Zurich University
of Applied Sciences

School of
Engineering
InES Institute of
Embedded Systems

Secure Boot for System on Chip

Zurich University of Applied Sciences

Institute of Embedded Systems

Authors	Tobias Vögeli Thierry Delafontaine
Version	1.0
Last changes	July 17, 2020

Contact Information
c/o Inst. of Embedded Systems (InES)
Zürcher Hochschule für Angewandte Wissenschaften
Technikumstrasse 22
CH-8401 Winterthur

Tel.: +41 (0)58 934 75 25
Fax.: +41 (0)58 935 75 25

E-Mail: voegetob@students.zhaw.ch, delafthi@students.zhaw.ch
Homepage: `http://www.ines.zhaw.ch`

Zürcher Hochschule für Angewandte Wissenschaften

School of Engineering
InES Institute of Embedded Systems

Erklärung betreffend das selbständige Verfassen einer Vertiefungsarbeit/Masterarbeit im Departement School of Engineering

Mit der Abgabe dieser Vertiefungsarbeit/Masterarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat.

Der/die unterzeichnende Studierende erklärt, dass alle verwendeten Quellen (auch Internetseiten) im Text oder Anhang korrekt ausgewiesen sind, d.h. dass die Vertiefungsarbeit/Masterarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten Paragraph 39 und Paragraph 40 der Rahmenprüfungsordnung für die Bachelor- und Masterstudiengänge an der Zürcher Hochschule für Angewandte Wissenschaften vom 29. Januar 2008 sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:

Winterthur, 17.07.2020

Unterschrift:

J. Delafontaine

T. Vogeli

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Vertiefungsarbeiten/Masterarbeiten im Anhang mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Abstract

Although many embedded devices lack the required security, manufacturers slowly start to realize the need to secure their products. The company Enclustra wished to have a reference design to help developers starting secure development on their Mercury platform. Following the Platform Security Architecture (PSA) framework from Arm® three different use-cases defined by Enclustra have been analyzed. To implement these use-cases, different security features of the Xilinx Zynq Ultrascale+ were analyzed and implemented.

The outcome is a modular reference design, where different security features can be added, depending on the individual use-case. The base is a PetaLinux project to boot the Zynq Ultrascale+ with an encrypted and authenticated image. It uses the cryptography hardware and key handling implemented by Xilinx. In addition to the base project, features like multiboot or tamper detection can be added as modules to the reference design. The Linux Crypto API has been analyzed to use the cryptography hardware of the Zynq Ultrascale+ in Linux. Code examples for different algorithms have been created to simplify the implementation of the Linux Crypto API. To isolate critical applications, Arm® implements the TrustZone concept in their processor architectures. This concept was analyzed and implemented on the Xilinx evaluation kit zcu102, with OP-TEE as Secure OS. This enables a user to isolate critical applications or data and therefore, protect them even if parts of the system are compromised.

Finally, the thesis shows that the Zynq Ultrascale+ is built for secure product development. Not only cryptography features but also features for maintainability, reliability and secure execution of code are implemented. The reference design is a solid base to get started, using all these features.

Contents

1. Introduction

Security is a significant challenge in industrial systems. Attacks against critical systems are a harsh reality and can cause much harm to customers and the production companies. The security of such systems is often dangerously low. This is mainly due to the lack of knowledge and the missing maturity in the market. According to Kaspersky [1], the maturity in the industrial market is low but increasing, due to the strong negative impact of incidents. Limiting factors are the lack of skill and collaborations. Security has to take up a more critical role in the development of future systems. However, where and how to start? This is where Enclustra intends to help with the outcome of this thesis.

Enlcustra is an independent FPGA designer. They offer different FPGA and SoC modules, which contain the necessary parts used in a broad spectrum of applications. To add application-specific features, a customer can mount the modules on a custom base-board. This gives the customer the freedom to focus on the development of the base-board and relieves him from designing the complicated module. As a seller of these modules, Enlcustra supports its customers during the design process. Building a knowledge base about security simplifies the start into development with security in mind.

This work focuses on the Zynq Ultrascale+ produced by Xilinx. This device is a System on Chip (SoC), which combines a multitude of different processors, as well as Programmable Logic (PL) and hardened cores dedicated, to specific functions. The Zynq Ultrascale+ is equipped with up to four Arm® Cortex™-A53 Application Processing Unit (APU) cores, a dual-core Arm® Cortex™-R5 Realtime Processing Unit (RPU) and an Arm® Mali™-400 MP2 Graphical Processing Unit (GPU). Besides, this chip is built with security and safety in mind and has a lot of features to offer. The chip is equipped with a dedicated Platform Management Unit (PMU) for system and power management and to ensure functional safety. A dedicated Configuration Security Unit (CSU) handles the hardened security features like the AES core, the RSA core and the hashing core. It provides secure key storage and different methods to minimize key usage. Additionally, different parameters can be monitored, like voltage and temperature, to detect tampering and prevent data disclosure. The chip also contains the Arm® TrustZone technology, as well as the Arm® Cryptography Extension to support different cryptography methods. [2]

The objective of this work is to provide a template project on GitHub, which covers the security issues detected in different use-cases. A customer of Enclustra should be able to analyze his product and be able to start with the template project and build his product onto the template. An implementation guideline explains the different parts of the example. Xilinx as well provides implementation examples, but usually, they are restricted to bare-metal applications. In contrast, this project concentrates on the implementation with Linux as an Operating System and U-Boot as a Second Stage Boot Loader (SSBL). To reach this outcome, we set multiple intermediate goals:

a) Familiarize with the Zynq Ultrascale+ and its features and functions.

b) Analyze the different use-cases provided by Enlcustra. Provide for each case a Data Flow Diagramm (DFD) and analyze it according to the STRIDE model.

c) Analyze the different security features provided on the Zynq Ultrascale+.

d) Elaborate on how the discovered security issues can be removed, using the given security features. Consider also using external hardware.

e) Implement elaborated concepts and create a bootable system.

f) Test the implementation and show proof of concept. Show that the implementation covers the found security issues.

g) Generate a reference design, which will be provided as an open-source project on GitHub.

The task description in appendix A.2 gives a more detailed view of the project objectives.

To cope with this work, we used the Platform Security Architecture (PSA) framework provided by Arm®. Each use-case defined by Enlcustra was separately analyzed and could be mapped to a work package, covering some particular features of the device. The first use-case for example covered mostly the security hardware, like AES, and RSA. Therefore, the PSA framework was elaborated at first, and then it was applied to the use-cases.

This thesis is structured accordingly. Chapter 2 gives an overview of the systematic approach to secure development according to the Platform Security Architecture (PSA) framework introduced by Arm®. Chapter 3 analyses the use-cases, given by Enclustra and reveals possible threats. It covers the first step of the PSA framework, analyze. Chapter 4 examines the features of the Zynq Ultrascale+. This chapter covers the second step, architect, of the PSA framework. Chapter 5, covers the next PSA step, implementation, and shows the use of the different security features to remedy the security issues, discovered in Chapter 3. Chapter 6 shows how the implementations were tested, to prove that they work. This chapter covers the last step of the PSA framework, certify, or in this work more validation. The last chapter, Chapter 7, picks up the elaborated results and implementations, gives a conclusion and an overview of what could be done next.

1.1. Project Data

All data generated and gathered throughout the project was stored on GitHub. During the project we worked with an internal and private repository, which can only be accessed, by authenticated members. This repository can be accessed with the following link.

https://github.zhaw.ch/hpmm/BA20_rosn_02_Secure_Boot

A primary goal of this project was to create a reference design, which can be publicly accessed and maintained. Therefore, a separate Git repository was created. It can be accessed using the following link.

https://github.zhaw.ch/hpmm/BA20_rosn_02_Secure_Boot_reference_design

2. Secure Development

The ever-growing number of embedded hardware shows that we are dependent on these devices. They take up significant roles in a multitude of systems. According to Symantec [3], more than 57'000 attacks against IoT devices have been registered. ENISA [4] states, that attacks against Industial Control Systems are increasing. Therefore, security can not be looked at as secondary. Even though security is time-consuming and expensive, it has to take up a primary role in the development process and life-cycle of a product.

2.1. Platform Security Architecture (PSA)

Arm® provides the Platform Security Architecture (PSA) framework to help customers with a systematic approach to security during the development process. The four stages of the PSA framework are seen in Figure 2.1.

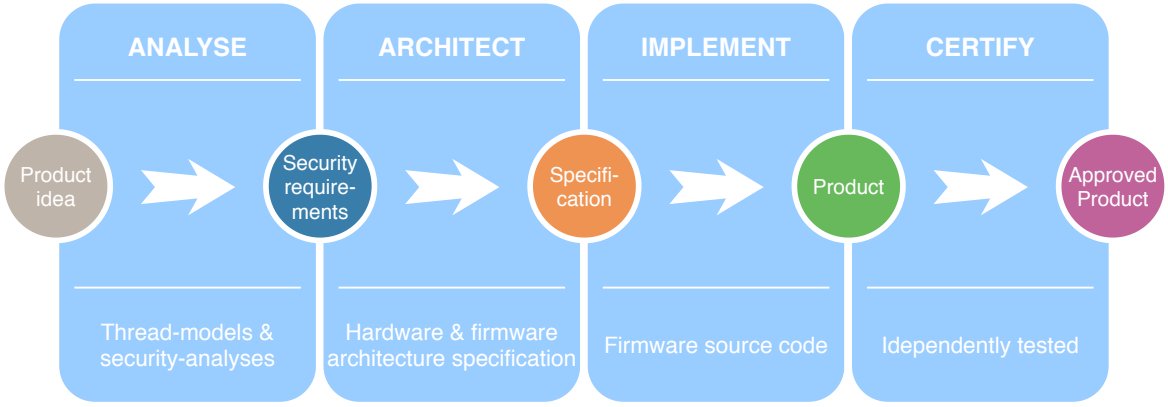


Figure 2.1.: The four stages of PSA

This process is Arm® specific, especially the last process. In this documentation we call the last process validation. [5]

The first stage, analyze, is further described in Section 2.2. During thread modelling, a product and its environment are analyzed, to derive security requirements. Chapter 3 describes the application of threat modeling to the use-cases in this project. In the second stage, architect, the previously derived security requirements are translated to hardware and software requirements. Therefore, this part is heavily dependent on what hardware and software you use. In this project, only the Zynq Ultrascale+ is used for development. Thus, the different features of the Zynq Ultrascale+ are analyzed in Chapter 4. During stage three, implementation, the product is developed. This stage is covered in Chapter 5. The last stage, the validation process, ensures your product meets the requirements. In this project, we made only a proof of concept verification. Though, a real product should be further tested. Of course, this step can also include certification for the product to reach a certain level of security.

2.2. Threat Modeling

The process of threat modelling helps to identify what possible weak points a product has for an attacker and which attackers to expect. In order to be successful, the main business goals of the

product have to be known. They help to focus on the right issues. Additionally, an understanding of the product is necessary to identify the weak points. The threat modeling process consists of five steps. [6;7]

1. The process starts by identifying basic security goals. They help to focus on the right issues.
2. The next step is to know your process and your assets. What could be worth trying to steal or tamper, and how is the data handled? This step involves not only internal processes but also interactions with the product from external sources, like human input. The output of this step is a Data Flow Diagramm (DFD), which shows processes, interaction, inputs and outputs.
3. After analyzing the assets, the attackers have to be analyzed. Which enemy has an interest in which assets? What is the attacker willing to do to get to his goal? What are his means? With the knowledge about the assets and the attackers, security issues can be detected in the DFD.
4. Usually the threats from the previous step are too numerous. Therefore the threats have to be evaluated, to concentrate on the most severe ones. The security goals defined in the first step should also provide guidelines to focus on the main threats. In the end, each selected threat should be countered by a requirement. A requirement can also tackle more than one threat.
5. The final step is to summarize the data in a table to structure and understand the thinking process.

The following sections should give an introductory overview and a general knowledge about the different proceedings. An example, following these steps, is given in Chapter 3.

2.2.1. Define Security Goals

To define security goals is necessary, in order to focus on the important threats. Security goals are common specifications for a product. They can emerge from industry standards or company standards. Usually, three to six security goals should be specified. The following example, called the CIA triad [8], defines three security goals.

1. **C**onfidentiality ensures that an embedded system's critical information, such as application code and surveillance data, cannot be disclosed to unauthorized entities.
2. **I**ntegrity ensures that adversaries cannot alter system operation.
3. **A**vailability assures that mission objectives cannot be disrupted.

Security goals can be different for each company or system. For one, confidentiality is a security goal to ensure that data cannot be disclosed to unauthorized users. For another, this could be irrelevant, because open-source software is used, and no confidential data can be disclosed. Then, integrity may be a security goal to ensure that system operations cannot be altered. Therefore, the security goals also depend on the use-case. [7]

2.2.2. Know your product and assets

In order to find any threats, the analyzer has to know the product, not only the internal processes but also external entities like users, servers, external sensors, and others. This knowledge gives an idea where adversaries may attack. In order to why they want to attack, the assets of a product have to be known. These can be log-in credentials, keys, certificates, software, firmware, but also device resources. To visualize the processes and entities in a clear way, a Data Flow Diagramm (DFD) is created. This DFD helps to further recognize threads. [6]

2.2.2.1. Create a Data Flow Diagramm (DFD)

The Data Flow Diagramm (DFD) is a two-dimensional visualization of interactions, processes and entities. It shows the exchange of data. In Figure 2.2 the main components of the DFD are introduced. The following paragraphs give a short introduction to each one of them. [9]

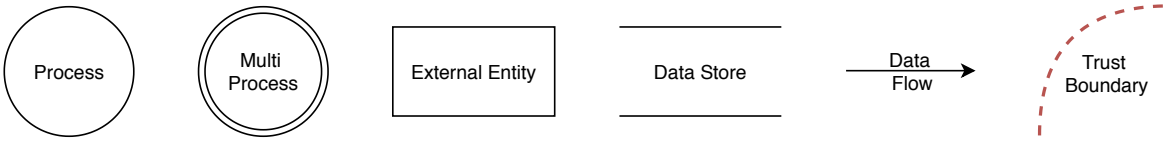


Figure 2.2.: Different components of a Data Flow Diagramm

Process A process handles input data and outputs it to other components of the application. An example could be a specific service, which processes data.

Multi-Process The multi-process represents a collection of processes. A multi-process could be a server with multiple services.

External Entity An external entity is an external component, which interacts with the system. External entities can be the user giving input information or an external sensor sending data.

Data Store A data store represents a location where data is saved. Examples for a data store can be a simple SD-card or a database.

Data Flow A data flow represents an interaction, or an exchange of data in the direction the arrow is pointing. Additionally, the communication protocol can be written onto the arrow.

Trust Boundary A trust boundary represents a boundary where two entities should not automatically trust each other, meaning that data exchanged between another is possibly tampered and thus not trustworthy.

2.2.3. Identify Adversaries

Now that the assets are known, the adversaries can be identified and analyzed. There is a multitude of threat agents, and each one has his level of motivation, resources, and skills. Figure 2.3 shows an overview about different threat agents. In Figure 2.3, the agents are mainly classified according to their skill and resources or possibilities.

In the following paragraphs the top threat agents according to the ENISA threat landscape from 2018 [4] are described. Listed on top are the agents with most responsibilities to attacks.

Cyber Criminals Cyber Criminals act out of financial profit. According to the ENISA threat landscape 2018 [4], they are the most active threat agent groups with a responsibility of more than 80% of all incidents. They act with medium to high resources and have a corresponding skill level. As they are mainly profit-driven, they will target fields with a high profit/expense ratio. [11]

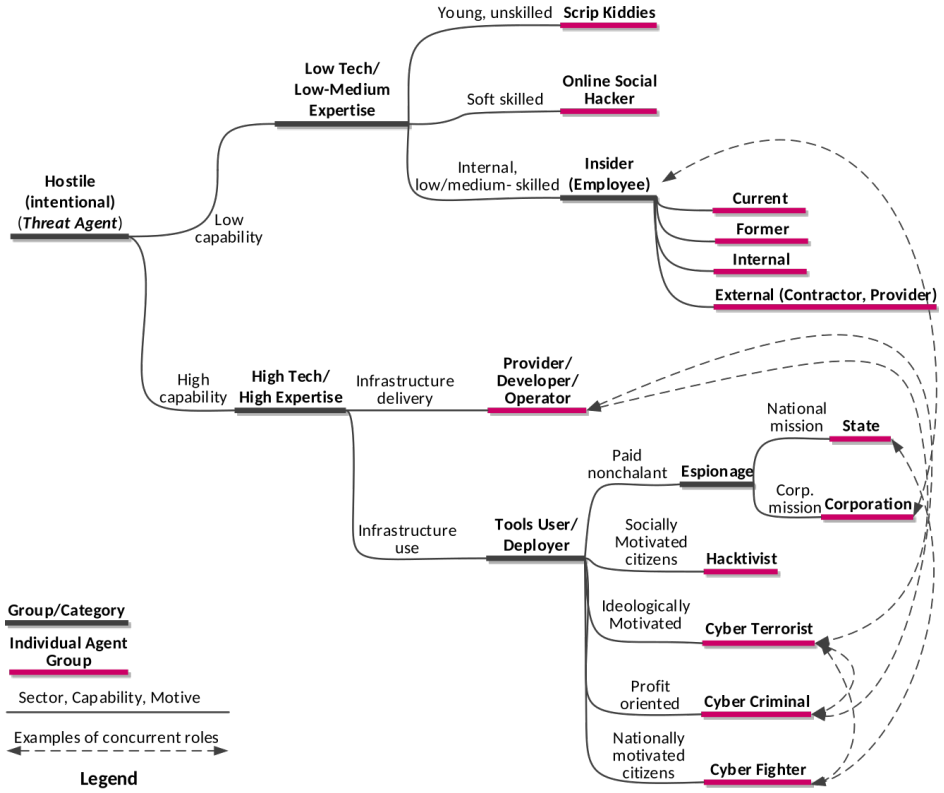


Figure 2.3.: Threat agents according to the ENISA Threat Landscape Report [10]

Insider Insider incidents contain intentional and unintentional attacks. Insider attacks are responsible for 25% of the incidents in the corporate environment [4]. Insiders act out of revenge or profit. They use their insider know-how or misuse their privileges to access systems and information. Because they know the system and its protection, the attacks are sometimes hard to mitigate. However, more damage is caused by unintentional actions, than misuse of privileges. [4;11]

Espionage/Cyberspies Cyberspies have a very high skill level. Their resources are also very high, especially if a state finances the group. They target state, company and military secrets, critical and military infrastructure. Attacks against Industial Control Systems and banks increased in 2018 [4]. Corporative cyberspies have almost the same resources as national cyberspies. [4;11]

Hacktivists Hacktivists act to draw the attention of the media. They launch defacement campaigns on target web sites to raise awareness about wrongdoings. Mainly, they use SQL injection to hack web sites or DDoS attacks to block web services. Hacktivists can be responsible for the leakage of confidential information found on hacked websites. They may not cause a lot of damage, but they can significantly harm the reputation of the target. [9;11]

Cyber Terrorists Cyber terrorists are mostly religiously driven. They use the web for propaganda to recruit new members and raise funds to finance their operations. Their capability, according to cyber-attacks, is relatively low, but the potential to recruit hackers is rising. Cyber terrorists would mostly aim at critical infrastructure to weaken national acceptance, thus targeting Industial Control Systems. [4;11]

Script Kiddies Script kiddies are the least skilled adversaries. They act out of fun, fame, and thrill. While they are currently unskilled, a script kiddie could become very efficient at hacking. Despite their low skill, the attacks can have a high impact, which shows the effectiveness of existing tools. [4;11]

2.2.4. Identify Threads with STRIDE

The STRIDE system is used for a structured approach. STRIDE was introduced and used by Microsoft to analyze their software and services for threats. Because of that, it is not a native threat model to analyze embedded systems. However, it can be easily applied. STRIDE is an acronym and stands for [9]:

- **S**poofing:
  Impersonating something or someone else to gain a non-legitimate advantage. Spoofing violates the authenticity of an application.
- **T**ampering:
  Modification of data or code while it is transmitted or stored. Thus, violating the integrity of an application.
- **R**epudiation:
  Deny to have performed an action and leaving no evidence to prove against, meaning no evidence can be linked to an attacker. This violates the non-repudiation of an application.
- **I**nformation disclosure:
  Exposing information to someone not authorized to see it or getting access to data without authorization. Information disclosure violates the confidentiality of an application.
- **D**enial of Service:
  Deny or degrade service to legitimate users and violating the availability of an application.
- **E**levation of Privilege:
  Gain capabilities without proper authorization. This violates the authorization of an application.

All six categories can be applied to the DFD components. Table 2.1 shows, which component is at risk from which category of STRIDE.

DFD-Element	S	T	R	I	D	E
(Multi-) Process	X	X	X	X	X	X
External Entity	X		X			
Data Store		X	X	X	X	
Data Flow		X		X	X	

Table 2.1.: STRIDE applied to the DFD components

2.2.5. Threat Evaluation Methods

The next step is to evaluate the threats. A lot of threads may have been found in the previous steps. Some may occur frequently, some not, others may cause a lot of damage, if they occur. Usually, there are too many threats to work trough. An excellent method to evaluate threads is the CVSS method. However, in this case, the threads are evaluated using the more straightforward risk analysis.

Each detected threat is evaluated according to their roughness of impact and their chance to occur. Validating each threat helps to rate the threats from high to low. High-rated threats should be processed first, then the following. A cost analysis for both sides, the developers and the attackers

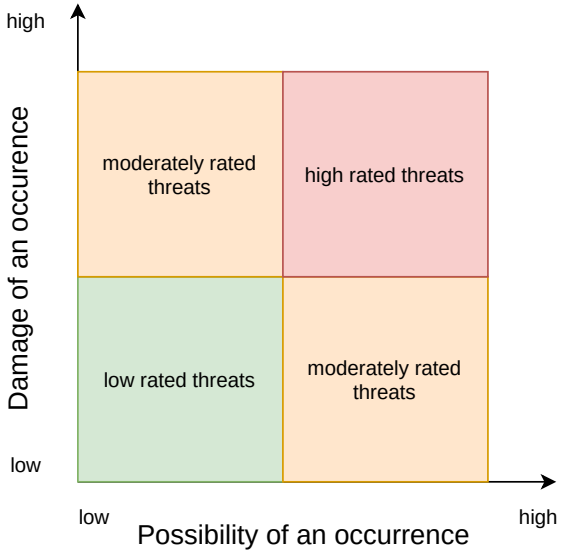


Figure 2.4.: Risk analysis for threat evaluation

may also be considered. It gives a more realistic feeling, what attackers may do to get what they want.

2.3. Apply Threat Modeling to Hardware

The original DFD entities and the STRIDE model originated from the IT world and are not directly applicable to hardware and embedded systems. Therefore, if hardware components are analyzed, the DFD elements are transferred to their hardware counterparts.

The multi-process represents processing hardware, and the database/datastore entity represents storage hardware, external entities I/O Ports and the data flow arrow communication between the different parts. Figure 2.5 shows the changed DFD elements.

Figure 2.5.: Different components of a DFD applied to hardware

Processing Hardware This element is used for everything, which processes data. This can be a CPU or an IP in the FPGA section of the chip. This entity is vulnerable to spoofing by running unauthorized code. Additionally, the processing unit may also be vulnerable to an elevation of privileges. Malicious code may access restricted hardware sections, which the code is not authorized to.

External I/Os This element represent every I/O interface. It includes simple I/O pins as well as JTAG interfaces. The vulnerability of this entity is denial of service. An I/O port can be unavailable, leading to unsuccessful communication. Additionally, only used I/O ports should be activated. Other I/O ports should be deactivated, as each I/O port represents a potential risk, and grants access to the system — especially ports used for development, as the JTAG interface.

Data Storage The representation of the data storage entity is self-explanatory. Data Storage is vulnerable to information disclosure and tampering. Every data storage should be looked at as readable and writable, especially external memory. Therefore, sensitive data needs to be encrypted and/or verified. Additionally, a data storage is also vulnerable to denial of service. If memory is unavailable, it can lead to errors during code execution.

Communication Flow This arrow represents the exchange of data and thus who can read and write data to which elements. Communication flow is only vulnerable to information disclosure. Observation of data flowing from one component to another is possible. As data exchanges very fast, targeted manipulation of data during the transport is impossible.

The STRIDE model also has to be adjusted to fit the vulnerabilities of these entities.

DFD-Element	S	T	R	I	D	E
Processing Hardware	X					X
External I/Os					X	X
Data Storage		X		X	X	
Communication Flow				X		

Table 2.2.: STRIDE applied to the DFD components for hardware

3. Security Analysis

In this chapter, a security analysis, as described in Chapter 2, is applied to three different use-cases. Enclustra derived the three use-cases from their work with customers. These use-cases derive from areas in which most of the customers had questions or concerns about security. In addition to the use-cases, three security goals have been defined. first, the three use-cases are described and then separately analyzed.

1. **Use-Case 1**: The customer deploys a new product. What does he have to consider, regarding his security analysis? This case includes the deployment of the product, as well as development and production.
2. **Use-Case 2**: The customer wants to update an existing product with new firmware or software. The update process discussed in this case is limited to updating by physical access. Update through the network is not considered, because then network-related security has to be also considered.
3. **Use-Case 3**: The customer wants to update his base-board. What does the customer need to change, especially when he is using the tamper detection unit?

The further description of each use-case should raise the awareness and start a thought process about possible security issues in the customers product. The first two use-cases follow the analysis process described in Chapter 2. Therefore, for the first two use-cases, at least one DFD is generated. In the first use-case, an example product is analyzed, to show the development process on a real product rather than only analyzing the hardware of the chip. At the end of a use-case, a table lists the identified vulnerabilities with an appropriate countermeasure. The last use-case has been analyzed differently because the threat modelling procedure could not be applied. Product security in the last use-case relies heavily on the previous development process for that product. Therefore, we point out possible mistakes and difficulties during hardware updates.

3.1. Define Security Goals

Based on the objectives of this thesis, we defined the following three security goals, derived from the specifications Enclustra gave us:

1. The device should only run authenticated software, maintaining a secure state throughout the device life-cycle.
2. Proprietary and secret software, should be handled confidentially and not be disclosed to unauthorized externals
3. The device captures any irregular modifications and informs the owner about incidents. Thus, safety precautions should prevent tampering.

3.2. Use-Case 1

Use-case 1 concentrates on the development, production, and deployment of a product. The section is divided into two parts. The first one covers the development and deployment of products and the second one the production process. Section 3.2.1 gives an overview of the hardware. What communicates with what? Where are trust boundaries? This part concentrates not on products but is necessary in our work to analyze the first use-case and understand the internals of the hardware. Afterwards, a realistic product is analyzed, which serves as an example for thread analysis.

3.2.1. Development

The development process is an essential part to launch a secure product. During development, many important decisions have to be made. A lot of them have a significant impact on the future life-cycle and success of the product. Before even getting into the details, it is crucial to understand what the product should do and how should it do it. Following questions give an insight into the product.

- What is the main business idea of my product? As mentioned in Section 2.2.2, the main goal of this question is to understand the basic concept of the product. This question is needed to generate a good DFD. It is also important to know which customers are targeted. Some may have other requirements for the product than others.
- What are my assets? How much are they worth? Assets are particularly crucial for threat analysis. Usually, an attacker goes only as far as it is worthwhile. Therefore, it is vital to know the effort of specific attacks, because no one is going through the immense effort of cracking a system if nothing valuable can be retrieved.
- Which could be potential enemies? This question is strongly related to the upper one. According to the assets and the value of the assets, the means of an attacker can be estimated. For example, if the product is only interesting for script kiddies, it is sure, that they will not launch a power consumption analysis attack, because the equipment and know-how needed to make such an attack, does not reflect their means.
- In which environment will the product be deployed? To what environment is the product exposed. The security requirements will be different if the product is inside building with restricted access, or if it is publicly accessible, especially for attacks, that require physical access to the device.

3.2.1.1. Hardware Analysis

Because we do not have a final product with specific requirements and assets, the STRIDE model was applied directly to the Zynq Ultrascale+ hardware. This application shows where different threats apply. Figure 3.1 shows an overview of the different components and what communicates with what. The introduced DFD elements for hardware are used in this analysis. For further simplification, different elements have been drawn inside boxes. These boxes signify in which sections of the chip, the components are located. When an arrow is pointing to a box, containing multiple items, it means, that this component can communicate with everything inside this box and vice versa, if the arrow points in the opposite direction.

The chip itself was defined as secure (to protect against hardware attacks read Section 4.5 about the tamper monitoring unit), meaning data stored inside the chip can not be accessed without the right privileges in software. Therefore, the trust boundary can be drawn outside of the chip. Critical information has to be either stored inside the chip or treated specially. If the information is confidential, it has to be stored encrypted, or if it has to be tamper-prof, it needs to be verified. Special attention has to be given to easily accessible entities, like SD cards or USB devices. Those interfaces, paired with insufficient security measures in software enable an attacker to deploy malicious software. The JTAG interfaces are also very dangerous and should only be active during development, as they give

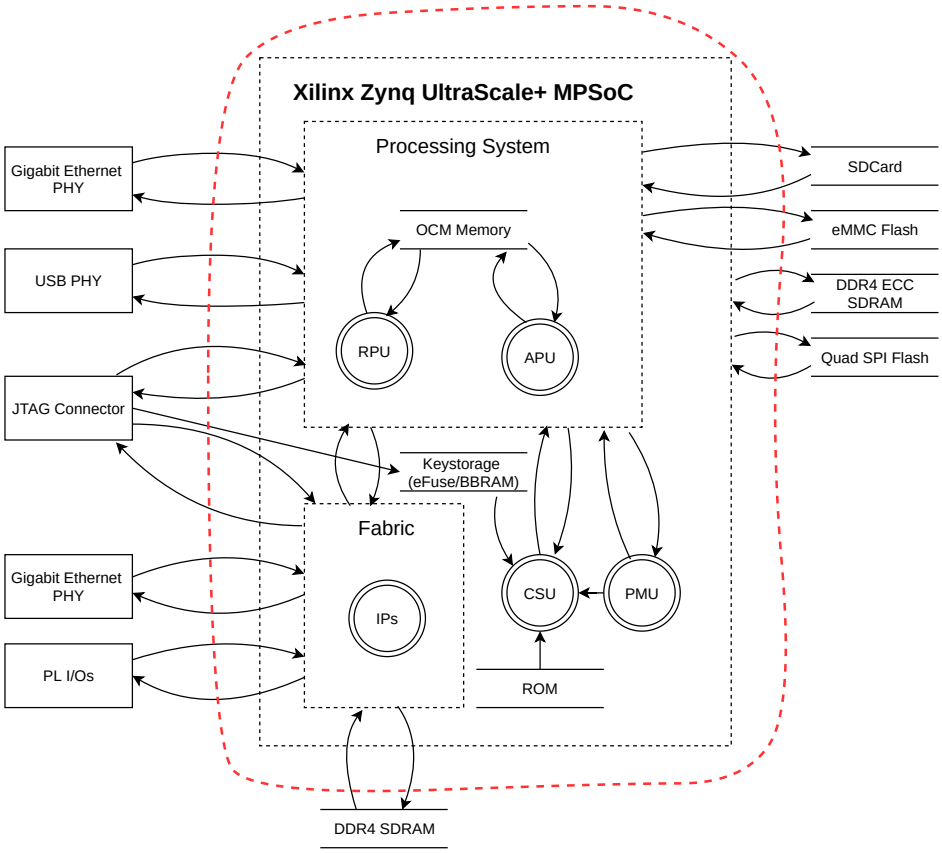


Figure 3.1.: DFD of the hardware [2]

an attacker full access to the device. The JTAG interface can be disabled by writing permanent bits in a specific register described in Section 5.2.3.1. For more details on the internal structure of the chip, refer to the technical reference manual [2].

A threat analysis has been performed for Figure 3.1 and is summarized in Table 3.1.

DFD Element	**Threat**	**Threat Description**	**Countermeasure**	**Designator**
Processing Hardware: APU RPU PMU IPs	Spoofing	A processor may execute malicious code. The CSU is not affected, because it can not run user code.	To counter spoofing threats, executed code has to be authenticated or started by authenticated software.	T1

	Elevation of Privileges	A processor may access restricted parts of memory or I/O by running not trusted code.	To prevent elevation of privileges, not trusted code execution has to be supervised. Access has to be restricted and granted by a supervisor.	T2
External I/O: JTAG, and others	Denial of Service	An external Entity may be inaccessible, which makes the device unusable. Such an I/O port would be the boot pins, which define from which storage the boot image will be loaded.	Restrict access to device hardware to prevent denial of service. Check the external hardware to be accessible before accessing it.	T3
	Elevation of Privileges	Elevated access to the device is possible through the JTAG interface.	To prevent elevation of privileges disable JTAG interfaces.	T4
Data Storage: SDCard eMMC Flash DDR4 SDRAM Quad SPI Flash	Tampering	Data stored outside of the chip may be altered.	To counter tampering, the access to the device has to be restricted. To detect tampering the information can be verified with certifications or hashes.	T5
	Information Disclosure	An attacker may gain access to externally stored data.	Encrypt critical data to prevent Information disclosure.	T6
	Denial of Service	If critical data storage is unavailable, like the boot storage, the device can not boot or can not work any longer.	Restrict access to data storage or use storage, which is more difficult to physically access.	T7

Communication Flow	Information Disclosure	Communication data to external entities may be analyzed.	To prevent information disclosure encrypt critical data.	T8

Table 3.1.: Threats Overview Table

3.2.1.2. Example Product

The following example product is used to measure the traffic on the streets. The product consists of two parts. One part is the measurement unit, which is placed locally in the street. The second part is a server, which serves as an interface to the user. The two parts communicate with each other through the web. A DFD can be seen in Figure 3.2. The measurement unit takes pictures of the traffic processes the data and saves it locally. After a defined time or when reaching a certain quantity of data, the data gets sent to the server via the web, where the user can see it. The measurement unit also sends data about its condition. The security goals set for this product are:

- Only verified software should run on the device.
- Measurement data has to be handled carefully and not be leaked.
- The device has to report security issues and malfunctions.

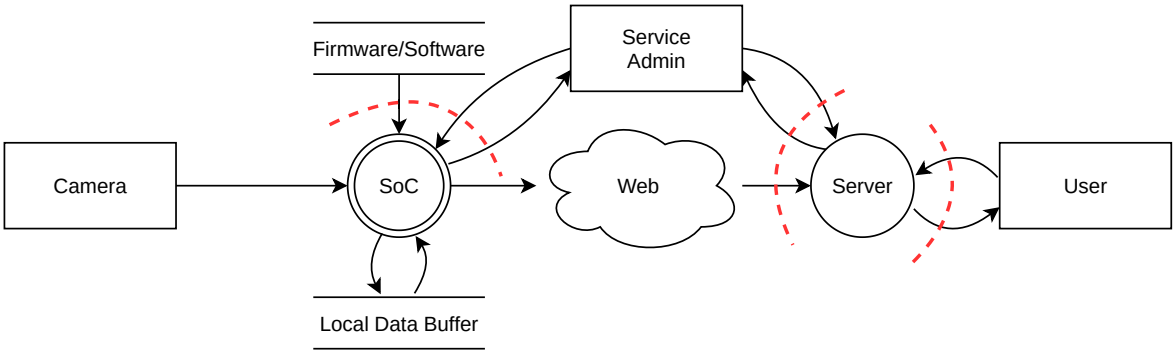


Figure 3.2.: DFD of the example product

The valuable asset for possible attackers is the measured data. The data is critical, because of privacy reasons. The software/firmware on the device is no secret, but the device needs to ensure that only authorized software can be executed. The main threat agent is the hacktivist, which wants to prove that privacy is not respected. He tries to attack the device physically or over the network.

DFD Element	**Threat Description**	**Countermeasure**	**Rating**	**Designator**
Firmware/Software	Software or firmware may be physically exchanged or tampered	Every loaded software or firmware has to be verified. If the verification does not match, the device goes to security lockdown and deletes all critical data.	high	T1

Local Data Buffer	Data can be tampered, making it unusable for analysis.	Save data in the internal storage.	low	T2
	An attacker may gain access to measured data.	Store data in the internal storage.	high	T3
SoC	An attacker may access the device through the service access ports.	Access to device is restricted with a locked casing.	high	T4
	Access through service access ports are not logged.	Log the access through service ports and send them to the server.	moderate	T5
	An attacker can access processed data during runtime with hardware attacks.	Monitor critical resources to detect tamper events.	low	T6
Server	An attacker may gain access to the server.	Restrict access to the server with log in credentials, to only accept authorized users.	high	T7
	Access to the server is not traceable.	Log every access to the server.	moderate	T8
	An attacker may send random data to the server to disable it or make it vulnerable	Received messages have to be verified, before they are further processed. In case of an overload a message will be generated to inform the system administrator.	moderate	T9
Web	An attacker may gain access to data during the transfer to the server (MITM attack).	Transferred data has to be encrypted.	high	T10

Table 3.2.: Threat table of the example product

The threats are rated according to their occurrence probability, their impact and their use to cost ration. The use to cost ratio can be evaluated for the implementation of the security features, but also the attacker. The high rated threats are focused first. Then the moderate ones. The low rated threats have to be further evaluated. In this case, where the expected attacker has low to moderate equipment and skills, the possibility of a hardware attack is not expected, because the material costs and the skill level, to launch such an attack are too high. This problem could also be addressed by making the casing more secure. With that also other threat risks will reduce.

Before deploying a product, it is essential to disable all development features. These features enable nearly unrestricted access to the device and can be very dangerous, if they are not disabled.

3.2.2. Production

Many challenges have to be overcome when it comes to production. This section should only provide a quick overview of decisions and challenges, which have to be handled. Furthermore, this analysis is specific for the Zynq Ultrascale+. Mainly in focus are problems related to decryption and verification. The Zynq Ultrascale+ is intended to use AES GCM for decryption with the possibility for different key sources. The main key sources, Battery Backup RAM (BBRAM) and eFuse, can only hold one key each. While the eFuse key is permanent, the BBRAM key can be exchanged. For more information about AES and its key sources read Section 4.2.1.

For verification the Zynq Ultrascale+ uses RSA. The key used to verify the image is not saved on the device, as it is a public key. Instead, the hash of the key is checked. The hash of the key can only be stored in the eFuse. For more information about RSA and its key sources read Section 4.3

The following list shows a collection of different issues during production. This problems mainly concern the AES master key stored either in the permanent storage (eFuse), or the non-permanent storage (BBRAM). AES is a symmetric cryptography method, meaning the same key is used to encrypt and decrypt. Thus, the key stored on the device is confidential. RSA, on the other hand, is less vulnerable, because it is an asymmetric cryptography method. Stored on the device is only a hash of the Primary Public Key (PPK), which ensures that the right key is used for validation. Beside of that, RSA gives more freedom to key revocation.

- How to handle a large number of devices and thus the keys used? Minimizing the leakage probability of a key is achieved by reducing the usage of a key to a minimum. Therefore, it makes sense to change the primary key for each device. However, how to keep track? Especially if future updates are considered, which also have to be encrypted using the same key. The Battery Backup RAM (BBRAM) gives the freedom to exchange the key during future stages. However, it also raises the risk, that the key is lost, in case the battery is empty.
- How to handle multiple software developer parties? Disclosure of a key can be reduced by handling the key as secret as possible. Optimally, the key can not even be seen by the one encrypting the data. A service should provide this action. Nevertheless, it has to be clear which parties handle keys and if it applies how they are exchanged?
- Who does program the keys to the device? This question mainly concerns companies, which produce their devices externally. In this case, two possibilities apply. First, everything is handled by the external producer, meaning the external producer has to have the keys. Second, the devices are produced externally, but some key features, like programming the keys, are done internally. This circumstance means more effort, but is more secure for the keys and provides some security in case the external producer sells unauthorized copies of the device. In case the keys are programmed internally, illegally sold devices from the external provider can not run encrypted software.
- How are additional keys handled? With additional keys, are meant keys, which will be used in a later state, not the keys used during boot. Where are those keys stored and how? The keys could be encrypted with the master key on the device to provide further security.

In any case, the following points have to be considered, to decrease leakage of keys:

- Use a key as little as possible. Use different keys for different things.
- Share keys as little as possible.
- Handle keys confidential. The less can be seen, the better.
- Exchange keys securely.

The more parties and keys are involved, the more complicated the whole process gets. Thus, the production process demands a good coordination and planning.

3.3. Use-Case 2

Use-case 2 concentrates on the life-cycle of a device, especially with updates to software and firmware. First, this section describes the procedure, to update software or firmware and second, the case, where a key has to be revoked and replaced by another one. The key revocation feature is specific to the Zynq Ultrascale+. While the RSA key can be revoked, the AES key can only be exchanged, when it is stored in the BBRAM. An AES key stored in the eFuse can not be exchanged.

3.3.1. Firmware / Software update

Security is an ever-evolving topic. Therefore, software and firmware have to be updated to match new upcoming threats. However, this topic is not only software but also hardware related. The developer has to consider making a product update-able already during development. For example, the product needs to have the possibility to import the new firmware. Also, the updated firmware has to be encrypted and signed with compatible keys. Thus, the keys need to be known. There has to also be a way to interact with the device in order to start the update process.

Updates are always a risk to make a device unusable. With a backup image available, the risk of breaking the device is reduced. An automated way to do this is with the multiboot registers. In this case, a so-called golden image is stored after the regular boot image. This golden image is a minimal, bootable image, which boots up and updates the regular one, in case the regular one is corrupt and can not boot. To further reduce the risk of breaking the device, the golden image should only be updated on special occasions. The system uses the golden image as a fallback image.

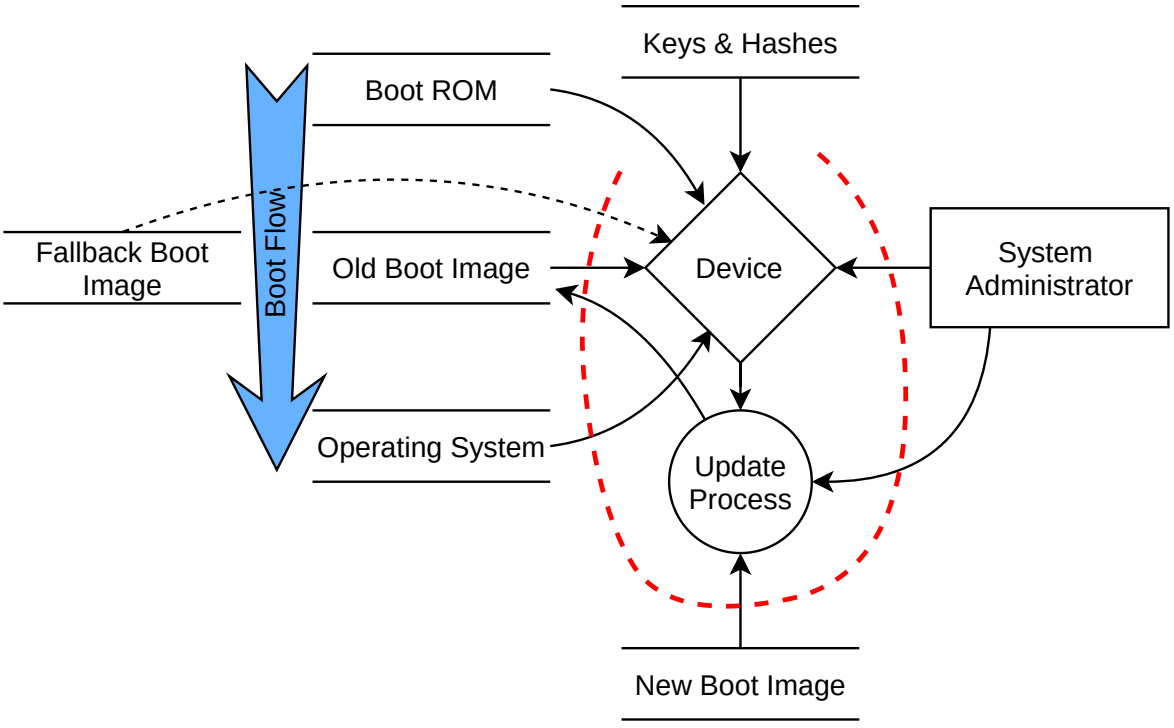


Figure 3.3.: DFD of the firmware update process

The DFD of a firmware update process is relatively simple. As seen in Figure 3.3, the update process has to be started by an external event. In our case, the event is caused by the system administrator, which starts the update process in our operating system. In a second step, the system administrator needs to make the new boot image accessible for the update process through a data connection. The easiest way would be with an SD-card or a USB stick. Then the system administrator starts the update process. In case the update process succeeded the product boots up on the new boot image.

In case the new boot image was corrupted or the update was not successful, the multiboot register increases and starts the fallback image. In this case, the update process has to be repeated.

In Table 3.3, a list of threats and their countermeasure is described. The most dangerous threat appears, if we replace the old boot image with a new corrupted one. This action does not have to be on purpose. Maybe the update process was not successful. In such a case, the device would be unusable, which is why the use of a fallback image is necessary. Section 4.1.1 describes further the procedure of the fallback image.

DFD Element	**Threat Description**	**Countermeasure**	**Rating**	**Designator**
Boot Process	An attacker may invoke the update process, to prevent normal service of the device.	The update process can only be invoked by authorized sources.	moderate	T1
System Administrator	An attacker can use the maintenance access ports, to inflict damage.	Restrict access to maintenance access ports.	moderate	T2
	Access and activity on the device are not traceable.	Log access and activity on the device.	moderate	T3
Boot Image	Loading a corrupted image can rend a device unusable.	Have a fallback image as recovery to boot, in case of a corrupted image.	high	T4

Table 3.3.: Threat table of the example product

3.4. Use-Case 3

The last use-case focuses on the update of hardware components. At Enclustra the Zynq Ultrascale+ is sold on a module, which mounts onto a customizable base-board. Because of that, the user can change or update the base-board while keeping the same chip. Many information about the hardware is stored inside the device tree and is needed to communicate and even to boot up. Therefore, it is necessary to update the hardware information and update the images to match the new base-board. Additionally, a few more things have to be considered, especially with security implemented, to ensure a successful update.

If this use-case is compared to Figure 3.1, the third use-case affects only parts outside of the trust boundary. Notably, external I/0 and external data storage, meaning that if the product was developed according to the previous security considerations, the change of the base-board should not be a security issue. Though if the chip was not secured enough during development, there can be a potential risk for attackers to gain illegitimate access to the device by mounting it on a custom base-board. If for example the JTAG pins were not deactivated, but only disconnected on the base-board, an attacker could access those, to gain elevated privileges on the chip. Other pins that may be delicate are the boot mode pins. These can prevent the device booting from the right boot storage. Nevertheless, this gives an attacker only access if the system does not authenticate the boot images. Therefore, in this section, the risks of bricking a device by updating its hardware are analyzed. How can the base-board be updated, with security implemented?

A potential risk rises by having the AES key inside the BBRAM. When exchanging the base-board, the key gets erased, because the BBRAM battery will be disconnected. On all modules of Enclustra, the battery is located on the base-board. In this case, the BBRAM key has to be programmed again. Everything in the eFuse registers will sustain. Therefore, the update process looks like this, if the AES key is stored inside the BBRAM:

1. Change the base-board.
2. Reprogram the key. If authentication is activated and enforced, the image to program the BBRAM key has to be signed. Optionally, because the AES key is programmed to the BBRAM, the key can also be exchanged with a different key than before.
3. Change the hardware configuration and regenerate the new boot-image with the new key, which has been stored to the BBRAM. Load the encrypted image to the device. Depending on what has changed on the base-board, the fallback image has also to be changed.
4. Boot from the new boot-image.

When the key is stored inside the BBRAM, it is not possible to lock yourself out. AES decryption can only be enforced, when using the eFuse key storage, which is permanent.

4. Analysis Security Features

The Zynq Ultrascale+ SoC combines a multitude of systems on a single chip. It is based on the Virtex Ultrascale+ FPGA from Xilinx. Beside the FPGA part the chip includes:

- a 64-bit quad- or dual-core Arm® Cortex™-A53
- a dual-core Arm® Cortex™-R5
- an Arm® Mali™-400 MP2 GPU
- several DSP blocks

The Arm® Cortex™-A53 features the cryptography extension, further explained in Section 4.2.1 and in Section 5.7. The device has 256KB of internal On Chip Memory (OCM) for secure storage and a variety of security features, which can help in securing the device. Among these are the Platform Management Unit (PMU), to manage power domains and ensure accurate clocking and the system monitoring unit to monitor temperature and voltage to maintain safety during run time. The Configuration Security Unit (CSU) is the heart of the Zynq Ultrascale+ security features. It holds all configuration registers and controls the hardened cores for AES encryption/decryption, key generation as well as hashing. The different methods and cores are further explained in this chapter. The CSU also handles secure key storage and is responsible for initializing the boot flow. The system memory management unit controls memory and peripheral access and works in conjunction with the Trust-zone technology described in Section 4.7.

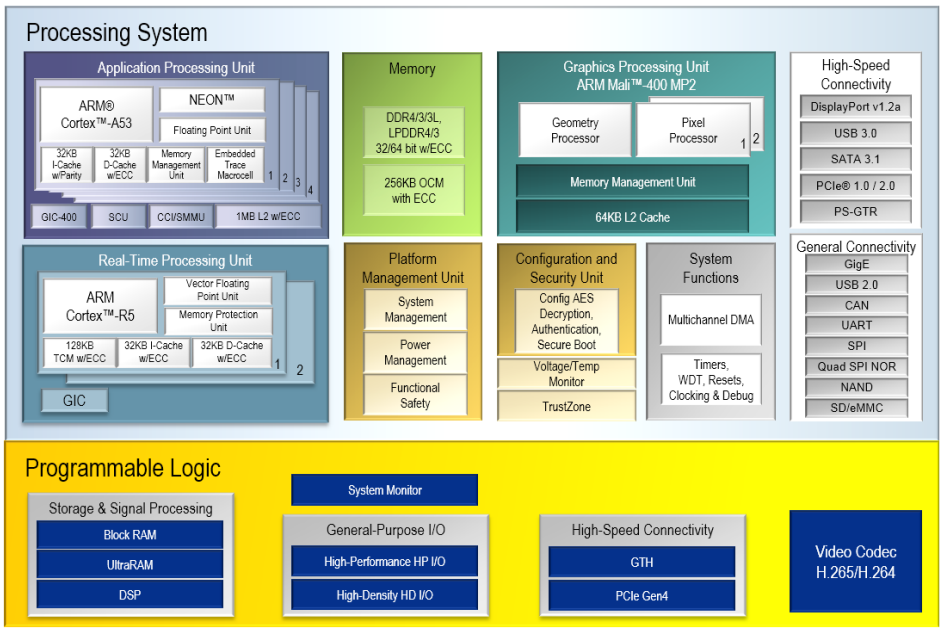


Figure 4.1.: Blockdiagramm of the Zynq Ultrascale+ MPSoC [12]

4.1. Boot Process

The boot-process of the device is divided into four stages. On the left side, figure 4.2 shows, which processor unit is executing the boot code. On the right side the storage can be seen, from which the boot code is executed. Zynq Ultrascale+ can boot from various sources. Those are:

- JTAG: JTAG boot is mainly for development and is only active in non-secure boot mode.
- QSPI: The Zynq Ultrascale+ can boot QSPI in configurations of 24-bit or 32-bit addressing either in single or in dual parallel mode.
- SD: Only FAT 16/32 file systems are supported. (Both, version SD 2.0 and 3.0, are supported. For more information: Technical Reference Manual [2]).
- eMMC: As on the SD cards, only FAT 16/32 file systems are supported.
- USB (2.0): The USB does not support multi-boot. USB is not supported in DDR-less systems.

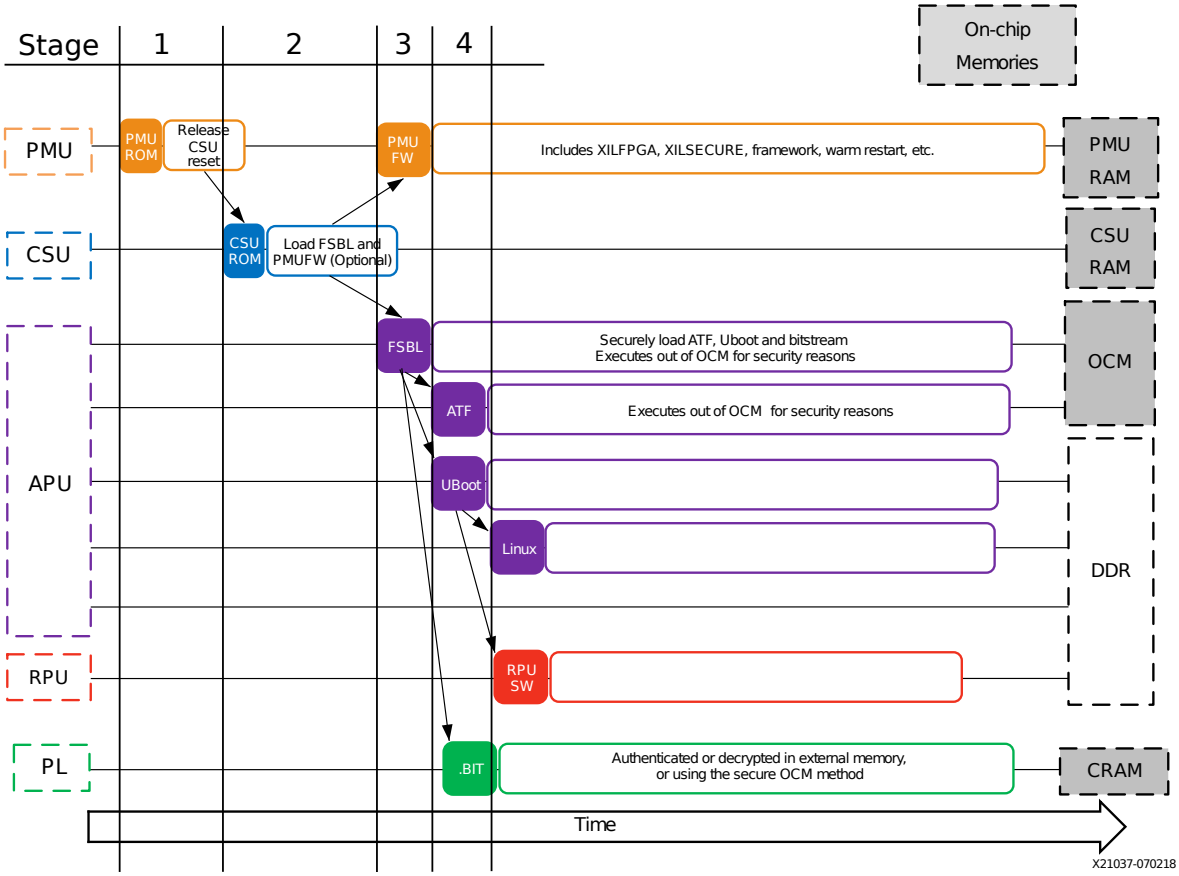


Figure 4.2.: Boot flow of the Zynq Ultrascale+ [2]

Stage 1 - Starting the PMU The device starts by booting the PMU ROM and sets up the system. The PMU is started first because it handles all process related wake-ups and resets. The PMU ROM is fixed and can not be adjusted. For future adjustments, a custom PMUFW is loaded to the PMU after the CSU ROM executed. The PMU ROM hands over to the CSU ROM and enters the server mode, where it monitors power. The CSU ROM configures the system according to boot related settings made in the boot header of the image and the eFuse registers. The device has two boot modes to continue from this state, secure and non-secure. The non-secure boot mode will not be further discussed in this work, as it is basically the same but without encryption and authentication.

"Encryption only" In secure boot mode however, the CSU checks if the image and header are authenticated and/or encrypted. Depending on the secure boot mode either one of them needs to be enforced. The "encryption only" secure mode can be activated by programming `ENC_ONLY` in the eFuse. This mode only relies on confidentiality and symmetric authentication provided by AES-GCM. Therefore, only the key stored in the eFuse can be used to decrypt image partitions.

"Root of Trust" The other secure boot mode "root of trust" enforces authentication. This is the intended way for secure boot, as it gives the liberty to use different key sources for decryption and allows loading the PMUFW from the CSU ROM. To enable it, at least the hash of the Primary Public Key (PPK) and in minimum one of the 15 `RSA_EN` bits in the eFuse have to be set. It is recommended to set all 15 bits to 1 when activating "root of trust". Other advantages above the "encryption only" secure boot mode are the possibility for different key methods in AES. The "encryption only" secure boot mode is bound to the eFuse red key method. "Root of trust" also supports among other features, rolling keys, operational keys and black keys. More about the different features is described in Section 4.2.5.2. Additionally, RSA keys can be revoked. The PPK can be revoked once. The Secondary Public Key (SPK) can be revoked up to 256 times. More about RSA key revocation is described in Section 4.3.3. [2;13]

Stage 2 The CSU ROM authenticates and decrypts the FSBL to the On Chip Memory (OCM). In the "encryption only", it is recommended, that the PMUFW is loaded by the FSBL. Otherwise, if both, the PMUFW and the FSBL, are loaded by the CSU ROM, the CSU ROM uses the same Initialization Vector (IV) and the same key to decrypt both of them. This violates the AES standard. Then the CSU ROM hands over to the FSBL, the first part running from the boot image.

Stage 3 The FSBL authenticates and decrypts the PMUFW, ATF and U-Boot. To do so, the FSBL is executed in Exception Level 3 (EL-3) (secure monitor mode). The PMUFW is loaded to the PMU taking over the monitoring task from the PMU ROM. The ATF is started out of the OCM and starts the secure monitor at EL-3. Then the FSBL hands over to U-Boot. In the "encryption only"mode, any bitstreams loaded in the FSBL have to be encrypted using the eFuse key. In cases, where the bitstream is encrypted using a different key, the bitstream has to be loaded either in U-Boot or Linux, using the XilFPGA library.

Stage 4 U-Boot then can further load and start other software or IPs. As U-Boot is an authenticated image, it can also load unauthenticated and unencrypted software.

4.1.1. Multi-boot

The multi-boot functionality enables the device to select between images to boot from. It can be looked at, as a register in the system, telling where to start looking for valid boot headers. Figure 4.3 shows that in a default case, the images are located one after the other. The multi-boot register starts with an offset of `0x0`. If the first image in figure 4.3 does not boot successfully, the multi-boot register is increased, until the next valid boot-header is reached. The increase of the multi-boot register increases the boot address by `0x800`. In this case, the multi-boot register is increased up to `0x200`, which corresponds to the boot address `0x100000`. Different boot storages have different search sizes. They are defined in table 11–2 on page 236 in the Technical Reference Manual [2].

This functionality prevents the device from becoming unusable, after an unsuccessful update. The fallback image has to be placed in a higher memory section. During boot, the multi-boot register increases until the fallback image is reached. More about secure firmware updates and fallback images is described in Section 5.3 and Section 5.4.

The multi-boot register can also be set manually, to boot from a defined offset. After manually setting the multi-boot register, the device has to be restarted with a soft-reset. A hard-reset also resets the multi-boot register, which results in booting from offset `0x0`.

Figure 4.3.: Structure of images in the storage selected to boot from

4.2. Encryption

The aim of encryption is to maintain confidentiality of the data. An unauthorized person should not be able to read and understand the encrypted data. Only the person or system, which owns the key, should be able to decrypt and read the data. Originally, encryption algorithm only ensured the confidentiality of data and not the integrity, which is the task of an authentication algorithm. But because the missing check for integrity leads to security issues, newer encryption algorithms also implement an integrity check. [14]

4.2.1. Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is a set of different algorithms for symmetric encryption. With a symmetric encryption, the same key is used to encrypt and decrypt the data. The data is split into blocks, which are individually encrypted with the key. The block size of AES is always 128 bit and the key size is either 128, 192 or 256 bit. The different AES algorithms are called modes. The base algorithm to encrypt a block is the same for all modes. The difference between the modes is how the data is split up into blocks and how the blocks are interconnected. [15]

4.2.1.1. AES Vulnerabilities

In general, AES is considered secure. However it can have some weaknesses, depending on the use case and the mode. New AES modes counter those weaknesses.

Brute Force Attack The default way to break an encryption is by trying different keys until the correct one is found. The number of attempts to find the right key is maximal $2^{keysize}$. In case of AES, the required resources are too high to make this attack feasible. But some vulnerabilities of AES break the algorithm, by reducing the number of attempts needed to get the correct key.

Pseudo-randomization The ECB mode in AES (section 4.2.2) encrypts every 128 bit block independent from the others. Identical blocks in plain text data result in identical encrypted blocks. If the data contains a lot of identical blocks, conclusions can be drawn between the encrypted and decrypted data. With a known data structure, like an image with defined headers and partitions, this is even more problematic. Newer modes like CBC and GCM use an algorithm to pseudo-randomize the encrypted data, resulting in unique encrypted data for every block. They are not affected from this problem.

Weak Keys Weak keys are primarily not a vulnerability of the algorithm, but the security of AES relies heavily on the strength of the key. Especially the entropy of the key is critical. Ideally, the key is randomly generated with a good random generator. A simple key that consists of encoded characters is not sufficient. The number of possible keys defines the chance to find the right key. If the number of keys can be limited, due to a bad key generator, the number of attempts is significantly reduced.

Tools like OpenSSL let the user input a password (which consists of encoded characters), that is longer than the AES key size and generates a randomized key out of it. OpenSSL strictly distinguishes between a key and a password. [16]

Padding Oracle Attack The padding oracle attacks concern the CBC mode of AES. In order to get the key, two ciphered blocks are sent to the decryption hardware. The last byte in the first block is changed to `0x01`. The attacker can now examine the returned plain text block. If the padding is correct he can conclude what the last byte of the key needs to be. If the padding is incorrect. The attacker needs to test the next possible solution. The attacker can make a total of 256 attempts to find the correct solution. For a key size of 128 bits this results in a maximum number of attempts of $255 \cdot 16 = 4080$ instead of 2^{128}‘, which is significantly less. Therefore, newer AES modes implement an integrity check in order to counter this vulnerability. They verify, that the data has not been manipulated after encryption. Only if the data is consistent, the hardware will send back a decryption result.

Differential Power Analysis (DPA) Differential Power Analysis (DPA) is a vulnerability of AES implementations in hardware. If a long ciphertext is decrypted, the key can be guessed by analysing the power consumption of the chip with Differential Power Analysis. It’s not a vulnerability of the AES algorithm itself, but because all hardware implementations of AES have this problem, it needs to be considered.

Some manufacturers implement additional steps in their chips, to reduce the amount of information leaked with the power consumption or to detect if the power supply is tampered with. Another solution is to split up the data into smaller sections and encrypt these with individual keys.

4.2.2. AES Electronic Codebook Mode (ECB)

Electronic Codebook (ECB) is the simplest mode for AES. As shown in figure 4.4 every 128 bit block is encrypted independently from each other. This results that the same plain text is also the same encrypted text. Thus, this mode does not provide pseudo randomization of the encrypted data, which makes it easy to identify the structure of the encrypted file. More about pseudo randomisation is described in Section 4.2.1.1. [15;17]

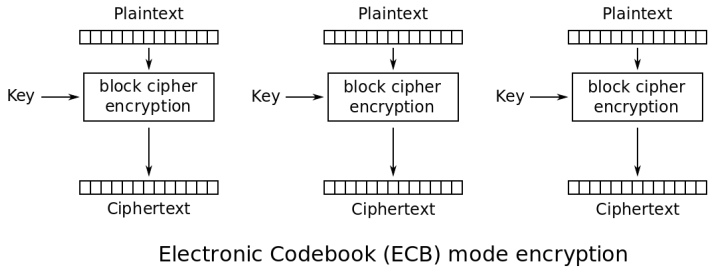


Figure 4.4.: Electronic Codebook (ECB) Encryption Process [18]

4.2.3. AES Cipher Block Chaining (CBC)

Cipher Block Chaining Mode (CBC) is a common mode for AES. In addition to the block cypher encryption of AES, every block is XORed with the proceeding encrypted block, before the encryption. This is shown in figure 4.5. This procedure achieves pseudo randomisation of the encrypted data which makes the algorithm resistant against some attacks. [17]

A disadvantage of this method is, the encryption process can not be done in parallel. A block can only be encrypted if the proceeding block is already encrypted. The decryption process although can be done parallel, as all the encrypted blocks are already available.

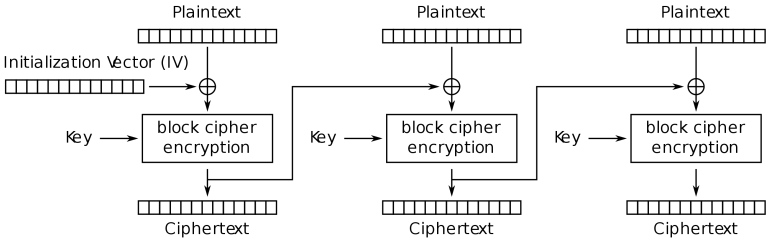


Figure 4.5.: Cipher Block Chaining Mode (CBC) Encryption Process [18]

The first block in the chain is XORed with the Initialization Vector (IV), which is a random 128 Bit block of data. The IV is needed to decrypt the data but does not need any protection. It can be sent in plain text alongside the encrypted data. The AES standard defines, that the probability to use the same IV and the same AES key together should be less than 2^{-32}. Therefore, using the same IV with the same AES key for multiple data violates the standard. The iv is usually randomly generated. [17;19]

Table 4.1 lists the different block sizes for CBC.

Algorithm	**Block Size**	**Key Size**	**IV Size**
AES-CBC-128	128 Bit	128 Bit	128 Bit
AES-CBC-192	128 Bit	192 Bit	128 Bit
AES-CBC-256	128 Bit	256 Bit	128 Bit

Table 4.1.: Cipher Block Chaining Mode (CBC) Block Sizes

4.2.4. AES Galois/Counter Mode (GCM)

Galois/Counter Mode (GCM) is a mode for AES, which implements a Authenticated Encryption with Associated Data (AEAD) cipher. This algorithm checks the encrypted data against the authentication tag during the decryption process. Thus, manipulated data can be detected. The associated data can be added during the encryption process and is only authenticated but not encrypted. It can hold various information about the encrypted data, like version number. Figure 4.6 shows that the same associated data is used to authenticate the encrypted data, as during the encryption process.

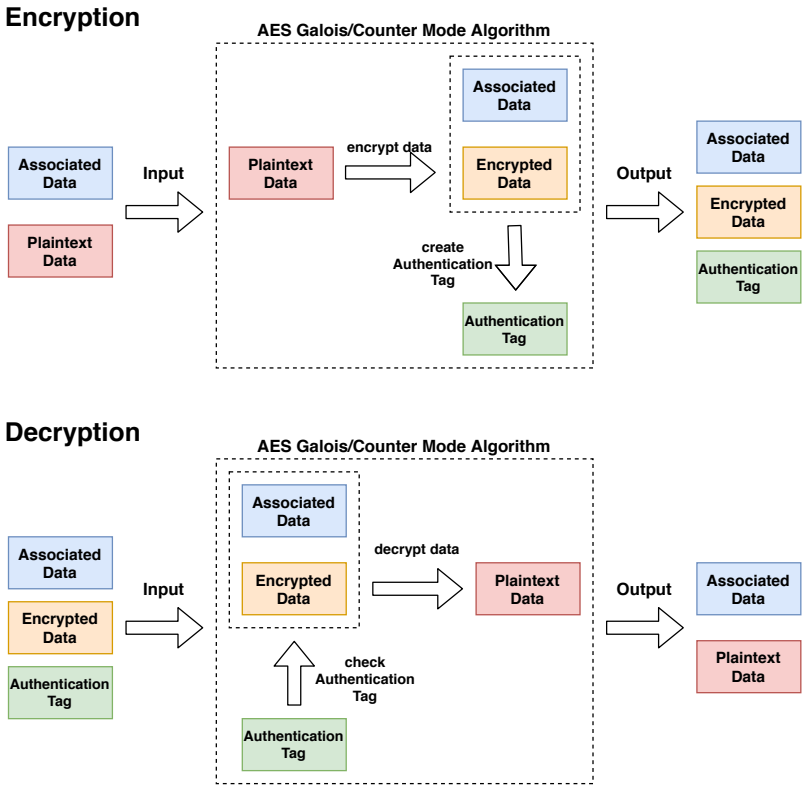


Figure 4.6.: AES GCM encryption/decryption process

A possible use case for this algorithm can look like this: A device needs a software update from the manufacturer. The user of the device should not have the plain text software, but should see which software and version he is using to update the device. When the software is encrypted with GCM, the software name and version is added as associated data. The user is therefore able to check, which software he is loading onto the device. If a someone tries to get the user to load an older software version on to the device, the user is able to see which version he is loading onto the device. Even if the software name and version in the associated data are changed, the authentication of the software fails, because the associated data was changed.

GCM uses a mre complex algorithm than CBC to pseude randomize the encrypted data. The method used for GCM does allow parallelization of the encryption and decryption process, which is an advantage over CBC. The IV used for GCM is 96 Bit long. Using the same IV and AES key is also not allowed in GCM mode. [17;19]

Table 4.2 lists the different block sizes for GCM.

Algorithm	Block Size	Key Size	IV Size
AES-GCM-128	128 Bit	128 Bit	96 Bit
AES-GCM-192	128 Bit	192 Bit	96 Bit
AES-GCM-256	128 Bit	256 Bit	96 Bit

Table 4.2.: Galois/Counter Mode (GCM) Block Sizes

4.2.5. Encryption Hardware Zynq Ultrascale+

The Zynq Ultrascale+ has built in hardware support for AES encryption. The Arm® Cryptography Extension of the Arm® Cortex™-A53 contains an instruction set for different AES modes. This extension consist of different hardware to accelerate the AES algorithm, but do not implement any key handling or possibility to boot from an encrypted image. In addition, Xilinx integrated full hardware support for AES Galois/Counter Mode (GCM) with a key size of 256 bit. The encryption hardware from Xilinx can either work with a user provided key or with a key from one of the key sources described in Section 4.2.5.2. Furthermore, this hardware enables the Zynq Ultrascale+ to boot encrypted images, as described in Section 4.1. [2]

4.2.5.1. Zynq Ultrascale+ AES GCM Hardware

The AES crypto engine is controlled by a triple redundant microblaze processor in the CSU. It can be configured by the `CSU Module registers`. With a bare metal application, the GCM crypto hardware can be used with the xilsecure [20] library provided by Xilinx. In a system running Linux, the GCM crypto hardware can be used by user space applications via the Linux Crypto API as described in Section 4.6. [2]

4.2.5.2. AES Key Sources

Table 4.3 lists the different AES key sources. During boot, the Zynq Ultrascale+ accepts keys from three sources. The BBRAM, eFuse or a key from either a rolling key method or an operational key method, which is stored inside the decrypted data. These keys can be used in plain text, encrypted with the key from the PUF or eFuse or obfuscated with the family key. The possible combinations are shown in table 4.3. After the system is booted, a software using the AES hardware can also provide a user key through the key update register.

None of the keys can be read by the software in any way. The key handling is made in such a way, that even if the software on the device is compromised, the keys can not be retracted from the device. The device can only be used to encrypt or decrypt data.

During the boot process, the boot header in the external storage is read. This header determines, which key is used as device key and the method from table 4.3, although some flags in the eFuse can restrict this selection. The key then is loaded from storage to the crypto hardware and decrypted if needed. All key handling actions are only performed by the CSU. The device key is only set once during boot and persistent until the next reboot. Excluded is the possibility to provide a user key in the key update register. For every encryption process can be selected, if the device key or a user provided key is used. [2]

AES Key Source	Description
BBRAM	Plain text key stored in Battery Backup RAM (BBRAM). This key can be reprogrammed and is lost in case the battery power is removed.
eFuse	The key in the eFuse is either stored plain text, encrypted with the PUF key (black key) or obfuscated with the family key. The eFuse key can only be programmed once.
`AES_KUP` register	The key update register serves as an interface to hand over a user provided key. A user provided key can only be used during run time and not during boot up.
Operational Key	This key is stored in the encrypted boot header on the external storage. Therefore, this key is either decrypted with the key in the eFuse or the BBRAM key.
Rolling Key	The rolling key technique is used to prevent key exposure with Differential Power Analysis (DPA). In cases where partitions of the image reach a certain length, DPA can successfully discover the key. This mostly targets bitstreams. In such cases, the images can be split into small block sizes, which are encrypted using their key. The key is always stored inside the previous block. [21]
Family Key	The family key can only be used to obfuscate the other keys. See Section 4.2.5.4.
PUF	The PUF key can only be used to encrypt the other keys. See Section 4.2.5.3.

Table 4.3.: AES Key Sources (Ultrascale+ Technical Reference Manual [2] page 262)

Table 4.5 shows the different possibilities to boot the Zynq Ultrascale+ from an encrypted image. Xilinx uses the terminology “red keys” for keys that are stored in plain text, “grey keys” for a key that is encrypted with the family key and “black key” for an encrypted key.

4.2.5.3. Physical Unclonable Function (PUF)

The Physical Unclonable Function (PUF) is a hardware generated key, with an individual, device dependent output. Therefore, it is used as an identifier or a fingerprint for a specific device. As the

Key Source	Key Storage State	
BBRAM	**RED**	plain text
eFuse	**RED**	plain text
	GREY	obfuscated with family key
	BLACK	encrypted with PUF key
Operational Key	**BLACK**	encrypted with BBRAM key
	BLACK	encrypted with eFuse key

Table 4.5.: Possible AES key sources available during boot [2]

name suggests, it is impossible to clone or reproduce the function and thus the key, as its generation depends on so many factors. To use the PUF as a key to encrypt data, gives the possibility, to only decrypt data on a specific device.

The PUF inside the Zynq Ultrascale+ is controlled by the CSU. The CSU provides the PUF as a service. Otherwise, the PUF can not be accessed or read and is only used for one purpose. A so called Key Encryption Key (KEK) and helper data can be generated with the PUF to encrypt a user provided AES key. Both the encrypted user key and the helper data are then stored either in the eFuse or in the boot image. On boot, the PUF regenerates the KEK using the helper data and decrypts the key.

Although all Zynq Ultrascale+ devices are equipped with a PUF. The PUF is not meant to be used in the field. The PUF in regular devices have not enough entropy to operate reliably in the field. To do so, devices, with tested PUFs have to be bought.

4.2.5.4. Family Key

The family key is a hard-coded key, which is the same in all Zynq Ultrascale+ devices. It can be used to encrypt a user provided AES key, which is then stored in the eFuse or in the boot image. Because the key is the same across the whole device family, the term obfuscated instead of encrypted is used. The family key can only be accessed from the CSU and is used to decrypt the obfuscated AES key. To get access to the family key, Xilinx has to be contacted. The key is subject to strict confidentiality.

4.2.5.5. Rolling Key Method

The bitstream and the software images of the Zynq Ultrascale+ are large enough to successfully extract the key with a DPA attack, as described in section 4.2.1.1. As a countermeasure, Xilinx implemented the rolling key method. An image is split up in smaller blocks, each encrypted with a separate key. As every key is only used for small blocks, it can not be guessed with a DPA attack. The key for one block is stored in the previous block, and only the first section is encrypted with the key from the eFuse or BBRAM. [21]

4.3. Authentication

Authentication is used for integrity and authenticity. While integrity shows, that data has not been altered, authenticity guarantees the source of the data. This is achieved by giving a signature aside of the data, defining a method to calculate it and a key as an identifier. The data is verified by comparing the given signature with the signature calculated using the given method with the data and the key. Because the data has an influence on the signature, a change of the data changes the calculated signature and thus, the authentication fails. Hashing functions are commonly used for authentication methods. [14]

4.3.1. Rivest Shamir Adelman (RSA) Method

RSA is an asymmetric-key cryptography method, which can be used for both encryption and authentication. But because the Zynq Ultrascale+ uses RSA only for authentication, encryption will be neglected. The main advantage over symmetric-key cryptography methods is that the key to verify the data is not the same as the one to sign the data. This results in having a pair of keys, a private confidential key, to sign data and a public non-confidential one to verify the data. This eases the handling of the key, as the private keys do not have to be delivered with the data. Because of the asymmetric keys, RSA relies heavily on one-way-functions. These are functions, which are easy to calculate in one way but very hard to calculate the other way around. It is easy to calculate $2 \cdot 5 \cdot 7 \cdot 9 =?$ by multiplying the numbers, but it is much harder to find out which prime numbers multiplied give 630. The algorithm depends on prime factorization. Because computers get faster, the key length has to be adjusted. Until 2023 RSA key sizes of at least 2000 bits are considered safe. After 2023 a key length of at least 3000 bits has to be chosen [22]. [14]

4.3.2. Zynq Ultrascale+ RSA

The Zynq Ultrascale+ is future proof according to the key length specifications and uses RSA with a key length of 4096 bits. The hardware acceleration consists of a modular exponentiation engine, support for $R \cdot R \bmod M$ pre-calculations and implementation for efficient processing of short public exponents. The RSA functionality is fully controlled by the CSU. RSA is supported during startup to authenticate boot images. It can be used in bare-metal applications through the xilsecure [20] library and in Linux through the Linux Crypto API described in Section 4.6. Because RSA is an asymmetric cryptography method and to provide flexibility and minimize the use of keys, Xilinx introduces a verification method with two keys. Both keys and signatures are stored inside the image. To prevent someone from exchanging both, the keys and the signatures, the primary key is verified by a hash stored inside the eFuse register. This key is then used to verify all following keys. Thus, the secondary key can be exchangeable for different data or image partitions. The secondary key then verifies the images or data. For further flexibility, Xilinx added a key revocation feature, which allows to invalidate keys. Key revocation is further described in Section 4.3.3. [2;14]

4.3.3. Key revocation

Key revocation can be used if the private keys get compromised. In such a case, the key has to be exchanged and invalidated, to prevent usage of the old key. Thus, the key hash in the eFuse register has to be invalidated. The Zynq Ultrascale+ has the possibility to revoke the primary key once. To provide more flexibility, the system introduces a secondary key. It gives the user the possibility to use different keys for different image partitions and the possibility to revoke the secondary key more than once.

To revoke the PPK, the hash of the new key has to be written to the eFuse. Additionally, the first key can be invalidated by setting the invalidate bit in the eFuse for the according key. The standard secondary key revocation is done with the `SPK_ID` eFuse registers. With this 32 bit register, up to

32 keys can be revoked. As shown in figure 4.7 the `SPK_ID` eFuse registers are compared bit-wise, for a bit to bit match. To get the maximum amount of revocations, the bits have to be flipped to ones separately (eFuse registers can only be burnt from a zero to a one). This leads to the following `SPK_IDs`: `0x0, 0x1, 0x3, 0x7, 0xf, 0x1f, 0x3f, 0x7f, 0xff, 0x1f, ...` If the user has to revoke more keys, the `USER_FUSE` registers can be used. They are also located in the eFuse registers. These seven `USER_FUSE` registers with 32 bits each, can revoke up to 256 keys. The `USER_FUSE` ID is not checked the same way, like the ones stored in the `SPK_ID` register. The `USER_FUSE` registers represent the IDs from zero to 256, as shown in figure 4.7. If the bit in the `USER_FUSE` register is set to one, then the key is revoked. For example, the key `0x56` (which is 86) is revoked, if the bit 86 is set to one. This means `USER_FUSE 2` bit 22 has to be set to 1. In order to select the `USER_FUSE` for ID validation, the `*.bif` file has to be adapted. The attribute `spk_select` has to be changed to `user-efuse`.[23]

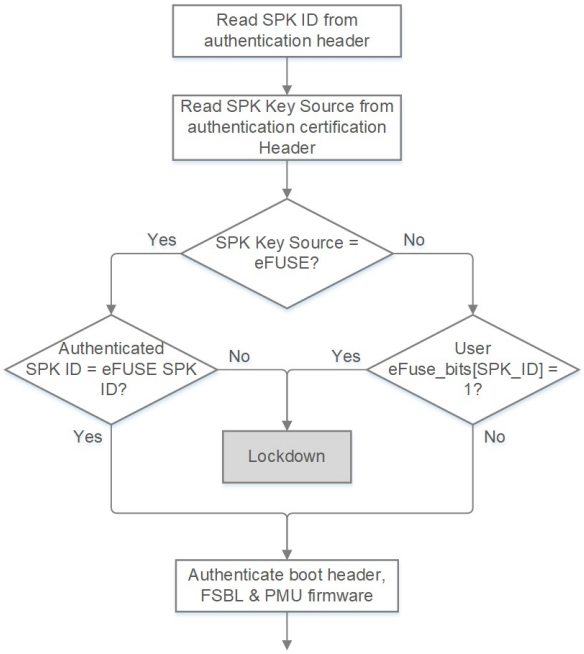


Figure 4.7.: Process flow of SPK_ID check [23]

Compared to the primary key revocation the revocation feature for the secondary key does not invalidate the secondary key, it does only check for the `SPK_ID` to be valid. Thus it only prevents the use of an old image with revoked IDs. Verification of the SPK is done by the primary key with the signature also stored alongside the keys.

4.4. Hash Algorithms

A hash algorithm creates an identifier of the given data. A good hashing algorithm is characterized by the fact that similar (but different) input data generates hash values, which is extensively different. It should not be possible to correlate the change of the data, to the change of the hash. Additionally, it should not be feasible to generate two different data blocks with the same hash value or to generate data with a specific hash value. Hash algorithms are used in many places to check if data has been modified during transfer or storage.

4.4.1. Secure Hashing Algorithm 2 (SHA-2)

SHA-2 is the successor of SHA-1 and consists of six hash functions (SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256). SHA-1 is no longer considered secure, as it is theoretically

possible to find the original data of a given hash or to produce a message which generates a specific hash. Therefore SHA-1 should not be used in critical applications. SHA-2 is considered secure against these types of attacks.

One vulnerability of SHA-2 is called Length extension attack. The SHA-2 output hash is the current state of the hash algorithm when all the input data has been processed. With the hash of an unknown data block, it is possible to reconstruct the internal state of the SHA-2 algorithm and continue adding new data to the hash. Therefore an unknown data block can be extended with new data, and a valid hash over the whole data block can be calculated, if the SHA-2 hash of the unknown data block was known. Depending on the use case, this vulnerability might or might not be a problem. The solution against this problem is, using a hash algorithm with a safe finalization function and therefore prohibiting the reconstruction of the internal state.

An overview of the different hash functions is given in table 4.7.

4.4.2. Secure Hashing Algorithm 3 (SHA-3)

The standard for hash algorithms is extended with SHA-3. Although SHA-2 is considered secure for most use cases, SHA-3 has two advantages over SHA-2. The algorithm for SHA-3 is improved for hardware implementations and therefore can achieve better performance.

If possible, new developments should use SHA-3, but there is no need to update applications from SHA-2 to SHA-3 at this time. The main advantage of SHA-3 over SHA-2 is better performance when done in hardware. The second advantage is the safe finalization function in SHA-3, which prohibits the reconstruction of the internal state. SHA-3 is not vulnerable against length extension attacks.

An overview of the different hash functions is given in table 4.7.

Algorithm		Output Size
SHA-1		160 Bit (20 Bytes)
SHA-2	SHA-224	224 Bit
	SHA-256	256 Bit
	SHA-384	384 Bit
	SHA-512	512 Bit
	SHA-512/224	224 Bit
	SHA-512/256	256 Bit
SHA-3	SHA3-224	224 Bit
	SHA3-256	256 Bit
	SHA3-384	384 Bit
	SHA3-512	512 Bit

Table 4.7.: Overview SHA functions

4.4.3. Hashing Hardware Zynq Ultrascale+

The Zynq Ultrascale+ has built in hardware support for hash algorithms. The Arm® Cryptography Extension of the Arm® Cortex™-A53 contains an instruction set for SHA-2. In addition the Zynq Ultrascale+ has hardware support for SHA-3-384. An example how these hardware accelerators can be used under Linux is given in section 5.7.

4.5. Tamper Monitoring Unit

The tamper monitoring unit is coupled with the system monitoring unit. It catches its alarms, to act according to the settings made in the tamper monitor register. According to figure 4.8 the tamper monitoring unit is active after the FSBL and the PMU firmware is loaded. Thus, hardware attacks during the period before are not recognized. This contains the CRC check of the PPK, the verification of the SPK and finally the authentication and decryption of the boot-header, the FSBL and the PMU firmware. Because these programs are very small, they are not a huge risk for key leakage.

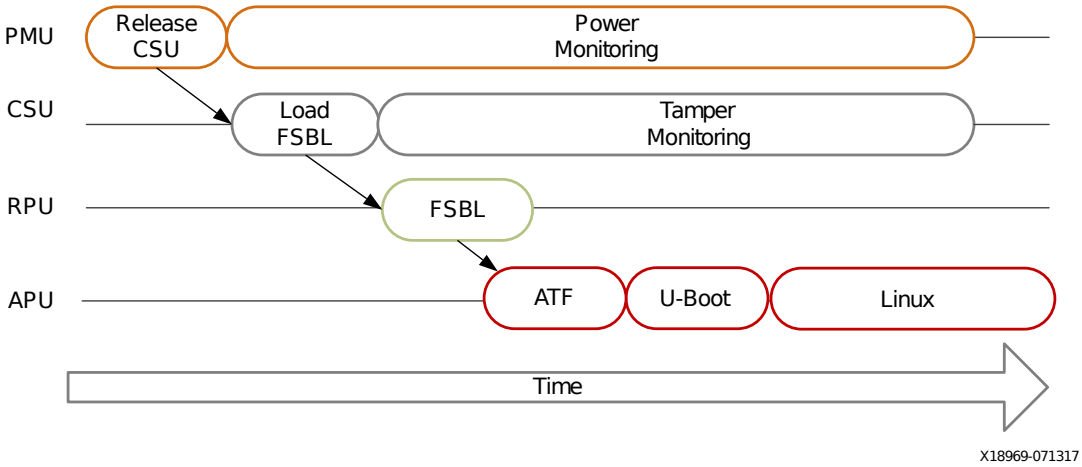


Figure 4.8.: Boot flow with monitoring units [13]

4.6. Linux Crypto API

To improve the performance of cryptography algorithms, processors have hardware support to accelerate cryptography functions. A user-space software can calculate cryptography functions in software at any time, but only the kernel has access to the crypto hardware. The Linux kernel provides the Linux Crypto API to user-space programs, to make the crypto hardware available.

Figure 4.9 shows, the Linux Crypto API is a layer between the applications requiring the crypto hardware and the drivers for the crypto hardware. This has the advantage that several applications can use the Linux Crypto API, and it provides a unified interface for hardware drivers and applications. A unified interface simplifies development and helps to keep applications portable between different platforms. [24]

The Linux Crypto API can also be used for cryptography algorithms that are calculated in software. However, it is not recommended, because the data has to travel through the interface of the Crypto API, which adds overhead. A software algorithm can be done directly in the user-space. [24]

Code examples for the Linux Crypto API can be in the Git repository 1.1. A description on how to use these examples is in section 5.7.

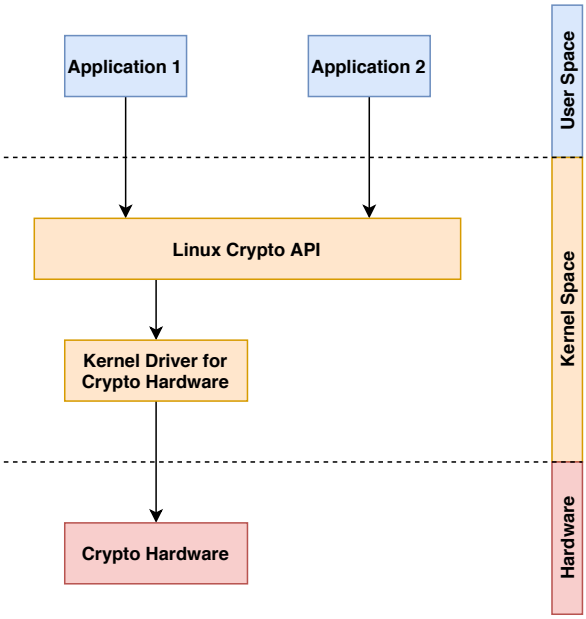


Figure 4.9.: Overview Crypto API

4.6.1. Xilinx Zynq Ultrascale+ Crypto Hardware

The Zynq Ultrascale+ contains hardware support for different hash and cipher algorithms. In addition to the Arm® Cryptography Extension of the Arm® Cortex™-A53, Xilinx implemented additional crypto hardware support. [25] [26]

The Arm® Cryptography Extension for the Arm® Cortex™-A53 contains the following features. [26]

- SHA-1, SHA-224 and SHA-256
- AES encryption and decryption
- Elliptic curve

In addition to the cryptography features of the Arm® Cortex™-A53, the Zynq Ultrascale+ has the following features implemented. [25]

- SHA3-384
- AES-GCM-256
- RSA-384

4.6.2. Interface Description

The Linux Crypto API uses sockets and system calls to communicate between the user-space and kernel space. An application using the Linux Crypto API has to select which algorithm should be used and configure the options, like the key size. After the algorithm is configured, the application can send the data to the Linux Crypto API and read the processed data back. [24]

A description on how to implement an application which uses the Linux Crypto API can be found in 5.7.

4.7. TrustZone

TrustZone is a concept to isolate different parts of a system, which is required if critical applications are executed. Linux implements an isolation between users and applications with different user permissions. Therefore different applications can not interfere with each other. However, with the complexity of the Linux kernel, the attack surface is huge. This makes it difficult to protect critical data if untrusted applications are running alongside trusted applications.

TrustZone distinguishes between the secure world and the non-secure world, as shown in Figure 4.10. Most parts of the system, including Linux, are running in the non-secure world and only the critical parts of an application are executed in the secure world. Usually, if an attacker compromises Linux and gains root access to the system, the attacker has full access to the whole system memory. With TrustZone, Linux has only access to the non-secure world, and the applications in the secure world are safe. [27]

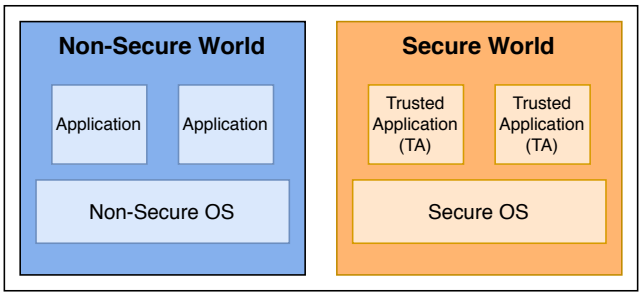


Figure 4.10.: Isolation between secure and non-secure world.

The concecpt for TrustZone is from Arm® and describes the security features implemented in the Arm® Cortex™-A53 architecture. In addition to the concept from Arm®, Xilinx implemented hardware in the Zynq Ultrascale+ to extend the features of TrustZone. [25]

4.7.1. TrustZone Concept from ARM

Arm® implements TrustZone in different processor series. This section describes the TrustZone features of the Cortex A series, especially the Arm® Cortex™-A53 which is part of the Zynq Ultrascale+. [25]

To isolate a critical application from other parts of the system, Arm® implements several hardware features in the Arm® Cortex™-A53 architecture. The processor has two security states, secure and non-secure. In the non-secure world, the processor has only access to the memory regions and hardware which are marked as non-secure. To access secure memory or hardware, the processor has to be in the secure state. In the secure state, the processor has access to both, the secure and non-secure marked hardware and memory sections. [27]

In the non-secure world runs a Rich Execution Environment (REE), an OS with a wide variety of features. A Trusted Execution Environment (TEE) runs in the secure world and contains only a minimal set of features to minimize the attack surface. The OS in the REE is typically Linux, but can be any other OS. The TEE runs usually a Secure OS, which is optimized to run in the secure world.

The central part of TrustZone is the hardware implementation to isolate the secure and non-secure world. Arm® provides software to use the TrustZone features, but it is possible to run custom software, which offers a different set of features. [27]

4.7.1.1. Execution Levels

To control the secure and non-secure world, the processor needs a control instance. The architecture already has different privileged levels called execution levels. The Arm® Cortex™-A53 has 4 different execution levels, ranging from EL-0 to EL-3. An application running in EL-0 has the lowest privilege on the system. An Operating System (OS) runs in EL-1 and is able to control and manage the different applications. The next level EL-2, is for a hypervisor, which has the rights to control multiple OSs. The highest level EL-3 is for a Secure Monitor. The Secure Monitor manages security features like TrustZone and has the highest access to the system. Figure 4.11 gives an overview of the different execution levels. The Secure Monitor in EL-3 runs always in the secure world. Only the non-secure world supports a hypervisor. The execution levels EL-1 and EL-0 can either run in the secure world or non-secure world. [27]

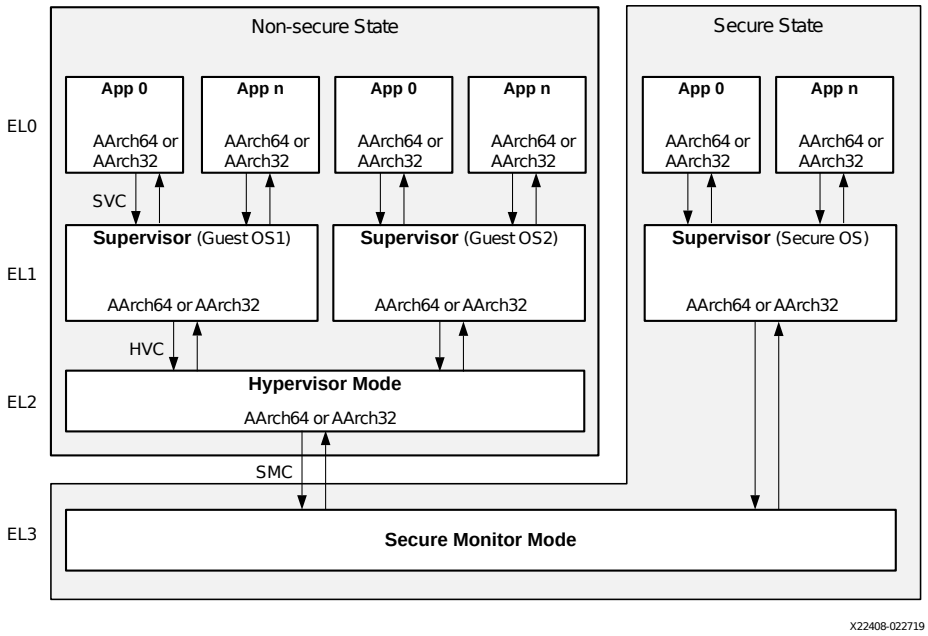


Figure 4.11.: ARM execution levels with Trustzone [27]

To communicate between the execution levels, different types of systemcalls can be used. An application can use the Supervisor Call (SVC) to request something from the OS. If a hypervisor is used, the OS can use the Hypervisor Call (HVC) to communicate with the hypervisor. The OS or the hypervisor can use the Secure Monitor Call (SMC) to communicate with the Secure Monitor. An application running in EL-0 can not issue an SMC. [27]

4.7.1.2. Communication between secure and non-secure world

The secure world is isolated from the non-secure world. Communication between applications running in different worlds needs to go through defined paths. The TrustZone concept from Arm® has two communication methods. The SMC and shared memory. The OS or hypervisor in the non-secure world can issue an SMC to request something from the Secure Monitor. An application in the secure world is able to access memory from the secure and non-secure world. Therefore it is possible to define a section in memory, where an application in the secure and non-secure world have access to. Via this shared memory it is possible to exchange data between the secure and non-secure world. [27]

4.7.1.3. Secure Monitor (ARM Trusted Firmware)

The software running as the Secure Monitor has to initialize the security features during boot. After the system is booted the non-secure part of the system can issue an SMC to access security functions. The Secure Monitor can either directly handle the request or forward it to an application running in the secure world.

The Arm® Trusted Firmware (ATF) is a Secure Monitor implementation from Arm®. It implements a number of standard SMC and can be expanded with custom SMCs. With the Secure Payload Dispatcher it is possible to load an OS in the secure world running in EL-1. The ATF implements the GlobalPlatform TEE API to communicate with a Secure OS. [28]

4.7.1.4. Secure OS (OP-TEE)

The OS running in the secure world is called Secure OS and the applications are called Trusted Applications (TA). Any OS could be used as Secure OS. However an OS, which is meant to run in the secure world has a number of advantages. The Secure OS should be kept simple, to minimize the attack surface. Further it has to be loaded by the Secure Payload Dispatcher from the ATF and has to know the GlobalPlatform TEE API to communicate with the ATF. [29]

OP-TEE is an OS, which is optimized to run in the secure world. It can be loaded with the Secure Payload Dispatcher and it supports the GlobalPlatform TEE API. The OP-TEE client is an application running in the non-secure world, it provides the GlobalPlatform TEE API to other applications running in the non-secure world. This provides a simple communication path between an application in the non-secure world and a TA, as shown in figure 4.12. A TA can offer a service via the GlobalPlatform TEE API and the applications in the non-secure world can access this service. The communication between the OP-TEE OS and client works through SMCs. [29]

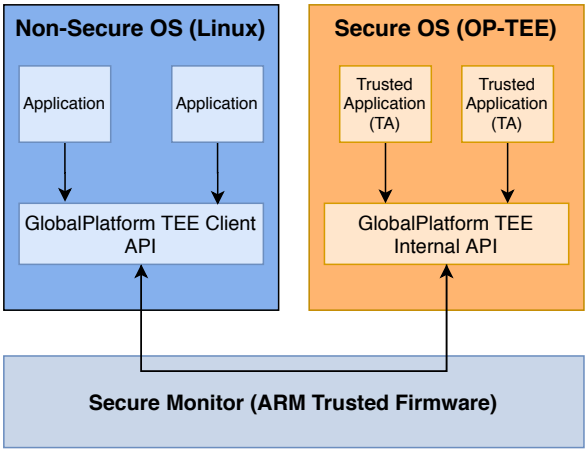


Figure 4.12.: GlobalPlatform TEE API to communicate between secure and non-secure world.

The TAs can be loaded automatically on startup or an application in the non-secure world can start a TA via the OP-TEE client. If the TA is loaded from the non-secure world, the TA can be encrypted, but has to be authenticated, to prohibit loading of a malicious TA. [29]

4.7.2. Additional TrustZone Elements from Xilinx

TrustZone is a concept to isolate applications on the Arm® Cortex™-A53 processor. But the Arm® Cortex™-A53 processor is only a part of the Zynq Ultrascale+, which contains much more, as shown in Figure 4.13. Xilinx added other parts to the Zynq Ultrascale+ to make and extend the features

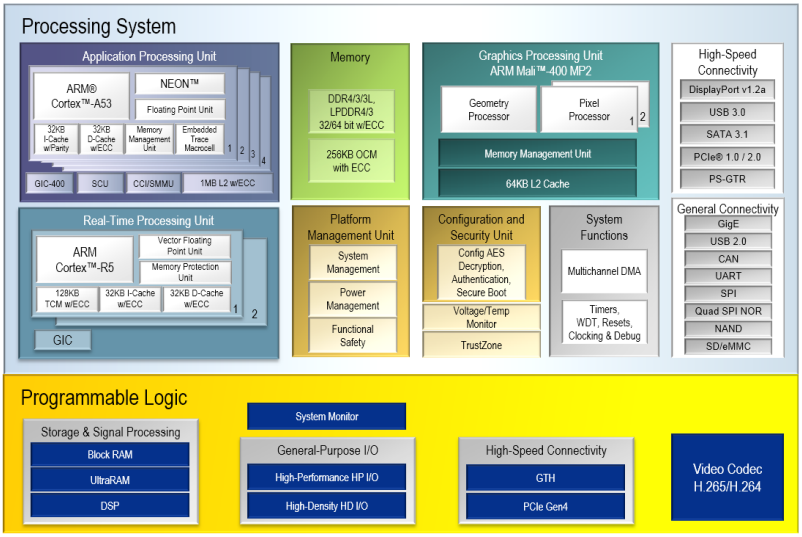


Figure 4.13.: Overview Zynq Ultrascale+ [2]

of TrustZone. The whole memory and all the peripheral hardware can be marked as either secure or non-secure. [27]

Apart from the Arm® Cortex™-A53, the Zynq Ultrascale+ contains also a Arm® Cortex™-R5. The Arm® Cortex™-A53 can run in both the secure and non-secure world. The Arm® Cortex™-R5 can be marked as either secure or non-secure. This makes a concept possible, where the Arm® Cortex™-A53 executes the non-secure tasks of an application, and the secure tasks are executed on the Arm® Cortex™-R5.

To communicate between the different processors, Xilinx implemented the Inter-Processor Interrupts and Communication (IPI). With the IPI one processor can issue an interrupt on another processor and communicate with 32 byte request and response buffers. [27]

4.8. Connecting Security Features to Security Requirements

Now that all the features are analyzed, this section shows which features are linked to which security issues. While in Chapter 3 the different issues are analyzed and grouped in different use-cases, the linking is done unstructured, because security features can eliminate multiple threads. Therefore, a security feature can be used in different use-cases. It is important to understand, that this list covers only security requirements emerging from detected threads in the security analysis of the use-cases. Additionally, Xilinx provides a lot of features to ease the usability of these features or provide certain features in addition to increase maintainability, but are not necessary.

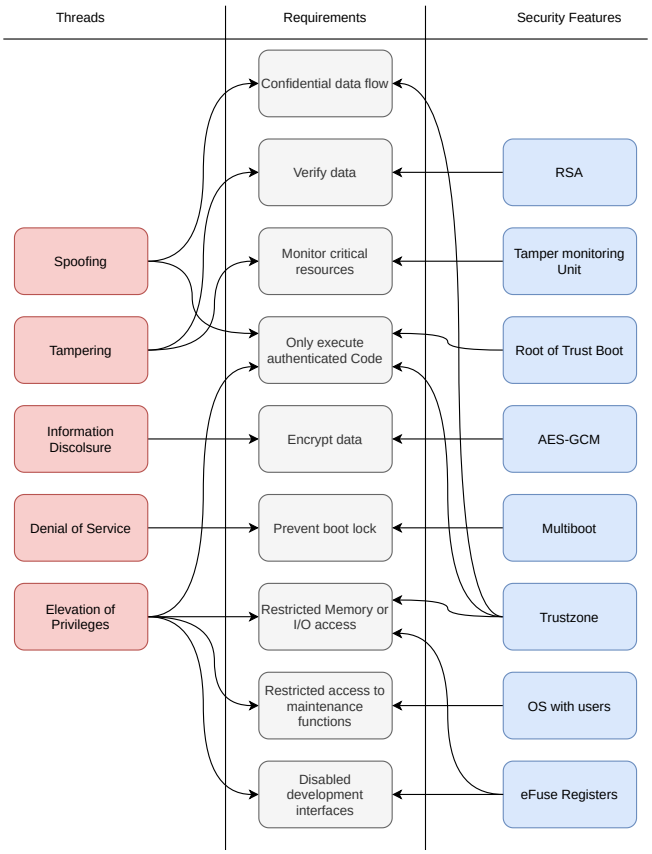


Figure 4.14.: Linking Threads, Requirements and security Features

5. Implementation

As Chapter 4 showed, the Zynq Ultrascale+ supports a range of features to prevent security incidents. This chapter shows how these features are implemented. In the first use-case, secure boot is the main component. Therefore, it is shown how to implement root of trust to authenticate images. Compared to an insecure device, secure build upon the concept that in each step, the previous bootloader authenticates the next software. This principle is called root of trust. An interrupt in this chain means that the software running is not trustworthy. Root of trust does not make a device all secure. If confidentiality has to be guaranteed, information stored on easily accessible places has to be encrypted. Thus, in this implementation, the boot image is also encrypted.

Data handled in Linux can be encrypted or decrypted using the Linux Crypto API. A framework to handle all security-related actions. Besides, the cryptography extensions from Arm® Xilinx provides also support for their hardware security features.

The second use-case covers the life-cycle and update process of a device. With the example of QSPI as the primary storage to boot from it is showed, how to implement an update procedure and the multiboot functionality in case the regular image fails to boot. With multiboot, the device can boot a second image stored in the memory. Additionally, the procedure of key revocation is explained. Key revocation applies only to RSA keys.

As discussed in Chapter 3, the hardware can be easily updated. Therefore, the implementation of the tamper monitoring unit will be shown. Together with the system monitoring unit, system variables can be tracked and to detect tamper events.

At last, it is showed how to implement TrustZone. Thus, it is described how get the Arm® Trusted Firmware (ATF) and OP-TEE running and how to load a Trusted Application (TA). Due to unresolved errors, OP-TEE could only be implemented on the evaluation board zcu102 from Xilinx.

5.1. Tools and Environment

This section describes the tools and the environment used for the implementation. All features were tested on a Mercury XU5 (ME-XU5-5EV-2I-D12E) module mounted on a Mercury PE1 board (ME-PE1-300-W). To generate the Bitstream and configure the hardware, Vivado 2019.2 was used. The official template from Enclustra was used, which can be downloaded from their website [30]. Further configuration and generating the boot images were made with PetaLinux 2019.2. The applications were manually compiled, although they can be directly integrated into the PetaLinux process. Nevertheless, because it is often very time consuming, the gnu-aarch64-none-linux-gcc compiler (9.2-2019.12) was used. The operating system of the host computer was based on Ubuntu 18.04.

5.1.1. PetaLinux or Yocto?

PetaLinux is the tool chain from Xilinx to create a Linux image for the Zynq Ultrascale+. It is built upon Yocto, which is an open-source project to create Linux distributions for different platforms. Instead of using PetaLinux, it is possible to use Yocto for development on the Zynq Ultrascale+.

The advantage of Yocto is, it works for many platforms and therefore, the same tools and workflow can be used for different platforms. Because Yocto is more often used, it has a better documentation, more examples and more guides available, than for PetaLinux.

However, there is also a disadvantage. Although Xilinx provides guides to work with Yocto, it is not officially supported by Xilinx. If any problem occurs, even if it is related to Yocto, Xilinx will not provide any support for it. Xilinx maintains a forum for PetaLinux, where most questions are

answered. Further, guides and examples for Yocto are not as good maintained as for PetaLinux and are sometimes out of date.

5.1.2. Getting started

This section describes how to create the first project for a Mercury Board from Enclustra. A description of how to install and use Petalinux can be found in the following user manuals and application notes from Xilinx. The following Guides focuses on the specialities of the Mercury Board.

- PetaLinux Tools Documentation Reference Guide (UG1144) [25]
- PetaLinux Tools Documentation Command Line Reference Guide (UG1157) [31]

Create Vivado project If further adjustments to the hardware have to be made, it makes sense to create a Vivado project as well. The user guide to the reference design from Enclustra [30] describes how to create a Vivado project.

- Mercury XU5 SoC Module Reference Design for Mercury+ PE1 Base Board User Manual [32]

Export Hardware After synthesizing the Vivado project, the hardware information and the bitstream have to be exported. To export the hardware access

`File->Export->Export Hardware...`

in the menu bar. It is essential to check `Include bitstream` in the options window, in order to use the export in PetaLinux.

Create PetaLinux Project In this case, none of the template board support packages from Xilinx will fit the chip, and an exported hardware will be later imported to the project. Thus, create a project with the `zynqMP` template.

```
petalinux-create --type project --template zynqMP --name <project-name>
```

Import hardware to the PetaLinux project This step is always necessary, after any changes in hardware. The new bitstream has to be exported and imported to PetaLinux. During this work, no issues were detected with the import. Though, since PetaLinux 2019.2 Xilinx introduced a new file format for hardware description files. The new files have the ending `.xsa`, instead of the older `.hdf`. However, older files can still be imported. If in previous steps no hardware has been exported, the exported hardware in the reference design from Enclustra [30] can be imported. For the board used in this work the `.hdf` file can be found in this path:

`Mercury_XU5_Reference_Design_for_Mercury_PE1_V6.2/SdkExport/system_top_5ev_2i.hdf`

It is essential only to have one exported hardware file in the folder, else PetaLinux will complain. Import the hardware description file to PetaLinux with the following command.

```
petalinux-config --get-hw-description=<PATH-TO-FOLDER-CONTAINING-HDF-OR-XSA>
```

Configure PetaLinux Project After importing the hardware description, the first configuring window appears. It can be exited by double-pressing the `ESC` key. The different components are configured separately in PetaLinux. There are two ways to configure components. If available, a `menuconfig` can be invoked, as seen after the hardware import, by typing:

```
petalinux-config -c <COMPONENT-NAME>
```

The different components with `menuconfig` are:

- `u-boot`
- `kernel`
- `rootfs` (default)

Other components can be configured by creating a configuration file. These components are:

- `pmufw`
- `bootloader` — FSBL
- `device-tree`

Build PetaLinux project Likewise, to configuring, build individual components with the `-c <COMPONENT-NAME>` flag. To build the whole project use:

```
petalinux-build
```

Packaging the project After building the project, all individual executables are generated individually but are not bundled together in a boot-able image. To generate a basic boot-able image use:

```
petalinux-package --boot --fsbl <FSBL-ELF> --fpga <BITSTREAM> --u-boot --pmufw <PMUFW-ELF>
```

With this image, the device will not boot securely. To generate an image for a secure boot, read the following section.

5.2. Secure Boot

The implementation of secure boot focuses mainly on packaging and settings in the eFuse registers. As explained in Section 4.1 two implementations of secure boot are possible. By setting the `ENC_ONLY` bit in the eFuse the "encryption only" secure boot process is activated. This boot mode is limiting, because this mode only uses AES for authentication and confidentiality. Additionally, the device can only use the AES master or device key stored in the eFuse register. Therefore, the other secure boot mode "root of trust" has been implemented. To activate the "root of trust" mode one has to write the `RSA_EN` bits in the eFuse. For testing purposes, Xilinx provides a feature to work with this secure boot mode, before writing the `RSA_EN` bits. In this mode, everything works as expected, with authentication, but the authentication of the PPK never happens. Instead, the bootloader skips this step and goes right to the authentication process. Therefore, neither secure boot nor key revocation works, because the checks between eFuse and the boot header are skipped. Nevertheless, the chip is not limited to authenticated images. It can run any images, as long as those secure boot bits are not set.

Too boot securely, the content of the boot-able image and the security features of each partition have to be defined. Instead of packaging the image only using the command above, which is very limited, an additional file is introduced. This `.bif` file is a description file of the boot image. It defines:

- The containing partitions
- Where the partition is executed

- If the partitions are encrypted, what key to use
- If the partitions are authenticated, what key to use
- Where and when partitions are loaded

A complete list and description of all definitions is available in the Bootgen User Guide [33]. Bootgen is the tool behind the `.bif` file. It is automatically installed when installing PetaLinux. In case the bootgen tool has to be used separately, it can be installed using the Vivado Design Suite Web installer [34].

5.2.1. Authentication

The key and signature paths have to be included in order to generate images with authentication. Thus, in the `.bif` file has to be declared, which key is used for which partition. Because RSA is an asymmetric authentication method with public and private keys, there are multiple possibilities to do so. The more straightforward way, which we implemented, was by providing the secret keys. The other possibility is by using only public keys. This possibility can be used to protect the secret keys. Listing 5.1 shows the settings when only public keys are used.

```
image : {
  /* Key revocation features */
  [auth_params] ppk_select=0; spk_select=spk-efuse; spk_id=0x00000000

  /* Define primary public key file */
  [ppkfile]primarypublickey.pem

  /* Define secondary public key file for the boot header*/
  [spkfile]secondarypublickey1.pem

  /* Define the signature file for the secondary public key */
  [spksignature]spk_signature.sig

  /* Define the signature file for the boot header and fsbl as they
     are together authenticated */
  [bhsignature]bh_signature.sig

  /* Define the signature file for the header table */
  [headersignature] header_signature.sig

  /* Enabling authentication and define signature file for individual
     images */
  [
  authentication    = rsa,
  spkfile           = secondarypublickey2.pem,
  presign           = signature2.sig,
  /* (Optional) Select a different spk_id and spk_id source */
  spk_select        = <spk-efuse/user-efuse>,
  spk_id            = 0x00000000
  ] image.bin
}
```

Listing 5.1: Example `.bif` file for authentication using only public keys

This way is more complicated than with secret keys because additionally, a signature has to be provided. However, it is a more secure way because the secret keys remain unrevealed. To generate the signatures and key files, use a toolkit like OpenSSL. The generated keys have to have a key length of 4096 bits. A good systematic description is available in the bootgen documentation [33] in the description about creating images using a HSM module, starting on page 73. The key revocation settings will be discussed in Section 5.5.

If the secret keys are available during the image creation, the `.bif` file looks a bit simpler. Instead of defining public keys and signatures, only the secret keys have to be defined. Bootgen automatically generates signatures and public keys in the process.

```
image : {
  /* Key revocation features */
  [auth_params] ppk_select=0; spk_select=spk-efuse; spk_id=0x00000000

  /* Define primary secret key */
  [pskfile] primarysecretkey.pem

  /* Define secondary secret key for the boot header*/
  [sskfile] secondarysecretkey1.pem

  /* Enabling authentication and define a secondary secret key for
     individual images */
  [
  authentication    = rsa,
  sskfile           = secondarysecretkey2.pem,
  /* (Optional) Select a different spk_id and spk_id source */
  spk_select        = <spk-efuse/user-efuse>,
  spk_id            = 0x00000000
  ] image.bin
}
```

Listing 5.2: Example `.bif` file for authentication using secret keys

An example of a full `.bif` file can be found in Listing A.1 in Appendix A.1. In this example, for each partition, a different key has been used. The additional parameter

`[fsbl_config] bh_auth_enable`

an be set, to test authentication and secure boot without blowing the `RSA_EN` eFuse. With this parameter set, authentication will be enabled. However, the PPK will not be checked with the hash stored in the eFuse, as well as the `SPK_IDs`. If neither the `RSA_EN` eFuse nor this parameter is set, nothing of the boot image will be authenticated.

Important The boot header and the FSBL need to have the same key defined. They usually are authenticated separately, but we noticed during testing, that the authentication throws an error, if two separate keys are defined.

5.2.1.1. Generate Keys for RSA

In both cases, at some point, keys have to be generated. Xilinx provides a way of using bootgen.

`bootgen -generate_keys pem -arch zynqmp -image boot.bif`

With this way, bootgen will generate secret RSA keys in the defined location. The location must exist, else bootgen will throw a segmentation fault. This command is only usable if a primary and one secondary key is used. In cases, which more secondary keys are used, bootgen will not generate the other keys. Thus, the second approach with OpenSSL is easier.

`openssl genrsa -out key.pem 4096`

Important is to generate a key with the key length of 4096 bits.

5.2.1.2. Generate Primary Public Key (PPK) Hash

To be able to write the hash from the PPK to the eFuse, it first has to be generated. Thus, the bootgen tool also provides a feature to generate the hash automatically.

```
bootgen -efuseppkbits efuseppkhash.txt -arch zynqmp -w -o test.bin -image boot.bif
```

Bootgen stores the hash in the defined file. In this example `efuseppkhash.txt`.

5.2.2. Encryption

As AES is a symmetrical method, it is much easier to implement. However, it has the disadvantage that the key files have to be available, for the generation of the image. The operational key method reduces the amount, in which the device key is used. While with the normal method, all the images would be decrypted using the device key, with the operational key method, only a small encrypted part of the boot header is decrypted with the device key. All following partitions are decrypted using their own key, saved in the previous partition. To further reduce the risk of a key leakage, the rolling key method can also be implemented. Listing 5.3 shows a basic `.bif` file, implementing encryption with the operational key method and the rolling key method for the `longimage.bit` file. An example of a complete `.bif` file can be found in Listing A.1 in Appendix A.1. More information about `.bif` file attributes can be found in the bootgen User Guide [33].

```
image : {
  /* Define source of device key */
  [keysrc_encryption] bbram_red_key

  /* Further options */
  [fsbl_config] opt_key

  /* Enabling encryption and defining key file for individual images */
  [
  encryption    = aes,
  aeskeyfile    = aeskeyfile1.nky
  ] image.bin

  /* (Optional) Enabling rolling key method for big image partition */
  [
  encryption    = aes,
  /* Attention!: the keyfile has to have the number of keys required for the amount of blocks */
  aeskeyfile    = aeskyfile2.nky,
  /* Define length and amount of blocks */
  blocks        = 2014(2);2048(2);8192(2);4096(*)
  ] longimage.bit
}
```

Listing 5.3: Example `.bif` file for Encryption using operational and rolling key method

5.2.2.1. Generate Keys for AES

To generate keys for AES nothing has to be especially done. When the bootgen command runs, it automatically generates new keys, if it does not find the defined ones. Only the part name has to be specified with the `-p <partname>` option, in addition to the usual options. The part name can be anything. Listing 5.4 shows that bootgen writes the part name into the key file. The key files may vary. This key file has been generated with the `.bif` file from the appendix A.1.

```
1  Device        <partname>;
2
3  Key 0         50
       C0F949817F0A00DD3A66117599936D7A14BAB349BB546E1CBBAD32F69278D3;
4  IV 0          A2E411F57D6045FE506B994F;
5
6  Key 1         83283
       F2FCA0F1A2643A8D62811CBB2DB6184C330CF9E926818FA098556ECBBD5;
7  IV 1          C28E78DAE84F9BF415FD142D;
8
9  Key Opt       96
       EEBD73F10EB683E8F4028D609AB86AA90E8D19920AB55BF507878C6DEE3E86;
```

Listing 5.4: Example of a generated AES key file

The first key is the device key, which has to be loaded, to the eFuse or BBRAM. This key should be the same across all generated key files. Also, the according IV should be the same. The second key is the individual key for each partition. This key and the IV are different from partition to partition. The last key is the operational key. This key and IV are also identical on all key files.

5.2.3. Write eFuse or BBRAM

The main possibility to write the eFuse and the BBRAM is through the xilskey library provided by Xilinx [20]. Xilinx already provides an example which only has to be complemented with the user-specific data. To use the example the description in the application note Programming BBRAM and eFuses [35] can be followed. While the BBRAM can be written a limitless amount, the bits in the eFuse can only be set once from 0 to 1.

5.2.3.1. eFuse Registers

Table 5.1 provides an overview about the most important eFuse registers. A complete list is available in the Zynq Ultrascale+ Devices Register Reference [36].

Register Name	Size (bits)	Description
RSA_EN	15	Enforces to boot with authentication. Enables secure boot.
PPK0_WRLK	1	Locks writing to the PPK0 eFuse.
PPK0_INVLD	2	Revokes the PPK0.
PPK0_0...11	12×32	Hash of PPK0
PPK1_WRLK	1	Locks writing to the PPK1 eFuse.
PPK1_INVLD	2	Revokes the PPK1.
PPK1_0...11	12×32	Hash of PPK1, in case of key revocation
SPK_ID	32	SPK ID to disable images with old secondary keys

USER_0...7	8×32	Eight user fuses. Can be used for enhanced key revocation or user specific features.
ENC_ONLY	1	Enforces encrypted boot partitions. Only the eFuse key can be used for AES. Enables secure boot.
AES_RDLK	1	Disables the CRC check for the AES key.
AES_WRLK	1	Locks writing the AES eFuse.
AES_KEY	256	eFuse register for the AES key.
BBRAM_DIS	1	Disables the BBRAM key.
JTAG_DIS	1	Disables the JTAG controller.
DFT_DIS	1	Disable Design for Test (DFT).
SEC_LOCK	1	Disable reboot into JTAG mode after secure lockdown.

Table 5.1.: The most important eFuse registers

5.3. Multi-boot

In our implementation, the device boots from QSPI. The system divides the QSPI into different sections. The first section is the boot image section. This section is where the images are stored. The second section is the boot environment section. There are the U-Boot environment variables stored. The last section is the kernel section, where the OS image is stored if the device gets the OS from QSPI. It is important to make sure that each image has its place in the storage, and is not intersecting another. In our example, the first boot image is stored in the section `0x000000` — `0x100000`. The fallback image is stored in the section `0x100000` — `0x200000`. The remaining sections are located after the address `0x200000`. Now, because the fallback image comes after the regular one, it has to be ensured that the update process does not accidentally overwrite the fallback image. For example, in the provided update program, it is ensured, that the image size is smaller than `0x100000` to prevent overwriting the fallback image. In addition to that, the image sections defined in the device tree have to be adjusted. Else, U-Boot may override some data when saving its boot environment variables. The sections can be adjusted during the configuration of the PetaLinux project. Access the configuration with:

`petalinux-config`

In the opened menuconfig the sections can be adjusted under

`Subsystem AUTO Hardware Settings-->Flash Settings -->`.

Now that space is allocated for the second image, it only has to be stored to the memory section. One way to copy images into the allocated memory section is inside the U-Boot console with the commands showed in Listing 5.5.

```
1  echo Load image from sd card (bootdev)
2  load mmc ${sdbootdev} 0x1000000 goldenImage.bin
3  echo Check that the image size does not exceed 0x1000000
4  if itest ${filesize} > 0x1000000
5  then
6  echo IMAGE SIZE BIGGER THEN 0x1000000! EXITING...
7  else
8  echo Image size ok.
9  echo Erase QSPI boot section 0x1000000 - 0x2000000
10 sf probe
11 sf erase 0x1000000 0x2000000
```

```
12  echo Write image to QSPI
13  sf write 0x1000000 0x1000000 ${filesize}
14  echo Check back written image
15  sf read 0x2000000 0x1000000 ${filesize}
16  if cmp.b 0x1000000 0x2000000 ${filesize}
17  then
18  echo IMAGE SUCCESSFULLY WRITTEN! EXITING...
19  else
20  echo IMAGE NOT SUCCESSFULLY WRITTEN! EXITING...
21  fi
22  fi
```

Listing 5.5: U-Boot commands to copy an image to the QSPI

Another way is by doing these steps in Linux, which is a little simpler. Listing 5.6 shows the different commands used in a Linux shell.

```
1  MAX_FILESIZE=16777216; # Max allowed filesize
2  FILENAME="goldenImage.bin";
3  FILESIZE=$(stat -c%s "\$FILENAME");
4
5  # Check file size
6  if [[ $FILESIZE -gt $MAX_FILESIZE ]]
7  then
8      echo "Image to big!! Leaving..."
9  else
10     flashcp -v "$FILENAME" /dev/mtd0;
11 fi
12 exit 1;
```

Listing 5.6: Linux commands to copy an image to the QSPI

In both ways, the image size is checked before writing the image to the storage medium. In cases where the image is bigger than the allocated space, parts of the following storage sections would get overwritten. In the case of a following backup image, this would be very bad.

5.4. Update Process Implementation

The update process varies from use case to use case and is dependable on both the boot and the update source used. Generally, the update procedure looks as shown in figure 5.1. The update process has to be initiated by an authorized source. Then the new image is read from the external memory source and checked. If the check was successful, the image can be written to the boot storage and the device rebooted. Else, the update process has to be stopped. The implemented update process is very rudimentary and serves only as a proof of concept. As described in the previous section, QSPI is used as the primary boot storage. Thus, to update the image, the allocated section in the boot storage has to be overwritten with the new image. To overwrite the image, we used the same approach as described in the previous section.

In the first step has to be ensured that only authenticated users can initiate the update process. Thus, we implemented the update process in Linux, which provides a user login system. In the next step, we check the image to be in the expected size, to avoid unintentional overwrites of the golden image, which in our case is stored directly after the regular image. Additionally, security checks, for example, authentication checks can be performed in order to provide even more security. After the new image is written, the device has to be restarted, to use the new image. In our case, the restart command is not automatically executed. Therefore, the user has to restart the device manually, making sure, that in the following restart the device boots from the correct boot storage and image. The start address of the image is seen at the end of Listing 5.7 after *QSPI Reading Src* on Line 16.

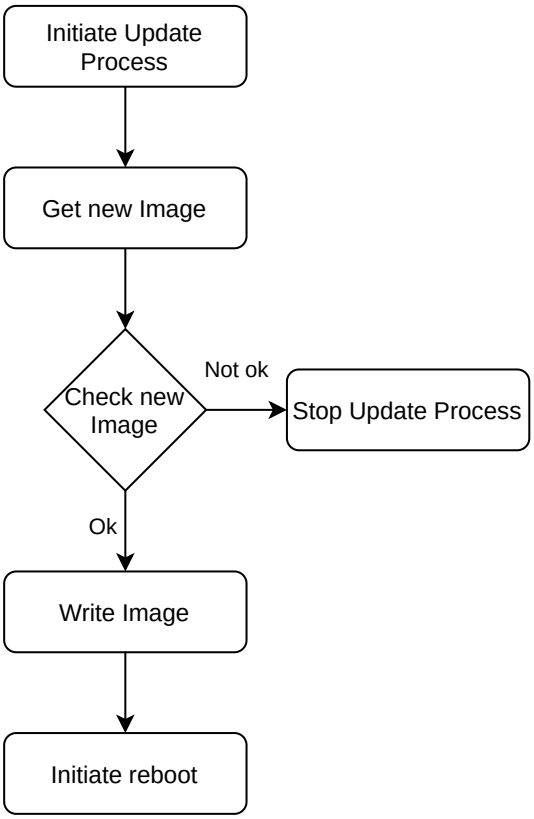


Figure 5.1.: General update procedure

```
Xilinx Zynq MP First Stage Boot Loader
Release 2019.2   Apr  2 2020  -  15:48:40
Reset Mode      :       System Reset
Platform: Silicon (4.0), Cluster ID 0x80000000
Running on A53-0 (64-bit) Processor, Device Name: XCZU5EV
Initializing DDR ECC
Address 0x0, Length 80000000, ECC initialized
Processor Initialization Done
================= In Stage 2 ============
QSPI 32 bit Boot Mode
QSPI is in single flash connection
QSPI is using 4 bit bus
FlashID=0x1 0x2 0x20
SPANSION 512M Bits
Multiboot Reg : 0x0
QSPI Reading Src 0x0, Dest FFFF1C40, Length EC0
...
```

Listing 5.7: The beginning of the boot log output

5.5. Key Revocation

Two major steps are required to revoke keys. First, the `.bif` file has to be adjusted to use the new keys, and a new image has to be generated, which is then loaded to the device. Second, the device eFuse has to be updated. Also, there are some differences, whether the primary or secondary key is revoked. The primary key is checked bit by bit with a hash stored inside the eFuse. Thus, in case of

revocation, the hash needs to be "replaced". (Because the eFuse can only be written once, the term replace is not 100% correct). The secondary keys can be changed as many times as needed because there is no direct connection between eFuse and secondary key authentication. In case of a revocation, the secondary key ID is marked as invalid, which prevents the use of old images with revoked IDs. Section 4.3.3 explains this feature in more detail. In every case, new keys should be generated. For more information about how to generate new RSA keys refer to Section 5.2.1.1.

5.5.1. Primary Key Revocation

Primary key revocation was never tested, in this work, because it would require to blow additional eFuse registers, which would limit the use of the borrowed board. Nevertheless, to implement revocation is nearly the same procedure, as for the initial implementation of the primary key.

The `.bif` file adjustments are simple and consists only of changing the `pskfile` attribute, to point to the new key. The procedure to write the eFuse registers is explained in the application note Programming BBRAM and eFuses [35] from Xilinx. Table 5.1 shows, that there are registers for two PPK hashes (PPK0 and PPK1) in the eFuse registers, as well as a register for each hash two invalidate or revoke.

5.5.2. Secondary Key Revocation

Secondary key revocation has only been tested with the dedicated eFuse registers, but not with the `USER` fuses. The steps for SPK identification via the `USER` fuses and the `SPK_ID` fuses only differentiate in the procedure the ID and the eFuse register are compared. The `SPK_ID` fuse is compared bit by bit ($FUSE_{SPK\_ID} = SPK\_ID$) the ID in the boot header, and on a perfect match, the key is valid. The `USER` fuse represents an array in which the revoked IDs are marked with a one. Thus, valid IDs are marked with a 0 ($USER\_FUSE[SPK\_ID] = 0$). Section 4.3.3 gives a more detailed explanation.

When revoking a secondary key the `.bif` file has to be changed a little further. Beside changing the keysource with the attribute `sskfile`, the ID with the attribute `spk_id` and the ID source with `spk_select` needs also to be changed. In order to maximize the use of available IDs read Section 4.3.3. Additionally the eFuse registers `SPK_ID` or `USER_0...7` need also to be updated. For information on how to write the eFuse registers refer to application note Programming BBRAM and eFuses [35] from Xilinx.

5.6. Tamper Monitoring

Only three tamper events could have been implemented for testing. Those were the tamper events for JTAG toggling, for temperature change and the manual tamper event trigger. Only these three tamper events could be tested due to the lack of material at home, which was the working place during the corona crisis. The implementation was entirely made in Vivado and then imported to the device via the bitstream. Figure 5.2 shows the settings made in the tamper registers. This settings control the reaction to different tamper events and enable additional security measures as erasing the BBRAM. Figure 5.3 shows the settings of the system monitor. The system monitor is relevant if threshold values have to be changed. In this project, the temperature values were adjusted to restrict device run-time between 35 and 85°C. In the case of the temperature restriction, we encountered some issues, which will be discussed in Section 6.4.

Registers control the tamper monitor and the system monitor unit. These register can be set in three different way. Because these registers are security-related, some ways may have restrictions or disadvantages over others.

One way is by loading the settings with a bitstream to the device. This is the easiest way to enable and control the feature, due to an interface provided inside Vivado. However, the drawback is, that the configuration is only taken over when the bitstream is loaded. Meaning, instead of being active after

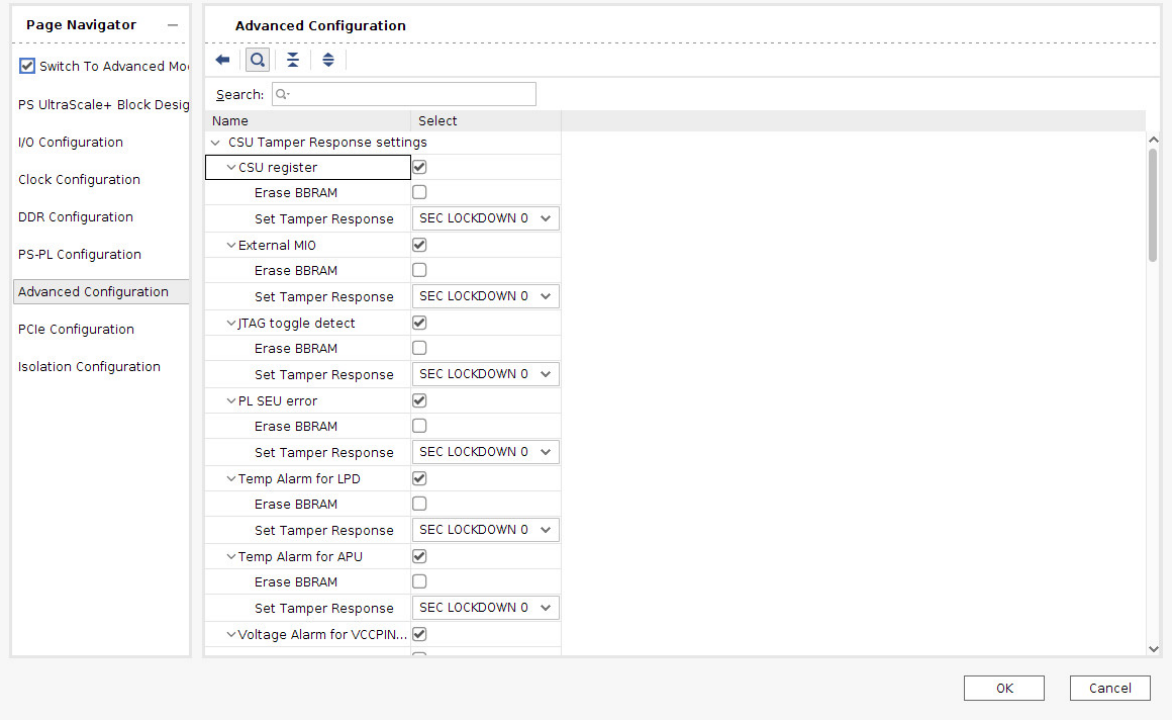


Figure 5.2.: Tamper Response settings

the FSBL is loaded, the tamper monitoring unit with the desired settings is active after the bitstream is loaded. Additionally, we had problems with the tamper events caused by temperature thresholds. Instead of locking the device and putting it to secure lockdown, the device waited until it reached the needed temperature to proceed. More information about this problem is given in Section 6.4.

Another way to configure the tamper and system monitor registers are via the AMS or CSU register drivers in Linux. These drivers provide an interface to the device registers via the sysfs file structure. A small description about the drivers are given on the Wiki page of Xilinx [37;38] or in the Software Developers Guide [13]. This way of reading and writing registers was mainly used for testing purposes. However, most relevant registers are not accessible due to security reasons. Because Linux is running as a normal OS it has only restricted access to security-related registers, for example, threshold values. To access these register with Linux, the PMUFW has to be authorized. In case of access, Linux makes a SMC which is handled by the PMUFW. However, the described solution in the Software Development Guide [13] did not work. The access to the register remained restricted. In any way, doing the main configuration of the tamper and system monitor in Linux is a security risk and should not be considered.

The last way to set up the tamper and system monitoring unit is by writing the registers during startup. Looking to figure 4.8, the tamper monitor gets active with the desired state during the execution phase of the FSBL. Therefore, a function needs to be created, which writes the desired values to the registers. This function then has to be called during the initialization process. Some registers can be set in the `psu_init.c` and `psu_init.h` file. Even though this is the most time consuming and circumstantial way, it is one of the most secure ways to do.

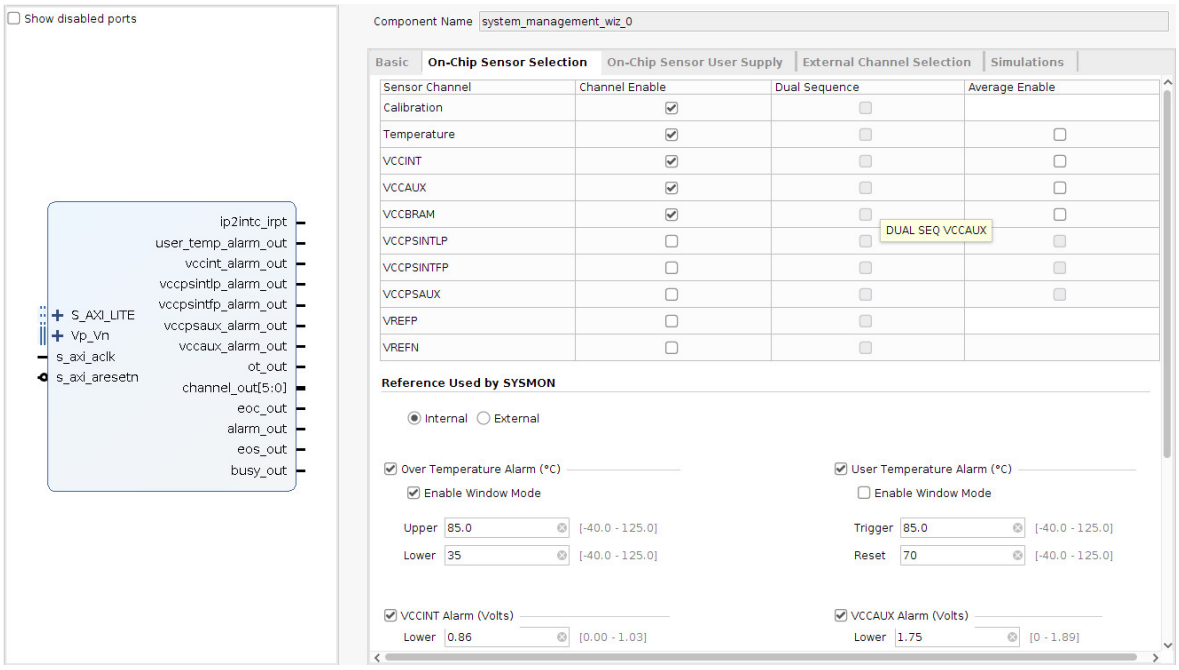


Figure 5.3.: System Monitor Settings

5.7. Crypto API Implementation

This section describes how to implement an application, which uses the Linux Crypto API. Section 5.7.1 describes how to access the Linux Crypto API for SHA and AES. ection 5.7.2 describes the differences in the Xilinx implementation. The different examples are explained in Section 5.7.2.

5.7.1. Procedure and Interface Configuration

The Linux Crypto API uses sockets and systemcalls to communicate between the user-space and kernel space. Depending on the algorithm, the application has to set different attributes like the key and other settings. The rough procedure is for all algorithms the same. The detailed procedure for SHA and AES are described in Section 5.7.1.1 and Section 5.7.1.2.

The entire session to encrypt or hash data with the Linux Crypto API is done with a socket. It has to be created in the first step and closed in the last step. After the socket is created, it can be used to send data or configuration to the Linux Crypto API or to read the processed data back. At first, the desired algorithm has to be selected. After the algorithm is chosen, the algorithm can be configured. The data is sent to the Linux Crypto API via the socket. Finally, the encrypted or hashed data can be read from the Linux Crypto API. The simplified steps to access the Linux Crypto API look like this: [24]

1. Open a socket.
2. Select the desired algorithm.
3. Set the configuration for the algorithm (when required).
4. Send the data to process to the socket.
5. Read the processed data.

5.7.1.1. SHA Hash Procedure

The procedure to hash data with the Linux Crypto API is done in the following steps. The SHA algorithm has no configuration except for the selection of the algorithm. The working example for hash algorithms can be found in Section 5.7.3.

1. Open a `socket` of type `AF_ALG`.
   To start the communication with the Linux Crypto API a socket has to be created. All further steps reference to this socket. Therefore, the Linux Crypto API can distinguish between different accesses. [24]

```
socket_desc = socket(AF_ALG, SOCK_SEQPACKET, 0);
```

2. `Bind` the `socket` to the desired algorithm `name` and `type`.
   After the socket is created, it has to be bound to the desired algorithm. To do this, the `type` and `name` of the algorithm has to be selected. For a hash algorithm the `type` is `hash` and the `name` `sha256` for a normal SHA-256. For the Xilinx implementation of the SHA3-384 algorithm, the name is `xilinx-keccak-384`. [24] [25]

```
struct sockaddr_alg sa = {
.salg_family = AF_ALG,
.salg_type = "hash",
.salg_name = "sha256",
};

bind(socket_desc, (struct sockaddr*)&sa, sizeof(sa));
```

3. For a hash algorithm like SHA, no settings like a key or key length have to be set.

4. Send the data, which needs to be hashed, to the Linux Crypto API.
   Create a file descriptor to transfer the data with the `accept` systemcall and send the data with the `write` systemcall. [24]

```
fd = accept(socket_desc, NULL, 0);
write(fd, data_to_hash, sizeof(data_to_hash));
```

5. Get the calculated hash from the Linux Crypto API.
   The `read` systemcall can be used to get back the calculated hash. The hash size of the algorithm has to be known. See table 4.7 for the different hash algorithms and according sizes. [24]

```
read(fd, hash, SHA_HASH_SIZE);
```

5.7.1.2. AES Encryption/Decryption Procedure

The procedure to use the Linux Crypto API to encrypt or decrypt data with the AES CBC algorithm is listed below. The procedure for AES GCM is similar, but uses different configurations for the additional functions of GCM. The examples for CBC and GCM in Section 5.7.3 show a complete implementation.

1. Open a `socket` of type `AF_ALG`.
   To start the communication with the Linux Crypto API a socket has be created. All further steps reference to this socket. Therefore, the Linux Crypto API can distinguish between different accesses. [24]

```
socket_desc = socket(AF_ALG, SOCK_SEQPACKET, 0);
```

2. `Bind` the `socket` to the desired algorithm `name` and `type`.
   After the socket is created it has to be bound to the desired algorithm. To do this, the `type` and `name` of the algorithm has to be selected. For AES CBC the `type` is `skcipher` and the `name` is `cbc(aes)`. [24]

```
struct sockaddr_alg sa = {
.salg_family = AF_ALG,
.salg_type = "skcipher",
.salg_name = "cbc(aes)"
};

bind(socket_desc, (struct sockaddr*)&sa, sizeof(sa));
```

3. Set key and key length.
   The `setsockopt` systemcall is used to set the key and key length. The key length selects if AES 128, 192 or 256 is used. [24]

```
#define AES_KEY_LENGTH 32 /* 16 byte -> 128 bit, 24 byte -> 192,
    32 byte -> 256 bit */

char aes_key[] = "aes key ...";

setsockopt(sd, SOL_ALG, ALG_SET_KEY, aes_key, AES_KEY_LENGTH);
```

4. Create a `msghdr` structure for further configurations.
   The `msghdr` structure holds the IV, the IV length, the data and the direction if the data needs to be encrypted or decrypted. Only the data is directly contained in the `msghdr` structure. The rest is in `cmsg` structures inside the `msghdr` structure. The `CMSG_SPACE(4) and CMSG_SPACE(20)` are the size for the two `cmsg` structures in the `msghdr` structure described below. [24]

```
struct msghdr msg = {};

/* buffer for op, iv and aad data length, used for msghdr (msg) */
char cbuf[CMSG_SPACE(4) + CMSG_SPACE(20)] = {0};
msg.msg_control = cbuf;
msg.msg_controllen = sizeof(cbuf);

/* pointer for elements from cbuf */
struct cmsghdr *cmsg;
```

5. Select the encryption/decryption direction.
   Create the first `cmsg` structure in the `msghdr` structure and select if the data should be encrypted or decrypted. The flag to select the direction is a 4 byte variable, therefore `CMSG_LEN(4)`. [24]

```
cmsg = CMSG_FIRSTHDR(&msg);
cmsg->cmsg_level = SOL_ALG;
cmsg->cmsg_type = ALG_SET_OP;
cmsg->cmsg_len = CMSG_LEN(4);
/* select encryption / decryption (ALG_OP_ENCRYPT / ALG_OP_DECRYPT
    ) */
*(__u32 *)CMSG_DATA(cmsg) = ALG_OP_ENCRYPT;
```

6. Select IV and IV length.
   Create the second `cmsg` structure in the `msghdr` structure to set the IV and IV length. The length of the IV is always 128 bit (16 byte) for AES CBC, but can be different for other algorithms. The size of the structure is 20 bytes, 16 bytes for the IV and 4 bytes for the length. [24]

```
char iv[] = "iv ...";

cmsg = CMSG_NXTHDR(&msg, cmsg);
```

```
4   cmsg->cmsg_level = SOL_ALG;
5   cmsg->cmsg_type = ALG_SET_IV;
6   cmsg->cmsg_len = CMSG_LEN(20);
7   ivp = (void *)CMSG_DATA(cmsg);
8   ivp->ivlen = 16;
9
10  /* set iv for cbc algorithm */
11  memcpy(ivp->iv, iv, 16);
```

7. Send the data to the Linux Crypto API.
   Add the data to encrypt or decrypt to the `msghdr` structure and send it to the Linux Crypto API. [24]

```
1   char iov_buf[AES_MSG_LENGTH];
2   struct af_alg_iv *ivp;
3   struct iovec iov = {iov_buf, AES_MSG_LENGTH};
4
5   char data[] = "Data to encrypt or decrypt ...";
6
7   memcpy(iov_buf, data, AES_MSG_LENGTH);
8   iov.iov_len = AES_MSG_LENGTH;
9   msg.msg_iov = &iov;
10  msg.msg_iovlen = 1;
11
12  sendmsg(fd, &msg, 0);
```

8. Get the encrypted or decrypted data from the Linux Crypto API. [24]

```
1   /* buffer to reveive data from kernel */
2   char buf[AES_MSG_LENGTH];
3
4   read(fd, buf, AES_MSG_LENGTH);
```

Table 5.3 shows the configuration, to use the Linux Crypto API with AES CBC. The data needs to be padded to the block size, the input and output data size is the same.

Table 5.5 shows the configuration, to use the Linux Crypto API with AES GCM.

Name	Description	Value	Type
Algorithm Type	Type of the used cipher.	`skicipher`	string
Algorithm Name	Name of the hardware under which the driver is registered.	`cbc(aes)`	string
AES Key Length	Select the AES key length.	`128`, `192`, `256`	int
AES Key	Set the AES key for encryption or decryption.	128 / 192 / 256 byte `key`	
IV Length	Select IV length.	`16` byte (128 bit)	int
IV	Initialization Vector (IV)	16 byte (128 bit) `iv`	
Encrypt/Decrypt	Select if the data should be encrypted or decrypted.	`"ALG_OP_ENCRYPT"` `"ALG_OP_DECRYPT"`	int
Data Length	Length of the data to encrypt or decrypt. Must be a multiple of block size.	Length in bytes	int

Table 5.3.: Configuration for AES CBC [24] [15]

5.7.2. Difference of the Xilinx Implementation

The Linux Crypto API is designed to work platform-independent and presents a consistent interface to the applications. This requires the hardware manufacturer to implement the drivers for the crypto hardware in a certain way. As described in Section 4.6.1, the Zynq Ultrascale+ has two different crypto hardware blocks implemented. The Arm® Cryptography Extension has drivers, who use the generic way to implement the Linux Crypto API. The examples, which use the Arm® Cryptography Extension, run on an x86 computer with Linux as well. The drivers for the crypto hardware from Xilinx do not use the usual way. Code, which uses this hardware, will not run on another architecture. [24]

5.7.2.1. Xilinx SHA Implementation

The driver for the SHA3-384 hardware of the Zynq Ultrascale+ uses the usual approach, but the algorithm is named `xilinx-keccak-384`. Apart from the name, the driver is platform-independent.

5.7.2.2. Xilinx AES GCM Implementation

Several points are implemented differently in the driver for the Xilinx crypto hardware for AES GCM. As the driver for the Linux Crypto API from Xilinx is not documented, these differences or limitations are not listed. The differences are found by using the interface and analyzing the results. It is possible that some of the missing features are implemented in a completely different way, which was not discovered during the project.

Key Handling The Zynq Ultrascale+ has built-in key handling. One key is selected during boot to decrypt the boot images. This key can be used to encrypt or decrypt further data. To select this key, a zero string must be passed to the Linux Crypto API as key.

Name	Description	Value	Type
Algorithm Type	Type of the used cipher.	`aead`	string
Algorithm Name	Name of the hardware under which the driver is registered.	`gcm(aes)`	string
AES Key Length	Select the AES key length.	`128`, `192`, `256`	int
AES Key	Set the AES key for encryption or decryption.	128 / 192 / 256 byte key `key`	
IV Length	Select IV length.	`12` byte (96 bit)	int
IV	Initialization Vector (IV)	12 byte (96 bit) `iv`	
Encrypt/Decrypt	Select if the data should be encrypted or decrypted.	`"ALG_OP_ENCRYPT"` `"ALG_OP_DECRYPT"`	int
AEAD Data Length	Length of the assosiated data, must be multiple of block size. (Data which is only authenticated, not encrypted.)	Length in bytes	int
AEAD Data	Assosiated data (Data which is only authenticated, not encrypted.)	aead data	
Data Length	Length of the data to encrypt or decrypt. Must be a multiple of block size.	Length in bytes	int

Table 5.5.: Configuration for AES GCM [24] [15]

Key length The key length is fixed to 256 bit, usually AES works with key lengths from 128, 192 and 256 bit.

Authentication tag length The authentication tag length is fixed to 16 bytes. It usually can be adjusted from 0-16 bytes.

AAD data The associated data of GCM is not implemented.

Buffer sizes When the buffer with the plaintext/encrypted data and the AAD data is passed to the Linux Crypto API, the length of the buffer has to be specified. Usually, for encryption, this can be calculated by AAD_LENGTH + DATA_LENGTH, and for decryption, it is AAD_LENGTH + DATA_LENGTH + AUTH_TAG_LENGTH which corresponds to the data which is actually passed to the Linux Crypto API.

In the Xilinx implementation, it is always DATA_LENGTH + AUTH_TAG_LENGTH (the AAD data part is missing, as it is not implemented). Although during encryption, only the data to encrypt is passed to the Linux Crypto API, the buffer size of the data has to include the authentication tag size.

The buffers to send and receive data are separate, the size of the buffer to receive data is calculated as usual.

Name and type AES GCM is an AEAD algorithm, therefore the type of the algorithm in the Linux Crypto API to select is `aead` and the name is `gcm(aes)`. Xilinx uses the type `skcipher` and the name `xilinx-zynqmp-aes`.

5.7.3. Examples

In this section, the different examples for the Linux Crypto API are described. Table 5.6 lists the different examples. The examples can be found in the Git repository 1.1.

Example	Hardware	Compatibility	Example file
SHA-256	Arm® Cryptography Extension	compatible with other architectures	crypt-api-sha256.c
SHA3-384	Xilinx crypto hardware	only on Zynq Ultrascale+	crypt-api-sha3-384.c
AES CBC	Arm® Cryptography Extension	compatible with other architectures	crypt-api-aes-cbc.c
AES GCM	not implemented in the Zynq Ultrascale+	x86 (general GCM implementation)	crypt-api-aes-gcm.c
AES GCM 256	Xilinx crypto hardware	only on Zynq Ultrascale+	crypt-api-aes-gcm-256-zynqmp.c

Table 5.6.: Overview examples

5.7.3.1. Security Reminder

These examples show how to access the Linux Crypto API and are not a complete tool to encrypt files or data. Algorithms like AES are only secure if used correctly. Read the section 4.2 to understand these algorithms. In general, the key has to be kept in secret and needs a high entropy. The IV must be generated randomly for every encryption. Do not use the key or IV provided in these examples in a production environment. [15]

5.7.3.2. Requirements

The compiler `aarch64-none-linux-gnu-gcc` Linux applications on the aarch64 architecture is required to compile the examples. It can be downloaded from Arm®. The provided makefile expects to find the compiler binaries under the following path. If the path is different, it has to be changed in the makefile.

`/opt/gcc/gcc-arm-9.2-2019.12-x86_64-aarch64-none-linux-gnu/bin/`

Some of the examples can run on the x86 architecture. Therefore, they can be tested on a host computer running Linux. The makefile contains the instructions to compile the compatible examples for the host. It requires the `gcc` and `make-utils` package for the host. If PetaLinux is installed, these

packages are already installed. It is also possible to add the application in PetaLinux and build it with the Linux image.

5.7.3.3. SHA Examples

Two examples are made for SHA. One is for SHA-256 and the other for SHA3-384. The example for SHA-256 can be found in `Software/crypto-api/crypto-api-sha256.c`. It uses the Arm® Cryptography Extension and should run on most Linux systems.

The example for SHA3-384 can be found in `Software/crypto-api/crypto-api-sha3-384.c`. It uses the Xilinx crypto hardware from the Zynq Ultrascale+. Most of the implementation is similar, but the name of the algorithm is `xilinx-keccak-384` instead of the generic name.

In both examples the data, which is hashed, is stored in the `data_to_hash` variable at the beginning of the code. The resulting hash is printed out to the terminal.

5.7.3.4. AES Examples

There are three examples for AES. One example for CBC and two for GCM.

The example for AES CBC can be found in `Software/crypto-api/crypto-api-aes-cbc.c`. It uses the Arm® Cryptography Extension and should run on most Linux systems.

For AES GCM two examples are available, `Software/crypto-api/crypto-api-aes-gcm.c` and `Software/crypto-api/crypto-api-aes-gcm-256-zynqmp.c`. The first one shows the usual implementation of the Linux Crypto API for AES GCM and runs on the x86 architecture. The second example shows how Xilinx implemented the driver for their crypto hardware in the Zynq Ultrascale+. Section 5.7.2 describes the differences.

Table 5.3 for AES CBC and Table 5.5 for GCM list the different options like keys or data length, which can be modified to use the examples. All the options can be set at the beginning of the examples in the sections marked `SETTINGS`. Some of the settings in the AES GCM for the Zynq Ultrascale+ are missing, as they are not implemented by Xilinx.

5.8. TrustZone Implementation

This section describes the procedure to implement TrustZone in a project. Section 4.7 describes the concept and functionality of TrustZone. Section 5.8.1 shows the implementation of the Arm® Trusted Firmware (ATF), Section 5.8.2 shows the implementation of OP-TEE.

5.8.1. Arm® Trusted Firmware (ATF) Implementation

The Arm® Trusted Firmware (ATF) is the Secure Monitor from Arm®. As it is required to boot the Zynq Ultrascale+, the ATF is included in every project. If TrustZone is not needed in a project, the ATF is not used after boot. To add custom SMC, the source code of the ATF needs to be modified and recompiled. The ATF can either be compiled on its own or as part of a PetaLinux project. [2]

5.8.1.1. ATF in PetaLinux project

The ATF is included in every PetaLinux project for the Zynq Ultrascale+ by default. The generated configuration can be found under the following path:

`<plnx-proj-root>/project-spec/meta-plnx-generated/recipes-bsp/arm-trusted-firmware/`

After the project is built, the binary file for the ATF can be found in: [31]

`<plnx-proj-root>/images/linux/bl31.elf`

5.8.1.2. ATF standalone project

To compile the ATF outside of a PetaLinux project, the source code is needed. The ATF source code from Xilinx can be found on their Github page:

https://github.com/Xilinx/arm-trusted-firmware

The compiler for aarch64 bare-metal target is needed (aarch64-none-elf). It can be downloaded from Arm®:

https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-a/downloads

To compile the ATF the following command has to be executed in the root directory of the ATF source: [39]

```
make CROSS_COMPILE=<compiler-path>/bin/aarch64-none-elf- PLAT=zynqmp bl31
```

The finished binary is placed at the following path:

`<atf-source-root>/build/zynqmp/release/bl31/bl31.elf`

5.8.1.3. Package ATF

The ATF binary has to be packaged with the other software binaries, in order to be executed on the Zynq Ultrascale+. In Section 5.1.2 is described, how to package a normal PetaLinux project. If PetaLinux is used to compile the ATF, the normal command to create the package can be used. The ATF binary is added automatically. [31]

```
petalinux-package --boot --fsbl <FSBL-ELF> --fpga <BITSTREAM> --u-boot --pmufw <PMUFW-ELF>
```

To generate the package with an ATF binary outside from the PetaLinux project, the following command can be used: [31]

```
petalinux-package --boot --fsbl <FSBL-ELF> --fpga <BITSTREAM> --u-boot --pmufw <PMUFW-ELF>
--atf <atf-source-root>/build/zynqmp/release/bl31/bl31.elf
```

5.8.2. OP-TEE Implementation

The following section describes how to create a PetaLinux project with OP-TEE. It is further explained how to compile and load a Trusted Application (TA). The OP-TEE project only worked on the development board `zcu102` from Xilinx. It does not boot on the Mercury board from Enclustra. The problem for this could not be found during this work. Therefore, this implementation describes only the solution for the `zcu102` board.

5.8.2.1. Setup PetaLinux project with OP-TEE

OP-TEE has an example project for the Zynq Ultrascale+, but some additional steps are required to get the project running. The OP-TEE Documentation. describes the example. [29]

The OP-TEE example is made for PetaLinux 2018.2 and does not work with version 2019.2. The project can be compiles after the version check is removed. However, it does not boot if, OP-TEE was not compiled with version 2018.2.

Since the OP-TEE example for the Zynq Ultrascale+ was created, OP-TEE was developed further and the example no longer works. The OP-TEE realeses from the time, when the example was created, are used to test the example. Table 5.7 lists the different parts of OP-TEE and the used releases.

Name	**Link**	**Git Commit**	**Date**
OP-TEE OS	https://github.com/OP-TEE/optee_os.git	e61fc00f9643fb55f2b1 9d1168d86b9b15d8d9c9	19.04.2019
OP-TEE Client	https://github.com/OP-TEE/optee_client.git	e9e55969d76ddefcb5b3 98e592353e5c7f5df198	14.05.2020 (latest)
OP-TEE Test	https://github.com/OP-TEE/optee_test.git	895c5caa9070a134bc12 acdc6d0ad0354aa1f644	28.02.2019

Table 5.7.: OP-TEE Version Overview

This are the required steps to create the OP-TEE project, the description is similar to the guide in the OP-TEE documentation, but has some additional steps.

1. Create a new directory for the project. This directory serves as the root directory for the project.
2. Place the Board Support Package (BSP) in the project root directory.
   The BSP for Xilinx boards can be downloaded from the Xilinx download page. The file for the zcu102 board is `xilinx-zcu102-v2018.2-final.bsp`. (As the OP-TEE example was made for PetaLinux 2018.2, the BSP for this version needs to be selected.)
3. Clone the OP-TEE build repository into the project root directory.
   `git clone https://github.com/OP-TEE/build`
4. Build the project.
   The cloned repository contains a makefile to create the project. By default, the project is generated for the zcu102 board from Xilinx. The makefile creates the project and starts to build, the first build will fail, as some additional configurations need to be made, as described in the next steps.
   ```
   cd build/
   make -f zynqmp.mk
   ```
5. Clone the OP-TEE OS and Test repository into the project root directory.
   (OP-TEE has 3 parts, optee_os, optee_client and optee_test. At the time of the implementation, the latest version of optee_client worked and only optee_os and optee_test needed older versions. The tested versions are listed in Table 5.7.)
   ```
   git clone https://github.com/OP-TEE/optee_os.git
   git clone https://github.com/OP-TEE/optee_test.git
   ```
6. Revert the OP-TEE repositories.
   Revert the repositories to the state, when the example was created. See Table 5.7.
   `cd optee_os`

```
git revert e61fc00f9643fb55f2b19d1168d86b9b15d8d9c9
cd ../optee_test
git revert 895c5caa9070a134bc12acdc6d0ad0354aa1f644
```

7. Change the configuration files to use the local repositories for OP-TEE.
   Open file `zcu102-2018.2/project-spec/meta-user/recipes-bsp/optee-os/optee-os.bb` and modify line 27:

```
26  ...
27  REPO ??= "git://<path-to-proj-root-dir>/optee_os/;protocol=file"
28  ...
```

   Open file `zcu102-2018.2/project-spec/meta-user/recipes-apps/optee-test/optee-test.bb` and modify line 21:

```
20  ...
21  REPO ??= "git://<path-to-proj-root-dir>/optee_test;protocol=file"
22  ...
```

8. Recompile the PetaLinux project.
   After the configuration is made, the project needs to be recompiled, this time it the build should complete without errors.
   `cd <path-to-proj-root-dir>/zcu102-2018.2/`
   `petalinux-build`

   After the build is finished, the following files should be in folder `<path-to-proj-root-dir>/zcu102-2018.2/images/linux/`:
   `zynqmp_fsbl.elf` (FSBL Binary)
   `pmufw.elf` (PMU Firmware Binary)
   `u-boot.elf` (U-Boot Binary)
   `system.bit` (Bitstream)
   `bl31.bin` (ATF Binary)
   `bl32.elf` (OP-TEE Binary)
   `image.ub` (Linux)

9. Package the binaries. Create a file named `boot.bif` with the configuration for the package:

```
1  image : {
2  [ /* fsbl */
3  bootloader,
4  destination_cpu          = a53-0
5  ] zynqmp_fsbl.elf
6
7  [ /* pmufw */
8  destination_cpu          = pmu
9  ] pmufw.elf
10
11 [ /* bitstream */
12 destination_device       = pl
13 ] system.bit
14
15 [ /* trusted firmware */
16 trustzone,
17 exception_level          = el-3,
18 destination_cpu          = a53-0
19 ] bl31.elf
20
21 [ /* u-boot */
```

```
22 exception_level          = el-2,
23 destination_cpu          = a53-0
24 ] u-boot.elf
25
26 [ /* op-tee */
27 trustzone,
28 exception_level    = el-1,
29 destination_cpu    = a53-0
30 ] bl32.elf
31 }
```

With the following command the package is created:
`bootgen -arch zynqmp -image boot.bif -o i boot.bin`

10. Boot the Zynq Ultrascale+.
The files `boot.bin` and `image.ub` are needed to boot the Zynq Ultrascale+. During boot, OP-TEE is loaded in the secure world and Linux in the non-secure world.

11. Start the OP-TEE client and run the self test.
The OP-TEE client can be started with the command:
`tee-supplicant &`
To check if OP-TEE is initialized and started correctly, the self test can be run:
`xtest` The system is now ready to load and execute a TA.

5.8.2.2. Setup Toolchain for Trusted Application (TA)

After OP-TEE is configured and running, a TA can be loaded and executed. The OP-TEE toolchain has to be set up to build a TA. As OP-TEE is builed with PetaLinux, the toolchain was not needed for this step.

The following script is an example of how the OP-TEE toolchain can be installed: [40]

```
1  mkdir -p $HOME/toolchains
2  cd $HOME/toolchains
3
4  # Download 32bit toolchain
5  wget https://developer.arm.com/-/media/Files/downloads/gnu-a
       /8.2-2019.01/gcc-arm-8.2-2019.01-x86_64-arm-linux-gnueabi.tar.xz
6  mkdir aarch32
7  tar xf gcc-arm-8.2-2019.01-x86_64-arm-linux-gnueabi.tar.xz -C aarch32 --
       strip-components=1
8
9  # Download 64bit toolchain
10 wget https://developer.arm.com/-/media/Files/downloads/gnu-a
       /8.2-2019.01/gcc-arm-8.2-2019.01-x86_64-aarch64-linux-gnu.tar.xz
11 mkdir aarch64
12 tar xf gcc-arm-8.2-2019.01-x86_64-aarch64-linux-gnu.tar.xz -C aarch64 --
       strip-components=1
```

It can be installed at any path on the system, it just needs to be sourced before the next steps:

`export PATH=$PATH:$HOME/toolchains/aarch32/bin:$HOME/toolchains/aarch64/bin` [40]

Clone the OP-TEE OS repository and build it:

`git clone https://github.com/OP-TEE/optee_os.git`

```
make CFG_ARM64_core=y CFG_TEE_BENCHMARK=n CFG_TEE_CORE_LOG_LEVEL=3
CROSS_COMPILE=aarch64-linux-gnu- CROSS_COMPILE_core=aarch64-linux-gnu-
CROSS_COMPILE_ta_arm32=arm-linux-gnueabi- CROSS_COMPILE_ta_arm64=aarch64-linux-gnu-
DEBUG=1 O=out/arm PLATFORM=zynqmp
```

Clone the OP-TEE client repository and built it:

```
git clone https://github.com/OP-TEE/optee_client.git
```

```
make CROSS_COMPILE=aarch64-linux-gnu-
```

5.8.2.3. Compile a Trusted Application (TA)

The Git repository https://github.com/linaro-swg/optee_examples contains a number of different example TAs. The examples consist of a `host` and a `ta` part. The `ta` part is the TA, which is running in the secure world, while the `host` part is the client application running in Linux, which communicate with the TA.

To prohibit the loading of a malicious TA, the TAs are signed. OP-TEE supports only a hardcoded key. The OP-TEE OS has a folder `keys` that contains the key, which is compiled into OP-TEE. During the build, the TA is automatically signed. The default key should be changed if the software is used in production.

Switch to the `host` folder of an example and run the command to compile the `host` part:

```
make CROSS_COMPILE=aarch64-linux-gnu- TEEC_EXPORT=<optee_client>/out/export/usr
-no-builtin-variables
```

Change to the `ta` folder to compile the TA.

```
make CROSS_COMPILE=arm-linux-gnueabi- PLATFORM=zynqmp
TA_DEV_KIT_DIR=<optee_os>/out/arm/export-ta_arm32
```

To execute the TA, copy the new binary inside the `host` folder and the `*.ta` file from the `ta` folder to the Zynq Ultrascale+. Move the `*.ta` file to the folder `/lib/optee_armtz/` and execute the binary.

6. Results

This chapter shows the outcome of the implementation. A goal of this work was to show a proof of concept, that the different implementations work as expected. In the following sections, the procedure to test and verify the various implementations are shown and discussed.

6.1. Secure Boot

To test secure boot, the `RSA_EN` had to be burned. Else the authentication could not have been tested.

In order to prove that secure boot is working following statements have to be confirmed. The statements target things, which are only stored inside the device (either eFuse or BBRAM). Changing a SPK for example, would not affect secure boot, because the bootgen generates all components for the authentication and none of them stored in the eFuse.

- An image with an incorrect PPK key should stop booting before the FSBL.
- An image with an invalid `SPK_ID` should boot until the image partition with the invalid `SPK_ID` is authenticated. Then it should go to secure lock down.
- An image with an incorrect AES master key should stop before the FSBL.
- An image with correct keys and valid `SPK_ID` should boot normally.

The complete output of a functional image can be found in the file

`BA20_rosn_02_Secure_Boot/Software/petalinux_prj/testing/normal/out.txt`

on the Git project and can be rebuilt with the current configuration of the PetaLinux project. The following sections describe the other three statements.

6.1.1. Image Authentication/SPK Revocation

The PPK can be changed by regenerating a new key and then generate the new image using the new key. When booting this image, the device outputs nothing, because the first output is generated during the execution of the FSBL. This output is exactly what we expected. Because the PPK in the boot header was changed with another key, the FSBL nor the boot header can be authenticated. The comparison of both hashes fails because they are not from the same key.

If the `SPK_ID` is changed on one partition of the image, the image boots fine until the previously adapted partition is loaded. In this case, the change can be quickly done in the `.bif` file. The output looks like booting a correct image until the boot flow stops and an error seen Listing 6.1 is shown.

```
428 Ppk ModularEx END
429 Ppk Exp = 1000100
430 Image's SPK ID : FFFDD2FC
431 eFUSE SPK ID: 46857465
432 XFsbl_SpkVer: XFSBL_ERROR_SPKID_VERIFICATION
433 XFSBL_ERROR_BITSTREAM_AUTHENTICATION
434 Partition 4 Load Failed, 0x66
435 ================= In Stage Err ============
436 Fsbl Error Status: 0x0
```

Listing 6.1: Console output of a boot image, when the `SPK_ID` is changed

In this case the SPK_ID of the bitstream was changed. The output clearly shows that the ID verification failed and the device goes to secure lockdown. The device can only get out of the secure lockdown by rebooting via a POR reset. This example also shows that the functionality of the SPK key revocation feature is working. The revocation feature for PPK with the user fuses was omitted, because it would only cause to burn unnecessary eFuse registers.

6.1.2. Image Decryption

To test decryption, the AES master key was changed, and a new image was generated. Same as when the authentication fails, the device outputs nothing. This output makes sense, because, with the wrong key, the decrypted data is unusable and therefore, no FSBL can be started, which results in no output to the user.

6.2. Multiboot

Multiboot will be activated in case an image is corrupted. Therefore, the device needs to images in the same storage. If the first one can be booted, everything should proceed with the first image. In case it is corrupt, the multiboot register has to jump to the position from the second image and proceed with the boot flow. QSPI has been selected in the test setup, which can be seen in line 10 of Listing 6.2. One image was placed in the memory starting at position 0x0000000. The second image was placed at the starting position 0x1000000. Listing 6.2 shows the beginning of an output using an intact image.

```
1  Xilinx Zynq MP First Stage Boot Loader
2  Release 2019.2   Jul  7 2020  -  13:01:40
3  Reset Mode      :       System Reset
4  Platform: Silicon (4.0), Cluster ID 0x80000000
5  Running on A53-0 (64-bit) Processor, Device Name: XCZU5EV
6  Initializing DDR ECC
7  Address 0x0, Length 80000000, ECC initialized
8  Processor Initialization Done
9  ================= In Stage 2 ============
10 QSPI 32 bit Boot Mode
11 QSPI is in single flash connection
12 QSPI is using 4 bit bus
13 FlashID=0x1 0x2 0x20
14 SPANSION 512M Bits
15 Multiboot Reg : 0x0
16 QSPI Reading Src 0x0, Dest FFFF1C40, Length EC0
17 ...
```

Listing 6.2: Beginning section of the console output with an intact image

On line 15, we see, that the multiboot register is set to 0x0 which results in the reading address of 0x0, seen on the following line 16. Therefore, this image boots up as expected.

To simulate a corrupt image, we simply overwrote or erased some memory sections between 0x0000000 and 0x1000000. If the device is restarted, a minor delay in the boot flow can be determined. Listing 6.3 shows the output.

```
1  Xilinx Zynq MP First Stage Boot Loader
2  Release 2019.2   Jul  6 2020  -  16:14:17
3  Reset Mode      :       System Reset
4  Platform: Silicon (4.0), Cluster ID 0x80000000
5  Running on A53-0 (64-bit) Processor, Device Name: XCZU5EV
6  Initializing DDR ECC
7  Address 0x0, Length 80000000, ECC initialized
```

```
8  Processor Initialization Done
9  ================= In Stage 2 ============
10 QSPI 32 bit Boot Mode
11 QSPI is in single flash connection
12 QSPI is using 4 bit bus
13 FlashID=0x1 0x2 0x20
14 SPANSION 512M Bits
15 Multiboot Reg : 0x200
16 QSPI Reading Src 0x1000000, Dest FFFF1C40, Length EC0
```

Listing 6.3: Beginning of the console output with a corrupt image

Here the boot storage is also QSPI. However, line 15 shows that the multiboot register has been increased to `0x200`, resulting in the new reading address `0x1000000`. If the multiboot register is multiplied with the step size of `0x800`, the result is precisely `0x1000000`, which can also be seen in line 16.

6.3. Update Process

The following points have to be verified to ensure a working update procedure. First, the image has to be written to the correct memory section and has to be boot-able. Second, the image has to be checked in size. Else the image might overwrite the following memory section. In the worst case, the following image is the golden image. At last, the update procedure should also be callable from within the golden image in case the first image is corrupted and needs to be overwritten. In this project, the golden and the regular image are the same. Therefore, the last point can be neglected.

To verify the functionality following procedure was played through. Like in the previous section, the first image, starting in the memory at position `0x0000000`, was changed to be corrupt and thus the multiboot register would jump to the next image. If the image is updated correctly, the multiboot register has to be `0x0`, because the corrupted image was updated with an intact one.

Listing 6.4 shows the beginning of the console output before and Listing 6.5 after the update was made.

```
1  Xilinx Zynq MP First Stage Boot Loader
2  Release 2019.2   Jul  8 2020  -  11:12:30
3  Reset Mode      :       System Reset
4  Platform: Silicon (4.0), Cluster ID 0x80000000
5  Running on A53-0 (64-bit) Processor, Device Name: XCZU5EV
6  Initializing DDR ECC
7  Address 0x0, Length 80000000, ECC initialized
8  Processor Initialization Done
9  ================= In Stage 2 ============
10 QSPI 32 bit Boot Mode
11 QSPI is in single flash connection
12 QSPI is using 4 bit bus
13 FlashID=0x1 0x2 0x20
14 SPANSION 512M Bits
15 Multiboot Reg : 0x200
16 QSPI Reading Src 0x1000000, Dest FFFF1C40, Length EC0
17 ...
```

Listing 6.4: Beginning of console output before the update

```
1  Xilinx Zynq MP First Stage Boot Loader
2  Release 2019.2   Jul  8 2020  -  11:12:30
3  Reset Mode      :       System Reset
4  Platform: Silicon (4.0), Cluster ID 0x80000000
5  Running on A53-0 (64-bit) Processor, Device Name: XCZU5EV
```

```
6   Initializing DDR ECC
7   Address 0x0, Length 80000000, ECC initialized
8   Processor Initialization Done
9   ================= In Stage 2 ============
10  QSPI 32 bit Boot Mode
11  QSPI is in single flash connection
12  QSPI is using 4 bit bus
13  FlashID=0x1 0x2 0x20
14  SPANSION 512M Bits
15  Multiboot Reg : 0x0
16  QSPI Reading Src 0x0, Dest FFFF1C40, Length EC0
17  ...
```

Listing 6.5: Beginning of the console output after the update

The output in Listing 6.4 shows that before the update, the device boots using the golden image. However, after the update Listing 6.5 shows that the first image beginning in the memory section `0x0000000` is booted. Therefore, the update was successful.

The last point to check is if images of too big size are rejected. To test that a file with the size of `0x1000000` + `0x1` has been generated. For this the command `truncate` was used.

```
1   truncate -s 16777217 toobigfile.bin
```

Listing 6.6: Generate a file with a certain size

The console output of the update script in Listing 6.7 shows that it handles too big images correctly and rejects them.

```
1   root@mercuryxu5:~# ./update_img.sh toobig.bin
2   Image to big!! Leaving...
3   root@mercuryxu5:~#
```

Listing 6.7: Console output when updating with a too big image

In contrast to the example before, Listing 6.8 shows the console output of a successful update.

```
1   root@mercuryxu5:~# ./update_img.sh QSPI.bin
2   Erasing blocks: 35/35 (100%)
3   Writing data: 8774k/8774k (100%)
4   Verifying data: 8774k/8774k (100%)
5   root@mercuryxu5:~#
```

Listing 6.8: Console output of a successful update

6.4. Tamper Monitoring Unit

The tamper monitoring unit can be tested in multiple ways. However, due to the lack of equipment, only three cases were tested.

6.4.1. Manual Tamper Event Trigger

The first case to be tested was the manual tamper event trigger. This trigger can be set by writing to the `csu_tamper_trig` register in the CSU. This register is meant for cases, where the processes have to be finished before an emergency shutdown. Thus, an occurring tamper event does not directly lock down the device but sets an interrupt, which is then handled by user software. During this interrupt, critical tasks can be processed, before setting the tamper trigger register, to lock the device. This bit can be set through the sysfs interface provided by Linux for the CSU and PMU registers. The interface is explained in the Software Developers Guide [13] on page 153. Listing 6.9 shows the output.

```
root@mercuryxu5:~# echo 0xffca0014 0x1 0x1 > /sys/firmware/zynqmp/
    config_reg
```

Listing 6.9: Manually setting the `csu_tamper_trig` register

The console output shows, that right after the writing the `csu_tamper_trig` register, the device goes to secure lockdown and does not respond to any inputs, nor gives any further outputs.

6.4.2. JTAG tamper Event

The second test case was the reaction to JTAG inputs. Figure 6.1 shows the different pin assignments of the JTAG connector. Triggering the JTAG tamper event can be achieved by toggling the pins on the right side (TMS, TCLK, TDO, TDI) with the VCC pin.

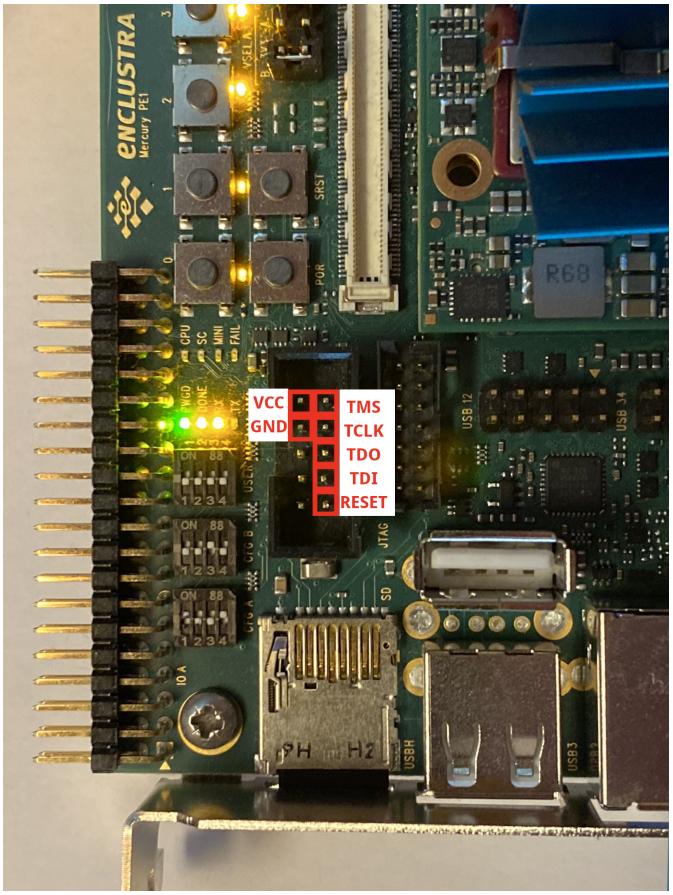


Figure 6.1.: JTAG pins on the Mercury board

The best way to observe the system going to secure lockdown is by running a program, which continuously outputs something to the console. One possibility, which we used, is by running the `read_temp` program, which will be explained in the last test case of the tamper monitoring unit. This program continuously reads and prints out the chip temperature. When one of the pins is toggled, the program stops immediately and the output freezes. In this state, no input will be accepted, until the device is restarted using a POR reset. This device reaction shows, that the JTAG tamper event is working.

6.4.3. Temperature Tamper Event

At last, we wanted to test if it is possible to allow the device to only run in a certain given temperature range. Outside this temperature range, the device should trigger a tamper event and go straight to secure lockdown. For testing purposes, the temperature range was set between 35 and 85°C, because during the run time the device runs at about 40°C. Besides, it is easier to heat the device than to cool it down.

What one would suggest on a cold boot (when the device boots up after a long time it was not running) is, that the device should not even start up. However, instead, the system boots up until U-Boot starts Linux. There the device holds on and waits until the device reaches 35°C and proceeds further with the boot process. The console output is showed in Listing 6.10 and Listing 6.11.

```
1200 [    3.731439] console [ttyPS0] enabled
1201 [    3.735038] bootconsole [cdns0] disabled
1202 [    3.735038] bootconsole [cdns0] disabled
1203 [   21.585288] GPIO IRQ not connected
1204 [   21.588683] XGpio: gpio@80000000: registered, base is 504
1205 [   21.594459] of-fpga-region fpga-full: FPGA Region probed
```

Listing 6.10: Console output during a cold boot

```
1379 [    3.731439] console [ttyPS0] enabled
1380 [    3.735038] bootconsole [cdns0] disabled
1381 [    3.735038] bootconsole [cdns0] disabled
1382 [   21.585288] GPIO IRQ not connected
1383 [   21.588683] XGpio: gpio@80000000: registered, base is 504
1384 [   21.594459] of-fpga-region fpga-full: FPGA Region probed
```

Listing 6.11: Temperature after the cold boot

Listing 6.10 shows that between line number 1202 and 1203, the device is waiting for something. Compared to a warm boot, the device is booted much faster. Listing 6.12 shows the console output of a warm boot as a comparison.

```
1200 [    3.730510] console [ttyPS0] enabled
1201 [    3.734111] bootconsole [cdns0] disabled
1202 [    3.734111] bootconsole [cdns0] disabled
1203 [    3.742091] GPIO IRQ not connected
1204 [    3.748841] XGpio: gpio@80000000: registered, base is 504
1205 [    3.754619] of-fpga-region fpga-full: FPGA Region probed
```

Listing 6.12: Console output during a warm boot

Listing 6.11 shows, that the device is just a little above 35°C after it started, which enforces the previously explained theory. Nevertheless, not only that is problematic, but also if the temperature is decreased after Linux is booted, the system does not react accordingly. Listing 6.13 shows, that if the temperature is decreased below the threshold of 35°C, the system does not trigger a tamper event.

```
1  Temperature: 35.516     C
2  Temperature: 36.448     C
3  Temperature: 36.215     C
4  Temperature: 35.967     C
5  Temperature: 34.925     C
6  Temperature: 34.910     C
7  Temperature: 35.780     C
8  Temperature: 34.987     C
9  Temperature: 35.314     C
10 Temperature: 35.811     C
11 Temperature: 34.832     C
12 Temperature: 34.879     C
```

```
13   Temperature: 36.464     C
14   Temperature: 34.039     C
15   Temperature: 34.816     C
16   Temperature: 35.998     C
17   Temperature: 34.506     C
18   Temperature: 35.500     C
19   Temperature: 34.552     C
20   Temperature: 33.044     C
21   Temperature: 34.552     C
22   Temperature: 34.086     C
23   Temperature: 35.034     C
24   Temperature: 34.552     C
25   Temperature: 32.578     C
```

Listing 6.13: Console output when the temperature is decreased within Linux

Also, for temperature increase, the device does not trigger a tamper event. To solve this problem, we asked in the Xilinx forum, about solutions, known problems or possible misunderstandings from our side. After one month, a Xilinx employee answered, that using the system management wizard from Vivado requires the bitstream to be loaded. Therefore, the tamper unit is not ready after the FSBL, but rater after the bitstream is loaded. He also suggested checking the AMS registers, if the temperature threshold values were set up correctly. After looking into the AMS registers, we saw, that the threshold values were not set. Therefore, the temperature alarm can not function because it needs the threshold value for comparison. The alarm will only be set when the measured value falls below the threshold value [36]. A possible solution is to set the values in the AMS registers during the execution of the FSBL.

6.5. Crypto API

The Linux Crypto API can be used to hash or encrypt data with the crypto hardware of the Zynq Ultrascale+.

To test the output of the Linux Crypto API, tools like sha256sum or OpenSSL are used to compare the output of the examples.

6.5.1. Verify SHA-256

The command line tool `sha256sum` calculates the SHA-256 hash of a file. The example for SHA-256 calculates the hash of the string `"abc"`. The hash for the same string can be calculated with `sha256sum` to compare it with the Linux Crypto API implementation.

```
1 $ ./crypto-api-sha256-aarch64
2 edeaaff3f1774ad2888673770c6d64097e391bc362d7d6fb34982ddf0efd18cb
3 $ echo "abc" > test-sha256.txt
4 $ sha256sum test-sha256.txt
5 edeaaff3f1774ad2888673770c6d64097e391bc362d7d6fb34982ddf0efd18cb  test-
      sha256.txt
```

Listing 6.14: Verify SHA-256 Implementation

6.5.2. Verify SHA3-384

The tool `openssl` can calculate SHA-3 hashes and is therefore used to test the SHA3-384 implementation. The example calculates the hash of the string `"abc"`, with OpenSSL the same hash is calculated for comparison.

```
$ ./crypto-api-sha3-384-aarch64
b727220940c0621e022627c1ff2f577b152f9873fbf33a4de9e5b8110d5bac847ec6
    cbbe6c54c523e8b9f33629f7aa04
$ echo "abc" > test-sha3.txt
$ openssl dgst -sha3-384 test-sha3.txt
SHA3-384(test-sha3.txt)=
    b727220940c0621e022627c1ff2f577b152f9873fbf33a4de9e5b8110d5bac847ec6
    cbbe6c54c523e8b9f33629f7aa04
```

Listing 6.15: Verify SHA3-384 Implementation

6.5.3. Verify AES CBC

The example for AES CBC can be tested with `openssl`. The same key and IV that are used in the example have to be passed to `openssl`. The output of `hexdump` has a different format as the example program, which makes it difficult to compare, however the outputs are the same.

```
$ ./crypto-api-aes-cbc-aarch64
\x7c\x4b\xae\xd3\x4e\x4f\x34\x2e\x44\xfe\x2c\x54\xfd\x77\x83\xf5\xeb\x99
    \x45\xd2\x09\x09\xae\xba\x0c\x32\xd4\x04\x29\x6d\x37\xe8
$ echo "This is a 32 byte message. abcd" > test-aes256.txt
$ openssl enc -aes-256-cbc -e -in test-aes256.txt -out test-aes256.enc -
    K 3031323334353637303132333435363730313233343536373031323334353637 -
    iv 20212223242526272021222324252627
$ hexdump test-aes256.enc
0000000 4b7c d3ae 4f4e 2e34 fe44 542c 77fd f583
0000010 99eb d245 0909 baae 320c 04d4 6d29 e837
0000020
```

Listing 6.16: Verify AES CBC Implementation

6.6. TrustZone

The implementation of TrustZone shows a working example for OP-TEE on the zcu102 board from Xilinx. The self-test from OP-TEE completes successful and it is possible to load and execute Trusted Application (TA)s. An application running under Linux can communicate with the TA in the secure world. However, this implementation has several open points:

1. **PetaLinux version**
   The version of PetaLinux used for the example project is 2018.2. It does not work with the newer version 2019.2. If the version check is removed, it compiles successful but hangs during boot.
2. **OP-TEE version**
   The example for the Zynq Ultrascale+ was made at the beginning of 2019, since then OP-TEE received many updates, and the example project no longer works. This example uses the OP-TEE version when the example was created and is missing possible security updates, which were added in the meantime.
3. **Different Boards**
   The OP-TEE example has only been successfully tested on the zcu102 board from Xilinx. If the project is modified for the Mercury board from Enclustra, it stops during the boot of Linux.

7. Conclusion

In this work, we followed through the four steps of the Platform Security Architecture (PSA) framework. The PSA framework was analyzed to understand the tools before working with them. Besides, the STRIDE model has also been analyzed and adapted to evaluate hardware components like the Zynq Ultrascale+. The Zynq Ultrascale+ is a System on Chip (SoC) with a multitude of security features. Most of them have been analyzed and implemented, which lead to a modular reference design. This reference design helps developers to start developing a tailored solution for their secure product.

Following the first step of the PSA framework, the following three use-cases were analyzed. The first use-case is how to develop a secure product on the Zynq Ultrascale+. The second use case deals with how images can be updated and leaked keys revoked. The third use-case is how to update the base-board on the Zynq Ultrascale+ while maintaining security.

However identifying threads was much harder, due to the lack of a real product with real goals and requirements. While the PSA framework was designed for exactly such cases, where product, goals and requirements are known, we had to think much more generally. Therefore, we also analyzed the hardware of the product, to give insight on which cases or product ideas security issues will arise. Thus, a product developer can quickly see what security issues he has to counter using the Zynq Ultrascale+.

The next step of the PSA framework was to analyze the features. Using the Zynq Ultrascale+, we analyzed its features and showed processes, security features and safety functions provided. The implementation of "root of trust", with the RSA and AES cores, show that secure product development is the way to go. Additional, features like key revocation and multiboot enhance maintainability and reliability of the product. With features like TrustZone and the tamper monitoring unit, more tools are given, to maintain security during runtime. Besides, the crypto hardware used for the secure boot can also be used with the Linux Crypto API, which provides a standardized and platform-independent interface to cryptography hardware.

The various features have been implemented to provide a reference design at the end of this project. The implementation consists of different modules, which can be merged, depending on the use-case. The base of this reference design is a PetaLinux project. The project has implemented support for both, encryption and authentication. Additionally, different modules can be added for other features like tamper detection or multiboot. This modules mostly consist of additional examples and descriptions on how to implement the feature.

Additionally, a module with examples on how to use the Linux Crypto API with different hardware and algorithms has been created. It shows how the Linux Crypto API works and how it can be used. On usual hardware, this knowledge has limited usage, because other tools like OpenSSL already use the Linux Crypto API and provide further functionalities, like key generation. However, the the crypto hardware of the Zynq Ultrascale+ is not supported by tools like OpenSSL. Therefore, it is required to know the Linux Crypto API to use the hardened cores in Linux.

Isolation of different parts in a system is necessary for applications, which handle confidential data. Usually the OS isolates the different applications from each other, but with a wide variety of functions OSs offer a large attack surface for privilege escalation. TrustZone isolates critical applications from the rest of the system. Thus, even the normal OS has no access to memory and peripherals assigned to secure applications. These secure applications run in the secure world, while standard applications run in the non-secure world. Although TrustZone is a helpful feature, it is not enough to just enable it. The application has to be redesigned around the concept of TrustZone. Critical parts have to be moved to the secure world, and an interface between the secure and non-secure world has to be designed. TrustZone itself is a vast topic. Therefore, we implemented Open Trusted Execution Environment (OP-TEE), an open-source program built on TrustZone, which implements a Trusted

Execution Environment (TEE) for TrustZone. Many problems have been encountered during the implementation. Even though most of them could be solved, OP-TEE could only run on the evaluation kit `zcu102` from Xilinx. The implementation shows how to solve most problems, but still, some remain.

Finally, the implementation has been tested to prove the concept. We could show that most of the features are functional. Nevertheless, some feature proofed to be more complicated and could not be implemented entirely. Those features were the tamper monitoring unit, where the temperature tamper event did not work and the Linux Crypto API, where some features of the AES GCM hardware, could not be found. No documentation about the usage and implementation of the AES GCM hardware driver was found. Therefore, the features had to be discovered, by analyzing code and functionality of the tool. Thus, some featured remained undiscovered. Finally, OP-TEE could not be fixed, to run on the mercury board.

To conclude, this thesis shows that the PSA framework was developed to be applied to a specific product. Though with some adjustments, it can also be used to analyze more general applications. The analysis of the features provided in the Zynq Ultrascale+ showed that the device is built for security. The Zynq Ultrascale+ provides also features to maintain security over the whole life-cycle of the product. Most of the features are easy and straightforward to implement. To further ease the development process, a reference design has been created. With the modular design of the reference project, it surely gives developers a head start developing their secure product.

7.1. Future Work

As pointed out in the conclusion, not all features could be elaborated as desired. The following sections provide a recommendation for future work on different features.

Tamper Monitoring Unit Only three tamper events have been tested during this thesis. Additionally, other tamper events could also be tested. However, more important would be to implement a function in the FSBL, which writes the AMS registers. Thus, the tamper and system monitoring unit could be configurated when the FSBL is executed. Maybe, this approach also solves the confronted issues with the temperature tamper event.

Linux Crypto API Some functions from the GCM mode of AES like associated data have not yet been detected or not been implemented in the Linux Crypto API driver. Therefore, the feature has to be found or implemented. Maybe in the future, Xilinx will provide a description for their Linux Crypto API implementation of the AES hardened core.

TrustZone The implementation for OP-TEE was only running on the evaluation kit (zcu102) from Xilinx, with an old version of PetaLinux and OP-TEE. The example provided from the OP-TEE project, could be adapted to neither the new version of PetaLinux, OP-TEE, nor to the mercury board used during the thesis. With further researches, a reason and a solution could be found for this problem. A good starting point is the Documentation for OP-TEE [29]. Optimally, board support for the mercury XU5 could be added to the OP-TEE Git repository [41] and maintained.

7.2. Reflection

This thesis enabled us to explore and analyze the security systems used in the Zynq Ultrascale+. It was beneficial that we carried out a similar project on the Zynq-7000 in the previous semester. Thus, knowledge from the previous project could be carried over and expanded.

It was interesting to work with the workflow according to the PSA framework. It provides a structured approach to secure development in the industry. After gaining experience with it, we are convinced that this workflow has the potential for future use. As newly trained engineers, it is our task to bring such workflows into the industrial companies.

Analyzing, implementing and testing the different security features of the Zynq Ultrascale+ allowed us to gain an understanding of how security features work, and how they are implemented. Especially, know-how on features like TrustZone, the Linux Crypto API or simply with U-Boot and Linux throughout this work, will persist even if the chip will change. Thus, this work serves as a knowledge base for future work in security development.

The most challenging part of this work was to organize and process the large amount of information to establish a good overview of all features. Besides, working with big projects like U-Boot and OP-TEE was also challenging. Especially when errors occurred, it was difficult to search for them, because of the amount of code and the unknown processes. This challenge taught us how to get an overview over a big project quickly and how to trace down processes for potential issues.

Finally, we can say that we are pleased with our work. Some task remained unfinished, which is always a drawback. Nevertheless, we were able to handle more than thought.

Bibliography

[1] Wolfgang Schwab and Mathieu Poujol. The state of industrial cyber security 2018. Technical report, June 2018.

[2] Xilinx. *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*, August 2019.

[3] Symantec. *ISTR Internet Security Threat Report*, February 2019.

[4] ENISA. *ENISA Threat Landscape Report 2018*, January 2019.

[5] Arm Limited. *Platform Security Architecture Overview White Paper*, June 2019.

[6] Suresh Marisetty. Five steps to successful threat modelling. Technical report, 2019. Accessed: 26.02.2020.

[7] Internet of things (iot) security architecture. `https://docs.microsoft.com/en-us/azure/iot-fundamentals/iot-security-architecture`, October 2018. Accessed: 23.02.2020.

[8] Michale Vai, David J. Whelian, Benjamin R. Nahill, Daniil M. Utin, Sean R. O'Melia, and Roger I. Khazan. Secure embedded systems. *Lincoln Laboratory Journal*, 22(1), 2016.

[9] Tobias Schläpfer. Embedded security frameworks for iot devices. Technical report, Institute of Embedded Systems (InES), 2020.

[10] ENISA. *ENISA Threat Landscape Report 2015*, January 2016.

[11] Peter Berlich, Stephan Neuhaus, Marc Rennhard, and Bernhard Tellenbach. Security requirements engineering and threat modeling. 2019.

[12] Zynq ultrascale+ xilinx webpage. `https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html#productAdvantages`. Accessed: 08.07.2020.

[13] Xilinx. *Zynq UltraScale+ MPSoC Software Developer Guide (UG1137)*, December 2019.

[14] Dietmar Wätjen. *Kryptographie*. Springer Vieweg, Braunschweig, Deutschland, 2018.

[15] NIST, FIPS. *Advanced Encryption Standardd (AES) (FIPS PUB 197)*, November 2001.

[16] Openssl documentation. `https://www.openssl.org/docs/`. Accessed: 02.04.2020.

[17] NIST. *Recommendation for Block Cipher Modes of Operation (NIST PUB 800-38A)*, December 2001.

[18] Block cipher mode of operation. `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation`. Accessed: 02.04.2020.

[19] NIST. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC (NIST PUB 800-38D)*, Novemebr 2007.

[20] Xilinx. *Xilinx Standalone Library Documentation (UG643)*, Decemeber 2019.

[21] Xilinx. *Developing Tamper-Resistant Designs with Zynq UltraScale+ Devices (XAPP1323)*, August 2018.

[22] Bundesamt für Sicherheit in der Informationstechnik. *Kryptographische Verahren: Empfehlungen und Schlüssellängen (BSI TR-02102-1)*, March 2020.

[23] Xilinx wiki zynq ultrascale+ mpsoc security features. `https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841708/Zynq+Ultrascale+MPSoC+Security+Features?focusedCommentId=100007939#ZynqUltrascale+MPSoCSecurityFeatures-RSASecondaryPublicKey(SPK)Revocation`. Accessed: 03.04.2020.

[24] Linux kernel crypto api documentation. `https://www.kernel.org/doc/html/latest/crypto/index.html`. Accessed: 14.07.2020.

[25] Xilinx. *PetaLinux Tools Documentation Reference Guide (UG1144)*, Oct 2019.

[26] ARM. *ARM Cortex-A53 MPCore Processor Cryptography Extension*, 2014.

[27] Xilinx. *Isolation Methods in Zynq UltraScale+ MPSoCs (XAPP1320)*, June 2019.

[28] Trusted firmware-a documentation. `https://trustedfirmware-a.readthedocs.io/en/latest/index.html`. Accessed: 14.07.2020.

[29] Op-tee documentation. `https://optee.readthedocs.io/en/latest/index.html`. Accessed: 13.07.2020.

[30] Enclustra downloads. `https://download.enclustra.com/`. Accessed: 11.05.2020.

[31] Xilinx. *PetaLinux Tools Documentation Command Line Reference Guide (UG1157)*, Oct 2019.

[32] Enclustra. *Mercury XU5 SoC Module Reference Design for Mercury+ PE1 Base Board User Manual*, September 2018.

[33] Xilinx. *Bootgen User Guide (UG1283)*, October 2019.

[34] Vivado design suite download. `https://www.xilinx.com/support/download.html`. Accessed: 13.05.2020.

[35] Xilinx. *Programming BBRAM and eFUSEs (XAPP1319)*, July 2017.

[36] Zynq ultrascale+ devices register reference (ug1087). `https://www.xilinx.com/html_docs/registers/ug1087/ug1087-zynq-ultrascale-registers.html`, February 2019. Accessed: 23.02.2020.

[37] Xilinx wiki zynq ultrascale+ mpsocams driver. `https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842163/Zynq+UltraScale+MPSoC+AMS`. Accessed: 04.07.2020.

[38] Xilinx wiki zynq ultrascale+ mpsoc power management - linux kernel. `https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842232/Zynq+UltraScale+MPSoC+Power+Management+-+Linux+Kernel`. Accessed: 09.07.2020.

[39] Xilinx wiki: Build arm trusted firmware (atf). `https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842305/Build+ARM+Trusted+Firmware+ATF`. Accessed: 13.07.2020.

[40] Op-tee toolchain. `https://optee.readthedocs.io/en/latest/building/toolchains.html`. Accessed: 14.07.2020.

[41] Op-tee github repository. `https://github.com/OP-TEE`. Accessed: 15.07.2020.

List of Figures

List of Tables

Acronyms

AEAD Authenticated Encryption with Associated Data. 27, 59

AES Advanced Encryption Standard. 2, 3, 17, 18, 20–28, 30, 43, 46–48, 53–60, 66, 67, 73–75, 80, 86

AMS Analog Monitor System. 52, 72, 75

APU Application Processing Unit. 2, 13

ATF Arm® Trusted Firmware. 2, 23, 38, 41, 60, 61, 63

BBRAM Battery Backup RAM. 2, 17, 18, 20, 28–30, 47, 48, 51, 66

BSP Board Support Package. 62

CBC Cipher Block Chaining Mode. 2, 25–27, 54–57, 59, 60, 73, 79, 80, 86

CIA Confidentiality Integrity Availability. 5

CPU Central Processing Unit. 9

CRC Cyclic Redundancy Check. 34, 48

CSU Configuration Security Unit. 2, 13, 21–23, 28, 30, 31, 52, 69

CVSS Common Vulnerability Scoring System. 8

DDoS Distributed-Denial-of-Service. 7

DDR Double Data Rate. 14, 22

DFD Data Flow Diagramm. 2, 5, 6, 8–13, 15, 18, 79, 80

DFT Design for Test. 48

DPA Differential Power Analysis. 25, 29, 30

DSP Digital Signal Processor. 21

ECB Electronic Codebook. 25, 26, 79

eFuse electronic Fuse. 2, 17, 18, 20, 22, 23, 28–32, 43, 45–48, 50, 51, 66, 67, 80

EL-0 Exception Level 0. 37

EL-1 Exception Level 1. 37, 38

EL-2 Exception Level 2. 37

EL-3 Exception Level 3. 23, 37

eMMC Embedded Multimedia Card. 14, 22

ENISA European Network and Information Security Agency. 4, 6, 7, 79

FAT File Allocation Table. 22

FPGA Field Programmable Gate Array. 2, 9, 21, 85

FSBL First Stage Boot Loader. 23, 34, 43, 45, 52, 66, 67, 72, 75

GCM Galois/Counter Mode. 17, 22, 25, 27, 28, 54, 56–60, 75, 80

GPU Graphical Processing Unit. 2, 21

HSM Hardware Security Module. 44

HVC Hypervisor Call. 37

ICS Industial Control Systems. 4, 7

IP Intelectual Property. 9, 13, 23

IPI Inter-Processor Interrupts and Communication. 39

IV Initialization Vector. 23, 26, 27, 47, 55, 57–59, 73

JTAG Joint Test Action Group. 2, 9, 12–14, 19, 22, 48, 51, 70, 79

KEK Key Encryption Key. 30

MITM Man in the Middle. 16

OCM On Chip Memory. 21, 23

OP-TEE Open Trusted Execution Environment. 2, 38, 41, 60–65, 73–76, 80

OS Operating System. 2, 36–38, 48, 52, 62, 64, 65, 74

PL Programmable Logic. 2

PMU Platform Management Unit. 2, 13, 21–23, 34, 69

PMUFW Platform Management Unit Firmware. 22, 23, 52

POR Power On Reset. 67, 70

PPK Primary Public Key. 17, 23, 31, 34, 43, 45–47, 51, 66, 67

PSA Platform Security Architecture. 3, 4, 74–76, 79

PUF Physical Unclonable Function. 28–30

QSPI Quad Serial Peripherial Interface. 22, 41, 48, 49, 67, 68, 86

REE Rich Execution Environment. 36

ROM Read Only Memory. 22, 23

RPU Realtime Processing Unit. 2, 13

SD Secure Digital. 14, 22

SDK Software Development Kit. 85

SDRAM Synchronous Dynamic Random Access Memory. 14

SHA Secure Hash Algorithm. 53, 54, 57, 60

SHA-1 Secure Hashing Algorithm 1. 32, 33

SHA-2 Secure Hashing Algorithm 2. 1, 32–34

SHA-256 Secure Hash Algorithm with 256 bits. 2, 54, 59, 60, 72, 86

SHA-3 Secure Hashing Algorithm 3. 1, 32–34, 72

SHA3-384 Secure Hash Algorithm 3 with 384 bits. 2, 54, 57, 59, 60, 72, 73, 86

SMC Secure Monitor Call. 37, 38, 52, 60

SoC System on Chip. 2, 21, 74

SPI Serial Peripherial Interface. 14

SPK Secondary Public Key. 2, 23, 32, 34, 47, 51, 66, 67

SQL Structured Query Language. 7

SSBL Second Stage Boot Loader. 2, 84

STRIDE Spoofing, Tampering, Repudiation, Information disclosure, Denial of Service, Elevation of privilege. 2, 8–10, 12, 74, 80

SVC Supervisor Call. 37

TA Trusted Application. 38, 41, 61, 64, 65, 73

TEE Trusted Execution Environment. 36, 74, 75

USB Universal Serial Bus. 22

XOR Exclusive OR. 26

Glossary

Arm® description. 36–38, 60, 61

Arm® Cortex™-A53 Application processor core from Arm®.. 2, 21, 28, 34–39

Arm® Cortex™-R5 Real-time processor core from Arm®.. 2, 21, 39

Arm® Cryptography Extension Hardware acceleration extension for Arm® Cortex A processors.. 2, 28, 34, 35, 57, 59, 60

Arm® Mali™-400 MP2 Graphics and multimedia processor core from Arm®.. 2, 21

bitstream A bitstream is the raw binary file which is programmed to the FPGA fabric. 23, 29, 30, 41, 42, 51, 52, 67, 72

bootgen Command Line Tool provided by Xilinx to generate Boot Images. 44–47, 66

GlobalPlatform TEE API API to communicate between applications running in the TEE and REE.. 38, 79

kernel space The kernel and it's modules are executed in the Kernel Space. Opposite of the User Space.. 35, 53

Linux Crypto API The Linux Crypto API gives an Interface to User Space programs to use hardware acceleration for cryptography functions.. 28, 31, 34, 35, 41, 53, 54, 56–60, 72, 74–76

OpenSSL Application to encrypt data. Supports a variety of algorithms.. 25, 44, 45, 72, 74

PetaLinux Command Line Tool provided by Xilinx to generate Linux Images. Based on the yocto project. 1, 41–44, 48, 59–64, 66, 73–75

RSA RSA is a asymmetrical cryptography procedure, which is used to sign data. The name comes from the first letter of the three inventors: Rivest, Shamir, Adleman. 1–3, 17, 18, 21, 23, 31, 41, 44, 45, 51, 74

Secure Monitor The Secure Monitor runs in EL-3 and has the highest access to the system. It controls the security features of the system.. 37, 38, 60

Secure OS OS which runs on execution level EL-1 in the secure world.. 36, 38

Secure Payload Dispatcher The Secure Payload Dispatcher is a part from the ATF to load a Secure OS.. 38

systemcall An application issues a systemcall to request something from the OS.. 37, 53–55

TrustZone TrustZone is a technology provided by Arm®, which enables secure execution of secure and non-secure software. 1, 36–39, 41, 60, 73–76

U-Boot An open Second Stage Boot Loader (SSBL). 2, 23, 48, 49, 71, 76, 86

user-space Programs in Linux run in the restricted User Space.. 34, 35, 53

Vivado Vivado is the main development Tool provided by Xilinx. It consists of a SDK and a FPGA Design Tool.. 41, 42, 44, 51, 72

Yocto The Yocto Project (YP) is an open source collaboration project that helps developers create custom Linux-based systems regardless of the hardware architecture. 1, 41, 42

Zynq Ultrascale+ Zynq Ultrascale+ is a SoC series from Xilinx.. 1–4, 12, 17–19, 21, 22, 28–31, 34–36, 38, 39, 41, 47, 57, 59–62, 64, 65, 72–76, 79

Listings

A. Appendix

A.1. `.bif` File Example

```
image : {

  /* Authentication declaration */
  [auth_params] ppk_select=0; spk_select=spk-efuse; spk_id=0x46857465
  [pskfile] keys/primary.pem
  [sskfile] keys/secondary0.pem

  /* Encryption declaration */
  [keysrc_encryption] bbram_red_key
  [fsbl_config] opt_key

  /* fsbl */
  [
    bootloader,
    destination_cpu         = a53-0,

    authentication          = rsa,
    spk_select              = spk-efuse,
    spk_id                  = 0x46857465,
    sskfile                 = keys/secondary0.pem,

    encryption              = aes,
    aeskeyfile              = keys/aes_p1.nky

  ] images/linux/zynqmp_fsbl.elf

  /* pmufw */
  [
    destination_cpu         = pmu
  ] images/linux/pmufw.elf

  /* bitstream */
  [
    destination_device      = pl,

    authentication          = rsa,
    spk_select              = spk-efuse,
    spk_id                  = 0x46857465,
    sskfile                 = keys/secondary1.pem,

    encryption              = aes,
    aeskeyfile              = keys/aes_p2.nky,
    blocks                  = 8192(*)

  ] images/linux/system.bit

  /* trusted firmware */
  [
    trustzone               = secure,
    exception_level         = el-3,
    destination_cpu         = a53-0,

```

```
53        authentication          = rsa,
54        spk_select              = spk-efuse,
55        spk_id                  = 0x46857465,
56        sskfile                 = keys/secondary2.pem,
57
58        encryption              = aes,
59        aeskeyfile              = keys/aes_p3.nky
60
61      ] images/linux/bl31.elf
62
63      /* u-boot */
64      [
65        exception_level         = el-2,
66        destination_cpu         = a53-0,
67
68        authentication          = rsa,
69        spk_select              = spk-efuse,
70        spk_id                  = 0x46857465,
71        sskfile                 = keys/secondary3.pem,
72
73        encryption              = aes,
74        aeskeyfile              = keys/aes_p4.nky
75
76      ] images/linux/u-boot.elf
77    }
```

Listing A.1: .bif file used in the project

A.2. Original Task Description

Bachelorarbeit Aufgabenstellung

BA20_rosn_02

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InES Institute of Embedded Systems

Studiengang	ET	Studenten	Thierry Delafontaine
Semester	FS 2020		Tobias Vögeli
Betreuer	Prof. Dr. Matthias Rosenthal Prof. Andreas Rüst Philipp Huber		
Ausgabetermin	10. Februar 2020	Abgabetermin	5. Juni 2020

Secure Boot für Ultrascale+ System on Chip

Themenbeschreibung

Nach der erfolgreichen Projektarbeit, welche einen sicheren Bootvorgang auf dem Zynq 7000 System-on-Chip untersuchte, soll in dieser Bachelorarbeit der Bootvorgang auf den neuen Xilinx Bausteinen der UltraScale+ Familie analysiert werden. Diese Systeme verfügen über eine Vielzahl neuer, integrierter Sicherheits-Features, welche verschiedenste sichere Anwendungen erlauben. In dieser Arbeit sollen diese vor allem für den Bootvorgang untersucht werden. Das Ziel ist ein Referenz-Design für die Firma Enclustra, welches optimal mit Enclustra Xilinx Modulen arbeitet.

Der Zynq Ultrascale+ enthält neben der FPGA Logik verschiedenste spezialisierte Prozessoren wie zum Beispiel eine Quad-Core-CPU (ARM Cortex A53), ein Realtime Prozessor (ARM Cortex-R5), eine GPU und eine eigene Security Unit.

Der normale Bootvorgang mit Linux auf den ARM Prozessoren ist nicht sicher, d.h. es werden keine Authentifizierungs- und Verschlüsselungsmechanismen verwendet, um die während des Bootvorgangs geladene Software zu überprüfen. Deshalb ist es notwendig, die folgenden potenziellen Bedrohungen für die Plattform zu berücksichtigen und das Gerät im Falle einer Bedrohung zu sichern:

1. Verhindern von Datendiebstahl aus dem Gerät (Datenschutz)
2. Verhindern des Klonen des Geräts
3. Verhindern von Denial-of-Service
4. Verhindern von Einfügen von Malware zur Änderung des Verhaltens des Geräts
5. Verhindern von Kompromittierung der sicheren Schlüssel

Ziel dieser Bachelorarbeit ist es, einen Bootmechanismus zu entwickeln, welcher die erwähnten Bedrohungen möglichst effizient verhindert. Dazu sollen die einzelnen Bootelemente (FSBL, Bootloader, Linux und Anwendung) während des Bootvorgangs authentifiziert und ver- respektive entschlüsselt werden. Eine weitere Schwierigkeit ist das Zusammenspiel der einzelnen Prozessoren (CPU, Realtime Prozessor, GPU), welches ebenfalls hohen Sicherheitsanforderungen genügen soll. Die Zynq Ultrascale+ Security Unit unterstützt diverse Authentifizierungs- (HMAC und RSA) und Verschlüsselungsfunktionen (AES), die ebenfalls untersucht und integriert werden sollen.
(siehe Übersicht auf Seite 257:
https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf)

Bachelorarbeit Aufgabenstellung

BA20_rosn_02

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InES Institute of Embedded Systems

Aufgabenstellung

a) Machen Sie sich mit der oben beschriebenen Thematik vertraut und arbeiten Sie sich in die Theorie des Bootvorgangs von Xilinx Ultrascale+ System-on-Chip ein.
b) Beschreiben Sie 2-3 grundlegende und konkrete Use-Cases mit den zugehörigen Aktoren, z.B. Code-Entwickler, Produktion, Anwender, etc. Use-Cases könnten sein: «Erstellen und freigeben einer neuen Firmware Version» und «Installation einer neuen Firmware durch Anwender»
c) Erstellen Sie ein Datenflowdiagramm (DFD) und eine zugehörige Bedrohungsanalyse (Threat Analysis) nach STRIDE (Unterlagen werden abgegeben). Leiten Sie daraus Security Requirements ab, welche Sie mit Ihrem Ultrascale+ Referenz Design erfüllen wollen.
d) Analysieren Sie grob die Anforderungen des Systems bezüglich Sicherheit und möglichen Angriffsszenarien. Untersuchen Sie dafür zuerst die einzelnen Bootsequenzen separat.
e) Untersuchen Sie die Ultrascale+ Security Unit und machen Sie sich ein Bild über deren Funktionen und Möglichkeiten.
f) Erstellen Sie ein Konzept für den Aufbau eines Prototypen mit dem zur Verfügung gestellten Xilinx Ultrascale+ Evaluationsboard und besprechen Sie dieses mit den Betreuern. Ziehen Sie auch mögliche externe Hardwareunterstützung in Betracht (zum Beispiel Secure-Elements)
g) Implementieren Sie das erarbeitete, sichere Bootkonzept soweit möglich auf dem Evaluationsboard und nehmen Sie das Gesamtsystem in Betrieb (Proof-of-Concept).
h) Bauen Sie eine einfache Testumgebung auf und zeigen Sie, warum die oben erwähnten Bedrohungen verhindert werden können. Zeigen Sie, dass/wie Ihr Referenzdesign die unter 2b) definierten Security Requirements erfüllt. (Es wird kein Penetration Test verlangt, aber zum Beispiel Tests mit falsch signierten Sourcen oder testen des Tamper Monitorings).
i) Testen und optimieren Sie das System.
j) Integrieren Sie die erarbeitete Testumgebung in einfach verwendbares Referenz-Design, welches als Open-Source Projekt auf GitHub zur Verfügung gestellt werden kann.

Reports, Bericht und Bewertung

Rechtzeitig vor der wöchentlichen Projektbesprechung ist ein kurzer Status-Report per Email an die Betreuer zu senden, mit Angaben zu folgenden 3 Punkten:

1. „Seit letzter Besprechung erledigt bzw. neue Resultate“,
2. „Bis zur nächsten Besprechung geplant“
3. „Probleme“.

Über die Bachelorarbeit ist ein Bericht zu schreiben. Der Bericht ist in 2-facher Ausführung (gebundene Papierversion und Link zu Repository) termingerecht abzugeben.

Halten Sie sich an das von der ZHAW vorgegebene Raster für die Strukturierung des Berichts.

Die Bewertung ergibt sich aus: Ingenieurfähigkeiten/Vorgehen (Gewicht 1/3), Resultate (Gewicht 1/3) und Bericht (Gewicht 1/3).

A.3. Time Chart

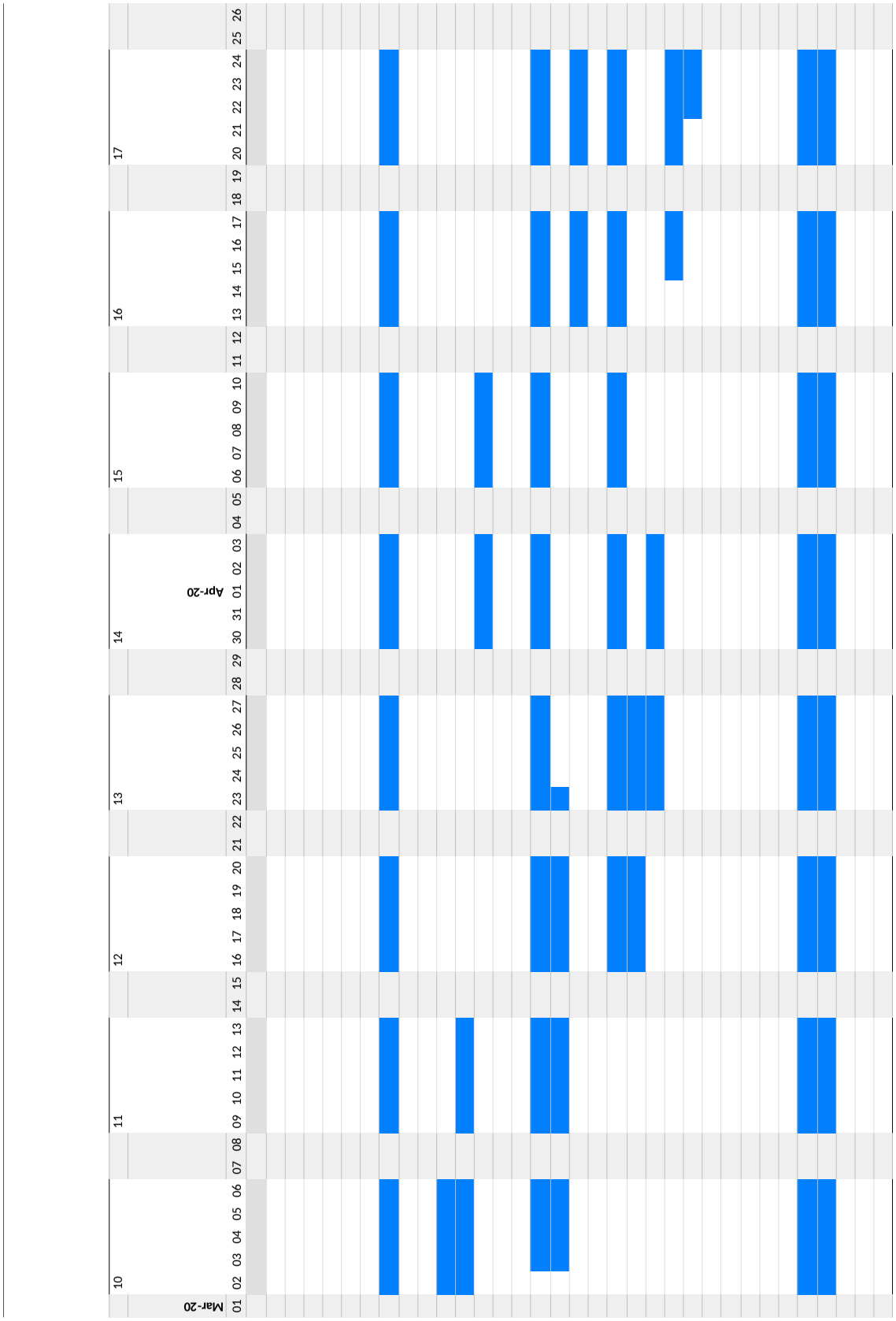
BA20_rosn02_secure_boot — 100%

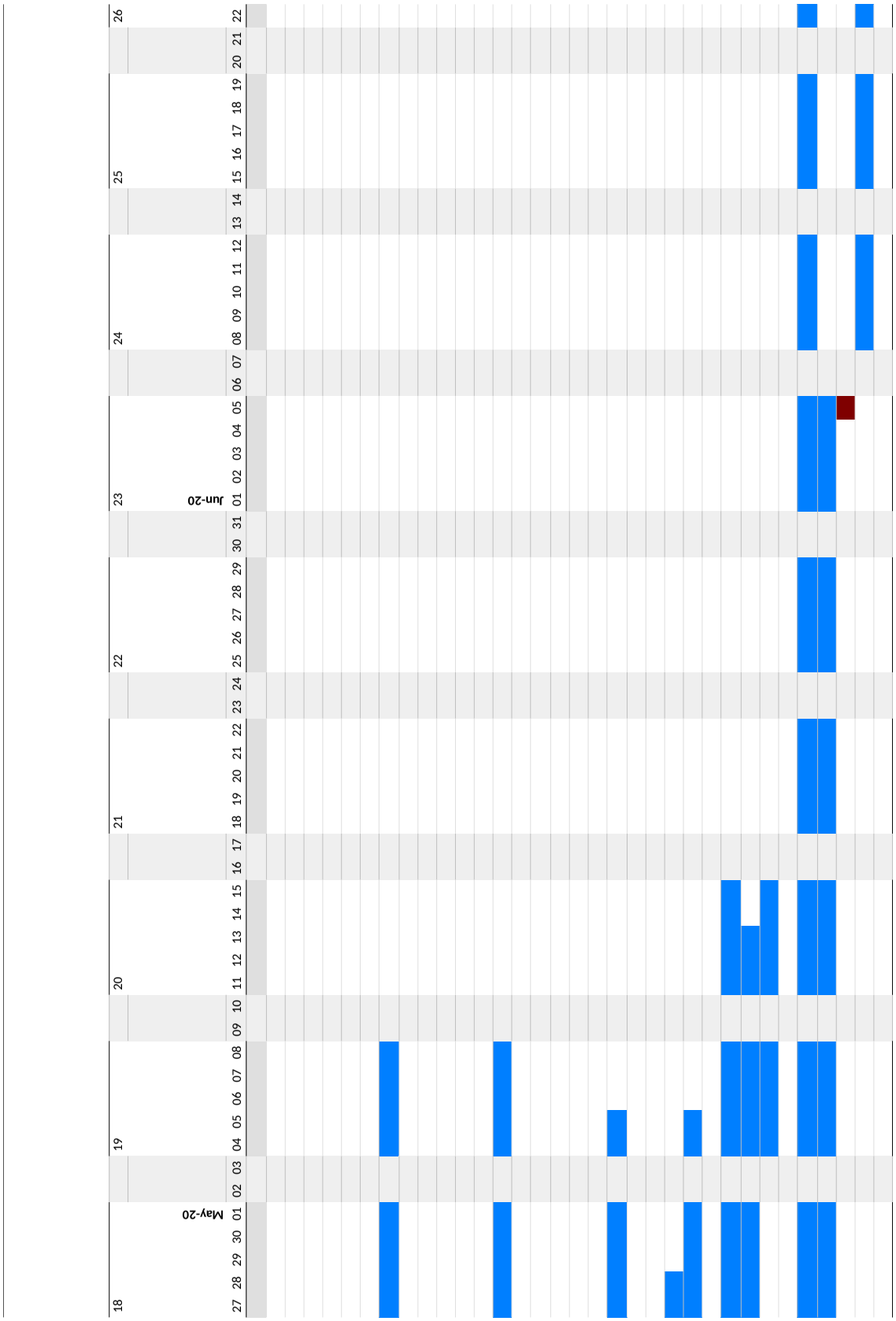
ZHAW InES

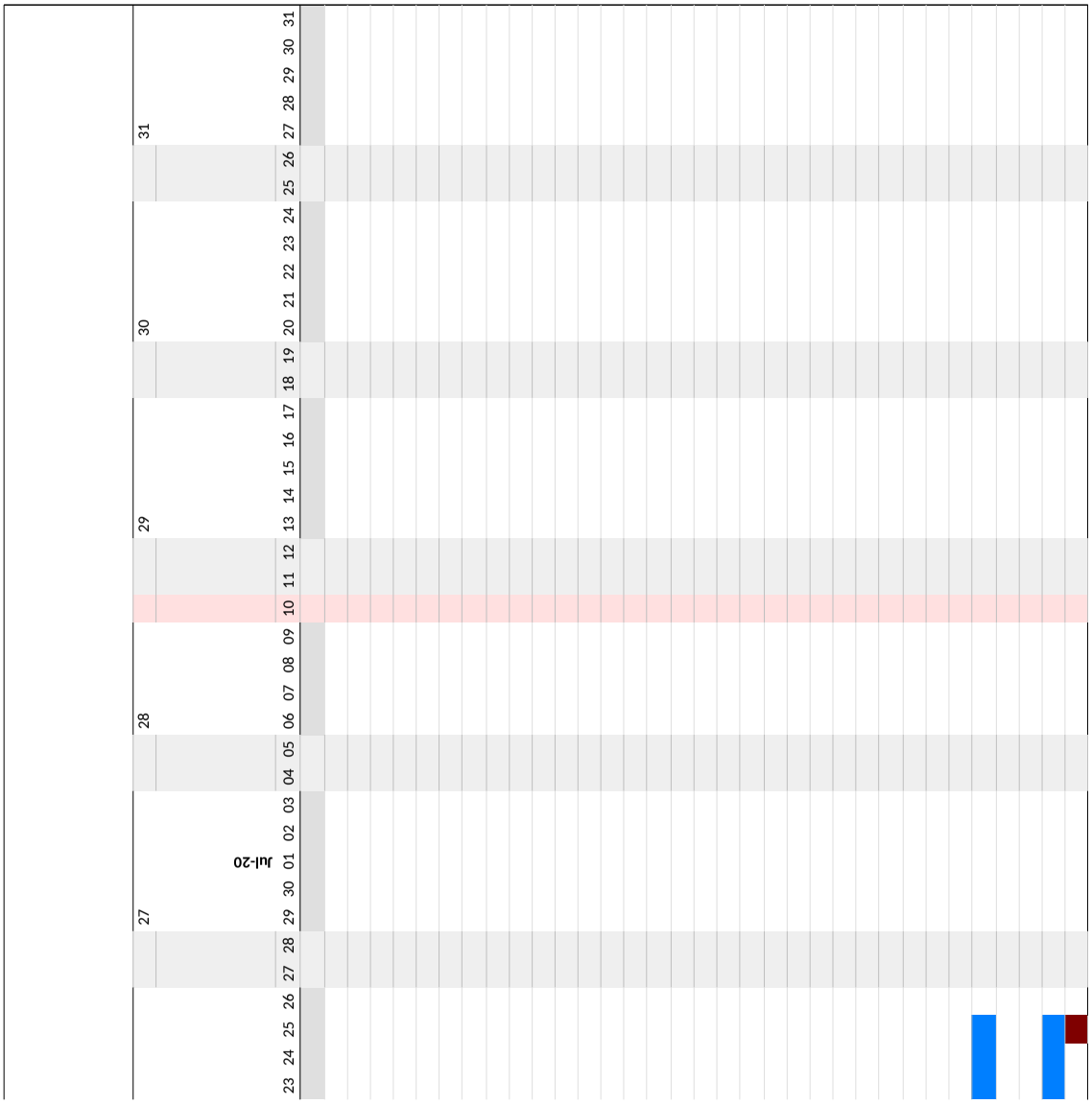
Tobias Vögeli (voegetob), Thierry Delafontaine (delafthi)

Friday, July 10, 2020

WBS	Task Names	Resource Names	Start	Duration	Finish	% Complete	Calendar days	Days Completed	Days Remaining
			10-Feb-2020	99	25-Jun-2020	100%	137		
	Get involved with the boot process	**voegetob, delafthi**	**10-Feb-2020**	**10**	**21-Feb-2020**	**100%**	**12**	**14**	**0**
	Learn about the boot process and the security(Xilinx Secure Boot)								
	Read the documentation	voegetob, delafthi	10-Feb-2020	10	21-Feb-2020	100%	12	**10**	**0**
	Make Introductory Presentation for enclustra	**voegetob, delafthi**	**13-Feb-2020**	**2**	**14-Feb-2020**	**100%**	2	**2**	**0**
	Elaborate 2-3 Use Cases with enclustra	voegetob, delafthi	20-Feb-2020	2	21-Feb-2020	100%	2	**2**	**0**
	Meeting at enclustra site in Zürich		**20-Feb-2020**						
	Analyze	**voegetob, delafthi**	**20-Feb-2020**	**57**	**08-May-2020**	**100%**	**79**	**12**	**0**
	Elaborate defined Use Cases	**voegetob, delafthi**							
	Analyze the threat modeling approach	voegetob, delafthi	20-Feb-2020	7	28-Feb-2020	100%	9	**7**	**0**
	Apply threat modeling to the US+	voegetob, delafthi	02-Mar-2020	5	06-Mar-2020	100%	5	**5**	**0**
	Analyze Crypto API	voegetob, delafthi	22-Feb-2020	16	13-Mar-2020	100%	21	**16**	**0**
	Analyze Tampering Unit	voegetob, delafthi	30-Mar-2020	10	10-Apr-2020	100%	12	**10**	**0**
	Optionally analyze PUF	voegetob, delafthi	27-Apr-2020	10	08-May-2020	100%	12	**10**	**0**
	Implement	**voegetob, delafthi**	**03-Mar-2020**	**39**	**24-Apr-2020**	**100%**	**53**	**15**	**0**
	Implement a Reference Implementation Part 1	**voegetob, delafthi**	**03-Mar-2020**	**15**	**23-Mar-2020**	**100%**	21	**15**	**0**
	Implement a Reference Implementation Part 2	**voegetob, delafthi**	**13-Apr-2020**	**10**	**24-Apr-2020**	**100%**	12	**10**	**0**
	Testing and Debugging	**voegetob, delafthi**	**16-Mar-2020**	**37**	**05-May-2020**	**100%**	**51**	**20**	**0**
	Debug Implementation Part 1	**voegetob, delafthi**	**16-Mar-2020**	**10**	**27-Mar-2020**	**100%**	12	**10**	**0**
	Make a Test Environment for Proof of Concept Part 1	**voegetob, delafthi**	**23-Mar-2020**	**10**	**03-Apr-2020**	**100%**	12	**10**	**0**
	Debug Implementation Part 2	**voegetob, delafthi**	**15-Apr-2020**	**10**	**28-Apr-2020**	**100%**	14	**10**	**0**
	Make a Test Environment for Proof of Concept Part 2	**voegetob, delafthi**	**22-Apr-2020**	**10**	**05-May-2020**	**100%**	14	**10**	**0**
	Finalize Result	**voegetob, delafthi**	**27-Apr-2020**	**15**	**15-May-2020**	**100%**	**19**	**13**	**0**
	Optimize Implementation	**voegetob, delafthi**	**27-Apr-2020**	**13**	**13-May-2020**	**100%**	17	**13**	**0**
	Generate a Reference Design for Github	**voegetob, delafthi**	**04-May-2020**	**10**	**15-May-2020**	**100%**	12	**10**	**0**
	Project Documentation and Presentation	**voegetob, delafthi**	**24-Feb-2020**	**89**	**25-Jun-2020**	**100%**	**123**	**75**	**0**
	Write Documentation in paralell	**voegetob, delafthi**	**24-Feb-2020**	**75**	**05-Jun-2020**	**100%**	103	**75**	**0**
	Delivery of project & documentation		**05-Jun-2020**						
	Make Presentation	**voegetob, delafthi**	**08-Jun-2020**	**14**	**25-Jun-2020**	**100%**	18	**14**	**0**
	Present final Presentation		**25-Jun-2020**						


Mar-20
Apr-20


18 19 20 21 22 23 24 25 26
May-20
Jun-20
18 19 20 21 22 23 24 25 26 27 28 29 30 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22


31
30
29
28
27
Jul-20
23 24 25 26 27 28 29 30 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

A.4. Meeting Protocols

A.4.1. 20.02.20

Anwesend: Gian Köppel (enclustra), Matthias Rosenthal, Philipp Huber, Tobias Vögeli, Thierry Delafontaine

Das Treffen fand bei der Enclustra GmbH in Zürich Binz statt.

Im Meeting wurden der Rahmen für die Bachelorarbeit besprochen. Als Rahmen wurden die Kundenarten der Enclustra GmbH analysiert und daraus Use-Cases abgeleitet, die den Umfang der Arbeit bestimmen sollten.

Folgende 3 Kundenarten wurden gefunden:

1. Kunden, die sich nicht um Security kümmern.
2. Kunden, die Angst haben, dass Schadsoftware auf ihre Geräte gespielt wird. Falls der Fall eintritt, in dem Schadsoftware auf das Gerät geladen wird, soll diese als solche erkannt werden und in keinem Fall ausgeführt werden. Das Gerät wechselt bei einem reboot auf eine Recovery Image und meldet den Fall. Als Beispiel wurde der Bund als Hersteller von Geschwindigkeitsmessgeräten genannt. Dabei spielt es keine Rolle ob diese Geräte kopiert werden, sondern nur dass sie so funktionieren wie sie sollten.
3. Als letztes sind die Kunden, die zusätzlich zur oben genannten Forderung einen Kopierschutz wollen. Diese verhindert dass Software kopiert oder reverse-engineered werden kann. Dazu muss die Software sowohl verschlüsselt sein, als auch gegen tampering geschützt sein. Im Falle eines Sicherheitsverstosses wird optimalerweise der Hersteller kontaktiert und gemeldet, dass etwas nicht in Ordnung ist.

Mit diesen Typen wurden 3 Use-Cases definiert.

1. Der Kunde designt ein neues Gerät. Auf was muss er achten, wenn er zu den oben genannten Gruppen gehört.
2. Update von Software/Firmware durch den Anwender oder das Netzwerk. Wie kann der Hersteller sicherstellen, dass auch die neue Software funktioniert. (Evtl. aktualisieren von Schlüsseln, etc.)
3. Aktualisieren der Hardware/Baseboard. Wie kann der Kunde Hardware ersetzen, ohne dass die Tamper Monitoring Einheit die Elektronik löscht/zerstört.

Aus diesen 3 Use-Cases, eignete man sich auf 5 Schritte.

1. Security Allgemein: Analysieren der Einzelnen Features (z. Bsp. eFuse Register), um dem Kunden sagen zu können, was bei welchem Fall hilft oder wo beim Falle eines Sicherheitsvergehen das Board zerstört wird. Ein anderes Beispiel ist der JTAG Anschluss. Es wird das Vorgehen erklärt, wie man seine Sicherheitssituation bewertet. Was sind die Gegner? Was die Bedrohungen? Was kann ich tun und welche Lösungen helfen bei was? Dies dient als Anfang um sich sein egeines Sicherheitssystem aus dem Referenzbaukasten zusammenzubauen.
2. Implementierung von Secure Boot mit einem Recovery Image.
3. Implementieren der AES Einheit in Linux, um Daten zu ver- und entschlüsseln.
4. Implementierung der Tamper Monitoring Unit. Wichtig ist auch herauszufinden, wie ich das Tampering weitermelden kann!
5. Implementierung der PUF Einheit.

Die Arbeit wird nach diesem groben Ablauf weiterverfolgt. Ein weiteres Meeting mit enclustra wird es an einem Donnerstag im März geben, der noch bestimmt werden muss.

A.4.2. 04.03.20

Anwesend: Matthias Rosenthal, Andreas Rüst, Philipp Huber, Tobias Vögeli, Thierry Delafontaine

Als erstes wurde der von uns vorgeschlagene Zeitplan überprüft. Nach dem vorgestellten Zeitplan sind wir bereits hinten nach. Grund dafür ist, dass wir uns entschieden haben, die Dokumentation parallel zur Arbeit aufzubauen. So ersparen wir uns doppelte Arbeit im Zusammensuchen der relevanten Informationen und zwingen uns durch das parallele Aufbauen das gelesene besser zu überdenken. Im Zusammenhang mit der Zeitplanung kam von unserer Seite die Frage, was das Verhältnis zwischen praktischer Arbeit und theoretischer Arbeit sein soll? Nach dem Meeting bei Enclustra ist uns bewusst geworden, dass viel Forschungsarbeit und Dokumentation, vor allem im Bereich der Anwendung eines Referenzmodells, nötig sein wird. Man einigte sich auf ein Verhältnis von 60% praktische und 40% theoretischer Arbeit, darunter versteht sich die Dokumentation der Bachelorarbeit, wie auch ein Manual, dass im Git veröffentlicht werden wird. Der Zeitplan muss dementsprechend überarbeitet werden und wird im nächsten Meeting nochmals angeschaut.

Als zweites kam die Frage, was die Resultate für die 3 Use Cases genau sind? Hier sind die Use Cases zur wiederholung kurz aufgeschrieben.

1. Use Case 1: Der Kunde designt ein neues Gerät. Auf was muss er achten?
2. Use Case 2: Der Kunde möchte ein Firmware/Software Update machen.
3. Use Case 3: Der Kunde möchte seine Hardware auswechseln.

Das Resultat des ersten Use Cases ist relativ klar. Der Kunde soll ein Manual und eine Referenzimplementationen haben, an denen er sich orientieren kann. Die Referenzimplementation beinhaltet ein System, dass sicher bootet. Es weisst eine Fallback Option auf und ist mit einer Referenzapplikation ausgestattet, die die Benutzung der Crypto API auf dem US+ zeigt. Beim zweiten Use Case wird es schon unklarer. Mögliche Ziele wären da eine Anleitung, wie Updates gemacht werden können. Ein wichtige Thema ist da vor allem das Keyhandling. Doch stellen sich die Fragen, wie Updates gemacht werden.

- Werden Update über Netzwerk gemacht. Dies würde den Prozess verkomplizieren, da noch weitere Schnittstellen angeschaut werden müssten (auf der seite Netzwerk).
- Sind die Images auf einer SDKarte, sind updates kein grosses Problem.
- Sind die Images auf dem QuadSPI und müssten mit der SDKarte aktualisiert werden, bräuchte man das entsprechende vorgehen. Entweder gibt es ein Fallback Image, dass immer gleich bleibt. Das andere Image wird dann jeweils aktualiesiert. Oder es gibt ein neues und eine altes Image, das neue wird zum alten und das alte wird überschrieben.

Der dritte Use Case kann im Moment nocht nicht geklärt werden, da zurzeit noch nicht bekannt ist ob ein Hardwareaustausch die Tampering Unit auslöst. In diesem Fall wäre es optimal die ersten zwei Use Cases zu klären und dann sich auf den dritten Use Case zu konzentrieren.

Zur Lösung dieser Fragen soll nochmals Kontakt mit Enclustra aufgenommen werden. Am besten soll aufgezeigt werden, was für Möglichkeiten wir uns ausgedacht haben, um zu entscheiden welche man weiterverfolgen soll.

Es wurde zusätzlich noch geklärt, ob die Crypto API von Linux näher angeschaut werden soll. Grund dafür ist, dass die Crypto Hardware auf dem US+ darüber gesteuert wird. Es wurde dabei entschieden, die Crypto API näher anzuschauen.

A.4.3. 11.03.2020

Anwesend: Matthias Rosenthal, Andreas Rüst, Philipp Huber, Tobias Vögeli, Thierry Delafontaine

Probleme sind beim erstellen der DFD-Modelle und anwenden des STRIDE Modells aufgetreten. Wieviel gehöhrt wirklich dazu? Wichtig ist eine Verbindung zur Praxis aufzubauen. Wieso soll das verwendet werden, was ist der Nutzen? Somit soll der Kontext aufgezeigt werden. Um dies auf die Use-Cases abzubilden, sollen exemplarische Beispiele verwendet werden. Wichtig ist jedoch Grenzen zu setzen. Im Rahmen dieser Arbeit kann nicht alles erarbeitet werden. Die entdeckten Threats sollen dann nicht nur mit einer Risikoanalyse gewertet werden, sondern auch aus der Kosten/Nutzen Sicht vom Angreifer. Was kriegt der Angreifer bei Erfolg, was wird er dafür aufwenden, dass es sich für ihn noch lohnt?

Für das abgemachte Meeting mit Gian Köppel von enclustra, wird er falls möglich zum nächsten refulären Meeting am Mittwoch um 08:30 eingeladen, um teilzunehmen. Dort werden wir ihn mit den Fragen vom letzten Meeting konfrontieren. Was sollen die Produkte sein, die aus den Use Cases hervorgehen? Dabei soll auch herausgefunden werden, ob wir zurzeit auf dem richtigen Weg sind.

Der Zeitplan V2 ist in Ordnung, jedoch wurden einige Fehler entdeckt. Diese sind zu korrigieren.

Es soll am Projekt weiterhin mit PetaLinux gearbeitet werden. Grund dafür ist, dass mit hoher Wahrscheinlichkeit, Kunden von enclustra auch mit PetaLinux arbeiten werden und nicht mit Yocto. Ausserdem sollte vieles sehr gleich funktionieren. Lösung für angetroffene Probleme können gelöst werden, indem man Yocto besser kennen lernt.

A.4.4. 18.03.2020

Anwesend: Gian Köppel, Matthias Rosenthal, Andreas Rüst, Philipp Huber, Tobias Vögeli, Thierry Delafontaine

Das Meeting wurde wie besprochen mit Gian Köppel von enclutra durchgeführt. Er konnte uns die Fragen, die beim letzten Meeting aufgetaucht sind beantworten. Es wurde ausserdem mitgetelt, dass wir uns auf PetaLinux konzentriere werden, weil Kunden mit hoher Wahrscheinlichkeit damit Arbeiten werden.

Der zweite noch offene Use Case wurde von Gian Köppel genauer spezifiziert. Dabei gehen wir auf den Fall ein, dass das Image mit dem golden Image im QSPI sitzt und der User/Anwender das Image über die SD Karte updated. Der Fallback soll dabei die Sicherheit geben, dass das Gerät immer gestartet werden kann, und somit auch ein neues Update aufgespielt werden kann. Die MCT Software um den JTAG auf den USB zu führen, funktioniert unter Windows nicht. Gian meinte, wir sollen uns an den Support richten, um dieses Problem zu lösen.

Die gezeigten DFD Analysen und STRIDE Modelle sind in Ordnung. In der nächsten Woche geht es darum diese umzusetzen. Die Crypto API ist noch in der Bearbeitung, doch wird sie in der nächsten Woche fertig gestellt.

A.4.5. 26.03.2020

Anwesend: Gian Köppel, Matthias Rosenthal, Philipp Huber, Tobias Vögeli, Thierry Delafontaine

Das Meeting wurde kurz gehalten, da keine grossen Probleme in der letzten Woche effectuating sind. Es ist das weitere Vorgehen besprochen worden. Da nicht genau klar ist wieviel Zeit für die nächsten Schritte gebrauch wird. Als erstes wird sicher die Tamper Monitoring Unit und die PUF angeschaut. Zusätzlich wird noch ein bis zwei Tage in das yocto investiert. Ziel ist es herauszufinden, wie umständlich es ist ein Projekt für Xilinx Produkte im yocto aufzsetzen.

Anschliessend ist uns bestätigt worden, die eFuse auf einem Board mit der RSA Hash zu brennen, um das ganze Vorgehen auch mit Autentifizierung zu testen. Wichtig ist, dass der Hash und Key klar dokumentiert werden, und nicht verloren gehen.

A.4.6. 02.04.2020

Anwesend: Gian Köppel, Matthias Rosenthal, Philipp Huber, Tobias Vögeli, Thierry Delafontaine

In der letzten Woche wurde die Möglichkeit angeschaut, Yocto für das Bilden des images anstatt PetaLinux zu benutzen. Dabei ist herausgekommen, dass die meisten Ressourcen veraltet und Yocto nicht von Xilinx unterstütz wird, da ja PetaLinux angeboten wird. Daraufhin ist etschieden worden weiter mit PetaLinux zu arbeiten. Das dies untersucht wurde, ist jedoch in der Dokumentation zu erwähnen.

Als zweites wurde besprochen, dass in einem geschützen Endprodukt die U-Boot CLI ausgeschaltet werden muss. Dementsprechend muss die Update-Software aus Linux gestartet werden. Dieses portieren sollte jedoch nicht zu viel Zeit verschwenden.

Die PUF im US+ ist lediglich für die Verschlüsselung eines Red Keys zu verwenden. Alles andere ist nicht vorgesehen. Darum werden wir uns nicht mehr mit der PUF beschäftigen, sondern mit dem Thema Trustzone und wie wir Hardware komplet mit Trustzone voneinander abtrennen können. Parallel dazu wird das Thema Tamper Monitoring angeschaut.

A.4.7. 09.04.2020

Anwesend: Matthias Rosenthal, Philipp Huber, Tobias Vögeli, Thierry Delafontaine

In der letzten Woche haben wir uns in die Themen Key Revocation, Tamper Monitoring Unit und Trustzone eingearbeitet. Das Thema Key Revocation ist klar und kann beendet werden. Die letzteren beiden Themen, werden in der nächsten Woche noch bearbeitet. Dabei wird uns Herr Rüst noch Arbeiten zu diesen Themen vom Init zukommen lassen. Das Thema Tamper Monitoring Unit und der 3. Use Case wird vermutlich nicht so viel Zeit in Anspruch nehmen. Trustzone wird eher Etwas, in das wir uns einarbeiten können. Dabei soll unbedingt mit einem Beispiel gearbeitet werden, da Trustzone seht anwendungsspezifisch ist. Herr Rosenthal hat vorgeschlagen ein Karten-Terminal als Beispiel Projekt zu nehmen.

Wir sind noch von Herrn Rosenthal auf einen aktuellen Eintrag im Xilinx Wiki aufmerksam (https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841862/Install+and+Build+with+Xilinx+Yocto) gemacht worden. Es geht dabei darum das Yocto Layers für Xilinx Produkte existieren. Diese Möglichkeit soll evaluiert und in der Dokumentation erwähnt werden.

Wegen dem Aufgebot vom Militär ist folgendes entschieden worden. Es soll wie gehabt weiter gearbeitet werden. Wenn es so weit ist kann immer noch entschieden werden.

A.4.8. 16.04.2020

Anwesend: Gian Köppel, Matthias Rosenthal, Andreas Rüst, Philipp Huber, Tobias Vögeli, Thierry Delafontaine

In der letzten Woche wurde an der Tamper Monitoring Unit und Trustzone weitergearbeitet. Bei der Tamper Monitor Unit ist das Problem aufgetreten, dass das Trustzone den Zugriff auf die Register verhindert. Dementsprechend können die Register nicht überprüft werden. Ausserdem ist noch unklar, wie der Triggermechanismus funktioniert. Ab wann wird getriggert? Was sind die Bedingungen um zu Triggern und wie kann ich die einstellen? Gian Köppel nannte uns die häufigsten Vorgehen beim Tampering. Darunter gehört:

- Einen Rippel auf die Versorgungsspannung legen um Glitches zu erzeugen
- Spannung sehr tief halten, damit mehr Strom gebraucht wird. Dann können mittels Differential Power analysis Informationen

Wegen dem Militär vom 18.05.2020 bis zum 03.07.2020 ist entschieden worden die Arbeit nach dem Dienst noch drei Wochen fortzuführen. Die Präsentationen werden dementsprechend erst im August stattfinden. Dies wird so Herrn Loeser mitgeteilt. Es ist wichtig zu beachten, dass die Diplome trotzdem noch rechtzeitig. Gegeben werden können, obwohl die Noten erst später nachgereicht werden.

A.4.9. 23.04.2020

Anwesend: Gian Köppel, Matthias Rosenthal, Andreas Rüst, Philipp Huber, Tobias Vögeli, Thierry Delafontaine

Während der letzten Woche wurden sowohl an der Tamper Monitoring Unit als auch an Trustzone gearbeitet. Tobias Vögeli hat zum Thema Trustzone eine Liste erstellt mit Punkten, die er bearbeiten will. Als Erstes will er auf die Themen Secure Monitor und Secure Monitor Call eingehen. Dieses Thema wurde bereits in der letzten Woche angefangen. Die Teile Secure Application und Secure OS werden als nächstes angeschaut. Dazu wird OP-TEE verwendet werden. Es muss aufgezeigt werden, wann man in der Secure World arbeitet. Dazu soll das Beispiel vom Kartenterminal, das bereits letzte Woche besprochen wurde, umgesetzt werden.

Der 3. Use Cases bietet keine Schwachstellen und ist nicht geeignet, um mit dem DFD oder Stride Modell zu analysieren. Das gewünschte Resultat der Analyse soll nicht Sicherheitsrisiken, sondern das beeinhalten auf was der Kunde achten muss, um sein Baseboard auszuwechseln, ohne sein Gerät zu zerstören. Die Frage stellt sich, was heisst das wenn man Sicherheit implementiert hat. Knackpunkt ist da vor allem der BBRAM, weil die Batterie auf dem Baseboard ist und der Inhalt des BBRAMs beim Auswechseln des Baseboardes gelöscht wird.

Als letztes ist noch die Struktur der Dokumentation angeschaut worden. Folgende Struktur eignet sich sehr gut:

1. Secure Development - Wie entwickle ich sichere Produkte
2. Security Analysis - Beschreibung und analyse der drei Use Cases
3. Security Features Analysis - Analyse der Sicherheitsfunktionen vom Zynq UltraScale+
4. Crypto API - Beschreibung der Crypto API
5. Implementation - Beschreibung der Implementation
6. Results - Validierung der Implementation

Diese Struktur beschränkt sich nur auf den Hauptteil.

A.4.10. 30.04.2020

Anwesend: Gian Köppel, Matthias Rosenthal, Andreas Rüst, Philipp Huber, Tobias Vögeli, Thierry Delafontaine

Zurzeit stellen sich Probleme, bei der Tamper Monitoring Unit. Während der Tamper Trigger problemlos funktioniert, ist die Reaktion des Systems auf Temperaturschwankungen unter die eingestellte Grenze komisch. Um dieses Problem zu lösen, soll im Xilinx Forum nachgefragt werden. In der folgenden Woche wird zusätzlich die Tamper Reaktion für einen JTAG toggle getestet.

OPTEE, eine Lösung von Trustzone, existiert für das Evaluation Board von Xilinx. Zurzeit läuft es jedoch nicht auf dem Mercury Board von Enclustra. Falls OPTEE nicht zum laufen gebracht wird, gibt es noch ein anderes TEE das getestet werden kann.

Heute wurde zusammen mit Herrn Rüst der erste Teil der Dokumentation angeschaut, der das letzte Mal abgegeben wurde. Dabei war vor allem zu bemängeln, dass zu schnell in die Details eingegangen und der Leser zu wenig ins Thema eingeführt wird. Es ist geplant in der nächsten Woche den nächsten Teil einzureichen.

A.4.11. 06.05.2020

Anwesend: Gian Köppel, Matthias Rosenthal, Andreas Rüst, Philipp Huber, Tobias Vögeli, Thierry Delafontaine

In der letzten Woche wurde vor allem an der Dokumentation gearbeitet. Im Bereich vom Tamper Monitor wurde nur noch getestet, ob das Triggern auch mit dem JTAG Port funktioniert. Obwohl bereits seit fast einer Woche die Frage zum Temperaturmonitoring veröffentlicht ist, ist noch keine antwort gegeben worden.

Auch im Beriech von Trustzone läuft es noch nicht. Das Projekt lässt sich zwar kompilieren und starten, doch nach dem FSBL bleibt das Gerät im Bootvorgang bei der Übergabe vom ATF zum OPTEE stehen. Das OPTEE scheint aus einem noch unbekannten Grund nicht zu starten.

In der nächsten Woche wird weiter an der Dokumentation gearbeitet. Ziel ist vor dem 18.5.2020 einen Teil der Dokumentation vorab abzugeben, damit er bereits korrigiert werden kann. Nebenbei wird noch an Trustzone und der Tamper Monitoring Unit weitergearbeitet.

A.4.12. 14.05.2020

Anwesend: Gian Köppel, Matthias Rosenthal, Andreas Rüst, Philipp Huber, Tobias Vögeli, Thierry Delafontaine

In der letzten Woche wurde vor allem die Dokumentation ergänzt. Weiter ist versucht worden, das OPTEE Projekt auf dem Mercury Board zum laufen zu bringen. Zurzeit stehen wir beim Problem, dass das OPTEE OS nicht startet. Dementsprechend bleibt der ganze Bootvorgang hängen. Um weiterzukommen, soll das OPTEE Projekt auf einem zcu102 Board getestet werden. Die ZHAW besitzt ein solches Board und wird es uns zur verfügung stellen. Zusätzlich soll in einem Forum nachgefragt werden, was mögliche Probleme sind.

In nächster Zeit wird Thierry Delafontaine abwesend sein. Darum werden die Meetings nicht wöchentlich durchgeführt, sondern nur wenn es Etwas zu besprechen gibt. Die Informationsmails werden weiterhin geschrieben.

Es ist geplant weiter an der Dokumentation zu arbeiten, um möglichst früh den nächsten Teil einzureichen. Zusätzlich ist noch besprochen worden, wie das zukünftig öffentliche Repo aussehen wird. Zum aktuellen Zeitpunkt wäre es möglich, dass die ganze Dokumentation veröffentlicht wird.

A.4.13. 02.07.2020

Anwesend: Gian Köppel, Matthias Rosenthal, Andreas Rüst, Philipp Huber, Tobias Vögeli, Thierry Delafontaine

Da die Arbeit in den letzten 8 Wochen unterbrochen war, lief nicht viel. In diesem Meeting ging es also hauptsächlich darum den Abschluss der Arbeit zu besprechen. In den letzten zwei Wochen wird vor allem die Dokumentation priorisiert. Tobias Vögeli konnte während dem letzten Meeting OP-TEE auf dem zcu102 Board laufen lassen und schrieb eine simple Applikation dazu. Auf dem Board der Enclustra läuft OP-TEE jedoch aus unbekannten Gründen immer noch nicht.

Neben der Dokumentation wird ein Vorschlag zum Aufbau des Referenzprojekts gemacht (öffenliches Git Repository). Es ist bestimmt worden, dass darin die Bachelordokumentation mit dem Referenzprojekt als PDF veröffentlicht wird. Da das Referenzprojekt Modular aufgebaut werden wird, sollen nebenbei einzelne Dokumentationen zu den einzelnen Modulen existieren, die entsprechend nachgeführt werden können. Die Bachelorarbeit dient dazu als Ausgangslage.

Noch nicht abgeschlossene Arbeiten, wie das Umsetzen von OP-TEE und die Tamper Monitoring Unit, werden nur weitergeführt, wenn genügend Zeit vorhanden ist. Diese Bereiche sind nicht essentiell für die Veröffentlichung der Referenzplattform und können dementsprechend auch später noch ergänzt werden.

Zur Zeit ist noch nicht ganz klar, wie genau die Bachelorarbeit abgegeben werden muss. Anforderungen zur Abgabe werden noch nächste Woche nachgereicht.

Als Vorschlag für die Bachelorpräsentation wurden die Termine 24.7, 27.7/ 28.7 vorgeschlagen. Diese Termine müssen noch mit dem Experte abgesprochen werden.

A.4.14. 09.7.2020

Anwesend: Gian Köppel, Matthias Rosenthal, Tobias Vögeli, Thierry Delafontaine

Es wird nur noch an der Dokumentation und dem Git Repo gearbeitet.

Heute wurde vor allem der Aufbau des öffentliche Git Repos besprochen. Das Repo wird in Modulen aufgebaut. Als Module zählen die einzelnen Features, welche optional zum Petalinux Projekt hinzugefügt werden können. Dabei bildet das Petalinux Projekt das zentrale Hauptmodul. Wichtig ist, dass in den Modulen jeweils der Aufbau der Ordnerstruktur innerhalb des Moduls und eine Anleitung vorhanden ist, wie das Modul zu implementieren ist.

Die Dokumentation wird als PDF im Grundverzeichnis abgelegt und dient als Referenz. Die hauptsächlichen Dokumentationen über den Inhalt des Repos, welche auch ergänzt und aktualisiert werden können, sind jedoch die README.md Dateien in den einzelnen Ordnern. Das Readme im Grundverzeichnis, gibt einen Überblick über das ganze Projekt und die Ordnerstruktur. Es soll auch einen Quickstart Guide beinhalten.

Ausserdem sollen im Petalinux Projekt alle generierten Schlüssel gelöscht werden, damit die Nutzer ihre eigenen Schlüssel generieren. Damit wird sichergestellt, dass niemand öffentlich Schlüssel für die Authentifizierung und Verschlüsselung/Entschlüsselung von Dateien einsetzt.

Die Abgabe wird spätestens um 17.07.2020 um 24:00 stattfinden. Dazu wird eine E-Mail mit der angehängten Dokumentation an alle beteiligten verschickt. Es kann sein, dass zu einem späteren Zeitpunkt, die Arbeit noch nachträglich gebunden, sowie auch auf einen SFTP Server der ZHAW hochgeladen werden muss.