University of Twente

Master Embedded Systems

Internship Neways Technologies B.V.

How to trust your embedded system

Student:	Geert Roks
UT supervisor	dr.ir. Marco Ottavi
Company:	Neways Technologies BV.
Internship period:	September 4th, 2023 - December 13th, 2023

January 7, 2025

Acknowledgment

This internship has been a great learning experience, which has exposed me to a lot of new concepts and ideas. Therefore I would like to thank the people that contributed to this experience. First of all, I thank Sander Huisman for providing guidance when I was stuck deep in a research rabbit hole. You introduced and helped me understand the Yocto Project, which I expect to be an invaluable skill for the rest of my career. Thank you to Marco Ottavi and Bruno Endres Forlin for making me think critically about security and help me improve my threat model. Furthermore, I want to thank Raymond Kaspers and Dennis Engbers, who made this internship possible and gave me the chance to this experience. Lastly, thank you to all colleagues at Neways Technologies B.V. for the warm welcome, the relaxing lunch walks and fun conversations.

Acronyms

SoC System-on-Chip

OCROM On-Chip Read-Only Memory

OCRAM On-Chip Random Access Memory

DDR RAM Double Data Rate Random Access Memory

PKI Public Key Infrastructure

CA Certificate Authority

PKCS Public Key Cryptography Standards

HSM Hardware Security Module

SPL Secondary Program Loader

OS Operating System

RoT Root of Trust

OTP One-Time Programmable

HAB High Assurance Boot

CST Code Signing Tool

SRK Super Root Key

CSF Command Sequence File

IVT Image Vector Table

IPC Inter-Processor Communication

SMP Symmetric Multiprocessing

AMP Asymmetric Multiprocessing

HMP Heterogeneous Multiprocessing

MU Messaging Unit

RPMsg Remote Processor Messaging

RDC Resource Domain Controller

ATF ARM Trusted Firmware (also refered to as TF or TF-A)

SMC Secure Monitor Call

TEE Trusted Execution Environment

REE Rich Execution Environment

TA Trusted Application

SW Secure World

NW Normal World

NS Non-Secure (referring to the NS bit of TrustZone)

TZASC TrustZone Address Space Controller

TZMA TrustZone Memory Adapter

TZPC TrustZone Protection Controller

Contents

1 Introduction

1.1 Assignment description

Neways Technologies B.V. is interested in improving the security of the devices they create, because the European Union has proposed "a regulation on cybersecurity requirements for products with digital elements" [1]. For this reason, the company would like to know how to establish a chain-of-trust in the boot process of an embedded device and what security that could provide for it. Additionally, during the internship, it has been decided to also put some research in understanding the ARM TrustZone technology as it can also help to improve the security of the device. Secondly, not relating to security, but the company wants to establish a bidirectional communication channel between a main processor and a co-processor within the same chip. These are the two main objectives of the internship and they need to be integrated in a build-layer for the build tool Yocto Project. In the end, a demo needs to be created that showcases the technologies that have been researched.

1.2 Development platform

The research topics are generally applicable to any System-on-Chip (SoC), but to confine the scope, the research is focused on the Variscite Dart-MX8M-Plus with the Var-DT8MCustomboard development platform. Neways is working on a project where these techniques will be directly applicable and the device for that project also uses the NXP i.MX8M Plus SoC, just like this Variscite board.

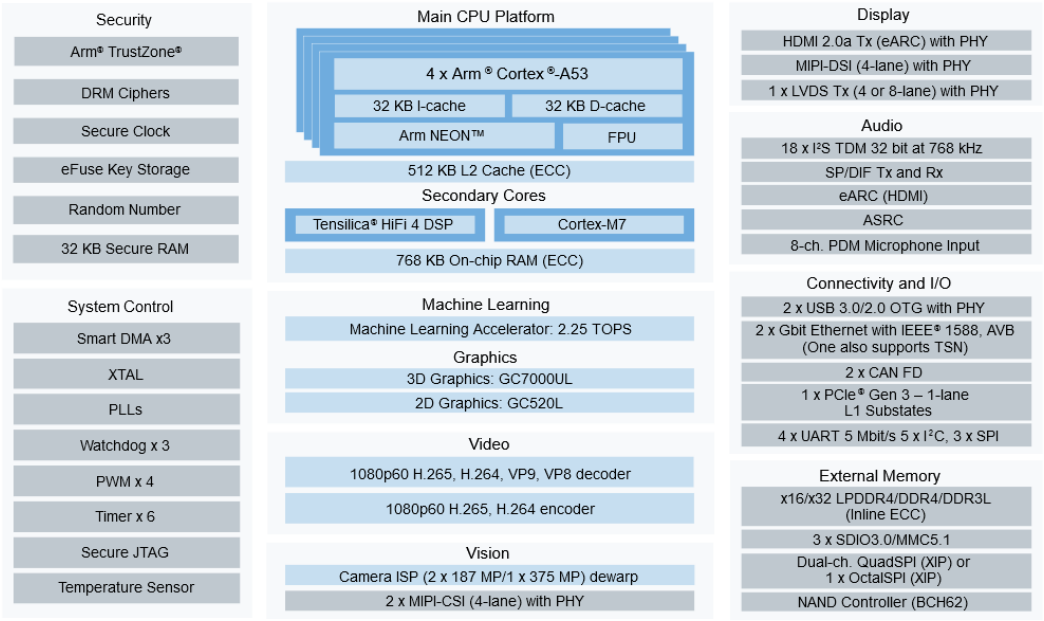


Figure 1.1: Diagram of the features of the NXP i.MX8M Plus SoC [2]

The NXP i.MX8M-Plus has a quad core Cortex-A53 processor cluster, which is the main processor of the SoC. Moreover, it provides some co-processors, such as a machine learning accelerator, a digital signal processor and a Cortex-M7 microcontroller core. The co-processors can be used to accelerate specific tasks which would be particularly inefficient to execute on the main processor. Furthermore, the SoC has many additional features, such as support for ARM TrustZone, peripheral busses (CAN, I2C, SPI, etc.) and more, see fig. 1.1.

1.3 Threat Model

Since the internship assignment concerns improving the security of a device, it is useful to define a threat model to be able to see whether the improvements are enough to reach the security goals of the company. A threat model helps to get a clear sense of what are the most critical vulnerabilities of the device. [3] introduced the STRIDE technique to help with threat modeling for developers. STRIDE is a mnemonic for Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege. This gives the developer a framework to think about possible threats and formulate actionable countermeasures.

Shostack describes in his book [4] different types of attackers or anti-users. An anti-user has two important properties: their skill and their motivation. Figure 1.2 shows an overview of anti-users defined by these two properties. The left bottom anti-users are the easiest to protect against and the right top are the toughest anti-users to protect from. These anti-user archetypes are useful to set security goals for a project, because securing every device for any kind of attacker is both unattainable and unreasonable.

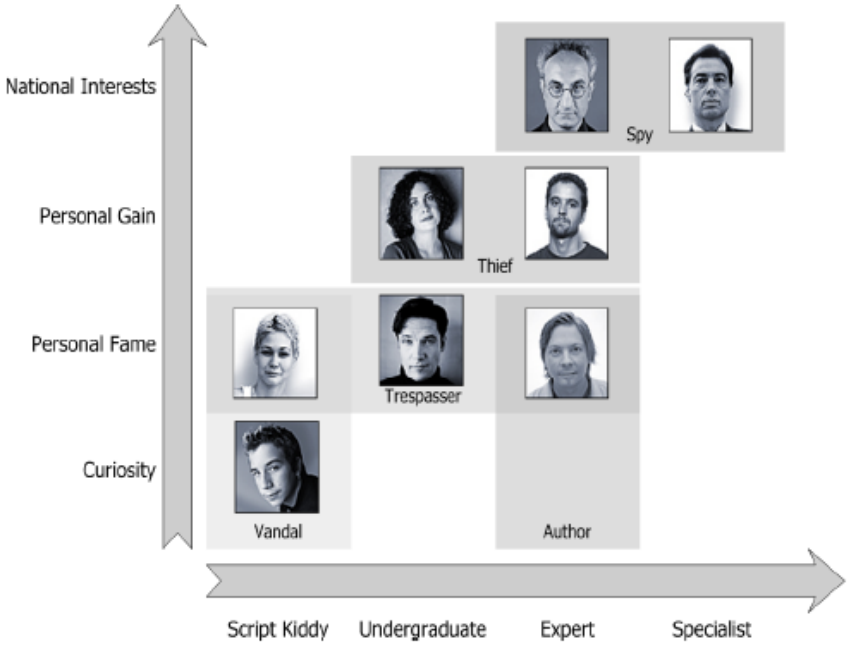


Figure 1.2: Diagram of types of anti-users described by their skill and motivation from [4]

The threat model for the demo assumes that an attacker has direct physical access to the device and has the capability to add malicious software to the unsafe portions of the system. The attacker does not have the ability to perform hardware level side-channel analysis or fault injection and is limited to the external ports of the device (USB, HDMI, Ethernet, micro SD-card and power barrel plug). This threat model is

inspired by the project that also uses the NXP i.MX8M Plus SoC which was mentioned in section 1.2. This project does not expect to attract the toughest anti-users and has anti-intrusion mechanisms in place, which should make it even harder to perform a hardware attack. The demo should be able to be resilient against most of the attacks of the undergraduates with a personal fame motivation and lower skilled anti-users. Also limit the attack surface as much as possible for the expert anti-user with the personal gain motivation.

1.4 Structure of the document

The rest of the document is structured as follows. Chapter 2 provides background on public key cryptography, and introduces the secure boot and ARM TrustZone technologies. This chapter also introduces methods to communicate between cores within the same SoC. Then chapter 3 shows how these technologies are implemented in a demo and chapter 4 discusses the results of the implementation. Finally, chapter 5 will conclude this report with future improvements which came up during research, but did not have time to be implemented. The appendices contain extra implementation details and development information.

2 Literature Review

2.1 Cryptography Background

When securing devices, cryptography will always be needed. Therefore, first a small section to provide some background on the cryptographic concepts that are used in this report.

2.1.1 Hash functions

A hash function is a mathematical function that can take an arbitrary length of data and produce a stream of characters with a fixed length in such a way that the input can not be deduced from the output. Also, for the same input the hash function will produce the same output every time. When there is a small change in the input, the hash will change very significantly. Therefore, when you have a known-good hash of a piece of data, then this can be used to verify the integrity of that particular data at any point. The data that needs to be verified is also hashed and compared against the know-good hash, when they match the integrity is assured. The hash is a space efficient and safe way to verify integrity of a piece of data, as long as the hash can be trusted. The FIPS 180 [5] or better known as SHA hash functions are often used and secure hash functions. Except for SHA-1, which is not considered secure anymore [6].

2.1.2 Public Key Cryptography

Cryptographic keys can be used to store and transport data in a state where the original data cannot be deduced (ciphertext), except for when the reader has the correct key to decipher the data to its original state. Symmetric key algorithms encrypt and decrypt data with the same key (the secret key). This makes it hard to use when the writer and reader of the data are not the same device, because the key needs to be shared in a safe manner, but the key was also meant to share the data in a safe manner. Also, if the key gets compromised, then the key provides no security anymore.

Public key cryptography or also know as asymmetric key cryptography solves this problem. An asymmetric key is a key pair, where one can be used to encrypt the data and the other is used to decrypt it. One of these keys is made public and the other kept private, therefore also called the public and private key. These keys are generated in a way that when you have one key, it is extremely hard to calculate the other. So, the security of public key encryption relies on the private key to stay private, whereas the public key can be freely shared.

The public key can be used by anyone to encrypt a message that only the holder of the private key can decrypt. The other way around, the holder of the private key can digitally sign a message by adding a signature to the end of a message. This signature is a combination of the message and the private key. Anyone can then use the public key to verify the signature by decrypting the signature and comparing it to the message that was send. When the signature matches the message, you know that the message was sent by the holder of the private key and that the message was not altered during transit.

2.1.3 Public Key Infrastructure

Asymmetric keys are often managed by a Public Key Infrastructure (PKI). This infrastructure is a tree structure with a Certificate Authority (CA) at the root of this tree. A digital certificate, or also known as

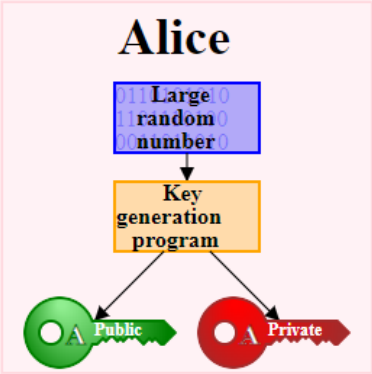


Figure 2.1: Asymmetric key generation: two keys that are mathematically related [7]

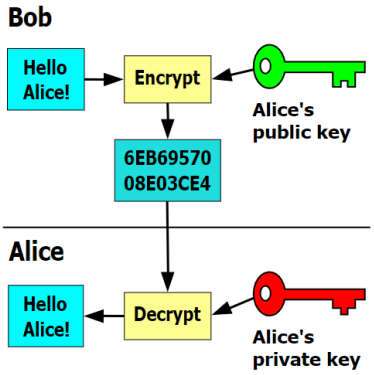


Figure 2.2: Public key encryption: message only readable by private key holder [8]

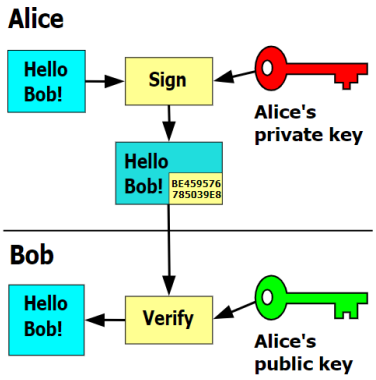


Figure 2.3: Private key signing: message readable by anyone, but sender is verifiable [9]

a public key certificate, is a document that proves the validity of a public key. The certificate contains the public key and information about the key, such as information about who generated the key (the subject), the validity period of the certificate and a digital signature of an entity that has verified the certificate (the issuer). When a device receives a certificate, it will check whether it trusts the issuer and that the certificate is still valid. If those checks are passed, then the device trusts the public key that is inside the certificate and can communicate securely with the subject of the certificate.

Any certificate can issue their own certificate, resulting in a tree with the CA at the root. The CA is a certificate itself, where the issuer and the subject are the same (also known as a self-signed certificate). This certificate needs to be trusted initially, also know as the Root of Trust (RoT). To be a trustworthy CA, it has to ensure that the certificates that it issues, have trustworthy subjects. When you set up and control your own CA, you can trust the certificates that it issues. However, you have to convince others to trust your certificates. Depending on your goals, this might not be a problem.

The advantage of a PKI is that all keys can be verified whether the developer actually generated them. All keys will have the same CA and only the developer will have control over the CA. Also, if a key gets compromised, then the certificate can be revoked. As long as the access to the CA is not compromised, then the PKI will protect all certificates that are issued by the CA.

2.1.4 PKCS #11

The Public Key Cryptography Standards (PKCS) are a set of public key cryptography standards developed by RSA Laboratories[1]. PKCS #1 [10], for example, describes the implementation for the RSA public key algorithm and PKCS #8 [11] describes a syntax for the private key information, which has the possibility to get encrypted by a password with a password-based encryption algorithm described in PKCS #5 [12]. Currently there are 15 PKCS standards, but some of them have already been withdrawn.

PKCS #11 [13] is the eleventh standard of this set. This standard describes a platform-independent API to interface with HSM. An HSM can safeguard and manage keys, as well as perform encryption and decryption with these keys. An HSM also has to option to store the keys on a cryptographic tokens. A token is a piece of hardware that can securely store data. For example, a smart card can contain a uniquely identifiable key that gives the holder of the card certain access rights to a building. An HSM contains slots in which tokens can be provided. In the example, the smart card readers next to the doors are slots to

[1] `https://www.rsa.com/`

what is probably an HSM availble on the network of the building.

The API that PKCS #11 describes is called *Cryptoki*, which is short for cryptographic token interface. This API has calls for the creation of tokens, as well as objects that get stored in the token. An object can be a piece of data, a certificate or a key (either public, private or secret). The objects need to be provided some attributes during creation, to define the actions that can be performed on them. The API can then make a call to digitally sign a piece of data using the private key in the token or to verify a signature using the public key. The PKCS #11 API provides a simple and portable interface to any device that supports this API.

2.2 ARM TrustZone

Modern ARM based SoCs often have ARM TrustZone technology build into the chip. This technology can help in establishing system-wide hardware-enforced security. There are two flavors of TrustZone: *TrustZone for Cortex-A*[2] and *TrustZone for Cortex-M*[3]. Only *TrustZone for Cortex-A* is discussed, because *TrustZone for Cortex-M* is not supported on the Cortex-M7 core included in the NXP i.MX8M-Plus SoC.

2.2.1 What is TrustZone

TrustZone implements domain-based isolation enforced by hardware directly into the SoC. It not only isolates the processor, but also the memory, system bus, interrupts and peripheral access. [14] explains that a TrustZone-enabled processor can execute instructions in four different privilege levels (Exception Levels – EL0-EL3) and two security domains (Normal World (NW) and Secure World (SW)) at any given time. Switching between the NW and SW is done by the ARM Trusted Firmware (ATF), which runs with the highest (EL3) privilege. EL3 is also called "monitor mode". The other privilege levels are used by the NW and SW. EL2 can optionally be used for a hypervisor, EL1 is used for running the Operating System (OS) kernel and EL0 for executing the applications.

The NW can use the Secure Monitor Call (SMC) to request the SW to run [14], [15]. A context switch will then be performed via the monitor mode. The NW execution state gets frozen and the SW context is started and executes anything that the NW asked it to. In a multi-core SoC, each processor can independently switch between worlds.

TrustZone also provides some optional components to isolate memory and cache, based on the security domains. The TrustZone Address Space Controller (TZASC) configures and manages on-chip memory to physically isolate the SW memory from the untrusted NW memory. The TZASC sits between the system bus and the memory chip. It reads the Non-Secure (NS) bit from the Secure Configuration Register (SCR) to determine whether a memory access is requested by the NW or SW. The SW can access both memory regions of the SW and NW, whereas the NW is only able to access its own memory regions. Furthermore, the TrustZone Memory Adapter (TZMA) provides the same functionality as the TZASC, but targeting off-chip ROM and SRAM [15]. Cache-level isolation is accomplished by adding an extra bit to the cache table which indicates to which world a cache-line belongs. Lastly, the TrustZone Protection Controller (TZPC) is another optional TrustZone component which can restrict system devices to either the SW or NW.

2.2.2 Trusted Execution Environments

The main use case of TrustZone is to establish a Trusted Execution Environment (TEE). An TEE is a secure OS that runs in the SW. Often an TEE is lightweight and has a minimal feature set to reduce

[2] `https://www.arm.com/technologies/trustzone-for-cortex-a`
[3] `https://www.arm.com/technologies/trustzone-for-cortex-m`

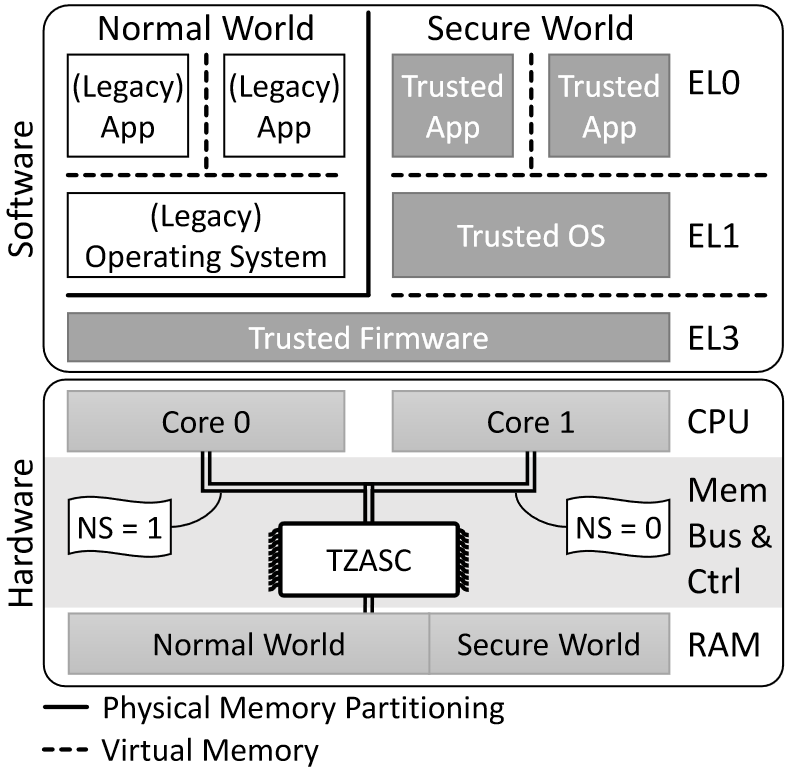


Figure 2.4: The software and hardware components of TrustZone [14]

attack surface. To make a distinction between this secure OS and the non-secure OS that runs in the NW, the non-secure OS is often referred to as Rich Execution Environment (REE). Just like in the REE, a TEE can run applications in a user-space. The TEE has, however, a secure user-space which runs Trusted Applications (TAs). These TAs run with the lowest privilege level (EL0). A TA is stored in the file system of the REE. They are however still trusted, because they are digitally signed (and sometimes encrypted) to ensure that the application is in the exact state as the developer has written it.

A TEE is mostly used to manage and store secrets in a way that they are never exposed to the untrusted NW. The TEE can run TAs that manage secure storage or cryptographic devices. The REE can then make API calls to request certain data to be signed/verified or encrypted/decrypted, without ever having to touch the private/secret keys. This means that if the REE is ever compromised, the most important and sensitive data is still safe in a harder to break system with minimal attack surface. This technique can be used to establish secure network connections (ie. store wifi credentials or manage a TLS handshake), authenticate to a cloud service, securing over-the-air firmware updates [16], managing and securing a (relational) database [17] and more.

2.2.3 OP-TEE

There are multiple runtimes that can function as TEE, but a well-supported, open-source and portable implementation that runs on the i.MX8M-Plus SoC is OP-TEE [18]. OP-TEE is divided in multiple components. The `optee_os` repository contains all components for the SW implementation, such as the secure kernel and a set of secure user space libraries for the TAs. Furthermore, the `optee_client` supplies the NW with an API for communication with OP-TEE and a user-space daemon (`tee-supplicant`). These components can communicate with the SW using the SMC which is supported by the OP-TEE driver that

is available since Linux 4.12 [19]. Optionally, `optee_test` supplies a test utility (`xtest`), which can be used to do regression testing and testing the API. OP-TEE is very minimal and does not provide a scheduler, so the execution is driven by calls from the NW.

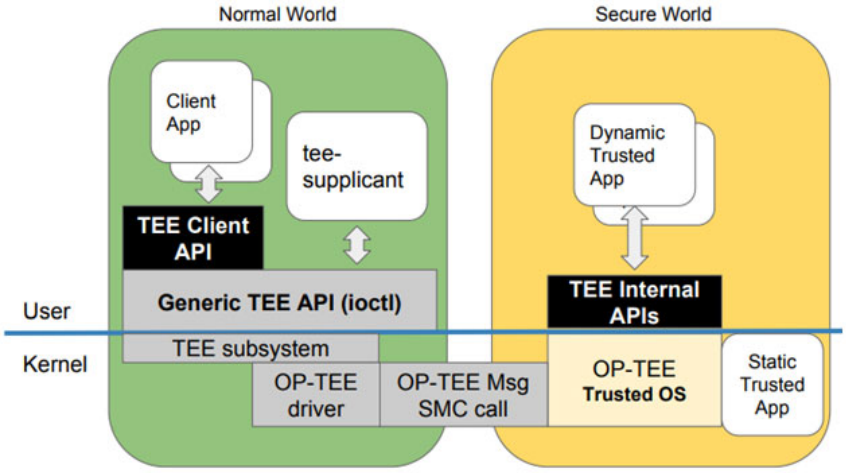


Figure 2.5: Overview of the components of OP-TEE [20]

2.3 Boot Process

This section first investigates how a device can boot from a power-off state to a fully running OS. Afterwards, the secure boot mechanism is discussed in general and the NXP implementation of it. Finally, it is described how TrustZone affects this secure boot mechanism.

2.3.1 Unsecured Boot process

A computing device can use code to initialize certain hardware. This code can be used to boot the entire device. However, a program needs to be stored in memory, but this memory is also a device that needs to be initialized. So to start a device from complete power off, a "chicken and egg"-type of problem occurs.

The way to solve this, is to incrementally initialize systems of the device during the boot process. Each increment is called a boot stage. The first boot stage is started when the device is turned on. In this stage, also called the Primary Program Loader, the instruction pointer of the CPU is pointed to a hard-coded location in the On-Chip Read-Only Memory (OCROM). The processor executes the code that is stored here directly from this OCROM. This ROM contains very basic instructions to interface with the most important hardware, including instructions to read from the eMMC persistent storage. This first stage loads the next boot stage from the eMMC into the On-Chip Random Access Memory (OCRAM) and hands the control over to this next stage.

The next boot stage is the Secondary Program Loader (SPL). The goal of this stage, is to initialize the Double Data Rate Random Access Memory (DDR RAM) of the system. The reason that this step is needed and not just load everything at the same time, is that the size of the OCRAM is very limited, so only a very small program can be loaded. When the SPL has executed, the DDR RAM is available and larger boot stages are able to be loaded. The SPL loads the next boot stage into the freshly initialized DDR RAM and hands the control over to it.

Now that the DDR RAM is available, every other piece of hardware can get initialized by programs which are ran in the DDR RAM. The boot stage that does this is called the bootloader. There are many

bootloader variants, but the one that is used here is called Das U-boot, or commonly called u-boot. The SPL mentioned before is also available in many variants, including one from u-boot, called U-Boot SPL. Once u-boot has initialized all the peripherals, it will mount the root file system and load a kernel in the DDR RAM and hands the control over to it.

The kernel is a program that is part of an OS. It contains many device drivers that make the peripherals available to the OS. The kernel also contains a scheduler that organizes all tasks that need to be executed based on a priority given to the task. The OS has next to the kernel space programs, also user space programs. These programs are the applications that give the device its intended behavior. The kernel has access to all of the hardware, whereas the user space programs only have access to what the kernel allows them. Now that the kernel is loaded and has control, the device is considered to be booted.

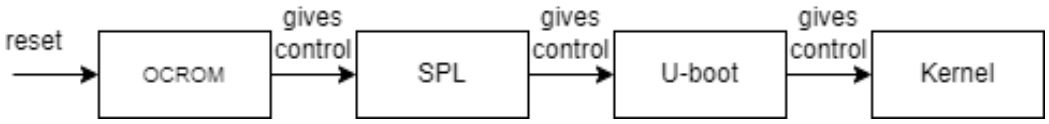


Figure 2.6: Boot flow

2.3.2 Secure boot

The boot process described in section 2.3.1 provides no assurances that the boot stage that was specified by the developer is loaded and given control. The memory of the system could have corrupted a part of the boot stage instructions, or someone could have maliciously changed any boot stage to run software that don't comply with the goals of the developer. Therefore a system that was booted in that way cannot be trusted to behave as is it should. [15] states that "to establish trust in a system we need to assure authenticity and integrity of the system". Secure boot uses cryptographic keys and hash functions to verify each boot stage. These are used to sign digital certificates, which can prove the authenticity and the integrity of the boot stage.

To have trust in the OS kernel, one has to be sure that all boot stages before it have executed as they should. So, U-boot has to be trusted, but it has the same problem as the kernel. This goes all the way down to the very first boot stage. It is easy to see that there is a need for a chain of trust, because only something can be trusted, when all that came before is trusted and trusts its successor. But to have a chain of trust, at some point something needs to be trusted initially. This is called the Root of Trust (RoT).

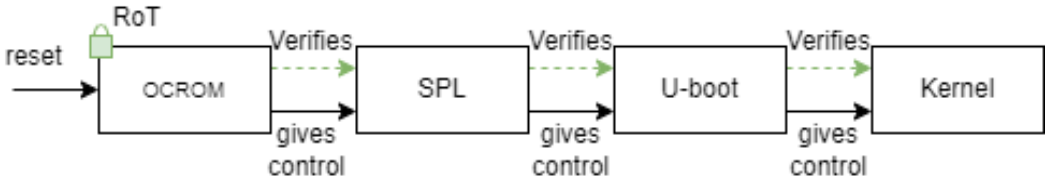


Figure 2.7: Secure boot flow

There are many components that can be used as a RoT. Some examples of these are One-Time Programmable (OTP) memory, Physical Unclonable Functions (PUF) and Trusted Platform Modules (TPM). OTP memory is a special kind of memory that can only be written to once, making this memory tamper-proof. A PUF is a device that whose output can be seen as a unique identifier for that specific device. A PUF usually uses some random process that gives the same output for the same input on the same device, but a different output on another device. Therefore a PUF can uniquely identify a device and prove its authenticity. The TPM is a secure cryptoprocessor that is defined in an international standard (ISO/IEC

11889). It contains a cryptographic processor with a random number generator, key generator and more cryptographic functions, as well as some secure storage to store keys.

These RoTs are then used to store a public key. The public key can be used to verify digital signatures that were created by the developer using the private key of the same key pair. Assuming the developer is the only one with access to the private key, the authenticity of the image can be proven. To prove the integrity of the image, a hash function is used to create a hash of the image and included in the digital signature. Then when the authenticity of the boot stage is verified by the public key, then the image is hashed and compared against the hash included in the signature. If the hashes match, then the integrity is also verified.

2.3.3 High Assurance Boot

The secure boot implementation of NXP on the i.MX8M-Plus is called High Assurance Boot (HAB)[21]. It uses OTP memory called e-fuses as the RoT. Figure 2.8 shows a simplified overview of how HAB works both in the development time and boot time.

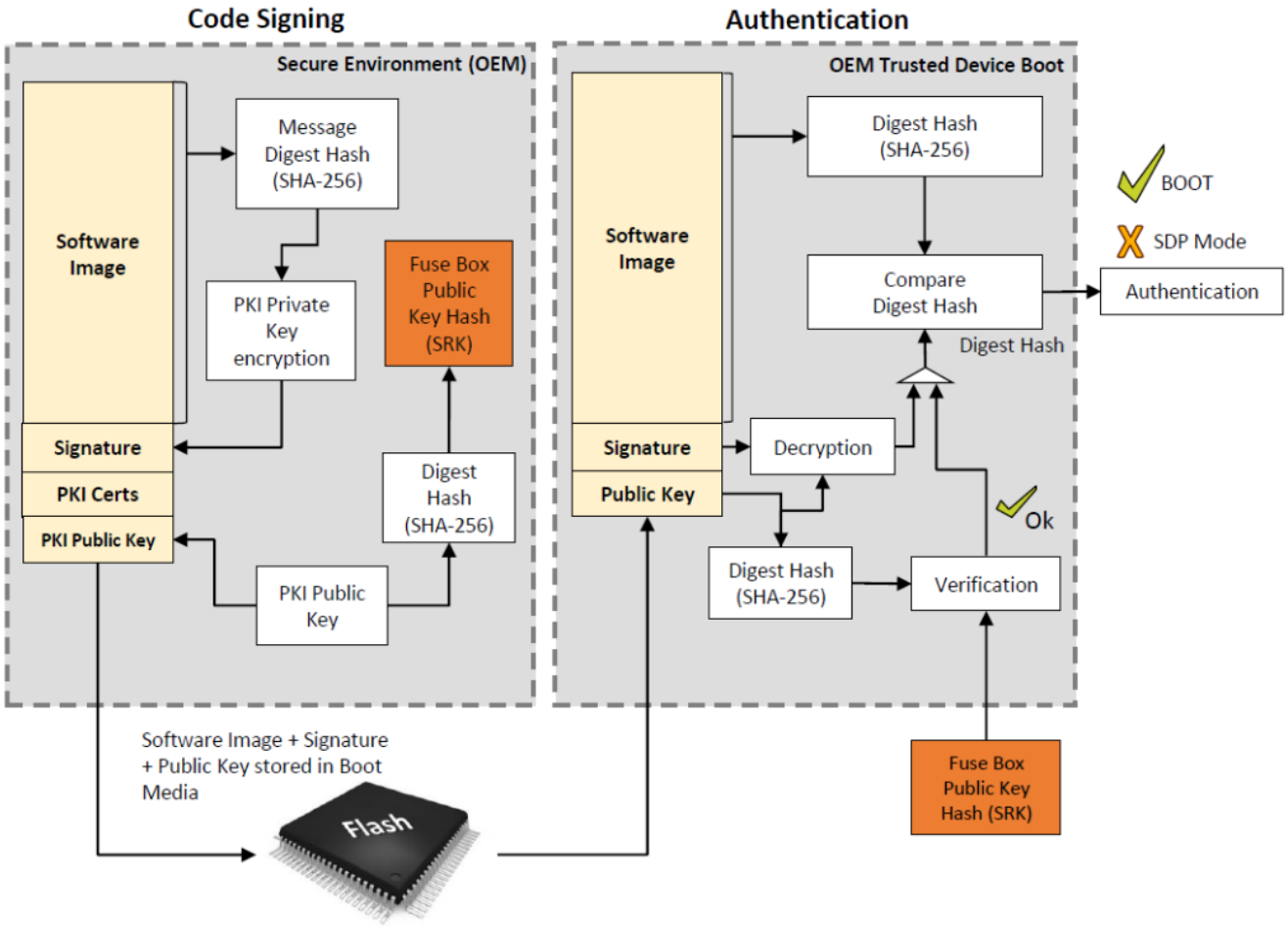


Figure 2.8: HAB Secure Boot process overview [21]

The left box describes the development time process. The *Software Image* is the boot stage. This is hashed and signed by a private key. It is then also stored on the boot medium. The corresponding public key is also added to the boot medium, but also hashed and stored in the e-fuses of the SoC.

However, before this can happen, a public/private key pair is needed. These are generated using the NXP Code Signing Tool (CST) [22] [23]. The CST generates a PKI tree, see fig. 2.9. The PKI tree consists of a CA at the root which issues 4 public/private key pairs called Super Root Key (SRK). Each SRK can also issue other key pairs. The public keys of the SRK are the ones that are hashed and stored in the e-fuses.

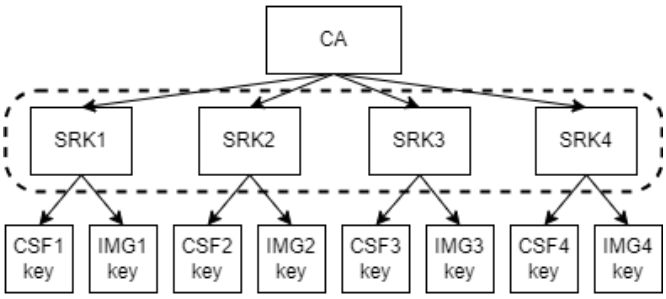


Figure 2.9: PKI tree

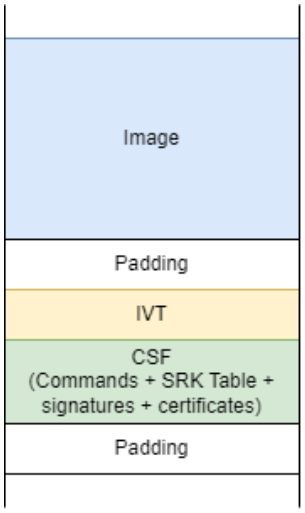


Figure 2.10: HAB memory layout

The right box of fig. 2.8 shows the boot authentication process of HAB. The public key that is stored in the boot medium is hashed and compared against the hash of the public key stored in the e-fuses. That verifies that the key pair is used of which only the developer should have the private key, thus verifying authenticity. The public key can then be used to verify the signature. Because the signature is a digitally signed hash of the actual image, the hash stored in the signature can be used to compare against the hash of the actual image that is supplied. If they match, then the integrity of the image is also assured and the boot stage can be executed.

Figure 2.8 leaves out one important detail. During booting, instructions are needed to execute these verification steps. The HAB program in the ROM of the SoC is responsible for this. It will read a Image Vector Table (IVT) and an Command Sequence File (CSF) from the image. The IVT is very small table with the memory locations of each part of the boot stage. The CSF contains the actual instructions, like which of the four SRKs need to be used and which parts of the image are signed. During the code signing process, the CSF is also signed by a CSF key to verify that the CSF is authentic. This CSF key is issued by one of the SRK keys. The SRK Table, signatures and certificates are all kept in the CSF. See fig. 2.10 for a schematic overview of the memory layout of a boot stage for HAB.

2.3.4 TrustZone and Secure Boot

For TrustZone to operate correctly and the SW to be trusted, the device has to be started with Secure Boot enabled. In fact, the trust in the secure world is based on the Root of Trust provided by the Secure Boot mechanism. With the addition of TrustZone, the boot flow presented in fig. 2.6 and fig. 2.7 is a bit altered. Figure 2.11 shows the bootflow with ARM TrustZone enabled. The ATF introduces specific names for the boot stages. BL1 is the BootROM which is stored in the OCROM. BL2 is the SPL and platform specific firmware, which initializes the DRAM, configures TZASC and validates the next boot stages (BL31, BL32 and BL33), before loading them and passing control to BL31. The BL3x stages are stored together in one

binary, called FIP (Firmware Image Package) or sometimes FIT (Flattened Image Tree) image. BL31 loads and initializes the ATF binary. If an TEE binary is present, BL31 will pass control to BL32, which will initialize the secure OS before handing the control back to BL31. When BL31 has finished initializing it wil pass control to BL33, which will start the NW bootloader and eventually load the NW OS kernel and any other firmware images, if specified.

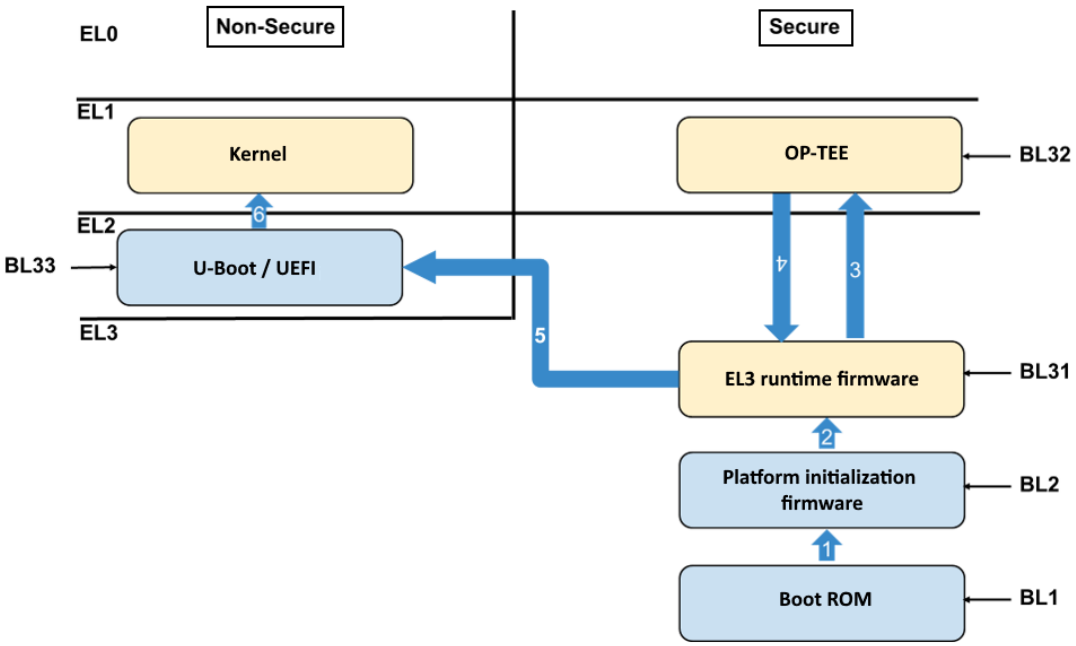


Figure 2.11: The boot flow with ARM TrustZone [24]

2.4 Inter-Processor Communication

Modern SoCs often have multiple cores in them. These cores can work together to speed up the calculations. There are multiple configurations to perform multiprocessing. Firstly, a Symmetric Multiprocessing (SMP) configuration contains the multiple of the same processors who split the computational load equally over them. Then there is also Asymmetric Multiprocessing (AMP), where there are also the same kind of processors used, but they have different loads. For example, two cores of a quad core CPU run a Linux kernel and two other cores run a real-time operating system like Xenomai. Next to having an unequal load, the cores also often have access to different peripherals and have their own address space[25]. Lastly, there is the Heterogeneous Multiprocessing (HMP) configuration. This configuration has processors which are not of the same kind. These different processors are often specialized in different tasks and can thus split the load in an unequal way, but which plays into both of their strengths making the computation more efficient[26]. However, HMP is also often referred to as AMP.

These multiple processors need a way to communicate, so that the main processor can outsource tasks to the specialized processors. To achieve this an Inter-Processor Communication (IPC) needs to be established. There are multiple ways to move data from one processor to the other.

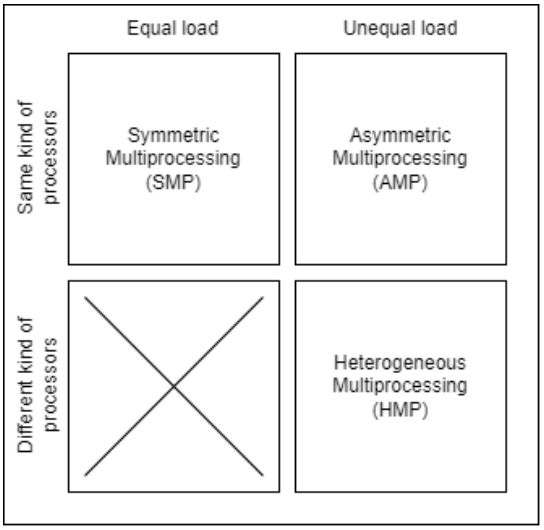


Figure 2.12: Diagram showing differences between SMP, AMP and HMP

2.4.1 Shared Memory

Shared memory can be used as a simple form of IPC. In fact, this is often the main way how the the processors in a SMP configuration communicate. As the name suggests, multiple processors share the same memory addresses, so that one core can write a message in this shared memory and another core can read from it.

This method needs the processors to synchronize, because if one processor tries to read from a memory address where another processor is writing to at the same time, then a race condition occurs. This synchronization can be provided by a semaphore. One processor can take the semaphore and perform its read or write instruction on the shared memory. If the processor is done, then it will release the semaphore again. While a processor holds the semaphore, no other processor can reach the shared memory and they will have to wait until it is free again. This ensures that only one processor at a time performs operations on memory.

2.4.2 Mailboxes

Some SoCs contain a specific piece of hardware that allows the processors to send messages to each other[27]. This hardware is called a mailbox and can interact with the common mailbox framework in Linux [28]. However, since this hardware module is mostly proprietary, the actual drivers for the device are very platform specific and need to be implemented for each new platform.

The NXP i.MX implementation of this hardware is called the Messaging Unit (MU). This piece of hardware has two interfaces, one for each processor. This allows for each processor to interact with the MU using their own clock speed. The MU manages the synchronization between the processors. It does this by using two sets of matching registers, one set facing processor-A and the other set facing processor-B. Each side has four, 32-bit read-only receive registers and four, 32-bit write-only send registers. The MU also has a status and control register, each of 32-bit and is able to send interrupts to the other processor to signal its state [29].

The mailbox can be used for very small messages of only a few words (four words in the case of the MU). For large message the mailbox can pass some frame information to notify the other processor where in shared memory the large message is. When the other processor has read the message in shared memory, it can then use an interrupt to signal that the data was written. In this manner, the registers and interrupts

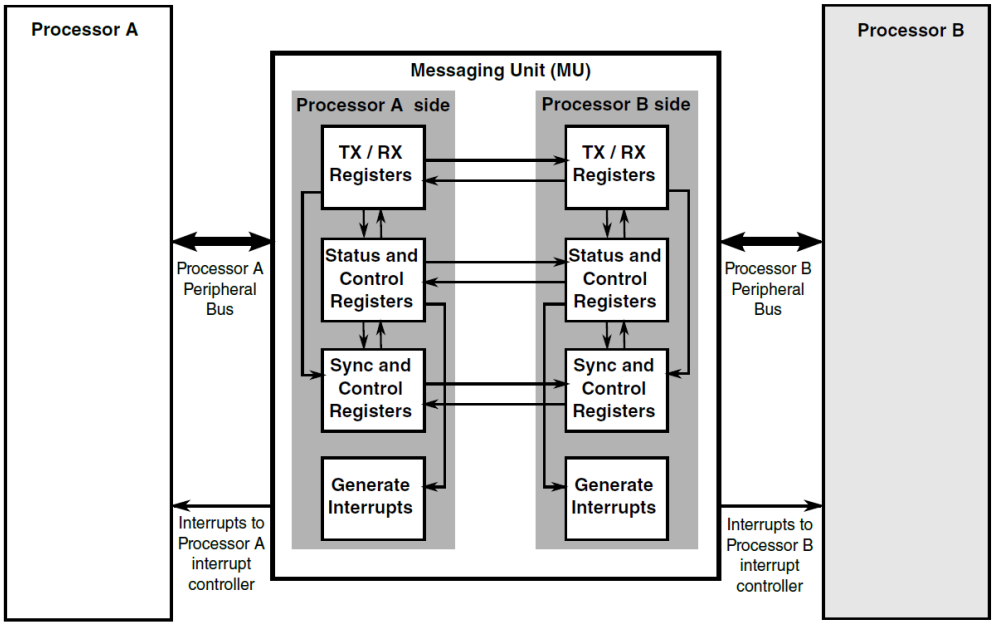


Figure 2.13: NXP i.MX mailbox hardware called Messaging Unit [29]

can be used to send messages and notify each other of their own state.

2.4.3 Remote Processor Messaging

The Remote Processor Messaging (RPMsg) framework implements both the shared memory and mailbox methods from before and builds an abstraction layer over them. This framework is intended to be a generic way to setup IPC [30] and is even integrated in the Linux Kernel [31]. It uses VirtIO, which is an abstraction layer over the host's devices. It provides a virtual IO bus on top of the platform bus of the host. The RPMsg bus registers to the VirtIO bus and a character device driver is then registered to the RPMsg bus. This character device allows the creation of RPMsg channels from the user space.

An RPMsg channel is denoted by a name string. The framework uses these names to keep track of the channels. Each channel can have multiple endpoints and each endpoint is denoted by a unique source address. The advantage of this is that a single channel can provide multiple receive callbacks. The sender can specify to which endpoint a message has to be send by setting the destination address in the message header. The header also needs the source address and the length of the payload. There is space reserved for flags, but these are not yet specified and thus unused [33].

The RPMsg message gets written to a VirtQueue. This is a part of VirtIO. A VirtQueue provides two Vrings, one for each direction of communication. A Vring is a circular buffer stored in shared memory. It manages this shared memory without the need for synchronization like semaphores, due to its single-writer single-reader circular buffering technique [33][34]. Also, the two processors can "kick" each other to notify of an update in the Vrings. When available, these inter-core interrupts are managed by a mailbox.

The framework has one "master", often the Linux Kernel, to which multiple "remotes" can connect to. At the time of writing, it is only possible to dynamically allocate RPMsg channels. A remote processor announces their existence to the master by sending a name service message. This message tells the master of the channel name and source address of the remote RPMsg service. The master then creates and registers a channel on the RPMsg bus with the name provided by the remote and sets their local address. To tell the

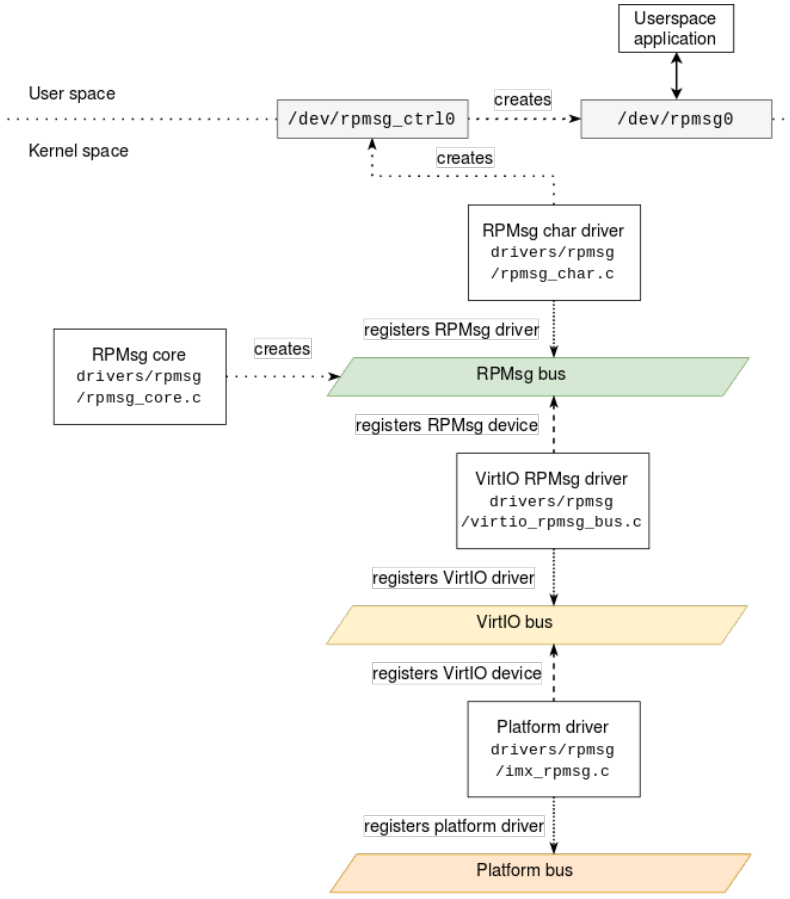


Figure 2.14: Driver structure of RPMsg framework [32]

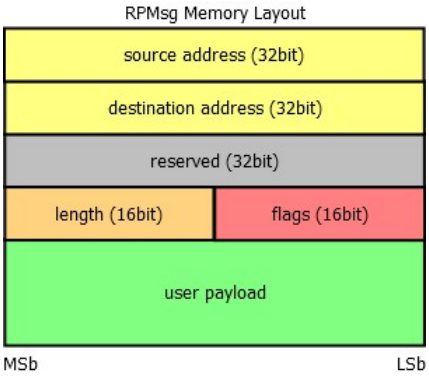


Figure 2.15: RPMsg header definition [33]

remote of this address the master only needs to send one message and the remote can then read the address from the source field of the message header. Now both sides are able to send messages to each other.

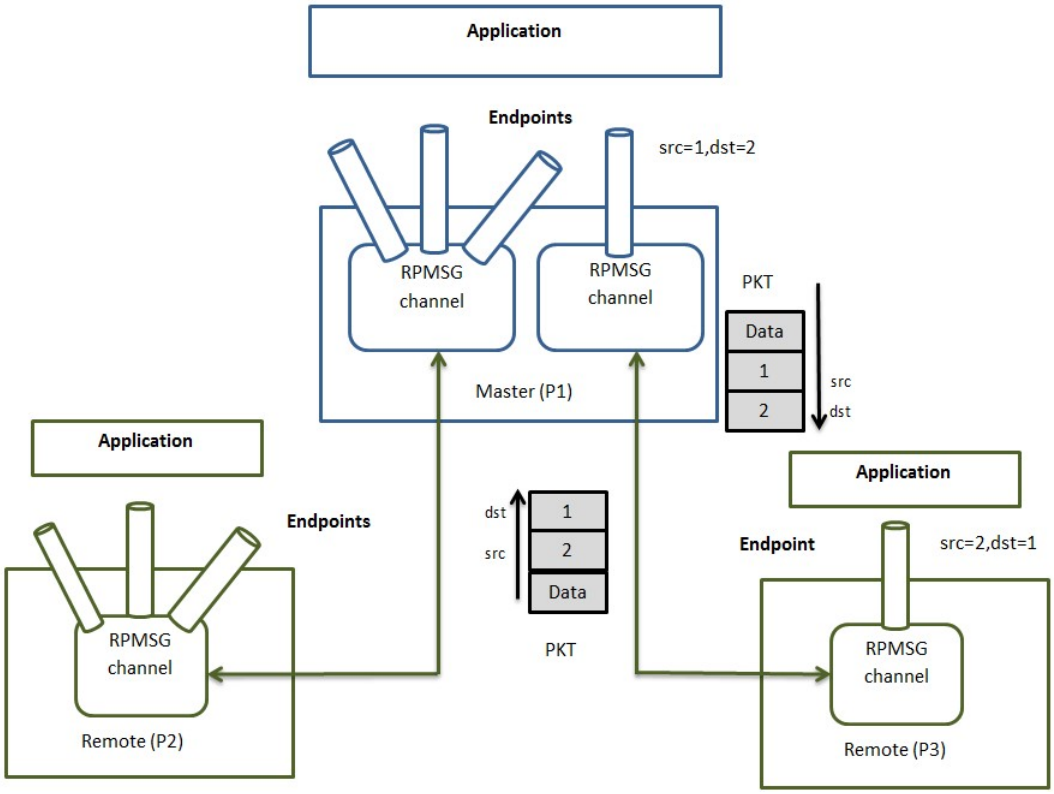


Figure 2.16: Diagram of two RPMsg channels, one channel having three endpoints on either side, the other only having one endpoint on either side [33]

3 Implementation

3.1 Demo Setup

A demo is created to showcase all research topics of this internship. The demo is developed for the NXP i.MX8M-Plus development platform and all research topics are incorporated in a single Yocto Project layer.

The demo is a temperature sensor connected over SPI to the platform. The SoC has some extensions to the SPI protocol and therefore calls it ECSPI, but it can also be used as regular SPI. The Cortex-M core runs FreeRTOS and reads the sensor values. These values are sent over RPMsg to Linux running on the Cortex-A cores in the NW. The Cortex-A cores also run OP-TEE in the SW. A Linux user-space application receives the RPMsg messages and is also able to send instructions back to FreeRTOS. An offset can be given to FreeRTOS which will be applied to the temperature data. The Linux app also signs the data with a digital signature using the PKCS#11 API. The API will make calls to a TA supplied by OP-TEE (PKCS#11 TA), which will handle all keys and do the signing operations. This TA can also verify the data that it signed. Lastly, the demo setup is booted using the NXP implementation of secure boot: HAB.

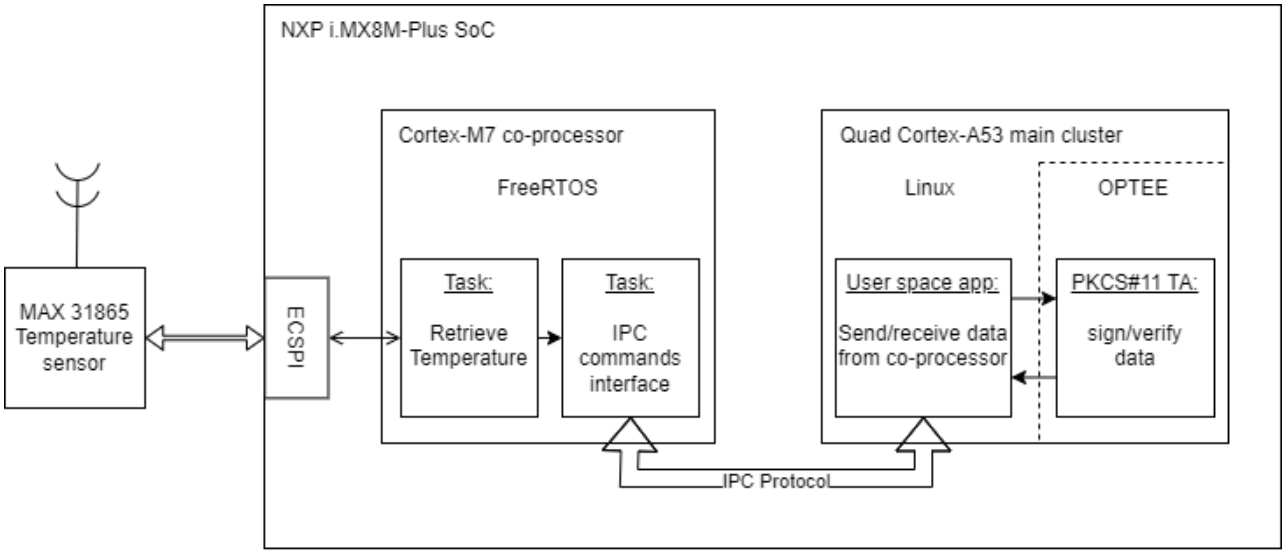


Figure 3.1: Diagram of data flow through the demo

3.2 Secure Boot

The HAB implementation is used for the demo. [35] provides an easy to follow tutorial to implement HAB on the NXP i.MX8M-Plus SoC. Since, HAB is well supported and a Yocto layer (`meta-variscite-hab`) is provided for its support, only a few additions need to made to the configuration to enable HAB.

Before HAB can be implemented in the image, a PKI tree needs to be generated. Appendix A.1 describes the steps that need to be taken.

Enabling HAB can be done in the `local.conf` file, or more permanently in the machine config of a custom layer.

```
# Enable Variscite HAB machine override
OVERRIDES =. "hab:"

# Provide path to NXP Code Signing archive
NXP_CST_URI="file://${TOPDIR}/../tools/cst-3.1.0.tgz"

# Set serial and password used during PKI generation
CST_SERIAL  ?= <serial>
CST_KEYPASS ?= <password>

# U-Boot device tree
# sign imx-boot images for both imx8mp-var-dart and imx8mp-var-som symphony (default
    configuration)
UBOOT_DTBS:mx8mp-nxp-bsp ?= "\
    imx8mp-var-dart-dt8mcustomboard.dtb \
    imx8mp-var-som-symphony.dtb \
"
# Use signed imx8mp-var-dart-customboard.dtb in the SD card image
 UBOOT_DTB_DEFAULT:mx8mp-nxp-bsp ?= "-imx8mp-var-dart-dt8mcustomboard"

# Kernel device tree
# Sign kernel device tree file:
SIGN_DTB:mx8mp-nxp-bsp ?= "${B}/${KERNEL_OUTPUT_DIR}/dts/freescale/imx8mp-var-dart-
    dt8mcustomboard-m7.dtb"

# GIT Repository for private certificates
# Note: CST_CERTS_REV is mandatory and used by var-hab.bbclass as well as CST_CERTS_URI
# If you don't know the sha256 checksum, then run without and the errors will contain the
    correct checksum
CST_CERTS_REV = <keys-git-commit-hash>
SRC_URI[cst-certs.sha256sum] = <sha256-checksum>
CST_CERTS_URI = <url-to-keys-git>
```

The `OVERRIDES` enables the HAB functionality and then some variables need to be set to use the PKI tree that has been generated. Yocto will then automatically sign the SPL, bootloader and kernel using the keys and build an image with them. The signing of images can also be done manually. Appendices A.2 and A.3 describe in detail how the signing process works.

3.3 ARM TrustZone

ARM TrustZone is used to establish a TEE using OP-TEE. Applications in the NW can make PKCS#11 API calls to the SW to generate key pairs and sign/verify data with those keys without having direct access to or any knowledge of the keys.

3.3.1 Setup and Configure OP-TEE

Support for OP-TEE on the i.MX8M-Plus is provided by the `meta-freescale` layer in Yocto. It is easily enabled in the `local.conf` file in the `build/conf` directory of your project.

```
MACHINE_FEATURES:append = " optee"
DISTRO_FEATURES:append = " optee"
IMAGE_INSTALL:append = " optee-os"
```

Or, to make the change more permanent, these features can be enabled in a custom layer. Add the features to their respective configuration file: `MACHINE_FEATURES` to a custom machine config, `DISTRO_FEATURES` to a custom distro config and `IMAGE_INSTALL` to a custom image recipe.

Additionally, OP-TEE provides a test-utility, which can be useful in development. This can be added to the image by appending `optee-test` to the `IMAGE_INSTALL` variable. When the device is booted, the OP-TEE configuration can be tested using the `xtest` command in Linux.

Lastly, OP-TEE contains a PKCS#11 library: `libckteec`. This library translates PKCS#11 API calls from the NW to SMC instructions for the SW. OP-TEE provides a PKCS#11 TA that performs the requested actions. This TA acts as a a software-based HSM. The `libckteec` library can be used in a C program using the `pkcs11.h` header files [36] or the `pkcs11-tool` command-line interface [37].

To add the `pkcs11-tool` to the image, the `opensc` and `libp11` packages need to be added to the image recipe. Also, `openssl` has support for PKCS#11 and can be used within OP-TEE as well.

```
IMAGE_INSTALL:append = " opensc libp11 openssl-bin"
```

3.3.2 Sign and Verify Data

The PKCS#11 TA is used to generate asymmetric keys and manage them. Using the `libckteec` library, data can be provided to be signed or verified with those keys.

Firstly, a token needs to be generated. A token needs a security-officer password (so-pin) and a user password (pin). These give certain privileges when logged in as either. Tokens are stored in secure storage together with their hashed pins protected by OP-TEE.They only needs to be setup once, as they only disappear when overwritten with a new token or a new image is loaded.

```
pkcs11-tool --module /usr/lib/libckteec.so.0 --slot <slot_index> --init-token --label
    mytoken --so-pin <super_pin> --init-pin --pin <user_pin>
```

Now that a token is available, a key pair can be generated and stored in this token. Keys, data and certificates are all objects that can be stored in the secure memory of a token. Object can have many attributes, such as whether the object is private or public, and whether the object lives only for the duration of a session or is permanently stored onto the token. Below is an example in C. First, the program starts a session as a users by logging in. Then two attributes are created, one for the public key and the other for the private key. These define the type of key, their capabilities, their name, etc. These are then used to generate the key pair as a temporary session object.

```
CK_FUNCTION_LIST *pkcsFunctions = NULL;
C_GetFunctionList(&pkcsFunctions);
CK_SESSION_HANDLE hSession = 0;

// Login to slot with slotPin
pkcsFunctions->C_Initialize(NULL);
pkcsFunctions->C_OpenSession((CK_SLOT_ID)slotId, CKF_SERIAL_SESSION | CKF_RW_SESSION,
    NULL, NULL, &hSession);
pkcsFunctions->C_Login(hSession, CKU_USER, (CK_BYTE_PTR)slotPin, strlen((const char*)
    slotPin));

CK_MECHANISM mech = {CKM_RSA_PKCS_KEY_PAIR_GEN};
CK_BBOOL yes = CK_TRUE;
CK_BBOOL no = CK_FALSE;
CK_ULONG keySize = 2048;
CK_BYTE publicExponent[] = {0x01, 0x00, 0x00, 0x00, 0x01};
CK_UTF8CHAR pubLabel[] = "rsa_public";
CK_UTF8CHAR priLabel[] = "rsa_private";
CK_OBJECT_HANDLE hPublic = 0;
CK_OBJECT_HANDLE hPrivate = 0;

// Create public key configuration
CK_ATTRIBUTE attribPub[] =
```

```
{
    {CKA_TOKEN,             &no,                sizeof(CK_BBOOL)},
    {CKA_PRIVATE,           &no,                sizeof(CK_BBOOL)},
    {CKA_VERIFY,            &yes,               sizeof(CK_BBOOL)},
    {CKA_ENCRYPT,           &yes,               sizeof(CK_BBOOL)},
    {CKA_MODULUS_BITS,      &keySize,           sizeof(CK_ULONG)},
    {CKA_PUBLIC_EXPONENT,   &publicExponent,    sizeof(publicExponent)},
    {CKA_LABEL,             &pubLabel,          sizeof(pubLabel)}
};
CK_ULONG attribLenPub = sizeof(attribPub) / sizeof(*attribPub);

// Create private key configuration
CK_ATTRIBUTE attribPri[] =
{
    {CKA_TOKEN,             &no,                sizeof(CK_BBOOL)},
    {CKA_PRIVATE,           &yes,               sizeof(CK_BBOOL)},
    {CKA_SIGN,              &yes,               sizeof(CK_BBOOL)},
    {CKA_DECRYPT,           &yes,               sizeof(CK_BBOOL)},
    {CKA_SENSITIVE,         &yes,               sizeof(CK_BBOOL)},
    {CKA_LABEL,             &priLabel,          sizeof(priLabel)}
};
CK_ULONG attribLenPri = sizeof(attribPri) / sizeof(*attribPri);

// Generate RSA key pair
pkcsFunctions->C_GenerateKeyPair(hSession, &mech, attribPub, attribLenPub, attribPri,
    attribLenPri, &hPublic, &hPrivate);
```

With the keys generated, they can be used using their object handles in the `Sign` and `Verify` functions.

```
// sign data
CK_MECHANISM sign_verify_mech = {CKM_RSA_PKCS};
pkcsFunctions->C_SignInit(hSession, &sign_verify_mech, hPrivate);
pkcsFunctions->C_Sign(hSession, data, dataSize-1, signature, &sigLen);

// verify data
pkcsFunctions->C_VerifyInit(hSession, &sign_verify_mech, hPublic);
pkcsFunctions->C_Verify(hSession, data, dataSize-1, signature, sigLen);
```

3.4 Inter-Processor Communication

Of the three IPC methods discussed in section 2.4, only the RPMsg method has been implemented in the end. The Shared Memory and Mailboxes have been tried, but unable to get working. The suspicion lies in that the Resource Domain Controller (RDC) of the i.MX8M-Plus SoC[29] was blocking access to these resources. The documentation of the RDC is not very clear and made it hard to configure.

An RPMsg connection is established between Linux running on the Cortex-A53 cores and FreeRTOS running on the Cortex-M7 core. NXP have developed a light-weight implementation of the RPMsg protocol called RPMsg-Lite [38]. This implementation is used on the FreeRTOS side. The Linux side is the master and uses the RPMsg drivers available in the Linux kernel.

The FreeRTOS side can start up first, it will first initialize an RPMsg-Lite instance and wait until the link state of the instance is true. Appendix C has some details on fixing problems when the FreeRTOS app is stuck on waiting for the link. When the link is up, RPMsg-Lite will need to initialize a VirtQueue and create an endpoint. A nameservice will be setup and anounce the RPMsg service and channel. Then the FreeRTOS side needs to wait for a response from the master in order to know what their address is.

```
#define LOCAL_EPT_ADDR (0x1E)
```

```
#define RPMSG_LITE_NS_ANNOUNCE_STRING "rpmsg-openamp-demo-channel"

rpmsg_lite_instance* my_rpmsg = rpmsg_lite_remote_init((void *)RPMSG_LITE_SHMEM_BASE,
    RPMSG_LITE_LINK_ID, RL_NO_FLAGS);
if (my_rpmsg == NULL)
{
    PRINTF("Invalid rpmsg instance\r\n");
}

rpmsg_lite_wait_for_link_up(my_rpmsg, RL_BLOCK)

rpmsg_queue_handle my_queue = rpmsg_queue_create(my_rpmsg);
rpmsg_lite_endpoint* my_ept = rpmsg_lite_create_ept(my_rpmsg, LOCAL_EPT_ADDR,
    rpmsg_queue_rx_cb, my_queue);
rpmsg_ns_handle ns_handle = rpmsg_ns_bind(my_rpmsg, app_nameservice_isr_cb, ((void *)0));

/* A small delay to allow the nameservice_isr_cb to get registered before the announce
    message is sent */
SDK_DelayAtLeastUs(1000000U, SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY);
(void)rpmsg_ns_announce(my_rpmsg, my_ept, RPMSG_LITE_NS_ANNOUNCE_STRING, (uint32_t)
    RL_NS_CREATE);
(void)PRINTF("Nameservice announce sent.\r\n");

/* Wait for Hello handshake message from master */
(void)rpmsg_queue_recv(my_rpmsg, my_queue, (uint32_t *)&remote_addr, helloMsg, sizeof(
    helloMsg), ((void *)0), RL_BLOCK);
```

RPMsg-lite has specific functions for sending and receiving. The code below gives two examples of how to use them. The MCUexpresso documentation [39] for RPMsg-Lite contains more examples of functions that are available.

```
//RPMsg-Lite: send to rpmsg
if (rpmsg_lite_is_link_up(my_rpmsg))
{
    char buffer[64] = "message to send";
    (void)rpmsg_lite_send(my_rpmsg, my_ept, remote_addr, buffer, 64, RL_BLOCK);
}

//RPMsg-Lite: read from rpmsg
if (rpmsg_queue_get_current_size(my_queue))
{
    char pPacket[64];
    uint32_t packetLength = 0;
    rpmsg_queue_recv(my_rpmsg, my_queue, (uint32_t*)remote_addr, pPacket, 64, &
    packetLength, RL_DONT_BLOCK);
}
```

The Linux side then has time to initialize as well in its own time. It will first need to open the "rpmsg_ctrl0" device and create an endpoint using the `ioctl` command. To create the endpoint a struct needs to be supplied with the channel name and the source and destination addresses. Extra flags can be added to the endpoint, for example, a flag that specifies that the read operation is non-blocking. When the endpoint is setup, then a handshake message can be send to the remote, so that it is aware of which address the master is listening on.

```
#define RPMSG_CREATE_EPT_IOCTL _IOW(0xb5, 0x1, struct rpmsg_endpoint_info)
#define RPMSG_DESTROY_EPT_IOCTL _IO(0xb5, 0x2)
#define RPMSG_CONTROL_DEVICE "/dev/rpmsg_ctrl0"
#define RPMSG_TTY_DEVICE "/dev/rpmsg0"
```

```
struct rpmsg_endpoint_info
{
    char name[32];
    uint32_t src;
    uint32_t dst;
};

struct rpmsg_endpoint_info ept_info = {"rpmsg-openamp-demo-channel", 0x2, 0x1E};
int fd = open(RPMSG_CONTROL_DEVICE, O_RDWR);
if (fd < 0)
{
    printf("%s not available\r\n", RPMSG_CONTROL_DEVICE);
    return 1;
}
/* create endpoint interface */
ioctl(fd, RPMSG_CREATE_EPT_IOCTL, &ept_info);
int fd_ept = open(RPMSG_TTY_DEVICE, O_RDWR);
int flags = fcntl(fd_ept, F_GETFL, 0);
fcntl(fd_ept, F_SETFL, flags | O_NONBLOCK);

char handshake[13] = "hello world!";
write(fd_ept, &handshake, sizeof(handshake));
```

Sending and receiving messages from the Linux side is done using character devices. As seen in the code snippet above, the the RPMsg device/endpoint needs to be opened like a file. The file descriptor can then be used with `read()` and `write()` functions from the `unistd.h` library.

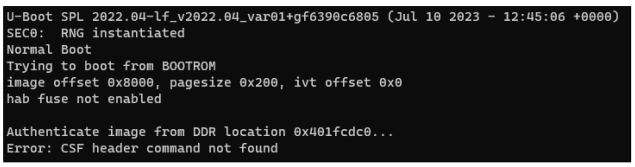
```
char data_buf[64];

//RPMsg Linux: write to rpmsg
write(fd_ept, &data_bus, sizeof(data_buf));

//RPMsg Linux: read from rpmsg
read(fd_ept, &data_buf, sizeof(data_buf));
```

4 Discussion

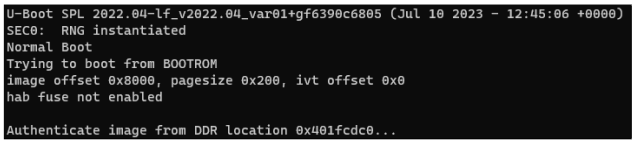
This chapter will discuss the results of the demo described in section 3.1. It is also subdivided into the three main topics.

4.1 Secure Boot

The HAB secure boot implementation for the demo works as expected. The signed images boot normally and images that are not signed or signed with the wrong key are detected by HAB. Figure 4.1a shows the output of the bootloader when no signature is provided. It clearly states that no CSF has been found. Figure 4.1b shows the output of the bootloader when the signature that is provided is not valid. In both situations the boot process hangs after it displays the message. When the signature is correct, the bootloader will continue and boot into U-Boot.



(a) Bootloader without signature



(b) Bootloader signed with incorrect key

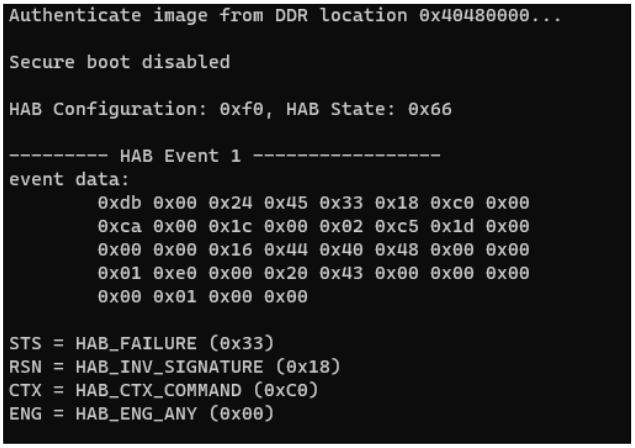
Figure 4.1: Bootloader output for bad HAB signatures

U-Boot will verify and load the kernel. If anything is wrong with the kernel image, HAB will display events with info about the error. These events can be decoded using the HAB API reference [40]. Figure 4.2 shows the HAB events for a kernel image without a signature and a kernel image with an invalid signature. The kernel still gets loaded and given control in the demo, because the device is not closed. That is also why the output says "Secure boot disabled". If it were to be closed using the instructions in appendix A.4.2, then no HAB events will be shown and the boot process will stop upon finding errors. If all signatures are valid, then the device will boot into the Linux kernel.

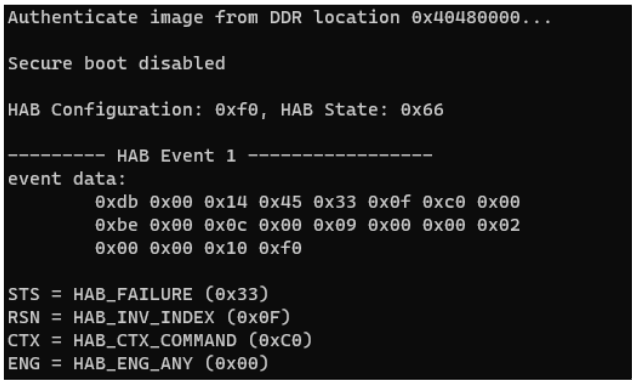
A big short-coming of the HAB implementation of the demo is that the Cortex-M7 application is not verified. HAB should be able to do it as suggested by [41]. However, it was unclear how to pad the image exactly. Therefore HAB was unable to find the CSF and thus not able to verify the image.

Furthermore, the root file system is also not verified by secure boot. This means that anyone could take out the sd card, mount the file system in another computer and edit the contents of it without HAB noticing. That means that user space commands (such as `ls` and `cd`) can not be trusted and files that are saved in the root file systems could be tampered with or the information leaked.

When these short-comings are a concern, then Encrypted boot [42] can be considered. This will use HAB to encrypt the bootloader code, but can also be extended to the entire image. If the root file system is encrypted and only a specific device has the key, then the data can not be tampered with and the Cortex-M7 app and user space files will be secure.



(a) Kernel without signature



(b) Kernel signed with incorrect key

Figure 4.2: HAB events shown for Kernel with bad HAB signatures

4.2 ARM TrustZone

Due to the secure boot implementation, the ATF boots into a verified state. Therefore the SW and OP-TEE can be trusted. They form a secure place to store secrets and sensitive data. The PKCS #11 TA is a recent addition to OP-TEE, but may be a strong application to store keys. Linux is able to use the keys, but not able to access them directly. The API seems to be as complete as needed, but some more research should be done to understand how to use it in a more permanent setting. The demo only kept keys as long as the program was running, because it only creates session objects.

The trust boundary of the demo is at the edge of the SW. Both the NW and Cortex-M are untrusted. When these domains don't have any secrets or sensitive data in them, then it is not a big problem for them to be untrusted. They still need to be protected by basic security measures, like strong passwords, minimal user permissions, limiting physical access and using best practices in IT infrastructure. These measures are like the initial walls before getting into an old city and OP-TEE being the citadel where the most valuable assets are kept.

4.3 Inter-Processor Communication

The demo shows that data can be send between the Cortex-A main processor and the Cortex-M co-processor using RPMsg. The framework is usable, but also complex and its documentation is quite minimal and scattered. The RPMsg-Lite implementation for the Cortex-M processor works well and is nicely documented, but the documentation for the Linux side is a bit lacking. This makes the framework not the easiest to work with.

RPMsg makes use of the shared memory and mailbox options, showing that these work well, but using those methods directly is a little harder. There has been an attempt to implement these, but it is suspected that the NXP Resource Domain Controller (RDC) is not configured to allow direct access to these peripherals. The documentation of the RDC is also very limited and confusing, making it hard to configure it. If the RDC is able to get configured, then the shared memory and mailbox methods might make the communication easier than using the RPMsg framework.

5 Future Improvements

The contents of this report lay a foundation for implementing security and communication in a heterogeneous embedded computing environment. These concepts can be extended to further improve upon them. A few directions of further research are presented here.

First of all, secure boot and TrustZone are assumed trusted for attackers that do not exploit hardware level attacks, as was specified in the threat model in section 1.3. However, there are resources that have successfully exploited some hardware bugs to break secure boot implementations and parts of TrustZone. [43] uses Side-Channel Analysis to find a timing in a smartphone SoC that is susceptible to a fault injection. The SoC leaks out some important information by Electromagnetic radiation, which was used to learn about the SoC using the Side-Channel Analysis technique without having any access to the code. Although the paper was not successful in all of their goals, they do show that important information does leak out and can be used to find vulnerabilities.

Another method, [44], shows that a power LED might also leak out sensitive cryptographic information, due to the power being used to perform the cryptographic operations. When more power is used, the led shines a little dimmer. This is not visible by human eyes, but even smartphone cameras can be used to catch these fluctuations in LED brightness. Also, while using a power LED is a less intrusive attack, also measuring the power usage directly from the power plug is another attack vector. Therefore having cryptographic algorithms that don't expose secrets in their power signature are needed, like developed in [45].

It is also important to understand that with the development of quantum computers, some cryptographic functions might become unsafe. For now, the hash functions and key pairs used in the demo are still considered secure. [46], [47] have looked into secure boot implementations that make use of cryptographic algorithms that are more resilient against quantum computers. These developments show that security is be a continuous topic that will need to be constantly updated against new forms of attacks.

Then there is the issue of updating the software on a device. A software stack may have a feature update or an important security update. These should be able to get written to the device, without compromising the Secure Boot implementation. [48] is an NXP document showing how a secure Over-The-Air update mechanism can be setup using Meder or SWUpdate. Furthermore, [49] shows how the JTAG port can also be secured. JTAG can be used to test and debug a chip and has access to a scan chain around the chip. This scan chain lets the developer set any input to any pin, therefore having full low-level control over the chip. When in the field, the device should not be accessed via this JTAG. So an easy solution is to completely disable JTAG, when shipping the device to a customer. However, when in-field maintenance is desired, the NXP chips also provide a secure JTAG solution which should only give access to an authenticated developer. These two documents show techniques to keep the device open for developers and while having it closed for anyone else.

Lastly, the implementation of Resource Domain Controller (RDC) held back the research of shared memory and mailbox for this particular SoC. Investigating the architecture and usage of the RDC will be useful to have fine control over the access rights of the peripherals of the SoC. This could give a developer the option to use a simpler shared memory setup, then using the extensive RPMsg framework. Also just understanding the RDC configuration will also give more insight in potential security vulnerabilities, because maybe some access rights are not configured properly. In addition to that, also the TrustZone Protection Controller (TZPC) should be investigated to give the same fine-grain peripheral access control to the TEE. Having the RDC and TZPC work together properly will also give a nice barrier for the simplest attacks and will push an attacker to the more difficult to perform hardware attacks.

Bibliography

[1] Accessed November 30th 2023. [Online]. Available: https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act.

[2] Accessed November 30th 2023. [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors/i-mx-8-applications-processors/i-mx-8m-plus-arm-cortex-a53-machine-learning-vision-multimedia-and-industrial-iot:IMX8MPLUS.

[3] L. Kohnfelder and P. Garg, "The threats to our products," Microsoft Inc., Tech. Rep., 1999. [Online]. Available: https://shostack.org/files/microsoft/The-Threats-To-Our-Products.docx.

[4] A. Shostack, *Threat Modelling: Designing for security.* John Wiley & Sons, Inc., 2014, ISBN: 978-1-118-80999-0.

[5] N. I. of Standards and T. (NIST), *Fips pub 180-4 secure hash standard (shs)*, Originally published as FIPS 180 in 1993. Updated over the years to improve security, 2015. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.180-4.

[6] B. Hayes/NIST, *Nist retires sha-1 cryptographic algorithm*, 2022. [Online]. Available: https://www.nist.gov/news-events/news/2022/12/nist-retires-sha-1-cryptographic-algorithm.

[7] By KohanX (talk) - Own work, Public Domain. [Online]. Available: https://commons.wikimedia.org/w/index.php?curid=24325809.

[8] By Davidgothberg - Own work, Public Domain. [Online]. Available: https://commons.wikimedia.org/w/index.php?curid=1028460.

[9] By FlippyFlink - Adapted work from Davidgothberg from encryption to signing, CC BY-SA 4.0. [Online]. Available: https://commons.wikimedia.org/w/index.php?curid=78867393.

[10] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, "Rfc8017: Pkcs #1 rsa cryptography specifications version 2.2," Internet Engineering Task Force, Tech. Rep., 2016. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8017.

[11] B. Kaliski, "Rfc 5208: Pkcs #8 private-key information syntax specification version 1.2," Internet Engineering Task Force, Tech. Rep., 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5208.

[12] K. Moriarty, B. Kaliski, and A. Rusch, "Rfc8018: Pkcs #5 password-based cryptography specification version 2.1," Internet Engineering Task Force, Tech. Rep., 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8018.

[13] R. Griffin and V. Fenwick, "Pkcs#11 cryptographic token interface base specification version 2.40," OASIS, Tech. Rep., 2015. [Online]. Available: https://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.pdf.

[14] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves.," in *NDSS*, 2019.

[15] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM computing surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[16] Accessed December 7th 2023. [Online]. Available: https://www.timesys.com/security/pkcs11-with-op-tee-securing-iot-device-keys/.

[17] D. Lu, M. Shi, X. Ma, *et al.*, "Smaug: A tee-assisted secured sqlite for embedded systems," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[18] Accessed December 7th 2023. [Online]. Available: https://www.trustedfirmware.org/projects/op-tee/.

[19] Accessed December 7th 2023. [Online]. Available: https://www.timesys.com/security/trusted-software-development-op-tee/.

[20] R. Wayman, *A gentle introduction to trusted execution and op-tee*, Slides of presentation on Linaro Connect Bangkok 2016 (BKK16-110), 2016. [Online]. Available: https://www.slideshare.net/linaroorg/bkk16110-a-gentle-introduction-to-trusted-execution-and-optee.

[21] *An4581 - secure boot on habv4 supported devices*, NXP account required. [Online]. Available: https://www.nxp.com/webapp/Download?colCode=AN4581&location=null.

[22] *Code-signing tool - user's guide*, Accessed December 4th 2023. [Online]. Available: https://sources.debian.org/data/main/i/imx-code-signing-tool/3.3.0%2Bdfsg2-1/docs/CST_UG.pdf.

[23] NXP account required. [Online]. Available: https://www.nxp.com/webapp/Download?colCode=IMX_CST_TOOL_NEW&location=null.

[24] Accessed December 6th 2023. [Online]. Available: https://docs.nxp.com/bundle/GUID-487B2E69-BB19-42CB-AC38-7EF18C0FE3AE/page/GUID-9280C0B6-CE8D-411D-905C-F97483AAE047.html.

[25] S. Muir and J. Smith, "Asymos-an asymmetric multiprocessor operating system," in *1998 IEEE Open Architectures and Network Programming*, IEEE, 1998, pp. 25–34.

[26] R. Kumar, D. M. Tullsen, and N. P. Jouppi, "Core architecture optimization for heterogeneous chip multiprocessors," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, 2006, pp. 23–32.

[27] Y. Poornima, S. R. Kalathuru, and P. G. Poddar, "Mailbox based inter-processor communication in soc," in *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, IEEE, 2017, pp. 1033–1037.

[28] Accessed December 6th 2023. [Online]. Available: https://www.kernel.org/doc/html/latest/driver-api/mailbox.html.

[29] NXP, *I.mx 8m plus applications processor reference manual*. [Online]. Available: https://www.nxp.com/webapp/Download?colCode=IMX8MPRM.

[30] O. Ben-Cohen, "Introducing a generic amp/ipc framework," *kernel.org*, 2011. [Online]. Available: https://lore.kernel.org/all/1308640714-17961-1-git-send-email-ohad@wizery.com/.

[31] Accessed December 6th 2023. [Online]. Available: https://www.kernel.org/doc/html/latest/staging/rpmsg.html.

[32] L. Noa, *Asymmetric multiprocessing: Rpmsg device and driver on linux and android*, Accessed December 5th 2023. [Online]. Available: https://technotes.kynetics.com/2018/Linux_rpmsg_char_driver/.

[33] OpenAMP, *Rpmsg messaging protocol*, Accessed December 5th 2023. [Online]. Available: https://openamp.readthedocs.io/en/latest/protocol_details/rpmsg.html.

[34] J. Palmer, *Oracle linux blog: Intro to virtio*, Accessed December 5th 2023, 2022. [Online]. Available: https://blogs.oracle.com/linux/post/introduction-to-virtio.

[35] Variscite, *I.mx8m high assurance boot (hab) / secure boot*. [Online]. Available: https://variwiki.com/index.php?title=High_Assurance_Boot_MX8&release=mx8mp-yocto-kirkstone-5.15.71_2.2.0-v1.1.

[36] Accessed December 7th 2023. [Online]. Available: https://github.com/OP-TEE/optee_client/blob/master/libckteec/include/pkcs11.h.

[37] Accessed December 7th 2023. [Online]. Available: https://optee.readthedocs.io/en/latest/building/userland_integration.html.

[38] Accessed December 5th 2023. [Online]. Available: https://github.com/nxp-mcuxpresso/rpmsg-lite.

[39] Accessed December 5th 2023. [Online]. Available: https://nxp-mcuxpresso.github.io/rpmsg-lite/index.html.

[40] NXP, "High assurance boot version 4 application programming interface reference manual," NXP, Tech. Rep., 2018. [Online]. Available: https://sources.debian.org/data/main/i/imx-code-signing-tool/3.3.0%2Bdfsg2-1/docs/HAB4_API.pdf.

[41] Accessed December 11th 2023. [Online]. Available: https://github.com/nxp-imx/uboot-imx/blob/lf_v2022.04/doc/imx/habv4/guides/mx8m_secure_boot.txt.

[42] NXP, "An12056 rev0: Encrypted boot on habv4 and caam enabled devices," NXP, Tech. Rep., 2018.

[43] C. Fanjas, D. Aboulkassimi, S. Pontié, and J. Clédière, "Exploration of system-on-chip secure-boot vulnerability to fault-injection by side-channel analysis," in *2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2023, pp. 1–6. DOI: 10.1109/DFT59622.2023.10313346.

[44] B. Nassi, E. Iluz, O. Cohen, *et al.*, "Video-based cryptanalysis: Extracting cryptographic keys from video footage of a device's power led," *Cryptology ePrint Archive*, 2023.

[45] K. A. Bayam and B. ÖRS, "Differential power analysis resistant hardware implementation of the rsa cryptosystem," *Turkish Journal of Electrical Engineering and Computer Sciences*, vol. 18, no. 1, pp. 129–140, 2010.

[46] V. B. Kumar, N. Gupta, A. Chattopadhyay, M. Kasper, C. Krauß, and R. Niederhagen, "Post-quantum secure boot," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2020, pp. 1582–1585.

[47] A. Wagner, F. Oberhansl, and M. Schink, "To be, or not to be stateful: Post-quantum secure boot using hash-based signatures," in *Proceedings of the 2022 Workshop on Attacks and Solutions in Hardware Security*, 2022, pp. 85–94.

[48] *An12900 - secure over-the-air prototype for linux using caam and mender or swupdate*, NXP account required. [Online]. Available: https://www.nxp.com/webapp/Download?colCode=AN12900&location=null.

[49] *An4686 - secure debug in i.mx 6/7/8m family of applications processors*, NXP account required. [Online]. Available: https://www.nxp.com/webapp/Download?colCode=AN4686&location=null.

A HAB Secure Boot implementation details

A.1 Generating a PKI tree

First, a PKI needs to be generated. The CST tool [23] contains a script that can do this. It needs two files: one file called `serial` containing eight hexadecimal characters and another file called `key_pass.txt` containing a password repeated on two lines. These two files need to be stored in the `keys` directory inside the CST directory.

```
tar xf cst-3.1.0.tgz
cd cst-3.1.0/keys
echo 13579BDF > serial
echo strong_password > key_pass.txt
echo strong_password >> key_pass.txt
```

When the files are created, the PKI tree can be generated using the `hab4_pki_tree.sh` script in the `keys` directory. The script will ask some questions to generate the tree as liked. While most questions are pretty self-explanatory, the last question regarding the `CA flag` is a bit vague. When answering `y` to this question, the PKI tree will have CSF and IMG keys for each SRK. Answering `n` will generate a PKI tree with only the CA and SRK. All keys are generated in PKCS#8 format and each key is protected by the same password.

```
./hab4_pki_tree.sh
Do you want to use an existing CA key (y/n)?: n
Do you want to use Elliptic Curve Cryptography (y/n)?: n
Enter key length in bits for PKI tree: 4096
Enter PKI tree duration (years): 20
How many Super Root Keys should be generated? 4
Do you want the SRK certificates to have the CA flag set? (y/n)?: y
```

The PKI tree is now generated. The SRK need to be combined into a table and a hash needs to be created of this table that can be stored in the e-fuses. The CST tool also provides a program for this, called `srktool`. The generated table and hash should be put in the certs/ directory of the CST. The tool takes some flags. `-h` sets the hab version and `-f` sets the hash format. `-t` specifies the filename of the SRK Table, and `-e` specifies the filename of the SRK Table Hash, with `-d` defining the hash algorithm used. `-c` takes a list separated by commas and no spaces of the SRK certificates that need to be in the table.

```
cd cst-3.1.0/certs/
../linux64/bin/srktool -h 4 -f 1 \
                       -t SRK_1_2_3_4_table.bin \
                       -e SRK_1_2_3_4_fuse.bin -d sha256 \
                       -c ./SRK1_sha256_4096_65537_v3_ca_crt.pem,\
                       ./SRK2_sha256_4096_65537_v3_ca_crt.pem,\
                       ./SRK3_sha256_4096_65537_v3_ca_crt.pem,\
                       ./SRK4_sha256_4096_65537_v3_ca_crt.pem
```

The values that need to be set to the fuses can be read from the hash file using the following command. The CST tool also generates a text file (`SRK_1_2_3_4_fuse.bin.u-boot-cmds`) with more clear instructions on how to set the e-fuses.

```
hexdump -e '/4 "0x%X\n"' < SRK_1_2_3_4_fuse.bin
```

! However, hold off with writing anything to the fuses, because they are **only able to be written once!** Wait until you are sure you have the correct commands and do it very carefully. One wrong character could brick your device. See appendix A.4 for more details on programming the e-fuses.

Lastly, it is useful to store the keys and certificates in a git repository. Then keys won't be lost. If the keys are ever lost, then a system would not be able to run any new images anymore, because the correct SRK table can't be embedded in the image and the original SRK table hash can't be changed in the e-fuses. It is also safe to store the certificates and private keys in a git repository, because the private keys are encrypted with the password provided during PKI generation. It is important to **NOT include the password file in the git repository**, otherwise the confidentiality of the private keys will be compromised. **Also don't add any CST scripts, intermediate files and the serial files in the repo.**

```
mkdir hab-pki-tree
cd hab-pki-tree
mkdir iMX8M    # or the name of any other platform, but a platform dir is necessary. Even
    if only one PKI tree is stored
cd iMX8M
mkdir crts keys
cp ../../cst-3.1.0/keys/*v3_{usr,ca}_key* keys
cp ../../cst-3.1.0/crts/*v3_{usr,ca}_key* crts
cp ../../cst-3.1.0/crts/SRK_1_2_3_4_{fuse,table}.bin crts
cd ../..

# store this directory to a git repository in any way you like
git init
git add .
git commit
git remote add origin <remote-repo-url>
git push origin main
```

A.2 Bootloader signing

A.2.1 Bootloader layout

Due to a new the architecture, multiple firmwares and softwares are required to boot i.MX8M family devices. In order to store all the images in a single binary the FIT (Flattened Image Tree) image structure is used for the main bootloader. So, the bootloader layer consists of 3 main parts, the HDMI or Displayport firmware, a Secondary Program Loader (SPL), and a FIT image containing U-Boot, the device tree for U-Boot, the ARM Trusted Firmware (ATF) binary and optionally an OP-TEE image for the secure world kernel. The SPL and FIT both need a Command Sequence File (CSF) which contain information about the certificate for each boot stage.

The first part of the bootloader layout is the HDMI/DP firmware. This already signed and distributed by NXP. It will always be authenticated regardless of security configuration. If the device does not need HDMI or DP support, then this can be disabled by e-Fuses and the firmwares are not required anymore. The next stages (SPL and FIT images) do need to be signed by the user.

A.2.2 Enable HAB support in U-Boot

The first step is to generate an U-Boot image supporting the HAB features. This enables the u-boot command `hab_status`, which allows the user to read the HAB logs and can support in debugging the HAB setup.

The support is enabled by adding `CONFIG_IMX_HAB=y` to the build configuration and then recompile the image.

A.2.3 Create the Bootloader image

The imx-mkimage project is used to combine all images in a single binary. The following files are required:

- U-Boot:
  - `u-boot.bin`
  - `u-boot-nodtb.bin`
  - `u-boot-spl.bin`
  - U-Boot DTB file (e.g. `imx8mp-var-dart-dt8mcustomboard.dtb`)
- ATF image:
  - `bl31.bin`
- DDR firmware:
  - `lpddr4_pmu_train_1d_dmem.bin`
  - `lpddr4_pmu_train_1d_imem.bin`
  - `lpddr4_pmu_train_2d_dmem.bin`
  - `lpddr4_pmu_train_2d_imem.bin`
- HDMI or DP firmware (Only in i.MX8M):
  - `signed_hdmi_imx8m.bin`
  - `signed_dp_imx8m.bin`
- OP-TEE (Optional):
  - `tee.bin`

The ATF image can be build using the imx-atf git repository and the DDR and display firmware can be downloaded from the NXP website. Or, instead of building and downloading them manually, they can also be taken from a Yocto build directory which already ran once. The Yocto build leaves all necessary images in `<build_dir>/tmp/work/imx8mp_var_dart-fslc-linux/imx-boot/1.0-r0` and the `git/iMX8M` directory in it. Copy all files to the 'uboot-imx/iMX8M' directory.

To create the bootloader binary, run `make` in `uboot-imx/iMX8M` with the `soc.mak` makefile. It will create a binary called `flash.bin` which contains all elements, except for the authentication data.

```
make -f soc.mak SOC=iMX8MP flash_hdmi_spl_uboot
```

This will generate some build logs. These logs are important for the signing steps later.

A.2.4 Sign the Booloader image

To sign the bootloader image, it takes two steps. First specify the signing process in a CSF file and then second compile the CSF file to a binary and add it to the rest of the bootloader image.

Create the CSF

First we need to create the Command Sequence File (CSF) for both the SPL and the FIT images. We need two CSFs, because the SPL will first be authenticated and then the SPL will extend the root of trust to the FIT image.

The CSF contains all the commands that the ROM executes during the secure boot. These commands instruct the HAB code on which memory areas of the image to authenticate, which keys to use for signing, etc.

Create SPL CSF (`imx8m-var-dart-spl.csf`) and copy it to also create the FIT CSF (`imx8m-var-dart-fit.csf`):

```
[Header]
    Version = 4.3
    Hash Algorithm = sha256
    Engine = CAAM
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS

[Install SRK]
    # Index of the key location in the SRK table to be installed
    File = "<relative_path>/crts/SRK_1_2_3_4_table.bin"
    Source index = 0

[Install CSFK]
    # Key used to authenticate the CSF data
    File = "<relative_path>/crts/CSF1_1_sha256_2048_65537_v3_usr_crt.pem"

[Authenticate CSF]

[Unlock]
    # Leave Job Ring and DECO master ID registers Unlocked to avoid kernel crash in
    closed devices
    Engine = CAAM
    Features = MID

[Install Key]
    # Key slot index used to authenticate the key to be installed
    Verification index = 0
    # Target key slot in HAB key store where key will be installed
    Target index = 2
    # Key to install
    File = "<relative_path>/crts/IMG1_1_sha256_2048_65537_v3_usr_crt.pem"
```

Then look to the build logs from creating the bootloader image. For the SPL look for `spl hab block` and copy the addresses. Add these addresses to a new `Authenticate Data` section in the SPL CSF, like so:

```
[Authenticate Data]
    # Key slot index used to authenticate the image data
    Verification index = 2
    # Authenticate Start Address, Offset, Length and file
    Blocks = 0x7e0fc0 0x1a000 0x2a600 "flash.bin"
```

Do the same for the FIT image, but take the addresses from `sld hab block` instead. Also, since the FIT image is build up from multiple binaries, each of these also need to be authenticated. To get the address from these, generate a script using the `soc.mak`. The output gives already some address, but in my case the device tree addresses were missing. To get all the addresses, run the script command which is displayed

in the output of the `make` command and add the U-Boot device tree name after it to print all necessary addresses.

```
make -f soc.mak SOC=iMX8MP print_fit_hab
TEE_LOAD_ADDR=0x56000000 ATF_LOAD_ADDR=0x00970000 VERSION=v2 ./print_fit_hab.sh 0x60000
    imx8mp-var-dart-dt8mcustomboard.dtb
```

Add these addresses to the FIT CSF under a new Authenticate Data section. The first set of addresses is for the FDT and IVT of the FIT to authenticate the FIT block, then there are individual address sets for the u-boot binary, ATF binary and U-boot device tree. Also if an OP-TEE binary is supplied, there will be an extra set of address for this.

```
[Authenticate Data]
    # Key slot index used to authenticate the image data
    Verification index = 2
    # Authenticate Start Address, Offset, Length and file
    Blocks = 0x401fcdc0 0x58000 0x1020 "flash.bin", \
             0x40200000 0x5B000 0xE1B50 "flash.bin", \
             0x402E1B50 0x13CB50 0xAEF0 "flash.bin", \
             0x970000 0x147A40 0xA150 "flash.bin"
```

A.2.5 Add CSF to the binary

To add the CSF to the binary, the CSF first needs to be in a binary format. The CST can transform the CSF to a binary. Do this for both the SPL and FIT CSF.

```
./cst -i imx8m-var-dart-spl.csf -o imx8m-var-dart-spl.bin
./cst -i imx8m-var-dart-fit.csf -o imx8m-var-dart-fit.bin
```

In the build logs from creating the `flash.bin` look for the `csf_off` address in the 'Loader IMAGE' section and the `sld_csf_off` address in the `Second Loader IMAGE`. These are the SPL CSF offset and the FIT CSF offset respectively. The CSF binaries need to be place in these locations.

```
cp flash.bin signed_flash.bin
dd if=imx8m-var-dart-spl.bin of=signed_flash.bin seek=$((0x2e600)) bs=1 conv=notrunc
dd if=imx8m-var-dart-fit.bin of=signed_flash.bin seek=$((0x59020)) bs=1 conv=notrunc
sync
```

Now the bootloader image is signed and ready to be flashed to an SD card, see appendix B

A.2.6 Test authentication by forcing wrong certificates

If the bootloader loads fine and the `hab_status` command in the u-boot command line does not show any hab events, then it is interesting to see whether certificate assertion events can be forced by intentionally creating a fault in the signing process.

The easiest way is to use a different key for either the CSF or IMG. For example, just use `IMG4_...` as the key for the SPL or FIT, while still specifying key index 0. Reassemble the bootloader and boot the device using this bootloader. The HAB authentication will fail and the boot process stops.

A.3 Kernel signing

Assuming you have a kernel Image that needs to be signed, we first need to pad it to the next address that has 0x00000 at the end. The kernel image contains the size and can be extracted using 'od'. Notice that address is stored in little endian. Then add the Image Vector Table (IVT). This table contains addresses for the different parts of the image. The IVT can be generated using a simple script from the variscite git[1].

[1] `https://github.com/varigit/var-hab-cst-scripts/blob/master/var-genIVT`

Supply the kernel load address of the chip (can be found in the `loadaddr` environment variable in U-Boot) and the size of the padded kernel image (result of the ‘od‘ command). Then the signature can be added. Create a CSF and make a binary of it. In the ‘Authenticate Data‘ of the CSF, also add the kernel device tree.

```
[Header]
    Version = 4.2
    Hash Algorithm = sha256
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS
    Engine = CAAM

[Install SRK]
    # Index of the key location in the SRK table to be installed
    File = "cst-certs/iMX8M/crts/SRK_1_2_3_4_table.bin"
    Source index = 0

[Install CSFK]
    # Key used to authenticate the CSF data
    File = "cst-certs/iMX8M/crts/CSF1_1_sha256_4096_65537_v3_usr_crt.pem"

[Authenticate CSF]

[Install Key]
    # Key slot index used to authenticate the key to be installed
    Verification index = 0
    # Target key slot in HAB key store where key will be installed
    Target Index = 2
    # Key to install
    File = "cst-certs/iMX8M/crts/IMG1_1_sha256_4096_65537_v3_usr_crt.pem"

[Authenticate Data]
    # Key slot index used to authenticate the image data
    Verification index = 2
    # Authenticate Start Address, Offset, Length and file
    Blocks = 0x40480000 0x00000000 0x01e00020 "Image_pad_ivt.bin", \
             0x43000000 0x00000000 0x00010000 "imx8mp-var-dart-dt8mcustomboard-m7.dtb"
```

```
# extract image size
od -x -j 0x10 -N 0x4 --endian=little Image
# output is in little endian, so size is 0x01e00000
# 0000020 0000 01e0
# 0000024
# pad the image
objcopy -I binary -O binary --pad-to 0x1e00000 --gap-fill=0x00 Image Image_pad.bin

# generate IVT using variscite script
./var-genIVT 0x40480000 0x1e00000
# Append the ivt.bin to the end of the padded Image
cat Image_pad.bin ivt.bin > Image_pad_ivt.bin

# Create CSF binary
cst-3.1.0/release/linux64/cst --i Image.csf --o csf_Image.bin
# Add CSF binary to end of Image
cat Image_pad_ivt.bin csf_Image.bin > Image_signed
# Zip the image
gzip -f Image_signed
```

The image is now signed and ready to be loaded onto an sd-card, see appendix B. If the image is a kernel, then don't forget to also load the kernel device tree onto the sd-card.

Test the authentication

When the kernel is loaded onto the sd-card, then boot the device and go into the u-boot console. Check if your kernel image has the same as the default kernel image name by typing `printenv image`. The default for `image` is `Image.gz`. If your image is named different, then change the environment variable using `edit image` . Then load the kernel into memory using `run loadimage` and notice the `Uncompressed size`, this is needed later. Also load the device tree using `run loadfdt`. Now you can check whether the signing process went correct by running `hab_auth_img $loadaddr 0x<size-of-image-in-hex> 0x<ivt-offset-in-image>`. The size of the image is the `Uncompressed size` that was shown during `run loadimage`. The `ivt` location is a bit more difficult. Remember that the signed image is build up as follows: `kernel - padding - ivt - csf`. So the IVT offset is the size of the kernel+padding. This is the value that came out of the `od` command. If all is correct then a message of `No HAB Events Found!` will be displayed. If some error messages about 'bad magic' are displayed, then the addresses in the `hab_auth_img` command are incorrect.

```
u-boot=> printenv image
image=Image.gz
u-boot=> edit image
edit: Image_signed.gz
u-boot=> run loadimage
12771359 bytes read in 537 ms (22.7 MiB/s)
Uncompressed size: 31463784 = 0x1E01968
u-boot=> run loadfdt
fdt_file=imx8mp-var-dart-dt8mcustomboard-m7.dtb
65536 bytes read in 5 ms (12.5 MiB/s)
u-boot=> hab_auth_img $loadaddr 0x1e01968 0x1e00000
hab fuse not enabled

Authenticate image from DDR location 0x40480000...

Secure boot disabled

HAB Configuration: 0xf0, HAB State: 0x66
No HAB Events Found!
```

A.4 Program e-fuses

CAUTION: This process is irreversible, once you write the keys to the e-fuses you can't change them. Be sure to do it right the first time!

A.4.1 Write SRK Table Hash to e-fuses

Yocto generates `SRK_1_2_3_4_fuse.bin.u-boot-cmds` when HAB is enabled. This file contains instructions with how to setup the fuses for your specified SRK Table hash. The values in this instruction file can also be found in the `SRK_1_2_3_4_fuse.bin` file which was created during the generation of the PKI tree. Those values can be displayed using `hexdump -e '/4 "0x%X\n"' < SRK_1_2_3_4_fuse.bin`. Both files will show the same values, but the `SRK_1_2_3_4_fuse.bin.u-boot-cmds` also provides instructions on how to use the values.

These values need to be written to bank 6 word 0-3 and bank 7 word 0-3 in the order displayed by the hexdump. The instructions file also shows this well. Go to the U-Boot commandline and follow the

instructions from your instructions file. **NOTE, these values are just an example! Use your own values**.

CAUTION: These are One-Time Programmable e-fuses. Once you write them you can't go back, so get it right the first time.

```
fuse prog -y 6 0 0xDA6B9ADB
fuse prog -y 6 1 0xDC9B55A1
fuse prog -y 6 2 0x93D10134
fuse prog -y 6 3 0x5CDC8DA3
fuse prog -y 7 0 0x143709F8
fuse prog -y 7 1 0xC6E305A7
fuse prog -y 7 2 0x3E718DA2
fuse prog -y 7 3 0xC6AC85B6
```

A.4.2 Closing the device

CAUTION: Again this irreversible! Only close the device when preparing a device for production and you are sure the HAB functionality works (ie. no HAB events pop up)

Closing the device is not necessary to enable HAB. In development do NOT close the device. When only the SRK hash has been written to the e-fuses, then will provide feedback with HAB events, but still boot the device even when the signing failed. So the device will never be bricked, when not closed.

To close the device, you have to write a specific fuse. This might be different for different models, so if you don't use the NXP iMX8M PLUS do check if it is the same. To close the iMX8MP execute the following command in the U-Boot commandline.

```
fuse prog 1 3 0x02000000
```

Once the device is closed, it won't boot any images that are signed with a SRK Table that creates a different hash than that stored in the e-fuses.

B Image flashing

B.1 Manually

This information about manually flashing images to an sd-card is just here for completeness. It is not recommended to actually use this technique for regular flashing. Rather use the script described in appendix B.2. It implements some safety features to protect against typos. The information here can be used to understand what the script does.

B.1.1 Bootloader

Manually writing the bootloader to the SD card is not necessarily hard. It only needs to be put on the correct location on the sd using `dd`. For iMX8M this generally means setting the `seek` option to `33`, however for the iMX8MP, the `seek` option needs to be set to `32` for some reason. Use the following command to write a bootloader binary to the sd. Be sure to replace `<x>` with the correct drive letter.

```
sudo dd if=bootloader.bin of=/dev/sd<x> bs=1K seek=32; sync
```

B.1.2 Kernel and Rootfs

To write the Kernel and root filesystem to the sd card, first a partition needs to be specified in the correct location on the SD-card.

Setup partition

The sd-card for the iMX8M needs a specific layout. First 8MiB of unallocated space for the bootloader and then a single ext4 partition for the root file system. You can setup this layout manually by using `fdisk`. However it is easier to first flash the sd using the `.wic` generated by Yocto or the Variscite SD setup script, see appendix B.2. Then when the partition is setup, you can overwrite the data that the scripts wrote with your own data.

If you do want to use `fdisk`, you will have to run `sudo fdisk /dev/sd<x>`, replacing `<x>` with the correct drive letter. List the partitions by typing `p`, to see whether there are already some the sd card. If there are already partitions, delete them all by typing `d` until there are no left. Then create a new partition by typing `n`. Choose primary partition type and partition number 1. The first sector has to have an offset of `16384` sectors for the bootloader, because 8MiB is 8388608 bytes and each sector on the sd-card is 512 bytes, so $8388608/512 = 16384$ sectors. The last sector can be the default to the end. Remove the ext4 signature when asked. The press `w` when back in the main menu to write the configuration to the partition table.

When the partition is made, it needs to be formatted, before it can be mounted. The filesystem format that needs to be used is `ext4`. Formatting the partition is easy with the following command, but do replace `sd<x>1` with the correct partition that you want to format.

```
sudo mkfs.ext4 /dev/sd<x>1 -L rootfs
```

Write Rootfs to ext4 partition

First mount the ext4 partition. Make sure you have a place to mount the sd-card, if not add a directory in `/media` where you can mount the sd-card.

```
sudo mkdir /media/sd-card
sudo mount /dev/sd<x>1 /media/sd-card
```

When the ext4 partition is mounted, then unpack the rootfs tar in the partition.

```
pv build_<build-dir>/tmp/deploy/images/<machine>/<image-name>-<machine>.tar.gz | sudo tar
    -xz --same-owner -C /media/sd-card
```

This rootfs already contains the kernel and device trees in its '/boot' directory.

Write Kernel to Rootfs

Mount the rootfs partition to the dev pc. Copy the kernel image and kernel device tree to the `/boot` partition of the rootfs on the sd card. Use `sync` to be sure that everything is written to the sd-card and then `unmount` the sd-card.

```
sudo mkdir -p /media/sd-card/
sudo mount /dev/sdc1 /media/sd-card/
sudo cp <path-to-build-dir>/Image.gz /media/sd-card/boot/
sudo cp <path-to-build-dir>/<name-of-dt>.dtb /media/sd-card/boot/
sync
sudo umount /dev/sdc1
```

B.2 Using Variscite script

Variscite has a very useful script[1] to write images to an SD card. It has some nice quality of life features, such as sanity checks. Using this script is less dangerous than using `dd` directly, because one small typo in the drive name can destroy your running system. This script can be found in the `meta-variscite-sdk` layer in `scripts/var_mk_yocto_sdcard/var-create-yocto-sdcard.sh`. You run it with

```
sudo MACHINE=imx8mp-var-dart sources/meta-variscite-sdk/scripts/var_mk_yocto_sdcard/var-
    create-yocto-sdcard.sh -d $(pwd)/build_wayland/tmp/deploy/images/imx8mp-var-dart/core
    -image-base-imx8mp-var-dart /dev/sdc
```

The script uses the same techniques as the manual process, but nicely automated in a script.

The script does have some quirks however. This script has been copied and some additions are made to remedy some quirks. For example, it only looks for `build_x11`, `build_xwayland`, `build_wayland` and `build-<machine-type>` build directories. An extra line is added to allow the user to specify a custom build directory name by setting the `YOCTO_BUILD_DIR` variable, like so:

```
sudo YOCTO_BUILD_DIR=build_poky MACHINE=imx8mp-var-dart sources/meta-variscite-sdk/
    scripts/var_mk_yocto_sdcard/var-create-yocto-sdcard.sh -d $(pwd)/build_poky/tmp/
    deploy/images/imx8mp-var-dart/core-image-base-imx8mp-var-dart /dev/sdc
```

Next to that the `-d` option has to be set if a different image is used, than the default `fsl-image-gui`. This option requires an absolute path and thus gets quite long. To resolve this, the `IMAGE` bash variable can be set to specify a non-default image. For example:

```
sudo YOCTO_BUILD_DIR=build_poky MACHINE=imx8mp IMAGE=core-image-base sources/meta-
    variscite-sdk/scripts/var_mk_yocto_sdcard/var-create-yocto-sdcard.sh /dev/sdc
```

[1] https://github.com/varigit/meta-variscite-sdk-imx/blob/hardknott/scripts/var_mk_yocto_sdcard/var-create-yocto-sdcard.sh

Also an option is added to allow for keeping a previous bootloader or rootfs/kernel. By default if these bash variables are not set, then both will be written to the sd-card. By setting `KEEP_BOOTLOADER=1`, no new bootloader will be written to the sd-card. So, if there is already a bootloader on the SD-card, then this one will remain there. For `KEEP_KERNEL=1`, it is the same, but for the rootfs and kernel. If the option is set to `0`, then the option is also disabled again.

```
sudo YOCTO_BUILD_DIR=build_poky MACHINE=imx8mp IMAGE=core-image-base KEEP_BOOTLOADER=1
    sources/meta-variscite-sdk/scripts/var_mk_yocto_sdcard/var-create-yocto-sdcard.sh /
    dev/sdc
```

These additions are stored in this Neways Azure DevOps git repository.

C Fix RPMsg waiting for link up problems

When developing for RPMsg, the Linux and FreeRTOS programs need to be rerun often. However, when an RPMsg connection has been made before and another program tries to use the same endpoints again, then no connection will be established, even if the previous programs are stopped. This will result in the FreeRTOS program to forever wait for the link between it and the Linux program to go up. A reboot will make the link available again, but this takes a lot of time during development.

To fix this issue, the RPMsg bus needs to be reset. This will restart RPMsg without having to restart the entire device. Just unload and reload the `rpmsg_virtio_bus` kernel module.

```
rmmod virtio_rpmsg_bus
modprobe virtio_rpmsg_bus
```

If this kernel module does not show up with the `lsmod` command, then you need to make this driver a kernel module instead of a built-in driver. To edit the kernel config using Yocto and bitbake, run `bitbake -c menuconfig virtual/kernel`. The `virtual/kernel` recipe is a pointer to the specified kernel, for example `variscite-linux`. This will open a blue menu. Go to `device drivers` and find the `RPMsg` driver. In this driver menu change the `virtio rpmsg bus` from built-in `[*]`, to module `[M]`. Then exit the menu and build the rest of the image. Warnings will show that a temporary change has been made, these can be ignored.

Now every time a new version needs to be tested of a program that interfaces with RPMsg, then just reload the `virtio_rpmsg_bus` kernel module and run the new version of the program.