the bootloader itself.
- **Image slot**. This region is simply flash memory located in the internal non-volatile memory. It corresponds to MCUboot's primary area.
- **Scratch area**. This last region is internal flash memory as well. It corresponds to MCUboot's scratch area.
- **Upgrade slot**. This region is also flash memory. It can be either internal or external. It corresponds to MCUboot's secondary area.
- **User area**. This region (representing the remaining the non-volatile memory) can freely be used by the image to store persistent data and cannot be used by the bootloader. This region is not mandatory (and not defined by MCUboot) but offers more usability to the image in a concrete scenario.

Flash Layout

Each flash segment is then composed of an integer number of flash sectors, in order to avoid conflicts between areas. A summary of the design layout is shown on Figure 4.2. Note that in this configuration, the only communications that are considered as exposed are those interacting with the secondary slot or the update server (as depicted in Figure 4.2). Indeed, the image uses the network interface (which is exposed) to download an update from the update server. Moreover, the secondary slot is assumed part of the external storage, so communications between the secondary slot and the processor are exposed. Such data flow happen either when the processor executes the bootloader (it will read from and write to the secondary slot) or the image (which will write to the secondary slot).

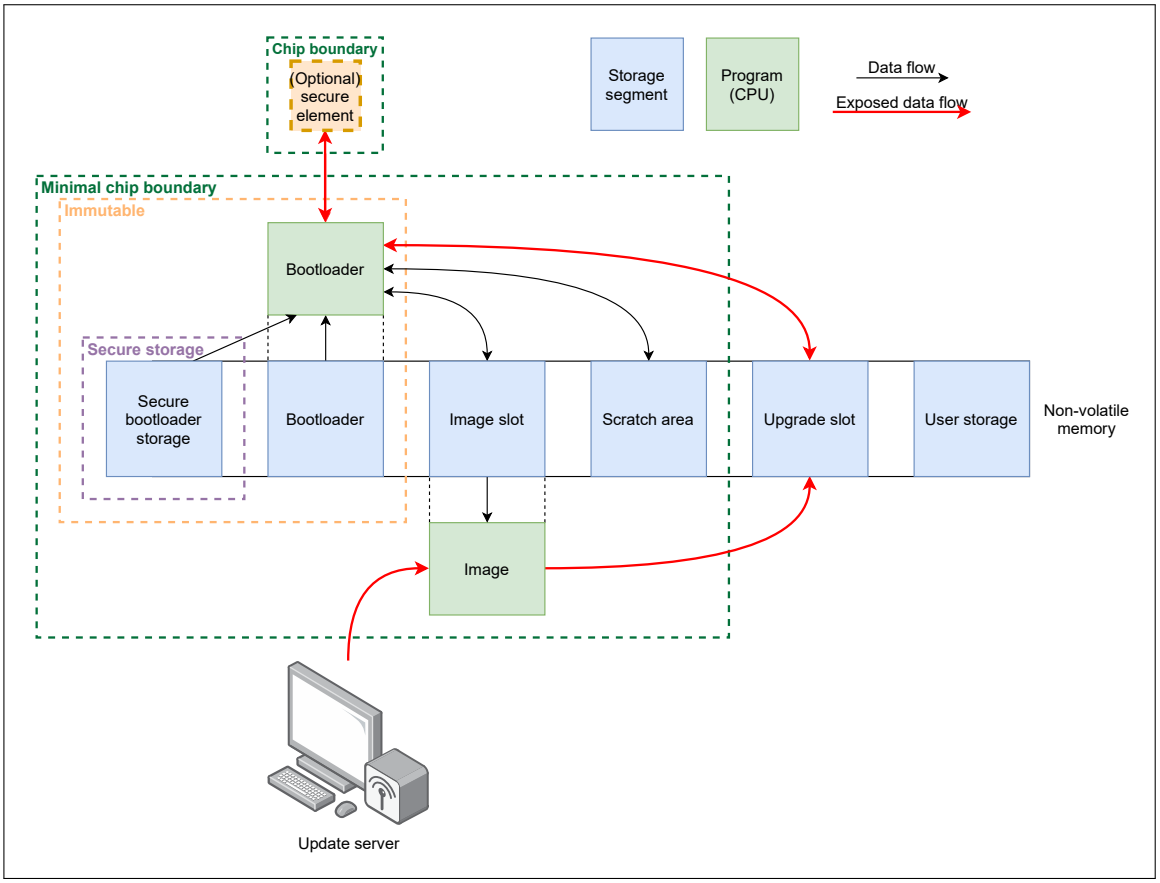

Figure 4.2: Design of the secure bootloader based on MCUboot.
Summary of the design layout, based on MCUboot. The non-volatile memory is split into segments. All segments except the secondary slot and the user storage have to be part of the internal storage, and the bootloader (and its secure storage) is immutable. The depicted data flow (not to be confused with the execution flow) shows the movements of storage content.

Thus, data stored in the secondary slot shall always be encrypted, while data in the other segments do not need to be. This is because the secondary slot might be part of the external storage which is more vulnerable to an attacker. On the other hand, the code in the primary slot and bootloader area will directly be executed by the processor and thus cannot be encrypted (the hardware might not be capable of decrypting data on-the-fly while executing code). The update

server could thus provide images that are already encrypted so that they can simply be stored in the secondary slot without leaking data. The scratch area could be encrypted in another design (and thus put in the external storage), but in this design it is not encrypted and must be part of the internal storage. This is imposed by MCUboot which chooses not to encrypt data in the scratch area during an update for technical reasons. That also means that the update process will have to decrypt and re-encrypt data on-the-fly. This is caused by the fact that an update will move data from a slot where data should be encrypted (the secondary slot) to one where it should not (the primary slot), and inversely. This helps ensuring that data never sits in clear in exposed storage, but this will also have a performance overhead on the update process.

In our case, the system model's user is a computer connected by USB to the board. We can represent the connectors by two interfaces: UART and debugging interfaces. The only implemented network interface in this case study is the UART interface, although we could imagine adding another interface (such as Ethernet, which is supported by the board). The update server, like the user, is represented by a computer connected to the board by USB. Thus, the microcontroller only communicates with the update server by UART. We will describe in more details the hardware security (which is sufficient to match the system model) in *Section 4.2.3*.

System Primitives

We will now detail the three design-specific primitives used in the attack game, introduced in *Section 3.7.1*: SAMPLE_CRYPTO(), CREATE_IMAGE() and INIT_BOARD(). SAMPLE_CRYPTO() strictly follows the initialization phase of the cryptographic protocol of MCUboot, described in *Section 2.4.4*. This means that SAMPLE_CRYPTO() will simply sample an ECIES key pair (k_{enc}^{pub} and k_{enc}^{priv}) and an ECDSA key pair (k_{sign}^{pub} and k_{sign}^{priv}). Those keys will be embedded in the bootloader or stored securely by the entity that generates updates. Those keys are thus accessible during the attack game by the corresponding entity, once they have been setup. In particular, we remind that the bootloader is installed in an immutable part of non-volatile memory, and embedded keys are thus also immutable. We assume that the update provider can securely store private keys. Public keys are considered as known by everyone. The location of different keys is explained in Figure 4.3.

CREATE_IMAGE(`version, software, keys`) must convert an input `software` to an MCUboot-compatible image. It will do so by passing `software` through the MCUboot image generation script along with `keys`, which will sign and encrypt `software` by following the cryptographic procedure described in *Section 2.4.4*. The script will also prepend a header. Since the image is encrypted, it is ready to be stored in the secondary slot so that the bootloader can verify it.

Finally, INIT_BOARD(`image, keys, bootloader, config`) must initialize the state of the device so that it is fully ready to be used. The first thing it needs to do is to install the bootloader on the board. To do that, it will simply store `bootloader` in the flash memory, in the bootloader area. It will also store the public signature key k_{sign}^{pub} in the bootloader area, and the private encryption

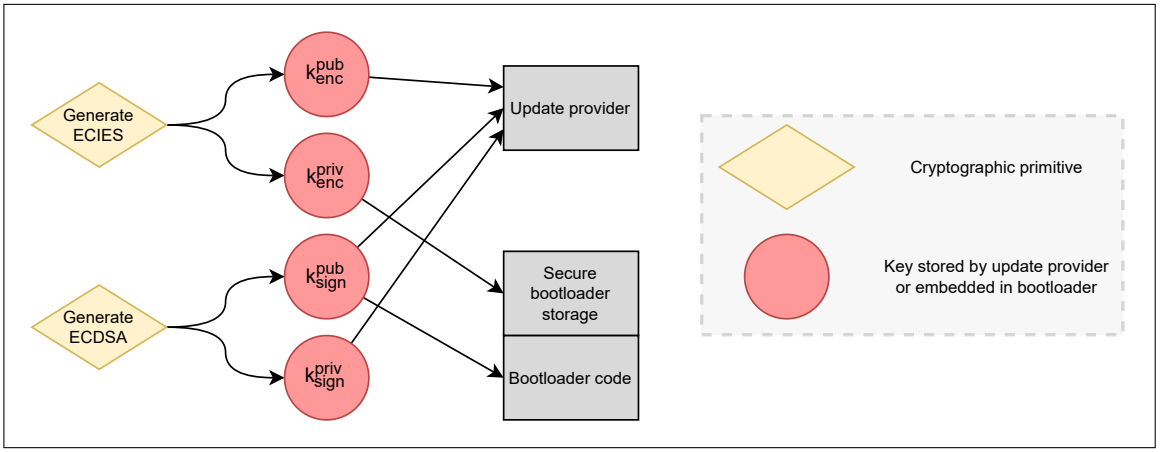

Figure 4.3: Location of keys during the initialization process

key k_{sign}^{priv} in the secure bootloader storage. Then, it needs to install the initial image, so it will store `image` in the secondary slot and let the bootloader execute so that it gets installed. Finally, both the bootloader and a valid image are installed, and so the board security can be enabled by applying `config` to the board. As we will see in *Section 4.2.3*, on STM32H753 this corresponds to setting the secure area to the bootloader area and enabling the readout protection.

4.2 Port

Let us now have a look at the part of the firmware which was implemented around MCUboot to get a secure bootloader that fits our threat model. A more technical description of files, variables and usage will be presented in *Section 4.4*.

4.2.1 Project

As MCUboot is only a library, a project still needs to be built around it, and MCUboot also has prerequisites that need to be implemented. This part will describe the files we added to get a working bootloader. As some information on the port of MCUboot are hardly found in its documentation, other online resources had to be found. As a result, the porting methodology and most of the structure and basis of the port are inspired by [5] and its example implementation [6]. The organization of the files of the bootloader is depicted on Figure 4.4.

Bootloader Files

The first necessary resource is a complete makefile. The compilation rules are mostly inspired by the structure of [6] and adapted to our own needs. Many files of the MCUboot library needed

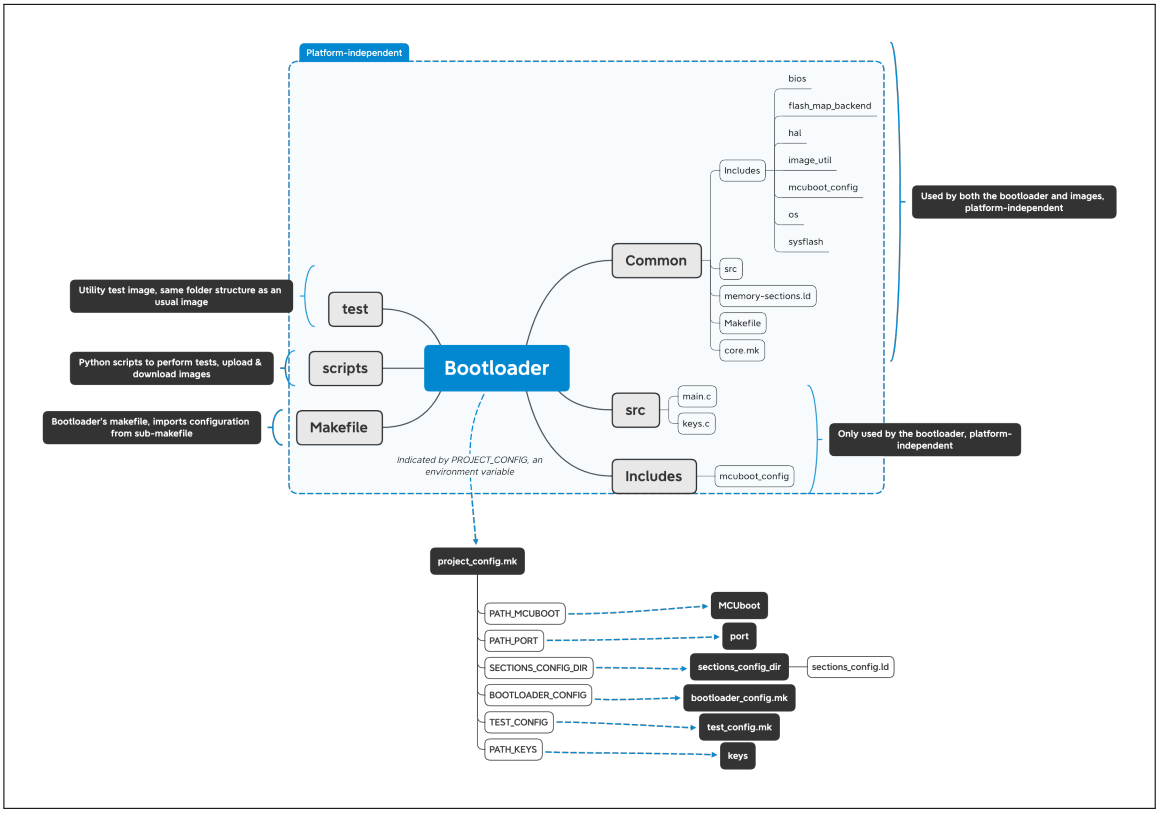

Figure 4.4: Organization of the files of the bootloader. All files withing the blue box are hardware-independent, the link with all hardware-dependent files is done with a single configuration file.

to be added to it, as well as files that contain software implementation of the cryptography used by MCUboot. All custom files added to the project will also be added to this makefile. Additionally, the makefile can define a few custom targets for convenience. In particular, the makefile can define a target that will directly program the bootloader into a connected device, in the bootloader slot. Targets can also be defined to modify the state of the board (for example, to set or unset the hardware security we will need later).

Another required component is a linker script that is compatible with MCUboot and that will be used by both the bootloader and corresponding images. In particular, it must split the flash memory into slots so that there is a well-defined bootloader area, both primary and secondary slots and a scratch area. The linker script also defines multiple constants that can be read by the bootloader and images, mostly indicating the starting address and size of different regions.

Then, a program entry point is necessary. Both the `main.c` file and the `startup.c` file are created to fulfill this role. The `startup.c` file defines the different interrupt handlers, prepare memory upon boot and calls the program main entry point. The `main.c` file then contains the start of the bootloader's program, which will initialize the drivers are call MCUboot.

MCUboot requires a configuration file with specific value defined depending on the

desired behavior. As some of those values are not properly documented, [7] was used as a reference of available symbols. Most notably, the symbols we need to define to fit our system model are `MCUBOOT_ENC_IMAGES` (to enable encryption of images in secondary slot), `MCUBOOT_SWAP_USING_SCRATCH` (to upgrade images using the scratch area), `MCUBOOT_BOOTSTRAP` (to always try to read secondary slot if primary is not valid) and `MCUBOOT_VALIDATE_PRIMARY_SLOT` (to always verify the primary slot before booting, even if no update is done).

Lastly, since the bootloader is the first program that is executed on the board upon reset, no drivers have been loaded. Furthermore, MCUboot requires a few drivers (most notably flash and a way to print logs, UART in our case) that are not included with MCUboot (to keep it as platform-agnostic as possible). Thus, we are required to implement flash and UART drivers, as well as any driver that will be used. A more complete description of drivers can be found in *Section 4.2.2*. MCUboot also requires a specific flash interface and symbols to use their common interface, so a `mcuboot_flash.c` file is added to link the flash drivers, MCUboot and the linker script layout together.

Common Files

Then, since MCUboot images will need to have access to parts of the bootloader (specifically, the core of its makefile, the linker script, its configuration and part of its drivers), the project is articulated around a `Common` folder. The `Common` folder will then contain everything that must be shared between the bootloader and at least some images. It also contains a default startup file, which is sufficient for most simple images. More precisely, the compilation rules and the core structure of the makefile are located in the `Common` folder, and both the bootloader and images will define their own minimal makefile while including the same core makefile. In addition, a more practical wrapper around the core makefile is made for images, in the form of another makefile. The goal is to define all targets that are common to all images in a single file, and facilitate the creation of images. In particular, it defines a few variables used by the core makefile, but most importantly targets used to generate and program images. Once an image has been compiled by the core makefile rules and an elf file is generated, an MCUboot-compatible file still needs to be generated. Firstly, a stripped version of the result is created in a bin file (by only keeping the binary content of the image). This stripped file then goes through MCUboot's image generation script (which will mostly prepend a header and sign and/or encrypt the content of the image), `$(PATH_MCUBOOT)/scripta/imgtool.py`. Two final products are thus generated in separate targets: a signed image (designed to be programmed into the primary slot) and a signed and encrypted image (designed to be downloaded into the secondary slot).

Some makefile targets require linker file symbols. For example, whenever MCUboot's image generation script itself is called, the size of an image header as well as the size of an image need to be passed as parameters. Programming an image into either the primary or secondary slots also require the address of the corresponding slot. Those values are project-wide constants,

but to avoid duplicating them in the makefile, it is preferable to read the symbols directly. The required symbols are defined in the linker script, so we might imagine that the makefile could simply parse the symbols to obtain their values. Unfortunately, the symbols themselves could be represented in multiple formats (decimal, hexadecimal, with different units, or defined as a computation) which would make the parsing harder. Even worse, they could be defined using other linker symbols, which would require a recursive parsing strategy. We could also define such symbols in a separate file, in a strict format, but then reading symbols from the linker script would also prove to be a problem. The key idea to solve this issue is to remind ourselves that any elf file generated using the linker script will itself contain, among its metadata, the values of symbols defined in the linker script. This means that a result elf file will always contain the necessary symbols, in a standard 8 bytes hexadecimal formal, right next to a string describing the symbol name. The makefile can thus fairly easily obtain the value of those symbols automatically by using a simple regexp to parse any output elf file (since the linker script is shared between any image and the bootloader). For this reason, some makefile targets start by parsing an elf file to recover a necessary symbol value, to be able to pass it as an argument to another program.

Image-Specific Files

A couple of image-specific files are required to create a new MCUboot-compatible image, one of which is a makefile. However, most of the required content of the makefile is already defined in the `Common` folder, so the new makefile only needs to define some image-specific variables (location of projects, additional `.c` or `.h` files, target name, encryption/signature keys, etc.). Another required file is a `main.c` file. Its existence (as well as a `main` function) is required since the image needs a entry point. The contain of the `main` function (i.e., the behavior of the image) can however freely be chosen, or even left mostly empty. The last required file is an image-specific configuration file, which mostly defines the logging properties of MCUboot's public interface. It is then possible, using the pre-existing files of the `Common`, to create a new MCUboot-compatible with only a few simple files. The image can even be capable of downloading an update by simply letting the utility BIOS handle communications. This is demonstrated by the provided `hello_world` image, which is a very minimalist yet fully functional image.

4.2.2 Drivers

In this section, we will present the drivers that are implemented in this project. All of those drivers were necessary at some point to have fully functional bootloader and images.

Drivers Are Shared

Note that drivers are located in the `Common` folder, which means they can be included in both the bootloader and any image. This is because the drivers are not specific to the bootloader

or MCUboot, so they might as well be part of the `Common` folder, in case an image doesn't have its own drivers. However, in order not to violate the system model (and not to introduce attack vectors), the bootloader and the images cannot *physically* share any code, meaning that an image needs to include its own copy of the drivers (and any other common code). A consequence of this is that we want to keep the drivers as minimal as possible, to avoid taking up too much space in flash memory. Indeed, the compiled binary code of some drivers might be included in the bootloader and both images, meaning they effectively can take three times as much space. It is also in general desirable to keep the bootloader as small as possible, to avoid limiting the space available to other programs (as the bootloader area will always be unavailable to other programs). For those reasons, many drivers include only basic features and generally use the simplest solution. In particular, they do not use interrupts, nor are they designed to be concurrent in any way. Additionally, drivers can individually be included or ignored in any image.

The drivers themselves are separated into two different parts, corresponding to a hardware-agnostic part and a port-specific part. Respective folders are `hal` and `drivers`. The content of `hal` is designed to be as independent from underlying hardware as possible. It mostly exposes a common interface usable by the bootloader or any image, or defines helper functions. The rationale is that the entire content of this folder should be common to every port, given some potential minimal platform requirement are met. On the other hand, the content of the `drivers` folder is entirely dependent on the underlying hardware. The goal is to regroup in a single folder almost every file that needs to be modified in order to port the bootloader to another platform. The content of the `drivers` folder constitutes the minimum set of hardware-dependent functions and constants that need to be defined in order to have a working bootloader. As such, the link between the bootloader and the content of the `drivers` folder is already made in the `Common` directory. Besides necessary definitions, the `drivers` folder can also add any new platform-specific definitions, or helper functions. A consequence of this is that a given port does not indicate which definitions are used by the bootloader (and are thus necessary) and which ones are specific to the port. It is thus not obvious to port the system based on an example port. The structure of a port folder is depicted on Figure 4.5.

List Of Drivers

For this reason, a port template can be found in a `port_template` folder. It contains an empty definition of every single device-specific function and constant that is used somewhere in the project. It shall be used as a reference in order to port the system. The implementation of a given function can then be inspired by the provided example port, since it necessarily exists. Another reason to regroup hardware-dependent drivers in such a way it to detect early if for some reason a device is incompatible with the bootloader. Indeed, it is easier this way to detect, for example, if some component is missing or if some hardware requirement contradicts the available prototypes.

The list of implemented drivers is the following:

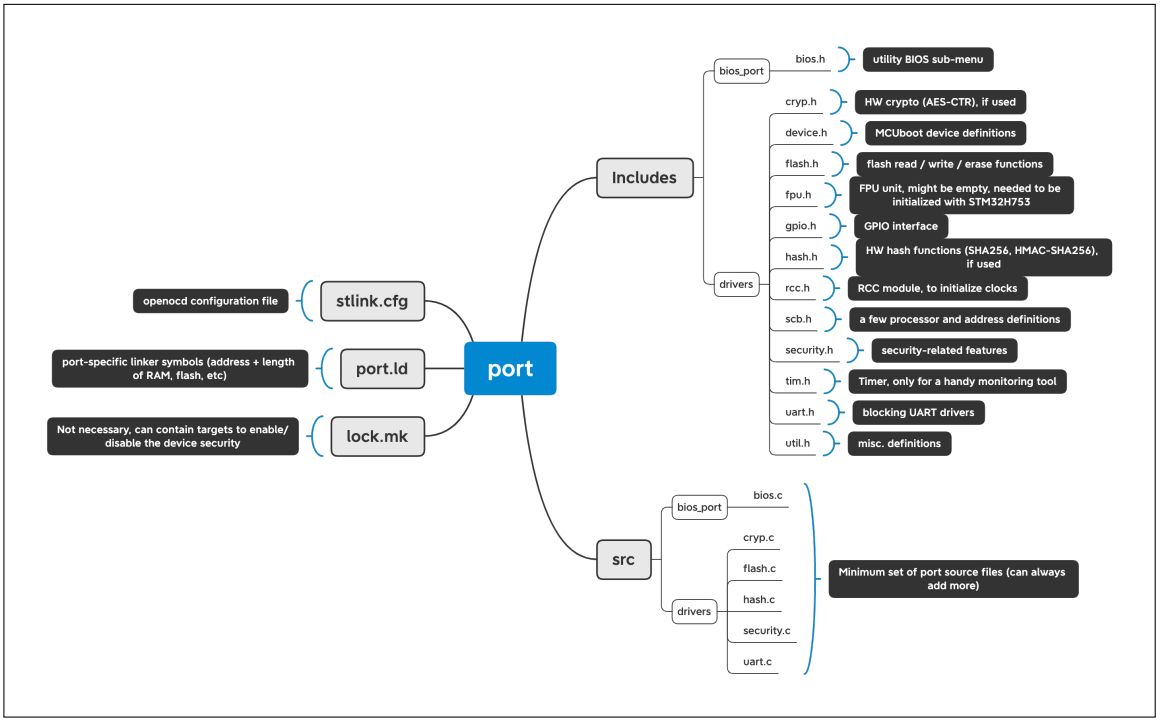

Figure 4.5: Organization of the files a port folder. It contains all hardware-specific definition related to a given port.

- `cryp`, the cryptographic module that performs AES-CTR. Only necessary if AES-CTR is configured to use the hardware implementation.
- `flash`, the flash memory interface.
- `fpu`, a module that needs to be initialized on STM32H753. Can be left empty if unnecessary.
- `gpio`, the GPIO interface. Only used to interact with the user (e.g., via a single push button)
- `hash`, the hash module that performs SHA256 and HMAC-SHA256. Only necessary if SHA256 or HMAC-SHA256 are configured to use the hardware implementation.
- `security`, the board protection mechanisms interface. Used to configure and monitor memory protections.
- `rcc`, the reset and clock control module. Used to initialize necessary clocks.
- `scb`, the system control block. Used to get access to the VTOR register and trigger system resets.
- `tim`, the timer module. Can optionally be left empty; only used to monitor MCUboot's execution time.
- `uart`, the UART module, used for logging and to issue commands to the utility BIOS.

- `util`, miscellaneous utility functions.

UART And Flash

Apart from UART and flash drivers, all drivers were implemented from scratch by following ST's reference manual [2]. The provided drivers were not satisfactory because we needed very few features, and extracting them from ST's drivers to make them compatible and reduce them to a minimal form would represent more work than simply implementing a minimal version from scratch. UART and flash drivers are notable cases as they are the most heavyweight drivers of all. Both are directly extracted from ST's HAL drivers (i.e., reduced to their minimal form and made independent by removing dependencies with other drivers) and potentially adapted for our needs. More specifically, the UART driver is reduced to its blocking, non-concurrent and non-DMA version. For this reason, the receiving part of the driver should always know in advance how many bytes it expects to receive. This is, in practice, not a problem since the size of a message can be sent in a first fixed-size message. Additionally, the image download and upload protocols were designed to work only with a blocking UART, for synchronization purposes.

On the other hand, the flash driver has been slightly modified to solve some inconveniences that are present in ST's drivers. In particular, the flash write operation does not wait for the end of a previous flash erase under certain circumstances when the board security is enabled (see *Section 4.2.3*). This means that if an erase operation is immediately followed by a write operation, the latter could fail and return an error code, and it would be necessary to retry the flash write as long as it fails, which would be a terrible practice. To solve this, the flash write routine now also waits for the end of any flash erase operation. Additionally, it is possible under certain circumstances to raise a persistent flash error flag between two flash write operations (for example, by trying to read the bootloader secure area while outside of the bootloader). It is incompatible with the fact that a write operation first checks error flags and fails if any is raised, before even trying to write to flash. In other words, it was possible to force the next write operation to fail only by reading a protected area, effectively crashing MCUboot. This has been fixed by clearing previous error flags before attempting a flash write operation.

Handling ECC Errors

Another flash driver issue is related to the Error Code Correction (**ECC**) of the STM32H753. Its purpose is to monitor the integrity of the flash memory. It is able to detect some errors in every flash word. An error could occur if an operation is interrupted, or if multiple writes are done to the same flash word without erasing the sector first. A single error will be signaled by an error flag, but two or more errors will immediately trigger a hard fault exception upon any read access to the corrupted word. This means that some corruption in specific areas of the flash memory occurred, it was possible for MCUboot to crash while trying to read this area. Since the flash memory is persistent, the bootloader would consistently crash after every reset. This has been

fixed (as proposed by [8]) by modifying the flash read operation to check for memory corruption before attempting to read any flash word. Returning an error at this point would cause MCUboot to immediately panic, when it could easily recover. The adopted solution is thus to silently hide the underlying error and act as if the flash word was empty. By design, MCUboot will then ignore the content of the corresponding area and erase it first if anything needs to be written to it. This way, MCUboot can gracefully recover from a non-fatal memory corruption, for example in the secondary slot header.

A last small change was done to the flash write operation, but only when a specific flag (`ENABLE_FLASH_CORRUPTION`) is defined. This flag allows partial and unaligned flash write operations, which if done repeatedly can systematically trigger a double ECC error. This flag is disabled in the bootloader and only intended to be used by the test routines, to corrupt some part of the memory and check that the bootloader can recover from it.

4.2.3 Hardware Security

In this section, we will present how the STM32H753 MCU fulfills the security requirements of the system model. First, let use introduce some of its features.

Readout Protection

The first interesting security feature is the readout protection. This is a setting of the board that can take three different levels: RDP0, RDP1 and RDP2 [1, pp. 30–31]. RDP0 represents a state of no added security. In RDP1, any debugging access attempt to the flash memory results in an error and the complete shutdown of the flash memory, meaning any subsequent access (even valid and by the bootloader) is refused, even after a system reset. To unlock the flash memory, a power-down reset should be done, meaning the board should be physically un-plugged before being usable again. This effectively means that access to flash is impossible from outside of the board, only the MCU can read its own flash. The last level, RDP2, is similar to RDP1 but all debugging features are permanently disabled, most of the board configuration (including its security) is permanently frozen, and the readout protection cannot be changed anymore.

Secure User Area

Another feature is the ability to define a user secure area. This area is secure in the sense that is will always be executed upon reboot, and some features are prohibited while the board is executing code in the secure area. It is the responsibility of the code in the secure area to signal when the processor exits the secure area (via a function called `exitSecureArea()`) [2, p. 249]. After this, any access (read, write or execute) to any flash memory in the secure area is impossible. The only time when this area can be access is between the board reset and the first call to `exitSecureArea()`. Additionally, while in the secure area, all exterior debugging or

configuration requests are ignored.

By locating the entire bootloader inside the secure area and enabling RPD2, the entire bootloader effectively becomes immutable, and it can safely embed secret keys in its code. The reason is that RDP2 prevents any debug feature or any change of security setting, while the secure area ensure only the bootloader is capable of reading and writing to itself. The bootloader is not technically read-only since it is authorized to overwrite itself, but if the bootloader is not programmed to ever overwrite itself, we can safely assume that its area is permanently frozen in flash. It is also possible to further enforce this by enabling a write protection, even though this design does not use it. This way, it becomes a root of trust. Additionally, it can contain a secure bootloader area in which private keys can be stored. In fact, this secure bootloader area encompasses the entire bootloader area in our case due to hardware requirements, but it can only be more secure that the model. In practice, the private keys can be embedded anywhere in the code of the bootloader. In this state, no interaction with the bootloader can be done from the outside, except from the interfaces explicitly exposed by the bootloader (like the utility BIOS which communicates via UART, if it is enabled during development, see *Section 4.2.5*). For this reason, we trust that no interactions (besides resetting the board) can be done from the outside with the bootloader while it is executing, in a production build.

Permanent Protections

Setting RDP2 is permanent and the bootloader can thus never be modified again, so depending on the application it might be desirable to only enable RDP1 since it is possible to disable RDP1 as well as the secure area, but this will trigger a mass erase of the entire flash memory. System model-wise, this means that the bootloader would technically not be immutable, but at least we can still guarantee that no sensitive image information can be extracted because disabling RDP1 would erase all data.

In fact, one should be careful with such security settings as it is even possible to permanently lock the board in a unusable state by only using the secure area. As we mentioned, it is not possible to interact with the board while the board is still in the secure area, meaning that if the bootloader crashes or never calls `exitSecureArea()`, it becomes permanently impossible to modify the board's settings from the outside. If the bootloader is not capable of disabling the secure area from within its own code, the board is then left in a permanently unusable state. A similar case can occur if the secure area is wrongly configured.

4.2.4 Hardware Cryptography

Let us now describe how we take advantage of available hardware cryptographic and hash modules.

By default, MCUboot uses software implementations of cryptographic primitives used in the

design (such as AES-CTR or SHA256). However, they left the possibility of modifying MCUboot to use any other compatible implementation, including those using hardware modules when they are available. As a reminder, MCUboot in our configuration uses the following cryptographic algorithms:

- *SHA256*, to compute hashes
- *HMAC-SHA256*, to sign an encryption key
- *AES-CTR*, to encrypt and decrypt images
- *ECDSA* with P-256, to sign images and verify their signatures

Among those, only ECDSA cannot be implemented using a specialized hardware module on the STM32H753 that can accelerate its execution. The three other algorithms can be performed by a specialized hardware module [2, pp. 1278–1279, 1247–1348]. This module is most likely tailored to a few algorithms and is thus very likely to execute much faster than any purely software counterpart. Additionally, it is a bit more secure as less data transition through volatile memory during computations, so there are less potential data leaks.

In particular, two different hardware modules can be used to compute SHA256, HMAC-SHA256 and AES-CTR: `cryp`, the cryptographic processor (which supports AES-CTR) and `hash`, the hash processor (which supports SHA256 and HMAC-SHA256). For this reason, two additional drivers are defined corresponding to those two peripherals. The three hardware implementations (using both drivers) are then used instead of software implementations when `USE_STM32_CRYP` is defined. As with other drivers, they were implemented from scratch by following ST's reference manual. It should however be noted than the implementation of AES-CTR raised an issue in ST's reference manual [2, p. 1306] where it is stated that AES-CTR counter is in a register called `CRYP_IV0L`, where in fact it is stored in `CRYP_IV1R` [9].

4.2.5 Utility BIOS

Finally, we will present here a tool that was created for development, testing and demonstration purposes.

Description

The tool is a utility menu that can simply be included in the bootloader itself by defining the `ENABLE_BIOS` variable in the bootloader configuration file. Alternatively, it can also be included in an application by adding the corresponding files to the application's makefile. If included in the bootloader, it will by default be executed right after the initialization of the drivers, before calling the MCUboot library. Additionally, it is executed only if the user requests it (for example

on our Nucleo board, only if the user push button is pressed during the boot sequence). This way, it can be ignored if accessing the utility BIOS is not desirable. It is intended to be used as a simple debug menu. It communicate via UART to print information or instructions and expects responses from the same channel. It will print commands and wait for a user response. The utility BIOS expects single-character responses, where different letters are bound to different actions or sub-menus.

It should however be noted that care should be taken before including the utility BIOS. As most features are designed as a way to interact with the bootloader or even bypass some of its functions, it also introduces easy ways to compromise the security of the bootloader. As such, the bootloader cannot be considered as secure (with regards to our threat model) as long as the utility BIOS is included in it. It should never be included in the bootloader in a final version and is only intended to be used as a debug tool, or for testing or demonstrations. The utility BIOS can however safely be included in any image without breaking the threat model, as the underlying hardware security should prevent sensitive features in the image, according to the system model.

Features

The functionalities of the utility BIOS were chosen to offer a way to monitor the state of the system and interact with it to allow some degree of debugging even if even when it is not possible to conventionally debug the system (which should not be possible given the system model). The complete list of actions that can be done on demand is the following:

- Upload an encrypted image to the device via UART and store it in the secondary slot. This can be used to test the upgrade feature of the bootloader without needing an image capable of downloading an update. It also means that it is sufficient for an image to include the utility BIOS in order to be capable of downloading an update, without implementing additional download protocols in the image.
- Force the bootloader to execute the image currently stored in the primary slot. This may prove useful when the bootloader is incapable of correctly verifying an image and thus refuses to execute it.
- Exit the utility BIOS and continue the program execution.
- Restart the device. This can for example be used to trigger the application of an update from software.
- Mark the image currently stored in the secondary slot for upgrade. This means that upon the next execution of MCUboot, the update (if valid) will be applied. This can be useful to test the upgrade feature if the image cannot indicate that the update should be applied.
- Read the content of the bootloader secure storage. This is especially useful to validate that the bootloader secure storage can only be read by the bootloader. Any image that includes the utility BIOS should not be able to see the content of this area.

- Print the start of the content of either the primary slot or the secondary slot. This can be useful to quickly monitor whether a given slot contains an image, or to check if it is encrypted or in clear.
- Download the entirety of a the current state of either primary or secondary slot via UART. This can be useful to recover the complete content of the currently installed image or update in order to thoroughly inspect it, if something went wrong.
- A port-specific print option. In general, should print whether hardware security is enabled. On our Nucleo, it prints the current status of the option registers (which indicate whether the secure area is defined and the current readout protection level).

Sensitive Features

In addition to that, an entire sub-menu is platform-specific. Its purpose is to configure the board security (which depends on the available hardware). In the template port, it is left empty. On the Nucleo board the actions can be used to modify the current readout protection and enable or disable the secure area.

Please note that it is possible to put the board in a permanently unusable state if one is not careful, as described in *Section 4.2.3*. For this reason, some actions are only enabled if a specific variable `DISABLE_SAFEGUARD` is set. This variable is not set by default and should explicitly be added to the makefile in order to enable those actions, to avoid accidents. Affected actions are mentioned explicitly below. The list of options specific to our Nucleo board is the following:

- Set the readout protection to RDP0 and unset the secure area. This can be useful to ensure that the bootloader always has a way to completely disable the board security (unless RDP2 is not enabled). It was desirable, during development, to make sure there is no way to permanently brick the board if we cannot disable the security from outside of the board.
- Set or unset the secure mode. The secure mode is fully reversible and can simply be seen as a prerequisite to enable the secure area.
- Set the secure area to the bootloader area. Requires `DISABLE_SAFEGUARD` to be defined.
- Set the current readout protection level to 1. Requires `DISABLE_SAFEGUARD` to be defined.

Finally, note that there is no option to enable RDP2. Since this option would be permanent, the BIOS does not allow to enable RDP2 itself to avoid any unwanted accident. Some code to enable RDP2 should be manually added (either in the bootloader or in an image). This way, it should not be possible to enable RDP2 by mistake.

Upload And Download Images

Some features of the utility BIOS allow to download or upload an image via UART. We shall explain here more precisely how this is done.

The main challenge when implementing the upload and download protocols is linked to the fact that we want the drivers to be kept minimal. More specifically, we want to use the UART driver to avoid adding an unnecessary new platform-specific driver. The UART driver itself is blocking and has a limited input buffer size. This means that part of the communication can be lost is too much data is received before being processed, which is problematic since images should keep their integrity during the transfer. Additionally, received data has to be stored in the flash memory and there is no guarantee that writing to flash memory is fast enough to avoid receiving too much data in the mean time.

The chosen solution is to use a script (on the computer-side of the communication) and synchronize it with the BIOS by using RDY ("ready") messages so that data are only sent when the other party is ready to receive more. Data are sent a single flash word at a time, since it is the smallest granularity at which flash should be written to. In both the download and upload cases, the BIOS is intended to initiate the protocol. It will thus always be the first to wait for a RDY signal. In practice, the RDY signal corresponds to any single-byte message, since sending any data can be enough to signal that more data are ready to be received. *Algorithms 9* and *10* show the procedures used to download an image from the device, executed respectively by the utility BIOS and the `image_download.py` script. *Algorithms 11* and *12* show the procedures used to upload an image to the device, executed respectively by the utility BIOS and the `image_upload.py` script.

Algorithm 9 Image download protocol, device side

```
1:  procedure UPLOADIMAGETOCOMPUTER()
2:      // Wait for a RDY signal
3:      _ ← UARTRECEIVE(1)
4:      UARTSEND(4, image_size)
5:      processed ← 0
6:      while processed < image_size do
7:          // Wait for a RDY signal
8:          _ ← UARTRECEIVE(1)
9:          data ← next FLASH_WORD_SIZE bytes of image
10:         UARTSEND(FLASH_WORD_SIZE, data)
11:         processed ← processed + FLASH_WORD_SIZE
```

Algorithm 10 Image download protocol, computer side

```
1: procedure DOWNLOADIMAGEFROMDEVICE()
2:     // Send a RDY signal
3:     UARTSEND(1,RDY)
4:     image_size ← UARTRECEIVE(4)
5:     file ← open new file
6:     processed ← 0
7:     while processed < image_size do
8:         // Send a RDY signal
9:         UARTSEND(1,RDY)
10:        data ← UARTRECEIVE(FLASH_WORD_SIZE)
11:        append data into file
12:        processed ← processed + FLASH_WORD_SIZE
```

Algorithm 11 Image upload protocol, device side

```
1: procedure DOWNLOADIMAGEFROMCOMPUTER()
2:     image_size ← UARTRECEIVE(4)
3:     erase recipient flash sectors
4:     processed ← 0
5:     while processed < image_size do
6:         // Send a RDY signal
7:         UARTSEND(1,RDY)
8:         data ← UARTRECEIVE(FLASH_WORD_SIZE)
9:         append data into the recipient flash sectors
10:        processed ← processed + FLASH_WORD_SIZE
```

Algorithm 12 Image upload protocol, computer side

```
1: procedure UPLOADIMAGETODEVICE()
2:     UARTSEND(4, image_size)
3:     processed ← 0
4:     while processed < image_size do
5:         // Wait for a RDY signal
6:         _ ← UARTRECEIVE(1)
7:         data ← next FLASH_WORD_SIZE bytes of image
8:         UARTSEND(FLASH_WORD_SIZE, data)
9:         processed ← processed + FLASH_WORD_SIZE
```

4.3 Evaluation

4.3.1 Security Analysis

In this section, we will assess the security of the implementation with regards to the threat model and try to convince the reader that it can be considered as secure. We remind the reader that some features that were implemented introduce vulnerabilities and should not be used in a production build. We thus consider in this section that no additional flaw was introduced by using readout protection level 2 and by completely disabling the utility BIOS from the bootloader.

Jailbreak

Let us start with the case of a jailbreak attack. According to the board's manufacturer, the readout protection level 2 is physically irreversible and permanently freezes the board security configuration and finally permanently disables debug features. They also claim that the secure user area becomes completely inaccessible as soon as the device exits the secure user area, until the next device reset. Combining all of that, we get that the bootloader area can only be modified from within the bootloader area itself. Since the bootloader area only contains the bootloader itself, and since it is programmed to never modify its own area, we claim that the bootloader is in fact immutable. We remind the reader that fault injections are not considered in this model, so the execution of the bootloader cannot be influenced by introducing glitches.

Additionally, the device is by default configured to always boot from the same address, which is the start of the flash memory, i.e., where the bootloader area starts. Since this starting address cannot be changed thanks to RDP2, the bootloader is in fact the only boot entry point of the device, which thus means that the bootloader is indeed a root of trust, since it is always executed after a reboot and is immutable. We also claim that the bootloader is designed to always choose between booting from an image in the primary slot or halting. This means that, in our threat model, a jailbreak attack can only be successful if the bootloader decides to execute the attacker image during phase 2 of an attack game. Since the bootloader is designed to always check the validity of an image before booting it (even after an update is installed), an attacker has only two ways to succeed in a jailbreak attack: they can either exploit the cryptographic scheme that was chosen, or perform a time-of-check to time-of-use attack.

We claim that a time-of-check to time-of-use attack during phase 2 of a jailbreak attack game is not possible. The attack would consist in modifying the image in primary slot between the time the bootloader reads its content and jumps to its location. During this time period, the bootloader is still executing and the attacker would thus have to directly modify the non-volatile memory. Since the primary slot is in internal memory, the attacker needs to directly modify the internal memory during phase 2. But since access to internal memory is not part of *Actions 3* and since all debugging features are disables thanks to RDP2, we claim that the jailbreak attacker

cannot modify internal memory during phase 2. Thus, a time-of-check to time-of-use attack is not possible.

Therefore, the only remaining option to successfully perform a jailbreak attack is exploit the signature verification algorithm. We claim that the underlying image verification algorithm is correct, meaning that an attacker cannot forge a signature without having access to the key without brute-forcing the cryptography. The private key necessary to generate a valid signature is stored in the entity that generates signatures, which we assumed could store the key securely. Additionally, we do not consider side-channel attacks, which could have been used to extract the key. Thus, the only remaining option for an attacker is to brute-force the signature algorithm, which is highly unlikely to succeed in a reasonable amount of time, by design. It is thus highly unlikely that a jailbreak attack is successful in this threat model.

DoS

Now, for a DoS attack to be successful, the device has to be unable to boot during phase 2. Since a factory reset is done between both phases, it is not sufficient for an attacker to brick the device during phase 1, they have to corrupt the state of the device during phase 2. One way to brick the device is to install a faulty image that would erase the secondary slot as well as itself, effectively corrupting the state of the device. However, we claim that this is unlikely as this reduces to performing a jailbreak attack, given that the DoS attacker can do strictly less actions that the jailbreak attacker.

Added to the fact that any way to influence the bootloader (other that modifying the content of both image slots) is not possible in this model, the only remaining way to successfully execute a DoS attack is to try to corrupt the state of the device by resetting it during updates. However, we claim that MCUboot is designed to be resilient to resets. It should always recover from resets and resume an interrupted update, as long as the non-volatile memory is not modified by anything other than MCUboot while the update is not fully applied or refused. So a DoS attacker has to modify the non-volatile memory during this time in order to successfully perform the attack. There are two ways to modify flash memory using the actions available to the attacker during phase 2: **factoryReset**() and **installImage**(*image*). Although **factoryReset**() indeed technically modifies the content of flash, it will fully restore the device to its initial valid state. Thus, the state right after a factory reset is the same as in the beginning of phase 2, so this will not help an attacker. On the other hand, **installImage**(*image*) is defined to do nothing if an update is ongoing, it can thus not be used to to corrupt the state of the device during an update. Therefore, it is unlikely that the state of the device gets corrupted by a DoS attacker during phase 2.

IP Theft

Lastly, we consider the case of an IP theft.

We claim that the underlying cryptographic algorithm used to encrypt images is highly unlikely to be broken, unless the attacker directly gets access to either the keys or the plaintext. We thus consider that to successfully perform an IP theft, an attacker is required to observe in some way a private key (either k_{enc}^{priv} or an intermediate secret) or the target image plaintext. The private encryption key, k_{enc}^{priv}, is stored securely in the secure user area, and the board's manufacturer states that this area is completely inaccessible once the software exits the secure user area. Thus, k_{enc}^{priv} can only be read by the bootloader. The attacker cannot influence the bootloader or access the internal memory where k_{enc}^{priv} would be copied if necessary. Furthermore, the key or any other intermediate product leave no trace as they are flushed from the RAM afterwards.

Therefore, the only way to successfully perform an IP theft is to get some knowledge on the target image during phase 2, without having access to any cryptographic secret. One solution would be to read the content of the image, either during its download, when its rests in storage or while both image slots are being swapped. However, the images are always produced in an encrypted state, so the attacker cannot observe its content during its download. Additionally, the attacker has only access to external memory, which means they can only observe the secondary slot. Since the content of the secondary slot is always encrypted, the attacker cannot observe the image plaintext at rest. The lack of access to internal memory during phase 2 also means that the attacker cannot observe the image plaintext in internal memory while it is being decrypted. Finally, we claim that MCUboot does not store any sensitive information in the secondary slot at any point of an update.

This means that the attacker cannot directly observe the content of the target image at any point of phase 2. So they have to infer information about an image by using only image ciphers. Since two different image are encrypted with AES-CTR using two independent ephemeral keys, the attacker cannot use the knowledge of any other ciphertext either to infer sensitive information about one of the images. Therefore, the attacker cannot do better than try to decrypt the image by brute-force or guessing randomly whether the image is valid or made of random data, which is considered as secure.

4.3.2 Secure Bootloader Correctness Verification

We will now try to assess the correctness of the implemented bootloader functionality-wise.

Methodology

The correctness of the bootloader is done by using tests to verify the properties, as we will explain here. Note that we are not interested in testing the MCUboot library itself, as it is already actively tested by its developers. We want to test the project as a whole, running on a development board.

Goal

The best-case scenario to prove the correctness of the software is to have exhaustive tests verifying every possible scenario directly on the board, while using a final production build. Unfortunately, two problems quickly arise. Firstly, it is not realistically possible to exhaustively test every possible scenario, as every combination of input images should be considered. The bootloader must be robust to resets, so the tests would need to reset the device at every possible step of an update, for each state of the flash memory, which would quickly lead to an explosion of the number of test cases. Since we want to test the software on the device itself, we are heavily limited by the speed of the flash memory, which would need to be rewritten many times, possibly between each tests. Hence, we will not exhaustively test the bootloader but rather try to find a relevant subset of tests that should cover interesting cases.

Secondly, since we want to want to perform tests in a realistic environment, the board security should be active. This means, in particular, that a debugging tool cannot be connected to the board in order to control or monitor its execution. In the case of the STM32H753, it is not even possible to program the flash directly while the readout protection is enabled. Only a running software can write to the internal flash. Moreover, the tests should also verify that the bootloader refuses to boot when no valid image is available, but that would imply that the device is locked in a state where no software can download an image to repair the state of the device. In other words, it should not be possible to test in a real-world situation that the bootloader refuses to boot without rendering the board unusable. For this reason, we choose to include a way to download images, monitor the state of the device and influence its execution directly from within the firmware. This way, we can simulate to some degree a debugging tool, even when the board security is enabled. A consequence of this is that we are not testing a real-world version of the bootloader, we will test a slightly modified version that is not compliant with the threat model (since it should not be possible to influence the bootloader). We however try to keep the changes to the bootloader itself minimal in order to avoid introducing or hiding bugs. All changes are only included in the code during a test build, in order not to unnecessarily pollute the code.

Test Helper Program

The interactions with the system will be done by a test helper program, which contains a tool that acts similarly to the utility BIOS. This means that the test helper menu will communicate via UART, print information and await commands. We will thus use a python script on the computer which will communicate with the board. It will issue commands to prepare the state of the images, execute the tests and recover the results. Since we want to avoid encumbering the bootloader itself, the test helper tool will not be included directly in the bootloader, but rather stored in flash elsewhere as a stand-alone application. One needs to remember that images are likely to be moved around during tests, in the primary, secondary and even scratch area, so the test helper cannot be in any of those. For this reason, the test helper tool will be stored in the only flash area that is never used by the bootloader: the user storage.

The test application contains a public interface to call a minimal test menu from anywhere while displaying information about the place from which the menu was launched. It contains some commands to reset the board, jump to the test helper image or exit the menu and continue the execution. The advantage is that this menu can be called from multiple places within the bootloader, which allows us have some control over the execution of the bootloader. In particular, the bootloader will call the minimal test menu at boot, when an image is validated, when the bootloader refuses to boot, in the hard-fault handler and twice during the update protocol. Each test menu identifies itself by printing a fixed sequence of two characters "$>" followed by a unique character identifier which indicates the place from where the test menu was called. This way, a python script communicating with the test menu can trace the execution path and check that it is as expected, or detect that a test is unsuccessful. Finally, the test script will issue commands to reset the board and upload images in the primary and secondary slots as required by the next test, before each test.

The test image itself contains a more complete version of the test menu which allows to extensively manipulate the state of the memory in order to prepare the tests. The list of available actions is the following:

- Trigger a hard-fault. This is useful to validate that the test menu can be called from within the hard-fault handler without issues.
- Corrupt the start of the primary, secondary or scratch area, using the modified flash driver that can issue partial write commands. This means that any read command issued to the corresponding area will issue a double ECC error, effectively triggering a hard-fault if the driver does not explicitly handle the error, and this error will persist even after a system reset, until this sector is erased. A systematic way to reproduce double ECC errors was desirable as this kind of error has been encountered during development and was particularly hard to reproduce or debug otherwise. We can now easily make sure that the bootloader can handle corrupted flash sectors and treat them as if its content was empty, without causing the board to crash.
- Read the content of the secure bootloader storage. This is necessary to check that no software outside of the bootloader can access the data contained in this area.
- Download an image on the board and store it in flash memory. This is similar to the feature presented in the utility BIOS in *Section 4.2.5*, but the current one can also store it directly in the primary slot. This feature was not included in the utility BIOS since the primary image should not be modified directly in general, but the test procedure will require to prepare different images in both slots.
- Mark the image in secondary slot for upgrade. This is necessary to verify that updates can be applied.
- Issue a write command to the bootloader area. This is useful to check that the bootloader area is indeed immutable. To avoid corruption of the device (and thus avoid further crashes

during the test procedure) in case the bootloader is not immutable, this function will try to first read a flash word and write back its content at the same address. This write command should still fail, even if the content of the word is the same.

- Erase the content of the primary or secondary slot. This is useful for tests that expect a specific slot to be empty.

Tests

Using the test helper tool and a python script, it is then possible to encode each test as a sequence of characters representing the commands that need to be issued as well as the identifiers of the test menus that the bootloader reached. For example, consider a test that simply checks that a valid image can boot. The test script will start by issuing a reset command, to ensure the bootloader executes from its beginning again. The bootloader should then reach the initial test menu again, and will indicate this. Then test script now needs to prepare the state of the flash memory: a valid image should be loaded into the primary slot, and the secondary slot should be empty. It will thus ask the device to jump to the test image, and the device will then print the test image menu. The test script will thus request to upload an image to the primary slot and send it. Still in the test image menu, the test script will request the secondary slot to be erased. Finally, it will request for a system reset, to get back to the bootloader, after what the bootloader will indicate again that the initial test menu has been reached. The test script will request that the bootloader continues its execution, so that MCUboot can execute and decide whether the image in primary slot is valid. Finally, the bootloader should indicate that it reached a point where it is going to execute the image, since it is considered as valid.

At any point during a test, the test script will abort the current test and report a failure if any unexpected response is received. Otherwise, the test is considered as successful if it could execute entirely without any mismatch. The list of main tests that are performed by the test script is the following:

- The bootloader secure storage should not be readable from outside of the bootloader.
- The bootlaoder area should not be writable.
- The hard-fault command should trigger a hard-fault. This test mainly serves as a validation of the fault handler menu.
- The bootloader should hang when both primary and secondary slots are empty.

Then, some tests are repeated with different combinations of varying parameters. Those parameters include the corruption of one or more flash sectors, and the number of resets that are issued during any update protocol. For each different combination considered, the following tests are performed:

Table 4.1: Size of different softwares. All sizes represent the size that the final firmware will take in flash when it is programmed on the device

Software	Bootloader (debug)	Bootloader (production)	*hello_world*	*FreeRTOS*
Space [kB]	101	85	52	241

- A valid image in primary slot should boot.
- An invalid image (signed with the wrong key or with a single byte modified after its signature) in primary slot should be refused
- A valid encrypted image in the secondary slot should be applied and executed
- An invalid image (signed with the wrong key or encrypted with the wrong key) in the secondary slot should not be applied.
- An update should be reverted successfully to recover the initial primary image.

4.3.3 Space And Time Requirements

We shall quantify here the flash memory space taken by the bootloader as well as different execution times.

Table 4.1 presents the size of different softwares. The size of softwares represent the amount of non-volatile memory necessary to program the software on the device. Two different versions of the bootloader are shown. The first one is a debug version which included the utility BIOS and an UART interface. This version corresponds to a version used during development. The second is a production version which does not communicate with the user and has no utility BIOS. This version corresponds to the final version that should be used to satisfy the system model. The `hello_world` image is a minimal yet fully functional image (i.e., capable of configuring the board security and downloading updates). The `FreeRTOS` image is an image that contains the `FreeRTOS` operating system, inside of which is a simple application similar to `hello_world`.

Table 4.2 presents the time necessary to authentify images or apply updates. All measures were taken with a logic analyzer. The measures represent the time between the end of the initialization of drivers and the end of the bootloader code. Each measure was taken five times, so the table contains the average measure, along with the standard deviation in parentheses. For each image, two sets of measures are shown, one representing the time necessary to authentify an image after a reboot, and one representing the time necessary to fully apply an update that is already in secondary slot. Additionally, measures show the execution time in two different setups. The first one corresponds to the execution of the bootloader when a software-only implementation of the cryptography is used, and the second one implements cryptographic acceleration using the available hardware modules (SHA256, HMAC-SHA256, AES-CTR).

Table 4.2: Execution time of the bootloader. All measures were taken 5 times. Table shows the mean of measures. Numbers between parentheses show the standard deviation.

	SW-only cryptography				HW-accelerated cryptography			
Image	Boot time [s]		Swap time [s]		Boot time [s]		Swap time [s]	
Hello world	2.173	(0.001)	17.798	(0.018)	1.982	(0.000)	12.654	(0.031)
FreeRTOS	3.031	(0.001)	38.722	(0.022)	2.027	(0.001)	16.397	(0.021)

4.3.4 MCUboot Port

We will now evaluate MCUboot itself. In particular, we will discuss the changes that were done to the MCUboot library during the implementation of the bootloader, and the challenges that were encountered to make MCUboot compatible with the STM32H753 MCU.

Changes

The MCUboot library advertises itself as being hardware independent, but it is not fully compatible with the STM32H753 due to its flash configuration, so some changes had to be done on fork of MCUboot to make it compatible with our bootloader. Some of the changes were proposed by [10].

Compatibility

Firstly, the minimal size with which MCUboot tries to write to flash memory could not take a value higher than 8 bytes [11]. This is incompatible with the STM32H753 since the latter one has a flash word size of 32 bytes, and a flash word should never be written in multiple parts. Therefore, MCUboot's maximal flash alignment was increased to 32 bytes. This leaded to a few more changes that still needed to write values shorter than 32 bytes. The image generation script parameters were changed to reflect the fact that it should now also accept this new value. Similarly, MCUboot uses an image magic number on 16 bytes to identify images, so this value was modified to a 32 byte value. The same value was modified in the image generation script, which needs to write the image magic number in the header of an image.

Another detail that was causing issues during development is that MCUboot writes some intermediate values in an image trailer in internal memory in order to avoid recomputing them in the future. The problem is that MCUboot was trying to write two 16 byte values consecutively without consulting to the current flash alignment in the `boot_write_enc_key` function, meaning that this could corrupt the flash memory by writing twice in the same flash word in our case. This function was thus modified to write the two values at the same time, in a single 32 byte flash write. If the flash alignment ever takes a value greater than 32, this functions should be modified

to add padding.

Implementation Of The Bootloader

After those changes, most of the remaining work around MCUboot consists in building the bootloader, implementing the required drivers and making the link with MCUboot. Unfortunately, MCUboot has very few instructions on the porting procedure. It does not include a template makefile that can be used as a reference, or a simple project template either. For this reason, we had to look for online examples and craft a makefile and project structure by getting inspiration from those online examples. To fully port MCUboot, one first needs a makefile that includes all files of the MCUboot library that will be used (depending on the configuration, some files are not required). Additionally, the cryptographic library used by MCUboot is external, so one also needs to include files from the chosen library, TinyCrypt. Even though the choice can be either TinyCrypt or Mbed TLS and TinyCrypt was chosen, some source files of Mbed TLS that are in principle not part of the public interface had to be included in order to successfully compile the project.

Additionally, MCUboot requires the project to follow a certain folder organization and to define and implement multiple functions and types. Since the complete list of required definition is not part of the documentation, the best way to get a bootloader to compile was to progressively solve compilation errors by including new files and definitions as they appear and look for exterior resources. In particular, an online example of an MCUboot port [5] was used as a reference, since it listed almost all required definitions to fully implement a bootloader using MCUboot, as well as a complete makefile template that we reused and adapted. The MCUboot interface may have evolved a bit since the time the article was written as we had to fix more compilation issues that were not listed.

In general, this part looks not documented enough as we had to look for other resources and learn by ourselves what is required to simply get a project to compile, with all dependencies satisfied. Given that the list of functions or constants that need to be defined is almost always the same (apart from a few ones which vary depending on the configuration of MCUboot), it would be fairly easy to include a template project to MCUboot, to avoid letting users learn by themselves what is required to port MCUboot, or at least to include a complete list of required definitions somewhere. Some were mentioned in the documentation, but information about this are scattered and incomplete.

Hardware Cryptography

As mentioned earlier, the default configuration uses a software implementation of cryptographic primitives, but MCUboot is designed to make it easy to plug in a custom implementation of the cryptography. Indeed, all functions of a given algorithm (e.g., SHA256) called by MCUboot are defined in a corresponding file, whose purpose is to act as a multiplexer for all available

implementations of the cryptography. It is then fairly easy to add another implementation; all that is required is to add a check for a custom flag and redirect to our own driver if the variable is defined. It is worth mentioning that using a hardware implementation of the cryptography uncovered a slight bug in MCUboot [12]. If the system is reset during the swapping operation of an update, it is resumed after booting again. However, it turns out that the underlying cryptography is never initialized while the swap is being resumed. This causes the underlying cryptographic module to be used before its initialization, which makes it unable to function properly. The bug has thus been fixed on our fork of MCUboot and reported in the original repository. It is likely that the bug was not discovered since the default software implementation of the cryptography does not need to initialize anything and functions properly even when it is never initialized.

Additional Changes To MCUboot

Finally, a few more changes were made to the MCUboot fork for convenience. Firstly, we added an option to the image generation script. The original script can generate images designed for the first or secondary slot, which means it can leave an image in clear, or encrypt it, depending on our needs. However, an image in clear (i.e., the state in which it is after being applied in primary slot) needs to retain information about its encryption in order to be re-encrypted by the bootloader using the key that originally encrypted it. While applying an update, this information is usually kept in the trailer of the image. The image generation script, however, does not include any encryption information in the trailer of an image when it is in clear. In other words, an image generated in clear and stored directly in the primary slot cannot be encrypted by the bootloader during an update. This is problematic since we might want to manually install the initial image directly in primary slot during the setup phase. As soon as another image is installed, the initial image will sit in clear in secondary slot, which can be in external memory according to the system model. If the image contains sensitive data, they can easily be recovered since they are in clear in external memory. We simply fixed the image generation script by adding encryption information in the trailer of an even if the image content is initially not encrypted. This way, the initial image can be directly installed in the primary slot and still be encrypted while being in the secondary slot.

Additionally, MCUboot made a few assumptions on the validity of the image in primary slot under certain circumstances. More specifically, it always assumed that a valid image is stored in the primary slot when it tried to read its header to prepare for an update. In practice, this means that if a valid update is detected in the secondary slot, the primary slot is treated as a valid image and its encryption is prepared for the swap, even when it does not contain an image, which leaded to unexpected behaviors. This has thus been fixed by first checking that the primary slot header corresponds to an image header, before trying to interpret its content. This verification has been added at a couple places of the MCUboot fork. More concretely, the bootloader is now capable of applying an update without crashing if the primary slot does not contain an image, which is the entire purpose of also using the secondary slot as a backup image. Otherwise, a single corruption in the header of the primary slot would completely prevent the bootloader

from booting ever again.

Finally, in order to allow the test script previously mentioned to reset the board at multiple steps of an update process, hooks to the test menu were added in a couple places of the MCUboot fork, but only if a test variable (`ENABLE_TEST`) is defined.

Possible Improvements

It should be noted that this project only uses a specific configuration of MCUboot, and a significant part of the library is never used. This means that the bootloader, along with all changes done to the MCUboot fork, might not work with other configurations, or might even have introduced incompatibilities. A better solution would be to try to integrate the changes directly to the MCUboot repository. Except for a few changes that are specific to this project, most of the changes should be done in a pull request so that other people can verify that no issues were introduced in the process. This way, a compatibility with STM32 devices could even officially be added to MCUboot. That might however require significant work as many configurations were not tested in this project an all incompatibilities an issues that might appear should be fixed.

Another possible improvement to the bootloader is linked to the way MCUboot swaps both image slots during an update. Currently, the swap is done by physically swapping data by copying them using a third temporary scratch area. This requires a lot of flash operations and is currently the most time-consuming part of an update in our case. It should be possible to significantly accelerate this process by exploiting the available hardware. The STM32H753 microcontroller has an internal flash memory that is equally split into two parts, called *banks*. The flash controller includes a feature called *bank swap* that allows to quickly virtually swap both banks without physically moving the data. It is then imaginable to implement a new swap strategy in MCUboot that would exploit this mechanism to quickly apply updates without requiring to copy a lot of data. This would certainly, however, be incompatible with the current layout and require changes to the way the system uses the flash memory.

4.4 Instructions

Let us now explain how to install and use this project. For a more technical description, please refer to the documentation located in the bootloader repository.

4.4.1 Prerequisites

Firstly, some tools are required to install the project and deploy it on a microcontroller. The list of required components is the following:

- A compatible embedded compilation toolchain. It is necessary to compile the project for a microcontroller. For the STM32H753, the GNU Arm Embedded Toolchain was used for this project as it supports Cortex-M7 processors.
- A compatible microcontroller. The only microcontroller supported by this project is the STM32H753, but others might be added by creating another port as described in *Section 4.4.4*.
- A tool to program the microcontroller flash memory. This will be used to program the bootloader and the first image into the board. For this project, OpenOCD was used, since it supports our microcontroller.
- A tool to communicate with the device via UART. This is not necessary strictly speaking, but it is very useful for the development and required to test the project.

If the platform is not currently supported and requires a new port, the new microcontroller needs to have at least the following features in order to be compatible with the current:

- It should have internal flash memory, and the available flash memory should be sufficient to store the bootloader, two entire images. Additionally, it should have at least a single flash sector remaining, along with some space that an image would need to store information, depending on the needs of the image.
- It needs some memory protections that matches the system model. More precisely, it needs a way to make the bootloader area immutable and to make the bootloader secure storage unreadable once the bootloader is done executing. Additionally, it needs to have a single boot entry point, which is the bootloader. Those mechanisms are requires to have a root of trust and to store secret keys securely.
- It also needs a way to disable debug features such as having access to the internal memory from outside of the microcontroller, or the ability to influence or monitor the state of the processor during the execution of the bootloader. This is necessary to match the system model, as most of the security becomes pointless if it is easy to control the bootloader directly from the outside.
- At least 16KB of RAM is necessary to execute the bootloader. This requirement is mentioned for completeness, as any microcontroller is likely to need RAM for any software anyway.
- The microcontroller needs some canal to download an update. This can for example be a UART driver (similarly to what this project uses), but it can be any physical way to reliably download data.
- Optionally, a UART interface is preferable to debug the port, as it is the primary intended way to interact with the bootloader without using a debug interface during its development. This is however not strictly necessary.

4.4.2 Installation

Let us now explain precisely the procedure one should follow to successfully install the bootloader.

First of all, you need to clone this repository. Additionally, you should pull submodules using `git submodule update -init -recursive`.

You'll then need to choose a port (i.e., platform for which drivers are implemented). You can either use the example port for STM32H753, or you'll have to implement your own (see *Section 4.4.4*). If you use STM32H753, an additional step is required to use the port.

To safely store keys on the STM32H753, we rely on the secure area. This requires two functions, `resetAndInitializeSecureArea` and `exitSecureArea`. Unfortunately, the code necessary to setup and use the secure area is under ST's license and cannot be shared in an open-source environment. For this reason, you will have to request ST's Secure Boot and Secure Firmware Update (SBSFU) yourself [13].

You should then create two files, `STM32H753/src/st/rss.c` and `STM32H753/Includes/st/rss.h`. They should contain the prototype and definition of two functions: `void resetAndInitializeSecureAreas(void)`, which will configure the secure area to correspond to the bootloader area (no more, no less), and `exitSecureArea(uint32_t vectors)` which should close the secure area and use the input vector table to jump to the application. More information can be found in the reference manual [2, p. 249].

Setting Up Keys

Then, cryptographic keys need to be setup. This includes an ECDSA key pair to sing the images and an ECIES key pair to distribute encryption keys. The keys used by the makefile to generate images should be in a `.pem` format, while keys embedded in the bootloader should be in a compatible `.c` format. For simplicity, the keys can simply be generated using the bootloader's makefile, by executing `make generate_keys OUTPUT_DIR=[some output directory]`. Alternatively, example keys are available in the `Bootloader/Common/keys_template` folder. They can be used in a development build (but should never be used in a production build).

Setting Up project_config.mk

You need to setup a `project_config.mk` file that will contain all information that are common to a given project (i.e., an instance of the bootloader and all compatible images, on a given device). It is used to make the link between generic files and port-specific or project-specific files. For STM32H753, `Bootloader/Common/project_config_stm32h753.mk` can be used as-is. Otherwise, the required variables are described in *Appendix A.1*.

To indicate which `project_config.mk` file will be used by the bootloader and images, the `PROJECT_CONFIG` environment variable should contain the path to your `project_config.mk` file. This can be set, for example, by using `export PROJECT_CONFIG=[some path]` in your shell.

Subsequently, three more files which are referenced by `project_config.mk` will need to be created. `bootloader_config.mk` and `test_config.mk` will contain the list port-specific source and header files required by the bootloader (resp. the test image), as well as any additional compilation option. They are described in *Appendix A.3*. For STM32H753, the provided `Bootloader/bootloader_config_stm32h753.mk` and `Bootloader/tests/test_config_stm32h753.mk` can be used as-is.

Additionally, a `sections_config.ld` linker file will contain the size of different MCUboot regions, as described in *Appendix A.2*. For STM32H753, `Bootloader/Common/template/sections_config.ld` can be used directly.

Building

Finally, you should successfully compile the bootloader by executing `make` inside the `Bootloader` directory.

4.4.3 Creating A Compatible Image

Now, you need an image that can be verified and run by the bootloader. A few images are already included for inspiration (`hello_world`, `FreeRTOS-Image`). Please note that the provided images are port-specific. They can be ported to other platforms, but their makefile should be adapted first.

Requirements

To begin with, an image needs a makefile, a startup file and a linker script. The template startup file `Bootloader/Common/src/startup_template.c` can directly be used. Similarly, the template linker script `Bootloader/Common/link_template.ld` is directly usable.

Note that the images must be defined to run from the primary slot, which means that their code must be placed in the section called `IMG` in their linker scripts. Additionally, the first words of the `IMG` region should be the interrupt vector table. If you use the template file `Bootloader/Common/src/startup_template.c` file but use your own linker script, the latter one must define that the `.isr_vector` section is placed at the beginning of the `IMG` area. Note that the bootloader will always jump to an image by executing its reset hanlder, so your reset handler should call your `main`.

Additionally, the bootloader will look for an image-specific configuration file that dictates the logging properties. For that reason, your image should include a file named `mcuboot_config/app_config.h`. Its content is described in *Appendix A.5*.

Image Makefile

To compile the image, it is possible to take advantage of the common makefile, `Bootloader/Common/Makefile`. It can take care of compiling from scratch and will provide targets necessary to generate signed and encrypted images, as well as targets to flash them, as long as some parameters are given.

First of all, a new image-specific makefile should be created. It needs to import the project configuration by using `include $(PROJECT_CONFIG)`. This way, it will have access to shared variables defined in it. Then, it can include the common makefile by using `include [path_to_git_folder]/Bootloader/Common/Makefile`. The list of required parameters can be found in *Appendix A.6*.

After that, the image can be compiled via the common makefile, by simply running `make`.

Alternatively, the makefile can be written from scratch, or by using your own (if your image already has one). However, some prerequisites must be filled to make your image compatible with the bootloader:

- `PROJECT_CONFIG` must still be included, as in contains variables that must be shared with the bootloader
- The linker script should include `memory-sections.ld`. You may also need to tell the makefile where to find this file, along with `port.ld` (in your port folder) and `sections_config.ld` (defined in `PROJECT_CONFIG`), which are both required by `Bootloader/Common/memory-sections.ld`. This can be done by adding `-L$(path to the parent folder of a .ld file)`, for all three of them, to the compilation flags of your makefile.
- The image should be defined to run from the primary slot, meaning its code should be contained in the `IMG` area (symbol defined in `Bootloader/Common/memory-sections.ld`)
- More specifically, the interrupt vector table should be placed in the very beggining of the `IMG` area.

Using Common Drivers

The image can implement its own drivers, but it can also include the common drivers that the bootloader already uses. Since some common drivers might be incompatible with your image, it

is possible to individually include or exclude drivers. Firstly, the desired driver source files should be included in the image makefile (as described above). Those are the source files of your port, as the common part of the drivers have no source files.

Then, the makefile should indicate which drivers should be included. There exist a variable for each driver that tells the common files whether this specific driver file should be empty or should contain definitions. By default, all drivers will be empty; including their source files is not sufficient to use them. This is described in *Appendix A.4*.

Including drivers individually might be useful because drivers may be inter-dependent, so including a driver X might automatically include driver Y, which might not be desirable. For example on the `FreeRTOS` image on STM32H753, the `security` drivers need to be enabled in order to use the BIOS, but this requires `uart` drivers, which is a problem since `FreeRTOS` comes with its own UART drivers. This image could thus only include the `security` drivers by setting `ADDITIONAL_CFLAGS += -DHAL_INCLUDE_SECURITY`.

Consequently, incompatibilities might appear when not using some drivers that are necessary (e.g., not including the UART driver but enabling MCUboot's logging). The missing definitions should then be defined by your image itself (this will effectively make the link the bootloader and your drivers). For this reason, *if any driver is not included*, it is necessary to add a `hal/hal_app.h` header file to your image. This file is automatically included by the common files and can then include your own definitions of missing functions or variables, which will be usable by the rest of the project. For example, it is possible to exclude the Flash drivers, in which case `hal/hal_app.h` will include its own implementation of the same Flash interface, or include another file that does exactly that. You can have a look at the FreeRTOS example (`FreeRTOS-Image/App/include/hal/hal_app.h`) for a concrete example which makes the link with its own Flash and UART drivers.

Update Capability

Finally, any image should be capable of downloading an update. This is because the bootloader should not download an image itself and this responsibility is left to the image.

More precisely, any deployment image should ***always*** include a way to download an update, no matter which way, and store it in the secondary slot of the Flash memory.

To obtain the boundaries of the secondary slot (or any slot, for that matter) in a source file, one should include a variable that is defined in the common linker scripts, for example via `extern uint32_t [symbol_name];`. It is then possible to use its value with `(uint32_t)&[symbol_name]`. The beginning of the secondary slot is defined by `_stImg2Header` and it has a maximum size of `_lnImg`.

Thus, an image should always include a way to download an encrypted image (like

the one freshly generated by the common makefile, i.e., `$(BUILD_DIR)/$(TARGET)_enc.bin`) and store it in the secondary slot (while checking its boundaries, if necessary). Lastly, the update should be marked for an update in order to be processed by the bootloader. This must be done by calling the `boot_set_pending(1);` function, defined in `$(PATH_MCUBOOT)/boot/bootutil/src/bootutil.c` (which should thus be included in the makefile). The update will then be processed after the next reset.

Alternatively, this can be ensured by including the utility BIOS in the image, since it is fully capable of downloading an image via UART and marking it for update, as mentioned in *Section 4.2.5*. To include the utility BIOS in an image, refer to *Appendix A.7*.

4.4.4 Porting

Let us cover how the bootloader should be ported to a new platform now.

Port Folder

As mentioned earlier, all of the files that are port-specific are regrouped in a single `port` folder. As many definitions and files are required, they are not explicitly listed here, but rather given in the form of the template port, `Bootloader/Common/template/port_template`. To implement a new port, one should thus copy this folder to a desired location and fill it accordingly. Each file or folder should be filled as follows:

- `stlink.cfg` will be used by OpenOCD to program the bootloader and images. It should contain every configuration necessary so that a program can be flashed as follows: `openocd -f $(PATH_PORT)stlink.cfg -c "program [...]"`.
- `port.ld`, which contains information about the device memory layout. All symbols defined in the template port must also be defined in any other port.
- `lock.mk`, which can contain memory protection-related targets. This file can be left empty as it is not used directly by the bootloader or any image, but it is included by default and can be used from the bootloader's makefile. Therefore, adding an `X` target in `lock.mk` makes it possible to issue a `make X` in the bootloader's makefile. It is intended to contain targets that can be used to set or unset the device protections, or monitor them.
- `Includes` and `src`, which contains many definitions and functions. Every function, constant or macro defined in any of their files must contain a valid body. Please refer directly to an implementation in the `Bootloader/Common/template/port_template` folder to get a more detailed description of the expected content.

Changes To MCUboot

In case the included MCUboot fork is not compatible with a given device, you may fork it and implement the necessary changes. You will then simply need to set the value of the `PATH_MCUBOOT` in the `project_config.mk` file accordinly.

Additionally, if you want to use some hardware cryptographic module, you will need to modify MCUboot so that MCUboot uses it. In particular, the location when the choice of the underlying implementation for cryptographic primitives is made in the `$(PATH_MCUBOOT)/boot/bootutil/include/bootutil/crypto` folder (`aes_ctr.h` for AES-CTR, `sha256.h` for SHA256, `hmac_sha256.h` for HMAC_SHA256 and `ecdsa_p256.h` for ECDSA using p-256). As those files work as multiplexers, you can for example use a new flag and modify those files to use your own implementation in case it is defined (like it is done in the provided fork, when `MCUBOOT_USE_STM32H753_CRYP`). In case you don't want to encumber those files, you can use intermediate files to link MCUboot's interface with your own, like the `$(PATH_MCUBOOT)/ext/stm32h753` folder does in our case.

4.4.5 Usage

We shall now explain how the secure bootloader should be used. Please note first that the plaftorm-specific security necessary to correspond to the system model do not need to be enabled during development. It only needs to be enabled in a deployment build (as described in *Section 4.4.6*), or for testing purposes. During development, the utility BIOS and the logging should typically be enabled in the bootloader.

Bootloader Installation

To use the bootloader, the first step is to install it on the device. Once it has successfully been compiled with the default makefile target, it is directly ready to be installed on the device. This can simply be done using the `make flash` target of the bootloader's makefile, which will use OpenOCD to program the board. It will automatically be placed at the beginning of the flash memory, in the bootloader area. It is possible, at any time, to connect to the device via UART with a tool like CoolTerm to monitor the execution of the bootloader, if the logging is enabled. Additionally, the utility BIOS (if enabled) can be launched depending on the value of `$(PATH_PORT)/drivers/Includes/gpio.h:GET_USER_INPUT_STATE()` while resetting the device. The utility BIOS expects singe-character responses, without terminating characters.

Image Installation

Once a compatible image is implemented and compiled, it is possible to directly flash an image in primary or secondary slot, without downloading it from an image, for development

purposes. For this, two targets are defined in the common image makefile (and can thus be used from the image makefile, it includes the common image makefile). They can be used by executing `make flash` and `make flash_sec`. The `flash` target will install the signed image in the primary slot, while `flash_sec` will install the signed and encrypted image in the secondary slot.

Testing

Lastly, some tests can be run to validate the functionality of the bootloader, but only *once basic functionalities of the bootloader have been manually tested* by validating manually that it can at least verify basic images, apply simple updates and download an image in secondary slot. First, `make install_tests` should be executed to compile the bootloader with the test flag and install it. Then, the test script can be run by executing `python3 Bootloader/scripts/test_main.py`. Finally, resetting the device will initiate the communication between both sides and start all tests.

4.4.6 Deployment

To deploy the bootloader along with a valid image in a real-world situation, a certain number of steps should be followed. One should make sure to follow them carefully, as according to the system model, the system will become immutable and impossible to correct if a mistake is made.

Verifications

First of all, the BIOS should be completely disabled in the bootloader. As mentioned earlier, the BIOS is useful during development but introduces vulnerabilities regarding the threat model. Additionally, you should make sure that tests are successful and that everything works as intended. Any bug left in the bootloader is irreversible. Since the keys will not be modifiable, one should also make sure that the private encryption key and the public signature key are both installed correctly and that no production or temporary key was left on the device. One should also verify that the secret key cannot be extracted, neither by reading memory from an image nor directly with a debugging tool. Similarly, the bootloader's configuration (e.g., the logging properties, the flash layout or the use of hardware-accelerated cryptography) will become permanent and should be verified entirely. In particular, it will not possible to increase the maximum image size afterwards.

Additionally, every image (including all future versions) should be capable of downloading an update to the secondary slot and marking it for upgrade. If an image that cannot download an update is ever installed, it is possible that the device gets locked in a state where it becomes impossible to replace the image in primary slot, and updates are effectively disabled indefinitely. Lastly, in case you need to enable the device security directly from software, you should make

sure that a first setup image is capable of doing so. You could simply enable the utility BIOS in the first image and make sure that the port-specific menu can enable the security.

Locking The Device

When everything has been verified, you can now setup the device. The bootloader should be flashed using `make flash` in the bootloader makefile, and the first setup image should then be flashed in primary slot using `make flash` in the image makefile. The bootloader should now launch the image. The next step is to enable the board security. This can be done either manually, or using the installed image, depending on the system. The first image can now be replaced by a "real" image whose features are the one the final product should have, and the capacity to enable the device security can be dropped. To install an image, proceed as usual: use the first setup image to download an update and mark it for update, then let the bootloader apply this update. At this point, the first setup image should be encrypted and in the secondary slot. It is thus still possible to apply it back using the primary image. If the setup image contains any sensitive feature, it should be overwritten by downloading anything with the primary image. This way, it cannot be applied back while it sits in secondary slot.

STM32H753

On the STM32H753, the procedure to lock the device is the following:

- *Enable the secure mode* by executing `optreg_set_security();` or by choosing the corresponding option in the utility BIOS.
- *Set the secure area* by executing `resetAndInitializeSecureAreas();` or by choosing the corresponding option in the utility BIOS. This will trigger a system reset.
- *Enable RDP1 or RDP1*. RDP1 can be enabled by executing `optreg_enable_rdp1();` or by choosing the corresponding option in the utility BIOS. This will cause the processor to stop executing until a power-down reset is performed. To enable RDP2, modify the drivers manually to enable it. The exact procedure is not describe here to avoid accidents and the reference manual[2] should be followed.

4.5 Summary

In this chapter, we presented the evaluation of our implementation of a secure bootloader that uses MCUboot, matched against our threat model. More specifically, we mapped the implementation to the threat model in order to later formally assess its security. We also presented a testing methodology that can be used even when the board security is enabled, in which case all

debugging features that can be used to interact with the bootloader are disabled. This methodology was used to assess the correctness of the system. We also evaluated the porting process itself. Finally, as the bootloader is supposed to be easily reusable, we describe all information necessary to install, use or port the system to another device.

Chapter 5

Related Work

5.1 Attacker Models

In this section, we will briefly introduce three (increasingly powerful) attacker models that are derived from the literature. More specifically, the attacker models of most references can be matched with one of the attacker models we will describe here. Note that all references do not always explicitly define a threat model. In multiple instances, some kind of attacks (e.g., probing, direct access to memory, communication attacks, etc.) are implicitly covered without being mentioned, while in other cases the attacker model is simply non-existent. It was therefore necessary to add some interpretation in order to classify them. This was done by comparing them to similar work that possess a more descriptive threat model.

Logical Attacker

The first attacker model is a logical attacker. This attacker model is mostly used in early research and is rarely used since, as it does not fully capture more recent attackers. This model corresponds to an attacker which has no physical access to the device. It represents attackers that only have remote access to the device and that can install a malware by using some vulnerability (the details of which do not matter in this context, we are only interested in their capability to install such malware). In this model, the attacker is thus fully represented by a malware installed on the device. It essentially means that the attacker can temporarily control the CPU and modify the state of the device, until it is reset after which point the control is lost until the device executes attacker-controlled code again. The non-volatile memory is therefore considered as untrusted, as an attacker might have previously modified it. This model is useful because it captures the bare minimum that a secure bootloader should protect against: we want to detect that malicious code is present and prevent its execution. For this reason, all instances of secure bootloader in the literature cover this model of attacker, even if no explicit mention is made. Additionally,

other threat models that we will consider will always represent attackers that are strictly more powerful than a logical attacker.

Non-Invasive Attacker

A second more involved attacker model corresponds to a non-invasive physical attacker. This model is used more often in practice since it is more realistic than a logical attacker. Indeed, in the context of the IoT where devices are present everywhere and in multiple forms, it is easily conceivable that an attacker might physically access a device. For this reason, this attacker model allows an attacker to perform non-invasive physical attacks on a device. Simply put, that includes most attacks that do not physically harm the device, nor break the boundaries of its components (e.g., chips, CPU, etc.). In practice, that means that the attacker might influence data that flows exposed on the board, or communicate directly with components of the device (e.g., any external memory, which could thus be manipulated without going through the CPU first) by connecting probes or tables to them. This model is much more common as it is a good compromise between an attacker model that require involved defense mechanisms or very specific hardware and attacker model that do not capture enough reality. As such, it is the default attacker model that is implicitly used in most secure boots (which is also true for the references of this work), unless specified otherwise. This model relies on some security properties offered by the underlying hardware, but those are common enough nowadays.

Invasive Attacker

The third attacker model is a more aggressive model that corresponds to an invasive physical attacker. Like the previous model, the attacker as a physical access to the device, but this time they can also perform invasive attacks. Each invasive attack might require a specific kind of hardware defense that is not necessarily common. For this reason, this model usually only allows a few specific invasive attacks and explicitly lists them. Such attacks can for example include decapping a chip, replacing a chip or invasive side-channel attacks. This model corresponds to a skilled attacker that has access to specialized equipment. Added to the fact that defending against such attacks require specialized hardware, this means that this model is much less common in practice. Only high-end devices that require more security and can afford such hardware use this attacker model by advertising some kind of hardening against more advanced attacks. This includes some Laptops or Smartphones, or devices for which security is more critical.

5.2 Practices

Let us now go through an overview of multiple implementations of secure bootloaders. Table 5.1 summarizes a classification of all secure boot implementations that were encountered in the literature in this work.

Note that in this table, the required hardware always implicitly include a way to have a root of trust (e.g., an immutable unique boot entry point), since this is mandatory to implement a secure boot. *Threat model* denotes the threat model that we attributed to the corresponded work. *Logical* corresponds to a logical attacker, *Non-invasive* corresponds to a non-invasive physical attacker and *Invasive* corresponds to an invasive physical attacker.

5.2.1 First Instance

One of the first instance of a secure bootloader was proposed in *A secure and reliable bootstrap architecture* [29]. This paper presents itself as a practical real-world system. It introduces a secure bootloader architecture that became the standard. The secure bootloader (which is embedded in ROM and thus trusted) works by computing a hash of its image and comparing it against an fixed value. Therefore, it cannot support updates. Additionally, a trusted external storage is used to allow a safe recovery in case of corruption. In fact, this work can be seen as an early proof-of-concept of the now standard secure boot architecture. However, the authors explicitly states that the rely on the assumption that the underlying hardware is not compromised. This means that their attacker model corresponds to a logical attacker.

5.2.2 Academic

Many academic works then followed this early model of secure boot, each proposing their own implementation on a specific device using their own design.

On implementing trusted boot for embedded systems [28]

This paper is another example of a simple secure boot architecture. This paper describes a secure bootloader That uses an integrated FPGA. The FPGA is used to implement a secure co-processor. Like the previous examples, the authors explicitly states that physical attacks are not addressed in this design. The attack model therefore corresponds to a logical attacker. Additionally, the authors are aware of a time of check / time of use (TOCTOU) vulnerability.

A Trusted Bootstrapping Scheme Using USB Key Based on UEFI [27]

This paper took a different approach on the secure bootloader. The bootROM itself (which usually contains the entire secure bootloader) is kept minimal and one of its only goals is to authentify a secure USB key using a private key embedded in the USB key, then execute its firmware. This architecture is thus linked to a particular USB key, it is required to boot and any

Table 5.1: Collection of secure boot implementations in the literature

Year	Source	Required Hardware	Optional Hardware	Threat Model	Academic / Industrial
2021	*CARE: Lightweight Attack Resilient Secure Boot Architecture with Onboard Recovery for RISC-V based SOC* [14]	CAU	-	Invasive	Academic
2021	*Boot process for a Mac with Apple silicon* [15]	T2 chip Secure enclave	-	Invasive	Industrial
2020	*Secure Boot and TLS 1.3 Firmware Update with FreeRTOS and wolfSSL on NXP "Freedom Board" K64* [16]	-	ARM TrustZone	Non-invasive	Industrial
2020	*Knox White Paper* [17]	ARM TrustZone Warranty fuse Rollback fuse	-	Non-invasive	Industrial
2020	*Android Verified Boot* [18]	-	ARM TrustZone	Non-invasive	Industrial
2020	*PureBoot* [19]	TPM Chip USB security token	-	Invasive?	Industrial
2020	*Security for Arm Cortex-M devices with FreeRTOS* [20]	ARM TrustZone	-	Non-invasive	Industrial
2019	*Lightweight Secure-Boot Architecture for RISC-V System-on-Chip* [21]	CAU	-	Non-invasive	Academic
2019	*An Implementation of Secure boot Using TPM in Embedded System* [22]	TPM	-	Logical	Academic

Year	Source	Required Hardware	Optional Hardware	Threat Model	Academic / Industrial
2018	*Secure Boot and Remote Attestation in the Sanctum Processor* [23]	PUF	-	Logical	Academic
2018	*Secure the Windows 10 boot process* [24]	TPM chip	-	Non-invasive	Industrial
2017	*Implementing a ARM-Based Secure Boot Scheme for the Isolated Execution Environment* [25]	ARM TrustZone	-	Non-invasive	Academic
2014	*A Practical Hardware-Assisted Approach to Customize Trusted Boot for Mobile Devices* [26]	TEE SE	-	Non-invasive	Academic
2013	*A Trusted Bootstrapping Scheme Using USB Key Based on UEFI* [27]	Secure USB key	-	Logical	Academic
2013	*On implementing trusted boot for embedded systems* [28]	FPGA	-	Logical	Academic
1999	*A secure and reliable bootstrap architecture* [29]	Trusted storage or trusted recovery server	-	Logical	Academic

other USB key should be refused. The secure bootloader itself is then contained in the secure USB key and will verify the image as we expect. The reason why the USB key is considered as secure is that it possesses a few security mechanisms. In particular, it contains a partition in which the private key is stored. This partition is then protected using a few security mechanisms that help keeping the key secret. This paper has no mention of any threat model, so it is assumed that the model used corresponds to a logical attacker model.

A Practical Hardware-Assisted Approach to Customize Trusted Boot for Mobile Devices [26]

This work proposes a more practical implementation of a secure bootloader. This secure bootloader aims at being compatible with most smartphones. Therefore, it has been implemented on an off-the-shelf smartphone, whose underlying hardware security is common and sufficient for a secure boot. This hardware includes a trusted execution environment (TEE), which provides logical separation between secure and none secure worlds, and a secure element (SE), which provides more cryptography primitives that are more useful against hardware attacks. Additionally, they claim that current secure boot solutions either prevent the use of third-party OS, or lack the ability to verify them. For this reason, they aimed at authorizing custom OSes meaning that it is possible to manually authorize a custom OS so that this one can successfully be verified by the bootloader. This is done by adding a new public signature key to the bootloader, which can only be done with the knowledge of a shared secret. Their threat model is well-documented, and they indicate that their design is vulnerable to some more advanced physical attacks, which are in fact invasive. Therefore, its threat model corresponds to a physical non-invasive attacker. They argue that some attacks cannot be mitigated with the hardware they used, which confirms that such invasive attacks require specialized hardware that is not always commonly available.

Implementing a ARM-Based Secure Boot Scheme for the Isolated Execution Environment [25]

This paper proposes an alternative implementation of a secure boot. Their solution is a multi-stage secure boot that uses ARM TrustZone and an FPGA. The FPGA is used to program the architecture of the device as well as the bootROM. Additionally, they offer encryption and decryption of images, in addition to the usual authentication. However, their threat model is not clearly indicated. We can however deduce that they are considering the equivalent of a physical non-invasive attacker from the fact that one of their examples mentions an attacker who is capable of modifying the external memory during the boot process.

Secure Boot and Remote Attestation in the Sanctum Processor [23]

This work proposes an additional dimension to the secure bootloader. They consider an honest but curious manufacturer which tries to extract secrets from the device. Their design aims at reducing the amount of shared secret with the manufacturer. It is based on physical unclonable function (PUF). PUFs are physical objects that possess a unique fingerprint, most often acquired

based on random variations induced by environmental conditions during the manufacturing process. Therefore, the PUF is used as a way to store a secret during the manufacturing process of a device without allowing the manufacturer to access it. From there, PUFs can be used as a challenge-response which is unique to each device and unknown to the manufacturer. A private key can then be derived from the PUF. However, the authors assume that the memory is trusted and only accessible by the processor, meaning that they consider a logical attacker model. Additionally, note that an alternative way to store a secret in a device without allowing the manufacturer to access it is to have access to memory that is initially writable and that can be made read-only afterwards. This is the solution we have chosen in our implementation.

An Implementation of Secure boot Using TPM in Embedded System [22]

This work offers a very descriptive approach to a secure bootloader that uses a trusted platform module (TPM) component and a secure element. They include a complete boot flow, which included the initialization protocol. Again, this paper does not cover physical attacks. Their threat model states that anything that is modified after the first boot will be caught, which corresponds to the notion of a logical attacker model. They also state that the setup phase must be attacker-free. This is common in most models (including ours), but is usually implicit, while this paper indicates it explicitly.

Lightweight Secure-Boot Architecture for RISC-V System-on-Chip [21]

This paper proposes an intricate implementation of a multi-level secure bootloader which relies on a hardware code authentication unit (CAU). The authors aim for an optimized implementation. This paper enumerates a complete list of attacks that the system protects against. There are no mentions of attacks on the communications, but they do mention some simple physical attacks, meaning that they considered a non-invasive physical attacker.

CARE: Lightweight Attack Resilient Secure Boot Architecture with Onboard Recovery for RISC-V based SOC [14]

Finally, this paper proposes a more optimized secure boot implementation. They use a code integrity and authentication unit. Their design is focused on using being space and energy efficient. Additionally, they provide a mechanism to allow safe recovery of a corrupted image to a static version stored in read-only memory. Their threat model is rather explicit and their attacker could correspond to a non-invasive physical attacker. However, they consider that an attacker can directly access all memory, including internal, without going through the processor first, as long as it is not protected by access control policies. Additionally, they mitigate some more involved physical attacks such as fault injections. Therefore, their attacker model corresponds more to some invasive physical attacker, which is uncommon in the research domain.

5.2.3 Industrial

Let us now present multiple secure bootloader implementations that are used as part of the industry.

Secure the Windows 10 boot process [24]

This document covers the functionalities of the secure boot that relies on a TPM chip and that is used with the Windows 10 operating system. Their implementation verifies the signature of the target OS, as is commonly done. However, personal computers are widely used by the society, and users often choose to install a different operating system. To allow such choices, the secure bootloader will consider as authentic either a Microsoft-signed product, or an OS that was manually approved by the user. Since allowing custom images in a secure boot could defeat its purpose, it must be done securely. Microsoft chose to disallow the modification of such settings in software altogether, to avoid introducing an attack vector. Only the user can manually change the settings and indicate once that an image should be considered as authentic. Additionally, some devices disable this feature entirely. This is the case, for example, for some Windows phones which are only meant to run a Microsoft-signed OS.

On a computer, the drivers can (and should) be modifiable, as they might be updated or customized. However, drivers may be necessary during the booting process. Therefore, they represent an attack vector during the booting process. Since the initialization of some drivers happen before Windows is able to launch its malware detection, an early launch anti-malware (ELAM) was implemented. It is a minimal anti-malware software that simply compares the hash of drivers against reference values and refuses to initialize it if there is a mismatch. This is essentially a way to extend the root of trust to drivers during the booting process. Lastly, Microsoft offers a trusted online verification server that allows a user to check the integrity of their device. This provides a way to identify a malicious modification of the system, even if it is located in the early phases of the boot process. The threat model itself does not mention any kind of invasive attack, so it certainly corresponds to a non-invasive physical attacker. This is expected since broadly available devices may not include specific hardware defenses necessary for more involved attacks.

Security for Arm Cortex-M devices with FreeRTOS [20]

This work provides an implementation of a secure boot which relies on Trusted Firmware-M (TF-M). TF-M requires ARM TrustZone and provides partition between secure and non-secure processing environments. The supported devices are similar to the one we used, but they targeted devices that possess a TrustZone, while the STM32H753 does not have one. They mention no threat model in this design, but we can safely assume that they implicitly use TF-M threat model since it is responsible for most of the security. This corresponds to a non-invasive physical

attacker model.

PureBoot [19]

This is an open-source secure bootloader designed for security-oriented phones and computers of the Librem series. They emphasize privacy and security as their main arguments. The bootloader is designed to require an external hardware. More precisely, a secure USB key is required to detect any tampering of the target device. While it is not strictly necessary to boot, the OS cannot prove that the boot firmware has not been tampered with. This way, a user can rely on an external USB key to monitor the integrity of its device. The bootloader itself uses a trusted platform module (TPM) chip as a hardware security mechanism. It also uses *Heads* [30] as part of the booting process which is an open-source secure bootloader. Heads is advertised as more secure than the default closed-source solutions. Although the security and confidentiality is one of the main selling arguments, PureBoot or the Librem series do not present a clear threat model. The threat model could partly be inherited from Heads. Heads mentions many attacks as part of the attacker capabilities, but some attacks are invasive physical attacks that cannot be fully prevented directly from software. It is thus unclear whether Heads actually prevents or mitigates such attacks. Since PureBoot does not mention them, it is uncertain whether their threat model actually corresponds to an invasive physical attacker.

Android Verified Boot [18]

This documentation describes the booting process of many Android devices. They refer to the booting strategy as a *verified* boot. Unlike a secure boot, a verified boot will still allow a non-authentic image to boot, but will warn the user and may require manual approbation. This can be useful in a context where it should be possible to run unauthorized code under certain circumstances, for example when the user is allowed to develop custom OSes on a device. Being able to temporarily (and knowingly) allow an image might thus be desirable. Additionally, the Android verified boot provide a way to add a public key to the root of trust int order to persistently authorize a custom operating system. This was, a user can choose to install an OS which is not verified by Android on their devices, while still verifying its integrity. Android provides a threat model to describe the security of their verified boot. They mostly target a scenario where an everyday-attacker steals one's smartphone and tries to install malicious code while remaining undetected. This corresponds to the definition of a non-invasive physical attacker.

Knox White Paper [17]

This white paper then proposes a secure boot for smartphones which uses ARM TrustZone. Knox is essentially an extension of Android's verified boot, where additional security is enforced. The first difference is that Samsung Know, unlike a verified boot, will refuse to boot any image that is not authentic. This means that it is not suitable for custom OSes. It does, however, raise

the security level since there are less attack vectors compared to a bootloader which allows custom OSes. Additionally, Samsung Knox relies on hardware to provide more security features. Such features include a rollback prevention (which, once a version is installed, should make it physically impossible to install any prior version, thanks to hardware fuses), or a fuse that locks the device in case any unapproved state is detected and that can only be unlocked by performing a factory reset. The threat model, however, does not mention invasive attacks, so it still corresponds to a non-invasive physical attacker model.

Secure Boot and TLS 1.3 Firmware Update with FreeRTOS and wolfSSL on NXP "Freedom Board" K64 [16]

This product proposes an implementation of a secure boot called *WolfBoot*, showcased on a particular device. This bootloader is closed-source so not much details is known about it, but it is advertised as a complete and stand-alone secure bootloader that can optionally use ARM TrustZone wherever it is available. It supports encrypted image and keeps the last installed version in storage to revert a corrupted update, much like MCUboot. Even though they advertise security, they do not provide a threat model. The only indication is that WolfBoot is inspired by the IEFT SUIT draft [31], which mostly corresponds with a non-invasive physical attacker model. Hence, we consider that WolfBoot follows this threat model.

Boot process for a Mac with Apple silicon [15]

Finally, this document describes the secure bootloader implemented on modern Apple devices. One uncommon particularity here is that Apple both the software and the hardware. Indeed, they use the processor that they manufacture themselves. A consequence of this is that their design can be very flexible, as it can be custom-tailored to the underlying hardware, and vice-versa. As such, they use a proprietary secure chip called *T2*. This chip is in particular responsible for verifying the secure bootloader itself as well as hosting the secure enclave, which is a physically separated area of the system on chip which contains privileged parts of the system. Apple mentions in their threat model that their system mitigates a few specific invasive attacks, so their attacker model corresponds to an invasive physical attacker.

5.3 Summary

In summary, the related work present many different implementations of a secure bootloader, each one with specific hardware security, and possibly threat model. The threat model is however lackluster or implicit in many cases. Still, three threat model categories can be derived from the related work (either used explicitly, or the attacker is implicit and corresponds roughly to this model). The weakest one is a logical attacker, which corresponds to an attacker that has no physical access to a device and is represented by a malware. A second intermediate model is a

non-invasive physical attacker, which corresponds to an attacker that also has physical access to the device but cannot perform attacks that would physically alter the device. Finally, a stronger invasive attacker corresponds to a physical attacker that can even physically harm the device in some way, depending on its model. In general, the earliest secure bootloader used the logical attacker model, while most recent one prefer the non-invasive attacker model, which is a good compromise between realistic attacker and attacks that can be mitigated without requiring very specific hardware. Some more practical work used an invasive physical attacker model since they had access to specialized hardware.

Chapter 6

Conclusion

In this work, we implemented a secure bootloader using MCUboot, a secure boot library. We also defined a corresponding two-phase attacker model which fully defines three kinds of attackers: a jailbreak attacker, a DoS attacker and an IP thief. Those attackers are defined to formally represent real-world attackers in our model. Both the bootloader and the threat model are designed to be easily reusable for other projects or for more concrete uses. As such, they represent a practical instantiation of the available related work that can be used directly or ported to another compatible device using the provided instructions.

Bibliography

[1] *Introduction to STM32 microcontrollers security*. 2019. URL: https://www.compel.ru/wordpress/wp-content/uploads/2019/09/en.dm00493651.pdf.

[2] *STM32H742, STM32H743/753 and STM32H750 Value line advanced Arm®-based 32-bit MCUs*. Feb. 20, 2020. URL: https://www.st.com/resource/en/reference_manual/dm00314099-stm32h742-stm32h743-753-and-stm32h750-value-line-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf.

[3] SEC 1: Elliptic Curve Cryptography. *The Elliptic Curve Digital Signature Algorithm (ECDSA)*. Tech. rep. May 21, 2009. URL: https://www.secg.org/sec1-v2.pdf.

[4] Andreas Rüst Tobias Schläpfer. "Security on IoT Devices with Secure Elements". In: 2019.

[5] *MCUboot Walkthrough and Porting Guide*. Nov. 4, 2020. URL: https://interrupt.memfault.com/blog/mcuboot-overview#references.

[6] *MCUboot Walkthrough and Porting Guide Github page*. Nov. 4, 2020. URL: https://github.com/memfault/interrupt/tree/master/example/mcuboot.

[7] *MCUboot Kconfig Github page*. Nov. 4, 2020. URL: https://github.com/mcu-tools/MCUboot/blob/master/boot/zephyr/Kconfig.

[8] *STM32H7: Bus fault when reading corrupt flash sectors*. Mar. 8, 2021. URL: https://github.com/zephyrproject-rtos/zephyr/issues/33140.

[9] *STM32H753ZI CRYP module, AES-CTR, counter register, documentation vs observations*. July 29, 2021. URL: https://community.st.com/s/question/0D53W00000uYmgqSAC/stm32h753zi-cryp-module-aesctr-counter-register-documentation-vs-observations.

[10] *Multiple SoCs have flash configurations unsupported by MCUboot*. Apr. 10, 2020. URL: https://github.com/mcu-tools/mcuboot/issues/713.

[11] "Multiple SoCs have flash configurations unsupported by MCUboot". In: 2021. URL: https://github.com/mcu-tools/mcuboot/issues/713#issuecomment-768336261.

[12] *bootutil_aes_ctr_init is never called during a swap recovery*. July 2, 2021. URL: https://github.com/mcu-tools/mcuboot/issues/713.

[13] *Secure boot & secure firmware update software expansion for STM32Cube*. 2021. URL: https://www.st.com/en/embedded-software/x-cube-sbsfu.html.

[14] Avani Dave, Nilanjan Banerjee, and Chintan Patel. *CARE: Lightweight Attack Resilient Secure Boot Architecture with Onboard Recovery for RISC-V based SOC*. 2021. arXiv: 2101.06300 [cs.CR].

[15] *Boot process for a Mac with Apple silicon*. Feb. 21, 2021. URL: https://support.apple.com/fr-ch/guide/security/secac71d5623/web.

[16] *Secure Boot and TLS 1.3 Firmware Update with FreeRTOS and wolfSSL on NXP "Freedom Board" K64*. Apr. 14, 2020. URL: https://www.wolfssl.com/secure-boot-and-tls-1-3-firmware-update-with-freertos-and-wolfssl-on-freescale-freedom-board-k64/.

[17] *Knox White Paper*. 2021. URL: https://docs.samsungknox.com/admin/whitepaper/kpe/hardware-backed-root-of-trust.htm.

[18] *Android Verified Boot*. Sept. 1, 2020. URL: https://source.android.com/security/verifiedboot?hl=en.

[19] *PureBoot*. 2020. URL: https://docs.puri.sm/PureBoot.html.

[20] *Security for Arm Cortex-M devices with FreeRTOS*. July 17, 2020. URL: https://www.freertos.org/2020/07/security-for-arm-cortex-m-devices-with-freertos.html.

[21] J. Haj-Yahya, M. M. Wong, V. Pudi, S. Bhasin, and A. Chattopadhyay. "Lightweight Secure-Boot Architecture for RISC-V System-on-Chip". In: *20th International Symposium on Quality Electronic Design (ISQED)*. 2019, pp. 216–223. DOI: 10.1109/ISQED.2019.8697657.

[22] Ko Jae-yong Kim Jin-woo Lee Sang-gil and Lee Chul-hoon. "An Implementation of Secure boot Using TPM in Embedded System". In: *Journal of The Korea Institute of Information Security and Cryptology* 29.5 (2019), pp. 949–960. DOI: 10.13089/JKIISC.2019.29.5.949.

[23] I. Lebedev, K. Hogan, and S. Devadas. "Invited Paper: Secure Boot and Remote Attestation in the Sanctum Processor". In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 2018, pp. 46–60. DOI: 10.1109/CSF.2018.00011.

[24] *Secure the Windows 10 boot process*. Nov. 16, 2018. URL: https://docs.microsoft.com/en-us/windows/security/information-protection/secure-the-windows-10-boot-process.

[25] H. Jiang, R. Chang, L. Ren, and W. Dong. "Implementing a ARM-Based Secure Boot Scheme for the Isolated Execution Environment". In: *2017 13th International Conference on Computational Intelligence and Security (CIS)*. 2017, pp. 336–340. DOI: 10.1109/CIS.2017.00079.

[26] Javier González, Michael Hölzl, Peter Riedl, Philippe Bonnet, and Rene Mayrhofer. "A Practical Hardware-Assisted Approach to Customize Trusted Boot for Mobile Devices". In: Oct. 2014. ISBN: 978-3-319-13256-3. DOI: 10.1007/978-3-319-13257-0_35.

[27] Abhishek Kushwaha. "A Trusted Bootstrapping Scheme Using USB Key Based on UEFI". In: *International Journal of Computer and Communication Engineering* (Jan. 2013), pp. 543–546. DOI: 10.7763/IJCCE.2013.V2.245.

[28] O. Khalid, C. Rolfes, and A. Ibing. "On implementing trusted boot for embedded systems". In: *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 2013, pp. 75–80. DOI: 10.1109/HST.2013.6581569.

[29] W. A. Arbaugh, D. J. Farber, and J. M. Smith. “A secure and reliable bootstrap architecture”. In: *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097)*. 1997, pp. 65–71. DOI: 10.1109/SECPRI.1997.601317.

[30] *Heads - Wiki*. 2021. URL: https://osresearch.net.

[31] *A Manifest Information Model for Firmware Updates in IoT Devices*. 2021. URL: https://datatracker.ietf.org/doc/draft-ietf-suit-information-model/.

Appendix A

Technical Instructions

A.1 project_config.mk

This file should define at least the following variables:

- `PATH_MCUBOOT`, the complete path to the MCUboot folder (either the default MCUboot submodule, or any fork of it if you need to make changes to it)
- `PATH_PORT`, the complete path to the port folder, as described in *Section 4.4.4*.
- `SECTIONS_CONFIG_DIR`, the complete path to the folder that contains a `sections_config.ld` linker script. This file configures the size of different MCUboot regions, as described in *Appendix A.2*.
- `BOOTLOADER_CONFIG`, the complete path to the `bootloader_config.mk` file. This file describes all port-specific source and headers files that are used by the bootloader, as described in *Appendix A.3*.
- `TEST_CONFIG`, the complete path to the `test_config.mk` file. This file describes all port-specific source and headers files that are used by the test image, as described in *Appendix A.3*.
- `PATH_KEYS`, the complete path to the `keys` folder, as generated by the bootloader's makefile. It defines keys that will be used by the image generation script or embedded in the bootloader.

Optionally, a few variables can be used to configure the toolchain by setting the value of the following variables:

- `COMPILER`, which contains the name of the compiler. Default: `arm-none-eabi-gcc`.
- `OBJCOPY`, which contains the name of the objcopy tool. Default: `arm-none-eabi-objcopy`.
- `OPENOCD`, which contains the name of the openocd executable. Default: `openocd`.

A.2 sections_config.ld

This linker file contains the size of different regions used by MCUboot. The following declarations are required:

- `_lnBootloader`, the length of the bootloader area. Should be a multiple of the flash sector size.
- `_lnBootloaderStorage`, the length of the secure bootloader area, which is at the end of the bootloader area. It is included inside the length of the bootloader area.
- `_lnImg`, the maximum length of an image. Should be a multiple of the flash sector size.
- `_lnImgHeader`, the length of an image header, which is included in the size of an image.
- `_lnScratch`, the length of the scratch area. Should be a multiple of the flash sector size.

Keep in mind that there are other constraints on the different sizes, depending on the device. Most constraints imposed by MCUboot or the bootloader are already defined in `Bootloader/Common/memory-sections.ld`, in the form of asserts.

A.3 bootloader_config.mk And test_config.mk

Those files are used to add source and header files to the bootloader and the test image on a per-port basis, in order to avoid having to modify their makefile for each new port. This can be done by defining the following variables:

- `SRC_FILES`, the list of port-specific source files.
- `INCLUDE_PATHS`, the list of port-specific header directories, of the form `-I[some path]`.
- `ADDITIONAL_CFLAGS`, a list of flags which will be used during the compilation. This can be either generic flags used by a compiler, or flags defined in this project. You can refer to *Appendix A.4* for a list of flags defined here.

`bootloader_config.mk` will be included during the compilation of the bootloader, while `test_config.mk` will be included during the compilation of the test image.

Example

For STM32H753, a port-specific `$(PATH_PORT)/src/st/rss.c` file is required to enable the secure area. Additionally, one might want to handle gracefully flash double ECC errors (which are specific to this boad). So `bootloader_config.mk` might contain:

```
ADDITIONAL_CFLAGS += -DSAFE_ECC_DBERRORS
```

and

```
SRC_FILES += $(PATH_PORT)/src/st/rss.c
```

A.4 Additional Makefile Flags

The list of makefile flags that are defined in this project and recognized by the software (typically if added in the `ADDITIONAL_CFLAGS` variable) is the following:

- `-DENABLE_BIOS`, which is required to use the utility BIOS. If used in the bootloader, BIOS will automatically be added and called. If used in an image, the BIOS's source files still need to be added in the image's makefile and the BIOS called in the image's code.
- `-DENABLE_TEST`, which add hooks to test menus in the bootloader. Irrelevant if used in an image.
- `-DHAL_INCLUDE_XXX`, which is required to use the HAL module `XXX` (which can be any module defined in *Section 4.2.2*). Otherwise, the corresponding header will simply be empty.
- `-DHAL_INCLUDE_ALL`, which works like every `-DHAL_INCLUDE_XXX` at the same time if it is defined. The bootloader always uses all drivers.
- `-DENABLE_FLASH_CORRUPTION`, which uses an alternative definition of the Flash HAL which can be used to corrupt a flash sector if the underlying hardware is susceptible to it. Only intended to be used with tests.

Additionally, some STM32H753-specific variables can be used when this port is chosen:

- `-DSAFE_ECC_DBERRORS`, which modifies the Flash driver to safely handle and silently hide Flash double ECC errors.

- `-DDISABLE_SAFEGUARD`, which allows the use of sensitive memory protection functions. If disabled, functions to enable RDP1 or the secure area will not do anything.
- `-DMCUBOOT_USE_STM32H753_CRYP`, which tells MCUboot to use the hardware acceleration modules to perform SHA256, HMAC-SHA256 and AES-CTR. Irrelevant if used in an image.

A.5 MCUboot Logs

MCUboot will always look for a `mcuboot_config/app_config.h` file, which will dictate its logging behavior. This file can be left empty (in which case MCUboot's logging will be completely disabled). In the bootloader, this concerns all logs (except for the utility BIOS, which will print information regardless of the configuration. if it is included). In an image, this only concerns the logging pf MCUboot's public interface, i.e., `boot_set_pending` and the MCUboot's flash interface. The list of logging options is the following:

- `MCUBOOT_HAVE_LOGGING`, which is required to have any logging from MCUboot. Not having it defined will disable all of its logging features, regardless of the following variables.
- `MCUBOOT_LOG_ERR_ENABLE`, which enables the error logging.
- `MCUBOOT_LOG_WRN_ENABLE`, which enables the warning logging.
- `MCUBOOT_LOG_INF_ENABLE`, which enables the information logging.
- `MCUBOOT_LOG_INTERNAL_ENABLE`, which enables logging about MCUboot's internal state.
- `MCUBOOT_LOG_DBG_ENABLE`, which enables debugging logs (warning: very verbose).

All of those variables can be defines by adding `#define [variable]` to the `mcuboot_config/app_config.h` file of the corresponding software (bootloader or image). They can be disabled by not defining them, or by setting `#undef [variable]`

A.6 Image Makefile

The list of parameters required by the common image makefile, `Bootloader/Common/Makefile`, is the following:

- `BUILD_DIR`, which contains the path of the output build folder ***which should be empty since it will be erased***.

- `SRC_FILES`, which contains the list of source files that need to be compiled.
- `INCLUDE_PATHS`, which contains the list of include paths, of the form `INCLUDE_PATHS = -I[folder1] -I[folder2] [...]`
- `TARGET`, which contains the target name (used in the name of the output files).
- `LDSCRIPT`, which contains the path to the linker script which contains the sections definition. This can for example be set to `[path_to_git_folder]/Bootloader/Common/link_template.ld`, or to your custom linker script.
- `KEY_SIGN`, which contains the path to the private signature key. In a `keys` folder generated by the bootloader's key creation tool, this correspond to `keys/imgtool_keys/key_sign_priv.pem`.
- `KEY_ENC`, which contains the path to the public encryption key. In a `keys` folder generated by the bootloader's key creation tool, this correspond to `keys/imgtool_keys/key_enc_pub.pem`.
- `PATH_MCUBOOT`, which contains the path to the MCUboot folder (which is already defined in `$(PROJECT_CONFIG)`)
- `PATH_PORT`, which contains the path to the chosen port folder (which is already defined in `$(PROJECT_CONFIG)`)
- `SECTIONS_CONFIG_DIR`, which contains the complete path to the folder that contains a `sections_config.ld` linker script (which is already defined in `$(PROJECT_CONFIG)`)

A.7 Including The Utility BIOS In An Image

To include the utility BIOS in an image, two steps are required. The first is to define the `-DENABLE_BIOS` flag in the image makefile.

The second step is to add all source files that are used by the utility BIOS. The list of source files is the following:

- `$(PATH_MCUBOOT)/boot/bootutil/src/bootutil_public.c`
- `Bootloader/Common/src/bios.c`
- `Bootloader/Common/src/image_util/image_util.c`
- `Bootloader/Common/src/image_util/image_download.c`

- `Bootloader/Common/src/image_util/image_upload.c`
- `$(PATH_PORT)/src/bios_port/bios.c`
- `$(PATH_PORT)/src/drivers/security.c`

To which is added any port-specific file required by the last two files; for STM32H753 only `$(PATH_PORT)/src/st/rss.c` is required.

If you are using your own image makefile, don't forget to add the following include paths (which are already included by default in the common image makefile):

- `$(PATH_MCUBOOT)/boot/bootutil/include`
- `Bootloader/Common/Includes`
- `$(PATH_PORT)/Includes`