Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Networked Systems and Services

Secure remote firmware update on embedded IoT devices

Scientific Students' Association Report

Author
Márton Juhász

Advisors
Dr. Levente Buttyán
Dorottya Futóné Papp

October 29, 2020

Contents

Abstract

The Internet of Things (IoT) means the Internet with embedded devices too. The IoT is becoming more and more important and ubiquitous in our everyday lives, for example in the infrastructure of smart cities, in modern industrial applications, in transport and in smart homes. The security of the IoT should not be overlooked as some of those applications are mission critical. Connecting the embedded devices to the Internet opens up a new and substantial attack surface, and the attacks targeting these devices are often successful. The attackers often exploit software vulnerabilities, which are hard to avoid, especially with low manufacturing costs. Such vulnerabilities can be fixed with software updates, however also it must be done securely, so that the update mechanism itself cannot be used for new attacks.

In this paper I propose a secure remote firmware update for embedded IoT devices, that satisfies the following criteria: it can be carried out remotely, without the need to physically approach each and every device; it is secure, otherwise it can be exploited to compromise devices at large scale; it is reliable and fail-safe, meaning if an update failed, the device automatically reverts to the last stable version of the updated component; and it has version rollback prevention, to prevent attackers from installing an old, potentially vulnerable version of the component through which they can compromise the devices again.

In the paper I give an overview of the security problems of the remote firmware update, present the design of my firmware update protocol, its security properties, and their analysis with formal methods, as well as report a prototype implementation done by me, which I did for the Raspberry Pi 3 Model B platform.

Kivonat

Az Internet of Things (IoT) a beágyazott számítógépeket is tartalmazó Internetet jelenti. Az IoT egyre fontosabb és áthatóbb szerepet játszik a mindennapi életünkben, például az okos városok infrastruktúrájában, modern ipari alkalmazásokban, közlekedésben és okos otthonokban. Az IoT biztonságát nem szabad elhanyagolni, mivel az említett alkalmazások között vannak kritikus fontosságúak. A beágyazott eszközök Internethez való csatlakoztatása egy új és lényeges támadási felületet nyit meg, és az ezeket az eszközöket célzó támadások gyakran sikeresek. A támadások sokszor szoftveres sérülékenységeket használnak ki, amiket nehéz elkerülni, különösen az alacsony előállítási költségek mellett. Az ilyen sérülékenységek javítása szoftverfrissítéssel lehetséges, ám ezt is biztonságosan kell megoldani, hogy ne maga a frissítési mechanizmus adjon lehetőséget újabb támadásokra.

Ebben a dolgozatban egy olyan biztonságos távoli firmware frissítést javaslok beágyazott IoT eszközökhöz, mely kielégíti a következő feltételeket: távolról végrehajtható, anélkül, hogy oda kellene menni minden egyes eszközhöz; biztonságos, különben ki lehetne használni nagyszámú eszköz kompromittálására; megbízható és üzembiztos, ami azt jelenti, hogy ha egy frissítés sikertelen, az eszköz automatikusan visszaáll a frissített komponens utolsó stabil verziójára; és tartalmaz verzió rollback prevenciót, amely megakadályozza a támadókat, hogy feltelepítsék valamely komponens egy régi, potenciálisan sérülékeny verzióját, mely segítségével újra kompromittálhatják az eszközöket.

A dolgozatban áttekintést adok a távoli firmware frissítés biztonsági problémáiról, bemutatom az általam tervezett firmware frissítési protokollt, annak biztonsági tulajdonságait, illetve ezek formális módszerekkel történő elemzését, valamint beszámolok egy általam készített prototípus implementációról, melyet Raspberry Pi 3 Model B platformra készítettem.

Chapter 1

Introduction

Internet of Things (IoT) systems are built from network connected embedded devices, and their security heavily relies on the security of those embedded devices. One of the most important security aspects in this context is the integrity of the software running on embedded devices. The reason is that unauthorized modification of software can result in arbitrary behavior of those devices, and as a consequence, loss of trust in the entire IoT system built upon them. In particular, protecting low level software, such as the operating system (OS) and the firmware, is important, because typically these components are responsible for implementing many security controls and they provide trusted services (e.g., in the form of system calls) to higher layer software.

Digitally signing software components, including the firmware and the OS kernel, and important data, such as configuration files, combined with some hardware based root-of-trust and a secure boot process, which ensures that software components are loaded and executed only if their signature is valid, can help protecting the integrity of software, but does not entirely solve the problem. In particular, signed code and verified boot ensure that the device runs intact code right after a reset, but software can also be compromised at run-time by exploiting design and implementation level vulnerabilities in it. For instance, an attacker may be able to execute arbitrary injected code on a device by exploiting software bugs, such as not checking the amount of data copied into a limited size buffer or using dangling pointers, leading to memory corruption. [5]

When software vulnerabilities are discovered, they need to be fixed, and embedded devices need to be updated with the fixed software. This applies to the OS and the firmware too. In addition, due to the potentially large number of embedded devices used in IoT applications and their often special operating environment, it is preferable that the update process can be carried out remotely, without the need to physically approach each and every device. Remote firmware and OS update is sometimes also called over-the-air (OTA) update, because the update may be downloaded by the devices over wireless communication links.

Security of the remote update process itself is of paramount importance, as we would like to avoid that attackers exploit an insecure update mechanism to install a compromised OS or firmware remotely at large scale. Potentially, such compromised updates may prevent any further legitimate update, leaving the control of all compromised devices in the hand of the attacker. Recovering from such a situation would require manual update of every device, which would be time consuming and expensive.

Besides security, the update process must be reliable and fail-safe, by which I mean that an unsuccessful update should not leave the devices in a state where they cannot boot and operate properly, but it should be possible to detect if the update failed and to load the last stable version of the updated software. At the same time, attackers should not be able to force a version rollback when the devices run a stable version of the software, because if that was possible, then they could force the devices to re-install an old, potentially vulnerable version of the software through which they can compromise the devices again.

In this paper, I introduce a secure remote update system and mechanism for embedded IoT devices that satisfy the above requirements on security, reliability, and version rollback prevention. The rest of this paper is structured in the five following chapters: First in Chapter 2, there are introductions to the related researches and technologies for the above mentioned problems. Then in Chapter 3, there are the design plans of my secure remote firmware update. And in Chapter 4, there are the implementation details of my secure remote firmware update. Then in Chapter 5, there are the my evaluations of my results. And finally, in Chapter 6, there is a summary of my work and some possible future improvements.

Chapter 2

Related work

Designing and implementing a secure remote firmware update architecture is a complex task with many important details. The TCG [6] contains guidelines and best practices for all the different details of a secure remote update process, and divides it to five main parts that are: Secure Development, Secure Update Signing, Robust Distribution, Secure Update Installation, Post-Update Verification and Attestation. Therefore I analyzed the following articles based on those five main aspects and the detailed guidelines in them.

ASSURED [1] is a secure and scalable update framework for the Internet of Things. It uses reliable cryptographic algorithms and keys with adequate strength, and places the most important keys in Hardware Security Modules. The updates are countersigned by the local controller of the devices. And it writes about the importance of carefully vetting any Certificate Authorities and other parties trusted in the signing process. It includes a recovery process with rollback protection by included version information. It authenticates and tracks the endpoints, and the endpoints should verify any updates downloaded and before installation. Update installation is done with attention to avoid time-of-check to time-of-use attacks. The distribution is automatic (without overloading networks or servers), but administrators can schedule updates. It also utilizes measured boot and remote attestation.

An other framework for incorporating secure remote updates into embedded designs [4] also highlights the need for reliable cryptographic algorithms/tools and who to trust in the signing process, but it also mentions to use a dedicated and air-gapped computer for production code signing. It includes a recovery process, but does not write about rollback protection. Similarly, it authenticates and tracks the endpoints, and the endpoints should verify any updates downloaded and before installation. The administrators can schedule the updates.

The article about secure firmware validation and update scheme for consumer devices in a home networking environment [2] focuses on the cryptographic side of the topic. And besides authentication, verification and rollback prevention, it also writes about encrypting the updates.

The article about design of a safe and secure bootloader for an RFID reader with a software implementation of a secure firmware updater [3] uses reliable cryptographic algorithms/-tools and keys with adequate strength. It also has a recovery process, however without rollback prevention. The device verifies any updates downloaded and before installation, and installs with attention to avoid time-of-check to time-of-use attacks. In addition the updates are encrypted.

The TCG guidance [6] also explains the popular concept of A/B Updates that uses two sets of partitions to store the firmware. At any point in time, the system is only running code from one of these, so the other one can be updated, and that is why it is also called as Seamless Updates. It requires enough storage space to hold two copies of the system image, but if an update fails, the older code is available in the other slot as a fallback.

In conclusion the articles covered most of the points in the TCG guidance [6], however there were some uncovered important parts. The Secure Development part was not discussed at all, but this is not surprising, as it is left to the group implementing a specific solution. Neither of the articles wrote about a revocation process, however due to the rollback protection, if an update was successfully installed a previous version can not be installed, so instead of revoking an update a newer shall be issued. The TCG guidance [6] suggests restricting update installation privileges and activities to a minimally sized, carefully coded, and tightly controlled Root Update Engine, which is a good practice, but was not considered by any of the articles.

ASSURED [1] provided a good starting point when I started to design my secure remote update process, and I used their stakeholders in my framework, but improved their model by using a separated OS for the security critical tasks. This model also improves on the concept of A/B Updates. I also followed the recommendations in the TCG guidance [6] during my development.

Chapter 3

Design

3.1 Stakeholders

The entire secure remote update framework consist of the following stakeholders:

- **Original Equipment Manufacturer:** It produces different types of Devices with their initial Images. During production, the Original Equipment Manufacturers cryptographic keys are securely installed on the Devices. It also creates and signs Updates for each type of Device.
- **Distributor:** If the Original Equipment Manufacturer does not want to build and maintain the Update distribution, it can be outsourced to a Distributor.
- **Controller:** It configures and manages the Devices within its administrative domain. Before a new Device starts its regular operation it must be customized, and that is when the Controllers cryptographic keys are securely installed on the Device. This makes it possible for the Controller to countersign the Updates and to specify constraints on them, such as which Device gets a particular Update and when.
- **Device:** The embedded IoT Device that is the target of the Updates.

3.2 Attacker model

Two different kind of attackers are considered:

- **Remote attacker:** Who may try to compromise the Updates between the Original Equipment Manufacturer and the Controller, or attempt to remotely exploit software vulnerabilities.
- **Local attacker:** Who may try to compromise the update process between the Controller and the Device.

A physical attacker is out of scope for this paper, because this paper focuses on software solutions and defending against a physical attacker requires tamper-resistant hardware solutions.

3.3 Objectives

In order to defend against attacks the Device shall only install signed Updates originated by the Original Equipment Manufacturer and approved by the Controller. The integrity of the running Images, future Updates and cryptographic keys, and the confidentiality of secret keys must be ensured. To achieve this, the following architecture is used.

3.4 Architecture

There are three types of Images that can be updated:

- **Firmware:** It includes low level software components, the Trusted Execution Environment and the boot loader for the OSs.
- **UpdateOS:** It is a trusted OS with minimal functionality, and contains only the required parts to verify and install Updates. So it does not even has network access in order to minimize its attack surface.
- **MainOS:** It is responsible for the main functionality of the Device. It has network access in order to fulfill the intended tasks of the Device and to download the Updates. It is extended with a self-testing mechanism that runs on startups and checks if everything works as intended.

When the Original Equipment Manufacturer releases an Update, it uploads the new Update to the Distributors repository. An Update contains one Image that needs to be updated and its version, together signed by the Original Equipment Manufacturer.

The Controller must periodically check the Distributors repository for Updates for all types of Devices that it manages. When it finds an Update in the Distributors repository for any type of Device that it manages, it must download that Update to its local repository. The process is illustrated in Figure 3.1.

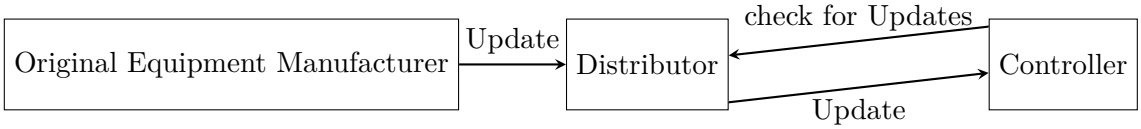

Figure 3.1: Communication between Original Equipment Manufacturer, Distributor and Controller

A Device must periodically send its Device State Report to the Controller that it is assigned to, and wait for the Controllers Response. A Device State Report describes the current state of the Device, and contains the ID of the Device and the version numbers of its current Firmware, UpdateOS and MainOS. It is signed by the Device.

When the Controller receives a Device State Report, it must reply with a signed Response that consists of the action the Device has to take and optionally an Update. The action can be one of the following: Continue operation, if there is nothing to be updated. Or update the specified software Image with the included Update in the Response. The process is illustrated in Figure 3.2.

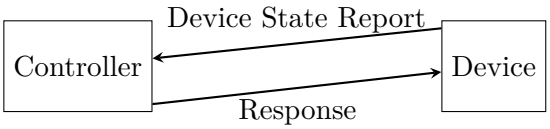

Figure 3.2: Communication between Controller and Device

When a Device receives the Controllers Response, one of the following actions must happen based on the Response: If the action in the Response is to continue operation, the Device closes the connection, and reopens it only, when it is time to send the next Device State Report. If the action in the Response is to update, the Device closes the connection, and installs the included Update. See the detailed process in Chapter 4.

To sign and verify the above mentioned artifacts the following cryptographic keys are used:

- **Update Image signing key pair:** It is unique for each type of Image for each type of Device.
  - **Private key:** The Original Equipment Manufacturer stores the private keys for each type of Image for each type of Device.
  - **Public key:** The Controller stores the public keys for each type of Image for each type of Device that it controls, and each Device stores the public keys for each type of Image that corresponds to its type.
- **Device State Report signing key pair:** It is unique for each Device.
  - **Private key:** The Device stores its own private key.
  - **Public key:** The Controller stores the public keys for each Device that it controls.
- **Response signing key pair:** It is unique for each Controller.
  - **Private key:** The Controller stores its own private key.
  - **Public key:** The Device stores the public key for its Controller.

In order to keep track of all the important events, the following log files are used:

- **updatelog_Firmware:** An entry consists of a version number and a state. The state can be stable or failed.
- **updatelog_UpdateOS:** An entry consists of a version number and a state. The state can be stable or failed.
- **updatelog_MainOS:** An entry consists of a version number and a state. The state can be stable, installed, executed or failed.
- **selftestlog:** It stores the results of the self-tests done by the MainOS. An entry consists of a version number and a state. The state can be successful self-test, failed self-test or failed root file system.
- **downloadlog:** It stores the downloaded Updates. An entry is the type of the downloaded Update (Firmware, UpdateOS and MainOS).

There are three partitions with different access restrictions:

- **BOOT:** It is where the Device boots from. It stores the verified Firmware, UpdateOS and MainOS Images; the cryptographic keys[1]; the three update log files; and the control file (nextOStoboot); as well as the configuration files. The control file, called nextOStoboot tells the boot loader which OS to boot (UpdateOS or MainOS). And the configuration files store the ID of the Device, the connection details to the Controller, the parameters needed by the self-tests and the time period to send a Device State Report. Furthermore, the BOOT partition must have hardware write protection, so that it is only writable by the Firmware and the UpdateOS.
- **images:** It holds the selftestlog, the downloadlog and the Responses downloaded from the Controller.
- **data:** Application data is saved here.

[1]For increased security it is better to store the cryptographic keys in a Hardware Security Module (HSM).

Chapter 4

Implementation

4.1 Used Components

4.1.1 ARM Trusted Firmware-A

As ARM is a widely adopted platform among IoT devices, this implementation is for ARM based devices as well and uses the ARM Trusted Firmware-A. "Trusted Firmware-A (TF-A) is a reference implementation of secure world software for Arm A-Profile architectures (Armv8-A and Armv7-A), including an Exception Level 3 (EL3) Secure Monitor. It provides a suitable starting point for productization of secure world boot and runtime firmware, in either the AArch32 or AArch64 execution states."[1] TF-A utilizes the ARM TrustZone technology, which provides system-wide hardware isolation for trusted software.[2]

4.1.2 OP-TEE

To further improve the secure operation of the Device the security subsystem can be separated from the normal OS. The TF-A supports this approach, as it can hold such a trusted component, which is available to the normal OS through the Secure Monitor of the TF-A. In this implementation the Open Portable Trusted Execution Environment (OP-TEE) was chosen as the security subsystem, because it is available on GitHub, has an active development and supports a variety of different devices. "OP-TEE is a Trusted Execution Environment (TEE) designed as companion to a non-secure Linux kernel running on Arm; Cortex-A cores using the TrustZone technology. OP-TEE implements TEE Internal Core API v1.1.x which is the API exposed to Trusted Applications and the TEE Client API v1.0, which is the API describing how to communicate with a TEE. Those APIs are defined in the GlobalPlatform API specifications."[3]

[1] `https://github.com/ARM-software/arm-trusted-firmware` (Accessed October 29, 2020)
[2] `https://developer.arm.com/ip-products/security-ip/trustzone` (Accessed October 29, 2020)
[3] `https://optee.readthedocs.io/en/latest/general/about.html` (Accessed October 29, 2020)

4.1.3 U-Boot

Das U-Boot[4], or U-Boot for short, is an open source boot loader for embedded devices, and it works with several different architectures. It provides a low-level interface through its commands, and those commands can be entered through its shell command interpreter. Its settings are saved in environment variables. These scripting abilities make it possible to create a customized boot flow.

4.1.4 Linux

The majority of the IoT devices run a GNU/Linux distribution as their OS, also OP-TEE is designed to work with the Linux kernel as the Rich Execution Environment (REE), so both the UpdateOS and the MainOS are Linux based.

- **UpdateOS:** Due to its minimal functionality it is a stripped down Linux, built together with an initial RAM file system as one image. The initial RAM file system stores everything the UpdateOS needs to perform its tasks, so there is no need for a separate root file system.
- **MainOS:** It is a normal embedded Linux image with a separate root file system. The root file system is packed into a single file so it can be verified.

4.2 Secure boot process

In order to build such a secure update process a secure boot process is also necessary, to boot into a known and secure state. In which all of the loaded components are digitally signed and each stage verifies the next stage before executing it. If the verification fails at any of the stages, the process is halted.

The TF-A has a feature, called Trusted Board Boot. "The Trusted Board Boot (TBB) feature prevents malicious firmware from running on the platform by authenticating all firmware images up to and including the normal world bootloader. It does this by establishing a Chain of Trust using Public-Key-Cryptography Standards (PKCS)."[5]

After reset, the process starts with the boot ROM verifying the read only part of the TF-A using the hardware root of trust. The hash of the signature verification public key used by the boot ROM code is stored in a special, one-time programmable memory, which is written during device customization, after which this signature verification public key can no longer be modified. Then the read only part of the TF-A verifies the updateable part of the TF-A, with the necessary public key stored in the read only part. Next the TF-A verifies the Secure Monitor before handing over control to it to initialize itself, and does the same for the secure payload, which is a Trusted Execution Environment, in this case OP-TEE. It also does the same for U-Boot, the boot loader of the Rich Execution Environment, but in this case the control is not handed back, so the boot process can carry on. The process is illustrated in Figure 4.1. All of the necessary public keys are stored in the updateable part of the TF-A.

[4] `https://www.denx.de/wiki/U-Boot/WebHome` (Accessed October 29, 2020)

[5] `https://trustedfirmware-a.readthedocs.io/en/latest/design/trusted-board-boot.html` (Accessed October 29, 2020)

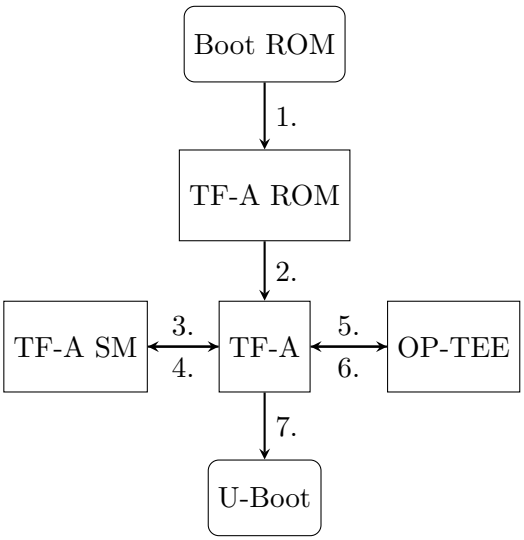

Figure 4.1: Trusted Firmware-A Trusted Board Boot: boot flow.

U-Boot supports verified boot through booting a Flattened Image Tree (FIT). "It is a flattened device tree (FDT) in a particular format, with images contained within. FIT supports hashing of images so that these hashes can be checked on loading. This protects against corruption of the image. However it does not prevent the substitution of one image for another. The signature feature allows the hash to be signed with a private key such that it can be verified using a public key later. Provided that the private key is kept secret and the public key is stored in a non-volatile place, any image can be verified in this way. The public key can be stored in U-Boot's CONFIG_OF_CONTROL device tree in a standard place. Then when a FIT is loaded it can be verified using that public key. Multiple keys and multiple signatures are supported."[6][7]

U-Boot has another signature verification public key baked into its code, which is used to verify the digital signatures of the Linux kernel images. U-Boot always checks the nextOStoboot control file, and acts according to what is indicated in that file. When the UpdateOS is about to be booted, U-Boot writes in the nextOStoboot file that the MainOS should be loaded next time, and verifies and executes the UpdateOS kernel. When the MainOS is about to be booted, U-Boot writes in the nextOStoboot that the UpdateOS should be loaded next time and checks the updatelog_MainOS file. If the new MainOS is about to be executed for the first time (its state is installed) U-Boot writes in the updatelog_MainOS file that it executed the new MainOS (its state is executed). If the stable MainOS is about to be executed (its state is stable) U-Boot does not change the updatelog_MainOS file. Then in both cases makes the BOOT partition write protected, and verifies and executes the MainOS kernel. Otherwise (latest state in the updatelog_MainOS is executed or failed) an unexpected error happened, for example a power loss during the operation of the UpdateOS, and U-Boot should not start the MainOS, so it resets the Device. The process is illustrated in Figure 4.2.

[6] `https://gitlab.denx.de/u-boot/u-boot/-/blob/master/doc/uImage.FIT/verified-boot.txt` (Accessed October 29, 2020)

[7] `https://gitlab.denx.de/u-boot/u-boot/-/blob/master/doc/uImage.FIT/signature.txt` (Accessed October 29, 2020)

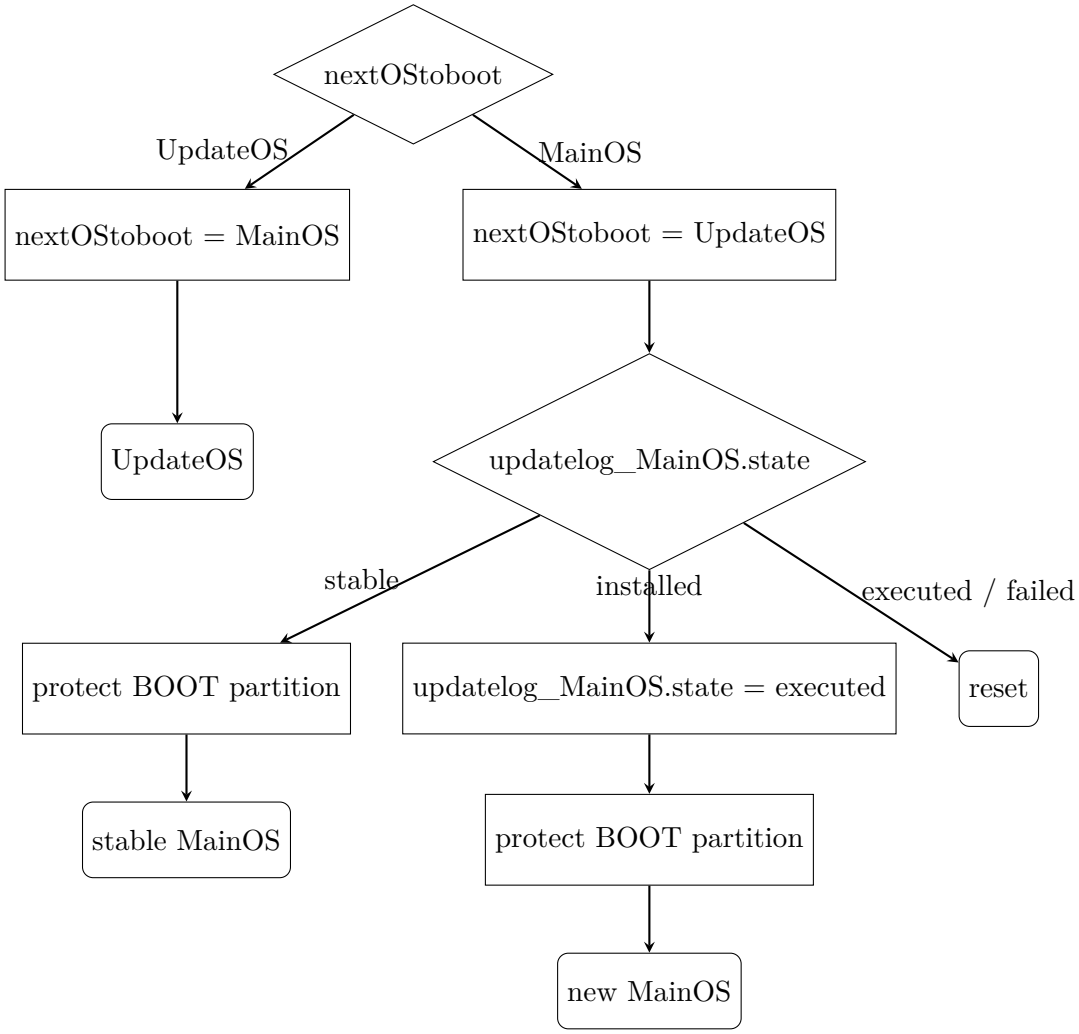

Figure 4.2: U-Boot: custom boot flow. The updatelog_MainOS entries are referring to the entries with the version number of the latest MainOS.

The verification of the root file system, is done with dm-verity. “Device-mapper is infrastructure in the Linux kernel that provides a generic way to create virtual layers of block devices. Device-mapper verity target provides read-only transparent integrity checking of block devices using kernel crypto API. […] The dm-verity was designed and developed by Chrome OS authors for verified boot implementation.”[8]

Finally, the MainOS kernel verifies the integrity of the root file system image on the BOOT partition, and on success, it mounts the root file system, after which the secure boot process is completed. See in Figure 4.3.

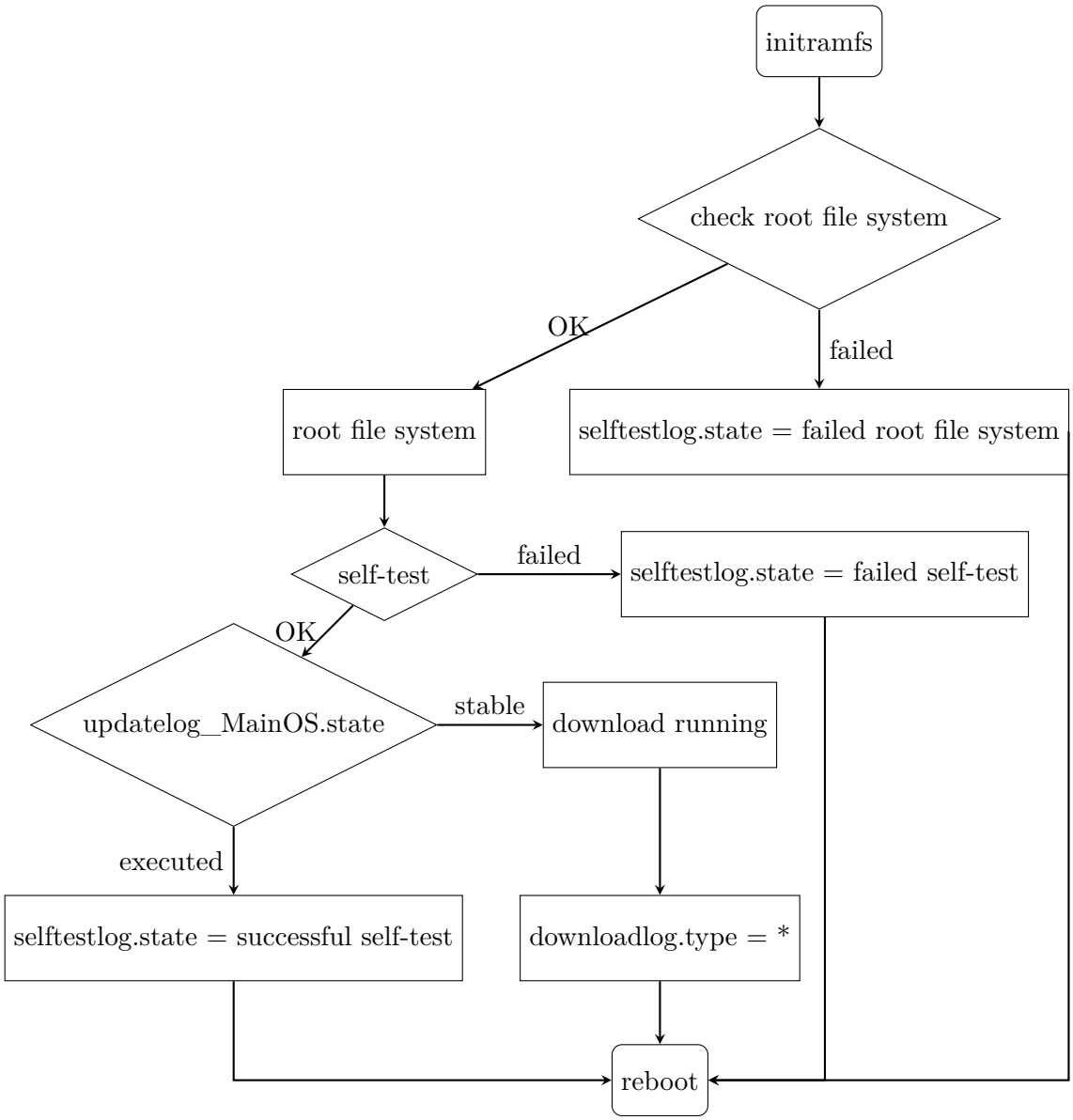

Figure 4.3: MainOS: control flow of the secure boot and update process. The selftestlog entries are referring to the entries with the version number of the currently running MainOS.

[8] `https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMVerity` (Accessed October 29, 2020)

4.3 Secure update process

First let's assume that the latest stable MainOS is booted, the download service is running and there is no Update on the Device. When the Controllers Response includes an Update, the download service in the MainOS, which sends the Device State Reports and processes the Responses, puts it to the images partition and writes the type of the Update into the downloadlog, then reboots the Device. See in Figure 4.3. Upon the next boot, the UpdateOS detects the Update from the downloadlog (Figure 4.4), verifies its digital signature, and on success, it places the new Image on the BOOT partition.

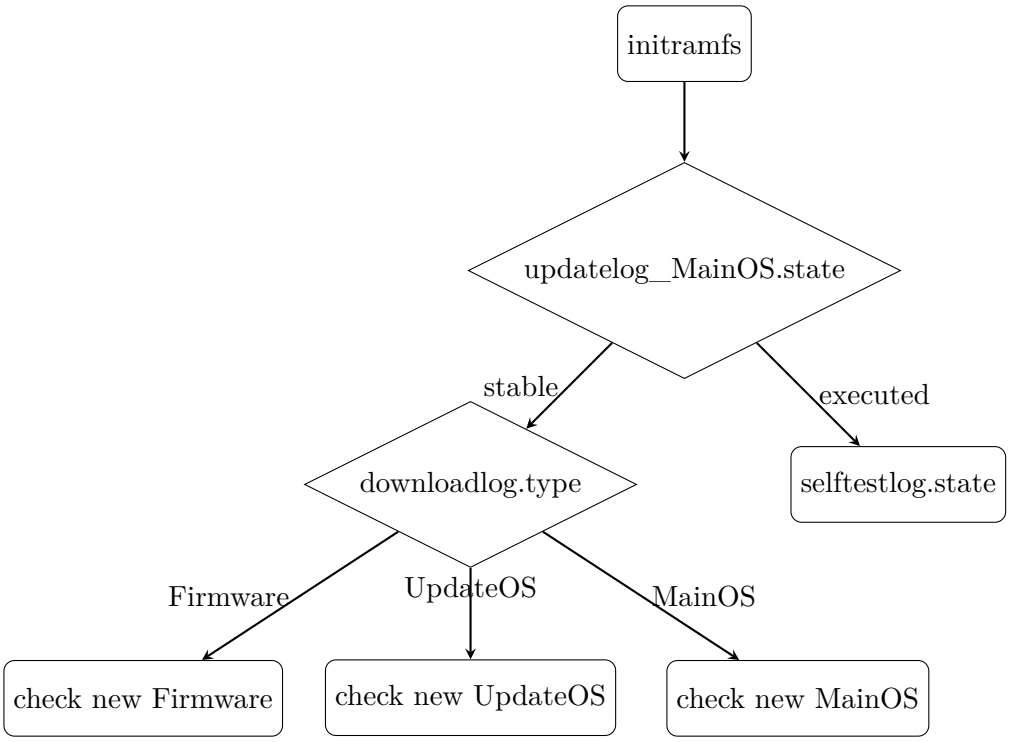

Figure 4.4: UpdateOS: overview of the control flow.

In case of an update of the Firmware or the UpdateOS, the new Image replaces the old one, as these must be thoroughly tested Images that function properly. The process is illustrated in Figure 4.5 and Figure 4.6.

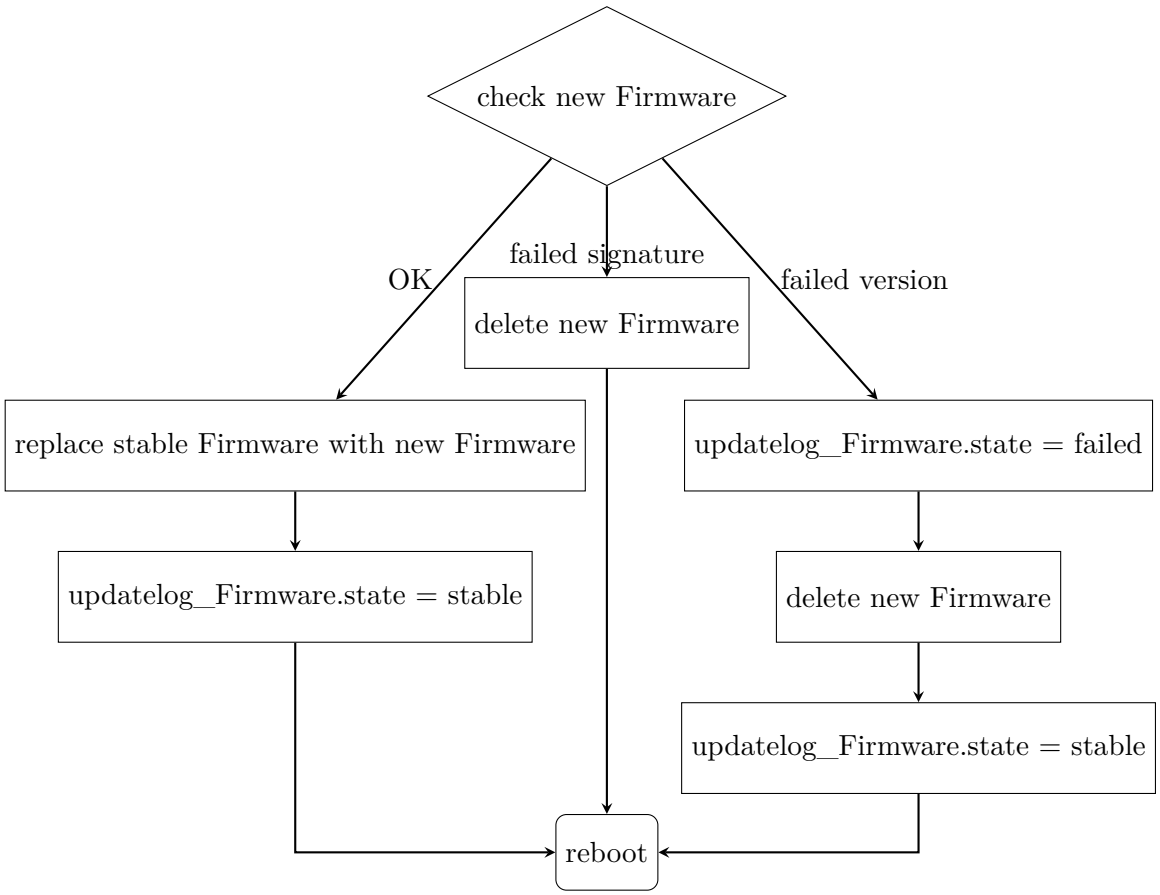

Figure 4.5: UpdateOS: control flow of the Firmware update. The updatelog_Firmware entries are referring to the entries with the version number of the new Firmware, except the entry after "delete new Firmware" that has the version number of the current Firmware.

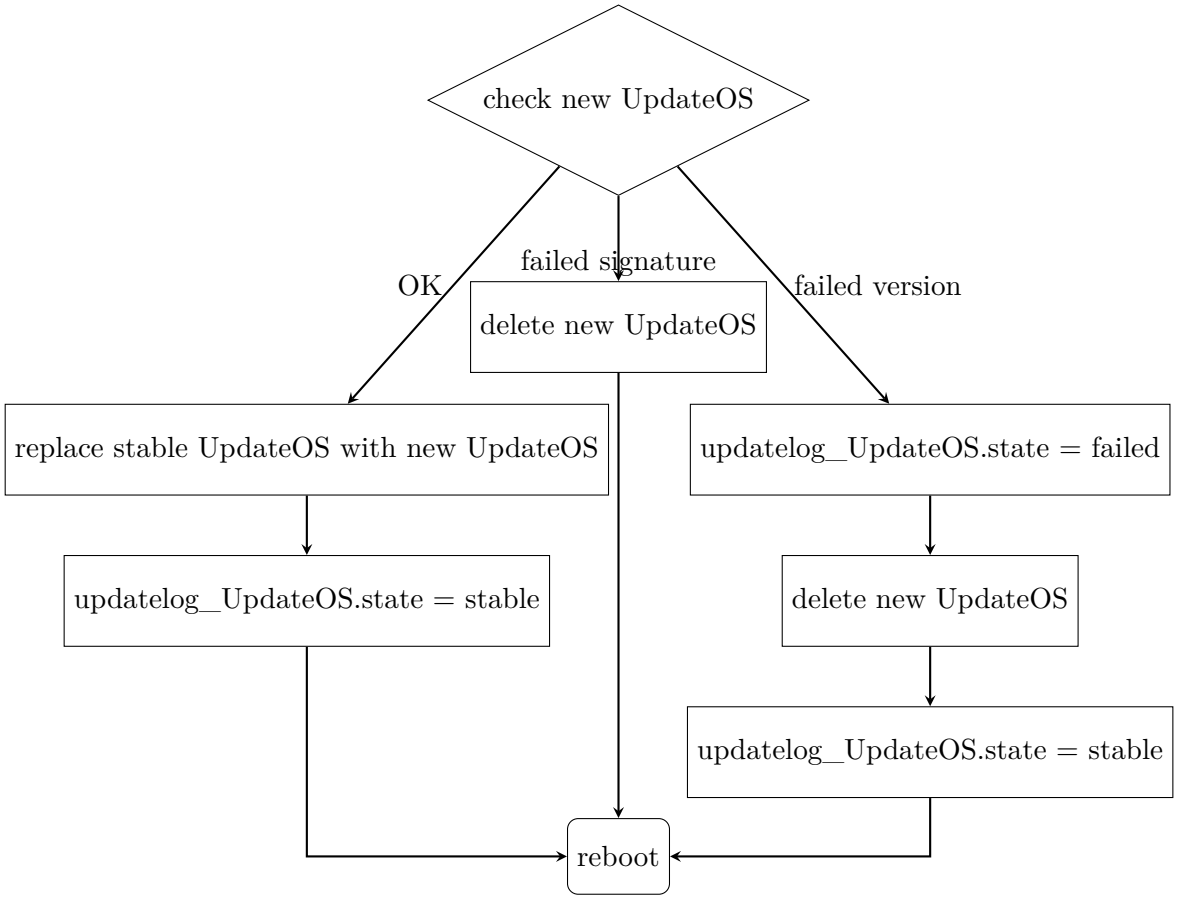

Figure 4.6: UpdateOS: control flow of the UpdateOS update. The updatelog_UpdateOS entries are referring to the entries with the version number of the new UpdateOS, except the entry after "delete new UpdateOS" that has the version number of the current UpdateOS.

However, in case of a MainOS update, both the new Image and the old image are kept on the BOOT partition. See in Figure 4.7.

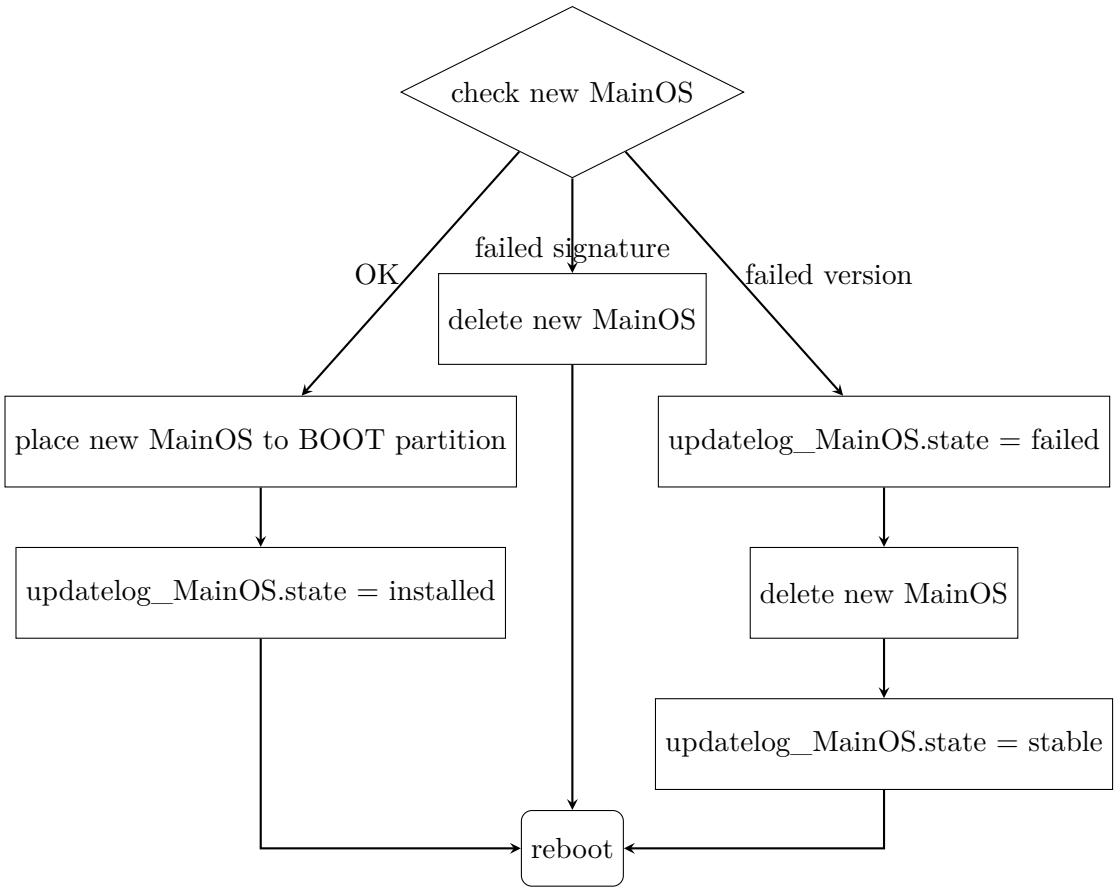

Figure 4.7: UpdateOS: control flow of the MainOS update (part 1). The updatelog_MainOS entries are referring to the entries with the version number of the new MainOS, except the entry after "delete new MainOS" that has the version number of the current MainOS.

If the digital signature verification fails or the version number in the Update is not greater than the version number of the stable Image on the Device, the new Image is deleted by the UpdateOS. In any case, an appropriate log entry is created in the corresponding update log file, except when the digital signature verification fails, in order to prevent putting untrusted version numbers in the update log files. Then the Device is rebooted. See in Figure 4.5, Figure 4.6 and Figure 4.7.

When the Device executes the secure boot process next time, U-Boot detects from the updatelog_MainOS that it should load and start the new MainOS for the first time. Then it writes in the updatelog_MainOS the version to be booted, and continues the secure boot process with the new MainOS. See in Figure 4.2.

After successfully verifying and mounting the new root file system, the new MainOS performs self-testing. The self-test consists of running the xtest test suite of OP-TEE and checking the network connection to the Controller. If everything goes well, the result of the self-testing is written in the selftestlog. See in Figure 4.3.

Upon next boot, the UpdateOS detects that the update was successful by observing the updatelog_MainOS and the selftestlog, so it deletes the old MainOS image from the device and logs in the updatelog_MainOS that the update was successful. However, the self-testing may fail or the new version of the MainOS may hang or crash. Such hangs or crashes are handled with a watchdog mechanism that reboots the Device. In this case, the UpdateOS detects the failed self-testing of the new MainOS by observing in the updatelog_MainOS that an new MainOS was booted, while missing any indication of a successful self-test in the selftestlog. When this happens, the UpdateOS deletes the failing update from the Device, logs the failure in updatelog_MainOS, and reboots the Device. The process is illustrated in Figure 4.8. After the reboot, the latest stable MainOS is loaded and executed.

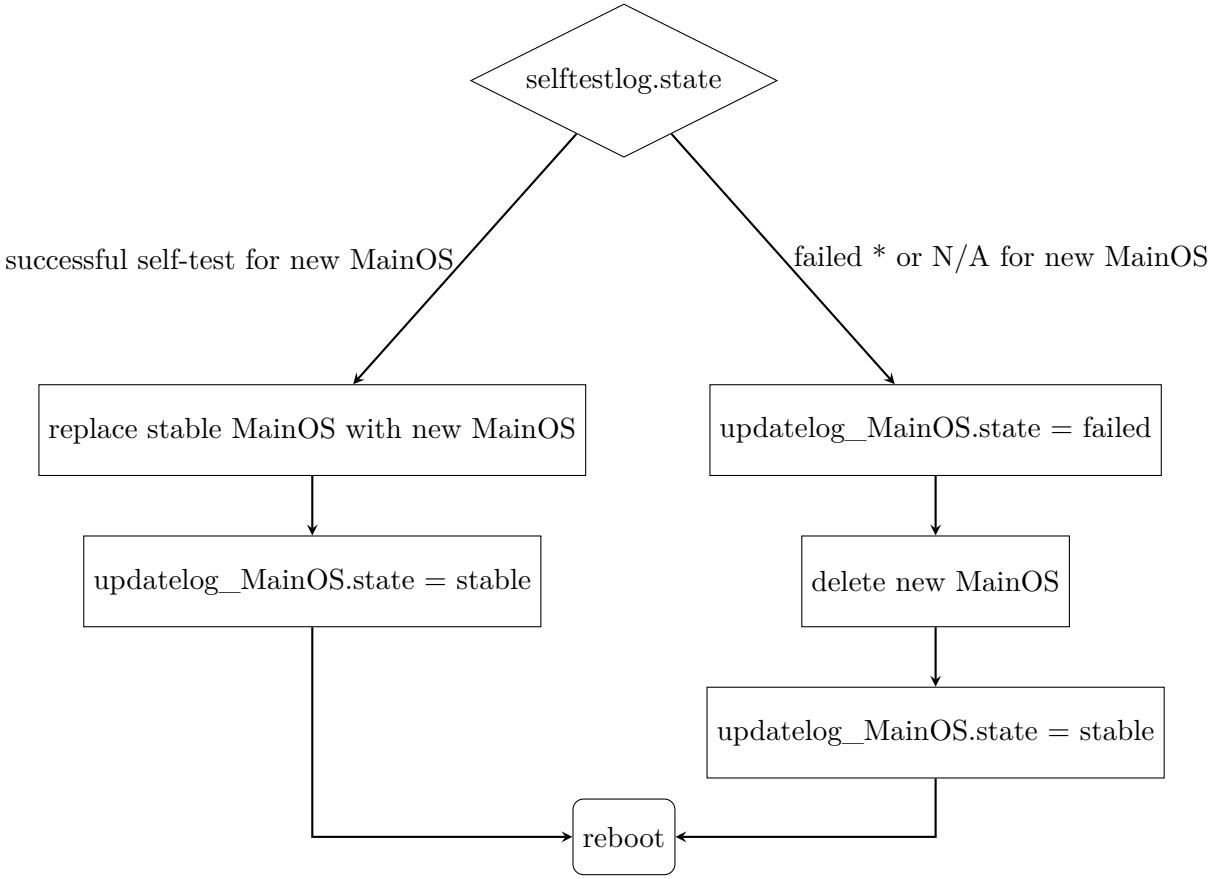

Figure 4.8: UpdateOS: control flow of the MainOS update (part 2). The updatelog_MainOS entries are referring to the entries with the version number of the new MainOS, except the entry after "delete new MainOS" that has the version number of the current MainOS.

4.4 Raspberry Pi 3 Model B

I implemented the above discussed secure update process on the Raspberry Pi 3 Model B. It is a widely used platform and it has a large software development community. It is also important, that it has the Broadcom BCM2837 System on a Chip (ARMv8-A) in it, which supports the ARM TrustZone technology, so it can run the TF-A.

Chapter 5

Evaluation

5.1 Formal verification

The described secure update process is complex enough to warrant for a thorough verification. For this reason, I used the UPPAAL model checker to model the secure update process and to formally verify its correctness. "UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.)."[1] It supports a limited version of Computational Tree Logic (CTL). My goal was to formally verify the following two requirements:

- **Update is possible:** When a given version of the MainOS is running and there is a functioning update available, it is possible to reach a state where this update is successfully installed.
- **Rollback is impossible:** It can never occur that a given version of the MainOS is successfully installed when a newer version was running and marked as stable in the past.

The "Update is possible" requirement can be formalized as following: Let's assume that initially the MainOS is running with the version number `v`, (`version == v`). Then the "Update is possible" requirement is the `E ((version == v) U (version > v))` CTL formula, which describes that a path exists in the model, where the version number goes from `v` to a number that is greater than `v` in the future. To formalize the "Rollback is impossible" requirement let's assume that initially the MainOS is running with an arbitrary version number `v`, (`version == v`) and it is not possible that somewhere in the future the version number goes from `v` to a number that is less than `v`. CTL formula: `A (not ((version == v) ==> F (version < v)))`.

[1] `http://www.uppaal.org/` (Accessed October 29, 2020)

Based on the previously discussed flowcharts I created three automata in UPPAAL, one for U-Boot, one for the UpdateOS and one for the MainOS. The "Update is possible" requirement can be formalized in UPPAAL: `E<> (mainos.running and (updatelog_MainOS_version > 2))` where the `mainos.running` is a state in the automaton of the MainOS that represents the stable running of the MainOS without errors, and the `updatelog_MainOS_version` is version number of the MainOS. To verify the requirement, I set the `updatelog_MainOS_version` variable to `2`, so I verified the requirements in a way that the model starts with 2 as the version number.

However, the "Rollback is impossible" requirement can not be formalized in UPPAAL. The `p ==> q` operator in the formula is eqivalent with the `A[] (p imply A<> q)` CTL formula. So there should be more than one CTL temporal operator in the beginning of the formula, and that is not supported by UPPAAL. To verify the requirement, I introduced the following auxiliary variables:

- `updatelog_MainOS_version_was_less_than_2`: Is true, when the automaton was in a state, where the `version < 2` formula was true.
- `updatelog_MainOS_version_was_3_and_state_0`: Is true, when the automaton was in a state, where the `version == 3` formula was true and it was with a stable version, `state == 0`.

With the help of the two auxiliary variables the "Rollback is impossible" requirement can be formalized in UPPAAL: `A[] ((not updatelog_MainOS_version_was_less_than_2) and (not (updatelog_MainOS_version_was_3_and_state_0 and mainos.running and (updatelog_MainOS_version < 3))))` that means the model will never be in a state that a version with version number less that 2 was running and after the stable running version with version number 3 there runs a version with version number less than 3. This solution satisfies the original requirement, which was formalized in a general way, because the value of the version number can be set to an arbitrary number, and by changing the semantics of the two auxiliary variables it is possible to verify that the "Rollback is impossible" for any version number.

5.2 Security

Security is achieved by installing only properly signed updates by the trusted UpdateOS and logging all relevant events to the update log files that cannot be modified by the potentially compromised MainOS or any applications running on it. Trust in the UpdateOS is based on the following factors:

1. The UpdateOS is signed and its signature is verified before loading and executing it.
2. The UpdateOS executes only for a limited amount of time.
3. The UpdateOS has a reduced functionality.
4. The UpdateOS can potentially be formally verified due to its stripped functionality.

5.3 Fail-safety

Fail-safety is achieved by using a watchdog mechanism that reboots the Device upon failures and by using log files in order to detect a failed self-test after an update. Moreover, the latest stable version of an updated Image is kept on the Device until the success of the update can be verified, so in case of failure, the Device can still boot the latest stable version.

5.4 Version rollback prevention

Finally, rollback protection is achieved by keeping information about updates in the update log files, which cannot be modified by the potentially compromised MainOS or the applications running on it, and by removing old versions from the Device after a successful update.

5.5 Limitations

Unfortunately the Raspberry Pi 3 Model B lacks a lot of security functions, for example it does not have a one-time programmable memory, and it does not have hardware write protection. As a result the implementations is only a proof of concept.

Chapter 6

Conclusion

An important security problem in IoT systems is the integrity protection of software, including the firmware and the operating system, running on embedded IoT devices. Digitally signed code and verified boot only partially solve this problem, because those mechanisms do not address the issue of run-time attacks that exploit software vulnerabilities. For this issue, the only known solution today is to fix the discovered vulnerabilities and update embedded devices with the fixed software. Such an update should be performed remotely in a secure and reliable way, as otherwise the update mechanism itself can be exploited to install compromised software on devices at large scale.

In this work, I proposed a system and related procedures for remotely updating the firmware and the operating system of embedded IoT devices securely and reliably.

First I researched the topic of secure remote firmware updates. Then I made my design plans based on that research. I designed a secure remote firmware update for embedded IoT devices, which is secure, fail-safe and prevents version rollback attacks. I also formally verified this design with UPPAAL.

Then I made a proof of concept implementation on the Raspberry Pi 3 Model B. However, it is a proof of concept only because the Raspberry Pi 3 Model B lacks some secure hardware elements. On another ARM A-Profile Device that has and utilizes those secure hardware elements the implementation is truly secure. And porting the implementation to such a Device is straightforward, with a possible improvement of using a formally verified microkernel such as seL4[1] as the UpdateOS.

[1] `https://sel4.systems` (Accessed October 29, 2020)

Acknowledgements

I would first like to thank my thesis advisors Dr. Levente Buttyán and Dorottya Futóné Papp, members of the CrySyS laboratory at the Budapest University of Technology and Economics. They provided me with valuable advice, guidance and encouragement.

The presented work was carried out within the SETIT Project (2018-1.2.1-NKP-2018-00004), which has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.

Bibliography

[1] N. Asokan, T. Nyman, N. Rattanavipanon, A. Sadeghi, and G. Tsudik. ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2290–2300, 2018.

[2] B. Choi, S. Lee, J. Na, and J. Lee. Secure firmware validation and update for consumer devices in home networking. *IEEE Transactions on Consumer Electronics*, 62(1):39–44, February 2016.

[3] Lukas Kvarda, Pavel Hnyk, Lukas Vojtech, Zdenek Lokaj, Marek Neruda, and Tomas Zitta. Software implementation of a secure firmware update solution in an IoT context. *Advances in Electrical and Electronic Engineering*, 14(4):389–396, 2016.

[4] Loren K. Shade. Implementing secure remote firmware updates. In *Proc. Embedded Syst. Conf.*, 2011.

[5] L. Szekeres, M. Payer, L. T. Wei, and R. Sekar. Eternal war in memory. *IEEE Security and Privacy Magazine*, 12, May-June 2014.

[6] TCG IoT-SG. TCG Guidance for Secure Update of Software and Firmware on Embedded Systems – version 1.0, revision 64. TCG Reference Document – Draft, July 2019.