

Heidelberg University,  
*Institute of Computer Engineering (ZITI),*  
*Novel Computing Technologies Group (NCT)*

Private publication. Not peer reviewed!

---

# **Setting up a Trusted Execution Environment (TEE) on the AMD Zynq UltraScale+ MPSoC platform using Arm TrustZone technology**

---

Author:	Jannik Schacht [REDACTED]
Matr. No.:	[REDACTED]
Course:	MSc Data and Computer Science
Module:	[REDACTED]
Examiner:	[REDACTED]
Date of submission:	13 <sup>th</sup> Feb 2024

## Abstract

Trusted Execution Environments (TEEs) make it possible to execute security-sensitive applications in isolation from a feature-rich operating system, thus minimising the attack surface. TEEs are already widespread in mobile devices like smartphones. But there are also several use cases for TEEs in SoC-FPGAs, especially in the context of cloud computing.

The main goal of this work is to establish a TEE on an AMD Zynq UltraScale+ MPSoC device by utilising the Arm TrustZone technology. The TEE shall work in conjunction with a Linux-based Rich Execution Environment (REE). OP-TEE OS is used as the Trusted OS. This report shows how the system has been planned and implemented. At first, the concept and potential use cases of TEEs are introduced. Then, the security features of the target hardware – the ZCU102 Evaluation Kit – are analysed. Based on the findings, the system is planned, whereby the design choices are discussed and justified. The implementation is documented in the style of a tutorial. As a result, the ZCU102 is running a Linux OS which is capable of invoking the execution of Trusted Applications in the TrustZone-based TEE. This system serves as a good foundation to start developing own Trusted Applications. The presented concepts may be transferred to other hardware architectures as well.

# Contents

<b>1. Introduction</b>	<b>1</b>
1.1. Need for a TEE on the Zynq MPSoC Platform . . . . .	1
1.2. Subject of this Work . . . . .	2
1.3. Related Work . . . . .	2
<b>2. Hardware Analysis</b>	<b>4</b>
2.1. AMD Zynq UltraScale+ MPSoC Architecture . . . . .	4
2.2. ZCU102 Evaluation Kit . . . . .	4
2.3. Relevant Components . . . . .	4
2.4. Key Management . . . . .	6
2.5. Boot Process . . . . .	7
<b>3. System Design</b>	<b>9</b>
3.1. Non-Secure World – Linux . . . . .	9
3.2. Secure World – OP-TEE . . . . .	9
3.3. Execution Hierarchy . . . . .	9
3.4. Secure Boot . . . . .	10
3.5. Memory Map . . . . .	12
3.6. Boot Image . . . . .	14
<b>4. System Implementation</b>	<b>15</b>
4.1. Tools . . . . .	15
4.2. Environment Setup . . . . .	17
4.3. Preparing the SD Card . . . . .	20
4.4. Creating PetaLinux Project . . . . .	21
4.5. Building & Booting a Basic Linux . . . . .	22
4.6. Helpful Customisations . . . . .	24
4.7. Secure Boot – Authentication . . . . .	26
4.8. Secure Boot – Encryption . . . . .	33
4.9. Setting up the TEE . . . . .	39
4.10. Developing Linux Applications . . . . .	48
4.11. Developing Trusted Applications . . . . .	49
4.12. Notes on Key Management . . . . .	49
<b>5. Results &amp; Discussion</b>	<b>51</b>
5.1. Achievements . . . . .	51
5.2. Future Work . . . . .	51
<b>List of Acronyms</b>	<b>53</b>
<b>Bibliography</b>	<b>54</b>

---

<b>A. Basic BIF-File</b>	<b>56</b>
<b>B. BIF-File with Authentication</b>	<b>57</b>
<b>C. BIF-File with Authentication (incl. Linux kernel)</b>	<b>59</b>
<b>D. BIF-File with Authentication &amp; Encryption</b>	<b>61</b>
<b>E. BIF-File with Authentication &amp; Encryption</b>	<b>63</b>
<b>F. U-Boot Script</b>	<b>65</b>

# 1. Introduction

The student project presented in this report has been carried out in the Novel Computing Technologies Group (NCT) at the Institute of Computer Engineering (ZITI) at Heidelberg University. However, this document is a private publication and is in no way affiliated with NCT, ZITI or Heidelberg University.

The increasing complexity of applications pushes developers towards using feature-rich Operating Systems (OSs), rather than developing bare-metal applications. One well-known example of this trend is the evolution of mobile phones in recent decades. However, more complex OSs entail new challenges. For example, the potential attack surface grows as their complexity increases. Therefore, the most relevant CPU designers/manufacturers have developed technologies like *Intel SGX* or *Arm TrustZone*, making it possible to establish trustworthy environments which are completely isolated from the primary OS. Such an environment, commonly referred to as *Trusted Execution Environment (TEE)*, can execute security-sensitive applications isolated from a *Rich Execution Environment (REE)*, thus minimising the attack surface.

As summarized in [1], various applications can benefit from the availability of TEEs. On mobile devices like smartphones or tablets, for example, they are used to protect mobile payment methods and authentication processes or to enforce copyright protection. Other common use cases are the protection of cloud applications against compromised cloud infrastructure or the secure analysis of sensitive data (e.g. with machine learning models).

## 1.1. Need for a TEE on the Zynq MPSoC Platform

In addition to the above examples in which TEEs are already widely used, TEEs are also a subject of current research on FPGA security – especially in the context of cloud computing. Various SoC-FPGAs such as the *AMD Zynq MPSoC* devices (i.e. chips with a heterogenous architecture combining CPU and FPGA) have been released in the last years. However, most TEEs are purely CPU-based and do not yet support FPGAs. Hence, it is a legitimate intention to extend CPU-based TEEs to the Programmable Logic (PL) of FPGAs [2]. This would allow for improved performance of applications that process security-sensitive data. For example, accelerators for machine learning [3] or networking [4] could be implemented using PL without taking the risk of increasing the attack surface.

Besides all these use cases, the primary motivation for setting up a TEE on the Zynq MPSoC platform in this project is creating a basis for further development of the *TruFPGA* protocol, which is proposed by Zeitouni et al. in [5]. This protocol is meant to solve the conflict of interests between the users of FPGA resources in commercial cloud platforms and the Cloud Service Providers (CSPs). On one hand, some clients wish to protect their intellectual property

by uploading an encrypted bitstream to the FPGA, such that it is not revealed to the CSP. On the other hand, CSPs need to protect their hardware from several known attacks that can be invoked via malicious bitstreams. One example is the denial-of-service attack shown by La et al. in [6]. Such attacks can be avoided by scanning the bitstream (or netlist) for malicious circuits before applying the configuration to an FPGA. This can be done by employing tools provided by the respective FPGA vendor or a virus scanner as presented by La et al. in [7]. However, in any case, the unencrypted bitstream or netlist must be revealed to allow for the scan.

To solve this conflict, the approach in [5] involves the FPGA vendor as a third party, which is assumed to be trusted by both the client and the CSP. The protocol utilises two TEEs, which must be provided by the vendor. One of them is called *trusted shell* and implemented on the target device using PL. The *trusted shell* is responsible for decrypting and configuring the bitstream after it has been checked. With SoC-FPGAs, new possibilities arise: The functionality of the *trusted shell* could possibly be ported to a CPU-based TEE within the same chip, thus leaving the entire FPGA to the client.

## 1.2. Subject of this Work

The goal of this work is to show how a TEE can be established on a System-on-Chip (SoC)-FPGA. As pointed out previously, there are many use cases for this, one of them being the further development of the *TruFPGA* protocol proposed in [5].

To achieve this goal, the security features of the given hardware shall be analysed first. Then, it shall be planned, how a system consisting of a REE and a TEE can be realised based on the available features. This involves defining a root of trust, securing the boot process, as well as selecting appropriate OSs for the REE and TEE. Finally, the system shall be implemented and tested. An AMD Zynq MPSoC device – more specifically the ZCU102 Evaluation Kit – shall be used for testing.

A secondary goal is to provide background information and knowledge about the required tools, such that the approaches presented in this work can also be applied to other Zynq MPSoC devices or even devices with different architectures.

## 1.3. Related Work

To the best of my knowledge, there are two profound elaborations about setting up a TEE on Zynq MPSoC devices:

The developers of OP-TEE, an open-source TEE that will be used in this project as well, provide instructions for building OP-TEE for Zynq MPSoC devices in their Wiki<sup>1</sup>. They also describe

---

<sup>1</sup><https://optee.readthedocs.io/en/3.14.0/building/devices/zynqmp.html> (Accessed: 2024-01-28)

how to obtain a boot image with Linux as [REE](#) and OP-TEE as [TEE](#), but they do not address the implementation of secure boot, which is mandatory. However, the provided instructions turn out to not work anymore.

In [8], Vögeli and Delafontaine provide very detailed instructions for the ZCU102 board. They explain how to establish a root of trust, set up secure boot and finally build and run Linux as [REE](#) and OP-TEE as [TEE](#). Unfortunately, these instructions only work with earlier versions of AMD's tools. They are not working with current software versions.

The technical documentation of the ZCU102 with information about the boot process and security features as well as information about the required tools is spread across many documents provided by AMD. Some of these documents (e.g. the guide for [BBRAM](#) programming) have not been updated recently and no longer work with current versions of AMS's tools.

## 2. Hardware Analysis

### 2.1. AMD Zynq UltraScale+ MPSoC Architecture

*Zynq UltraScale+ MPSoC* is a [SoC](#) platform, which is based on AMD's UltraScale+ architecture for FPGAs. In contrast to other UltraScale+ FPGA devices, the Zynq device family combines soft and hard engines by including a Processing System ([PS](#)) in addition to the Programmable Logic ([PL](#)). The platform integrates a multi-core Arm Cortex-A53 CPU for application processing, a dual-core Arm Cortex-R5F CPU for real-time processing, and an Arm Mali-400 GPU optionally (see [9]). Furthermore, the [PS](#) offers additional connectivity interfaces and units for platform management, memory management, system monitoring, security, and FPGA configuration. The [PS](#) and [PL](#) are connected by an AXI4-based communication bus. This makes it possible to, for example, implement custom peripherals on the FPGA and access them via software.

### 2.2. ZCU102 Evaluation Kit

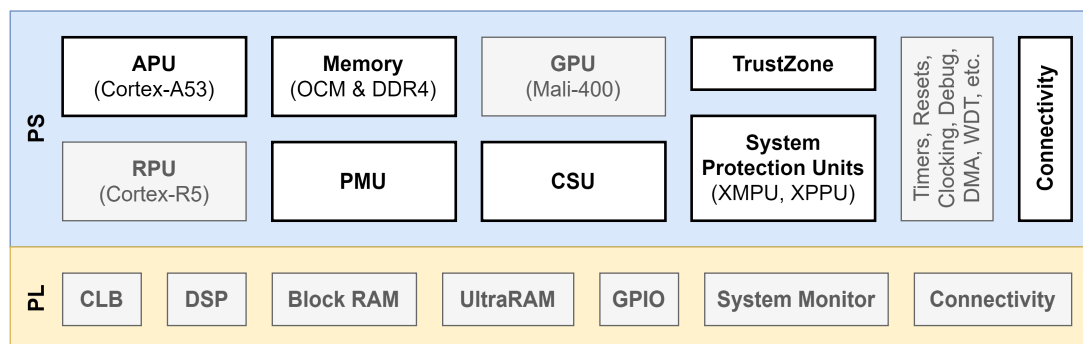
The hardware used in this project is the *ZCU102 Evaluation Kit*. It is equipped with a ZU9EG [SoC](#) from the Zynq UltraScale+ MPSoC family. Some key features of this device are (see [10]):

- 64-bit quad-core Arm Cortex-A53 (ARMv8-A architecture),
- 32-bit dual-core Arm Cortex-R5F (ARMv7-R architecture),
- Arm Mali-400 GPU,
- [PS](#) with 256 KB on-chip RAM (extended with 4 GB external DDR4 RAM),
- Boot from JTAG, QSPI, SD card, etc.
- SHA, RSA & AES hardware support,
- Arm TrustZone support,
- eFuse registers and Battery Backed RAM ([BBRAM](#)) for key storage,
- Physical Unclonable Function ([PUF](#)),
- 4-channel USB-UART interface,
- USB-JTAG interface,
- [PL](#) with ~600k logic cells, ~2.5k DSP slices, 32.1 Mb block RAM.

### 2.3. Relevant Components

Fig. 2.1 provides an overview of the essential components of the [SoC](#). The components most relevant to this project are highlighted and explained in the following. More information can be found in [11].





**Figure 2.1.:** Zynq ZU9EG component overview

## APU

The Application Processing Unit (**APU**) contains the quad-core Cortex-A53 and is a general-purpose processing unit that can be used to execute bare-metal applications but also complex **OSs** like Linux. Furthermore, it supports the TrustZone technology, meaning that it is capable of switching between a secure and non-secure state.

## Platform Management Unit

The Platform Management Unit (**PMU**) is responsible for basic system initialisation, power management and error handling. It is a processor that executes a non-exchangeable startup sequence when the system is powered up. The **PMU**'s ROM memory contains instructions for default functionality. During the boot process, custom firmware can be loaded to the **PMU**'s RAM, which may extend or replace features of the **PMU**.

## Configuration and Security Unit

The Configuration and Security Unit (**CSU**) is a processor that is responsible for providing access to hardware-based security features (authentication, decryption, hashing, storing and managing keys, etc.), monitoring temperature and voltage to detect tampering and for reconfiguration of the **PL**.

## Memory

The Zynq UltraScale+ MPSoC architecture supports a maximum address width of 40 bits, resulting in up to 1 TiB of physical address space. This can only be utilised by the **APU**, as the Real-Time Processing Unit (**RPU**) is limited to 32-bit addresses. To maintain compatibility, all relevant peripherals are mapped to the lower 4 GiB of the address space, which can be accessed with 32-bit addresses.

There are two main types of memory available. The internal memory has a total capacity of 4 MiB. It is used for software components and data that cannot or should not be stored in external memory for security reasons. This includes, for example, the [CSU](#) and [PMU](#) firmware and the eFuse registers. 256 KiB of the internal memory are available to the user. This memory region is referred to as On-Chip Memory ([OCM](#)). The internal memory can (and must) be extended by external DDR4 RAM. In this project, 4 GiB of DDR4 RAM are used.

### TrustZone & System Protection Units

The SoC provides hardware support for the TrustZone technology on system level (i.e. outside the [APU](#) itself). This is necessary because all transactions on the shared buses must be tagged as secure or non-secure, depending on the state of the master. In the case of the APU, both states are possible, because it supports TrustZone internally and can therefore switch between a secure and a non-secure state. Other bus masters (e.g. the [RPU](#)) do not support TrustZone internally. However, they can be statically assigned to either the secure or the non-secure world.

Furthermore, the system is equipped with multiple protection units, which verify that a bus master is explicitly allowed to access an address by assigning specific address ranges to either the secure world or the non-secure world. The Xilinx Memory Protection Units ([XMPUs](#)) are responsible for blocking illegal access attempts to memory, whereas the Xilinx Peripheral Protection Units ([XPPUs](#)) are responsible for blocking illegal access attempts to peripherals.

### Connectivity

Of the various interfaces available for external communication, the UART, Ethernet and SD card interfaces are relevant for this project.

## 2.4. Key Management

There are multiple options for storing and managing keys. This involves two types of memory: The eFuse registers are one-time programmable and mapped into the global address space. The [BBRAM](#) can be reprogrammed and is volatile. It keeps its content as long as it is powered by a battery that is mounted on the ZCU102 board. The [BBRAM](#) is not mapped into the global address space. It can only be accessed by the [CSU](#).

The eFuse memory is predestined to hold the hash value of a public key used by an asymmetric authentication method (here: RSA), where it does not matter whether the key is exposed, as long as it cannot be manipulated. For encryption with a symmetric method (here: AES), the [BBRAM](#) is more suitable as a key storage. Since the same key is used for encryption and decryption, it must be protected from being read out. However, it is not important to protect it from manipulation, as it is not used to ensure authenticity.

In terms of RSA authentication, the system supports the use of secondary key pairs. This means that an arbitrary amount of RSA key pairs can be generated and each Secondary Public Key (**SPK**) can be signed using the Primary Secret Key (**PSK**). Then, any of the Secondary Secret Keys (**SSKs**) can be used to sign software components. This approach can help to protect the **PSK** from being compromised. The Primary Public Key (**PPK**), whose hash value is programmed into the eFuse, can be revoked only once, whereas the key revocation mechanism is capable of revoking up to 256 **SPKs**.

Furthermore, it is differentiated between different key types. Plain text keys are called *Red Keys*. If one does not want to store a plain text key on the device, there are two more options. The key can be obfuscated, which means it can be encrypted with the family key provided by the vendor upon request (it is the same key for the entire device family). An obfuscated key is called *Grey Key*. Alternatively, a key can be encrypted with the *Key Encryption Key*, which is generated by the **PUF** and therefore unique. An encrypted key is called *Black Key*.

To reduce the risk of certain attacks when using AES encryption, the system also supports the use of rolling keys and operational keys, such that every key is used only for a small portion of data.

Further information on key management is provided in [11, p. 255 ff].

## 2.5. Boot Process

The boot process of the device is divided into multiple stages. The exact number of stages depends on how the user configured the boot image. For example, if a bare-metal application shall be executed, there is no need for a second-stage bootloader. Fig. 2.2 shows the boot process for a very common configuration, where a Linux **OS** is booted.

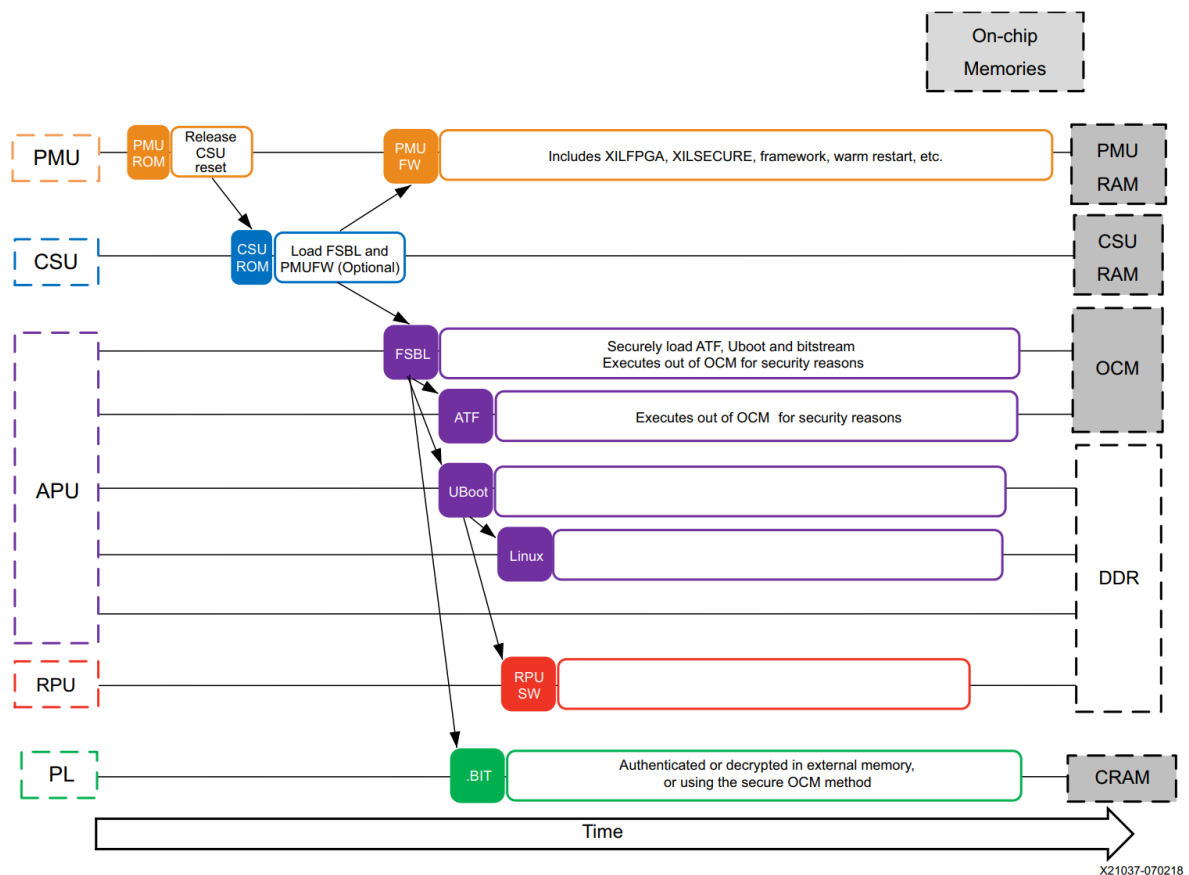
When the system is reset or powered up, the **PMU** at first executes the startup sequence from **PMU** ROM and performs the necessary initialisations. Then, it hands over to the **CSU** which reads the boot image from the desired boot source. It extracts the partition containing both the First Stage Bootloader (**FSBL**) and the **PMU** firmware from the boot image. Depending on the configuration, it may have to authenticate and decrypt the partition. Finally, it copies the **PMU** firmware to the **PMU** RAM and the **FSBL** to the **OCM**. Then, the **CSU** hands over to the **FSBL**. From this point, the **PMU** runs its new firmware and works independently.

The **FSBL** authenticates and decrypts other components contained in the boot image. This includes the Arm Trusted Firmware (**ATF**) (which is always required, even if TrustZone is not used) and U-Boot (which acts as the second-stage bootloader). The **ATF** is copied to the **OCM**, whereas U-Boot is placed somewhere in the external RAM. The **FSBL** also reads and authenticates the bitstream from the boot image. On success, it sends the encrypted bitstream

to the **CSU**, where it is decrypted and written to the configuration memory. Finally, the **FSBL** hands over to the **ATF** and then to U-Boot.

In the last stage, U-Boot can further load software. In this case, it loads the Linux kernel. U-Boot may load any software from any source, not necessarily from the boot image. If secure boot is enabled, U-Boot is an authenticated software component and may verify the authenticity of software components itself, if desired.

Further information on the boot process is provided in [11, p. 278 ff].



**Figure 2.2.:** Boot sequence of the Zynq MPSoC device [11, p. 279]

## 3. System Design

### 3.1. Non-Secure World – Linux

In the non-secure world, Linux shall be used as the **REE**. Linux is open-source, free to use, compact, highly customisable and provides many features by default. Furthermore, there are many applications and drivers available for Linux and it is the most used **OS** for cloud computing. Another reason for choosing Linux is that AMD provides tools that make it easy to build a Linux kernel for Zynq MPSoC devices.

### 3.2. Secure World – OP-TEE

Even though it would be possible to run bare-metal applications in the TrustZone, it is common to use a Trusted **OS** to implement the **TEE**. It provides a certain level of abstraction and therefore makes it easier to develop Trusted Applications (**TAs**), which will also be more portable. It also isolates multiple **TAs** from each other and implements communication with the **REE**.

There are several Trusted **OSs** available. However, multiple factors limit the number of options. The Trusted **OS** should be open-source, free to use, compatible with the Arm TrustZone technology and with the target hardware. Furthermore, it should provide a client application for the non-secure world that can be executed on Linux. Good documentation or online support would be helpful.

The *OP-TEE* project best meets these criteria. It is specifically made to run in the Arm TrustZone in combination with a Linux system as **REE**. In addition to a client application, it also provides a test suite which can be used to validate the correct functionality of the **TEE**, as well as multiple example **TAs**, which can be used for testing but also as a good starting point for **TA** development. OP-TEE is well-documented in the corresponding Wiki<sup>1</sup>. It explicitly supports the ZCU102 boards (even though this is poorly maintained at the moment).

### 3.3. Execution Hierarchy

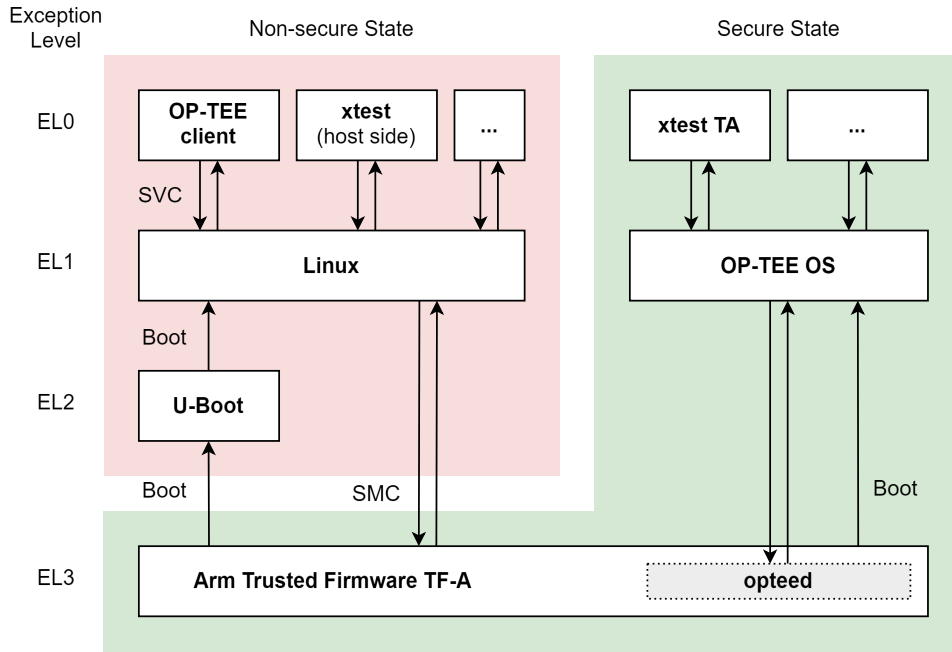
Next to Linux and OP-TEE, there is one more software component that is essential for the **TEE**: The *Secure Monitor* is a mandatory part of the TrustZone architecture. It must be executed by the CPU in the secure state and manages the context switches and communication between both worlds. From the non-secure world, the **OS** can communicate with the secure monitor via Secure Monitor Calls (**SMCs**). Inside the secure monitor software, these requests are delegated to a Secure Payload Dispatcher (**SPD**), which handles them either locally or forwards them to the Trusted **OS**.

---

<sup>1</sup><https://optee.readthedocs.io/en/latest/> (Accessed: 2024-01-28)

The Arm Trusted Firmware (**ATF**) is a secure monitor implementation provided by Arm. The term *Trusted Firmware-A* (**TF-A**) refers to a particular version of the **ATF** that is made for the Armv7-A and Armv8-A architectures. Since OP-TEE provides an **SPD**-plugin for it (called *opteed*), **TF-A** shall be used as the secure monitor.

The desired execution hierarchy is shown in Fig. 3.1.



**Figure 3.1.:** Desired hierarchy of software components

During the boot process, the **FSBL** hands off to the **ATF**, which shall call OP-TEE OS first. After OP-TEE OS has finished its initialisation, it returns to the **ATF**, which shall then call U-Boot. Finally, U-Boot may boot the Linux kernel as usual.

In the Linux root file system, the OP-TEE client, the test suite (*xtest*) and example **TAs** shall be installed. Furthermore, a **TEE** driver must be included in the Linux kernel. When a **TA** is being executed, the OP-TEE client can communicate with the Linux kernel via Supervisor Calls (**SVCs**). The Linux kernel can utilise the **TEE** driver to communicate with the **ATF** through **SMCs**. Within the **ATF**, OP-TEE's payload dispatcher delegates the calls to the OP-TEE OS, which in turn runs the **TA**. This communication path works in both directions.

It must be ensured that each component is executed with the exception level assigned in Fig. 3.1, such that it has the required privileges.

### 3.4. Secure Boot

Foremost, secure boot shall ensure the authenticity of software components before execution, to build up a chain of trust. However, in some cases, it may be desired to ensure confidentiality as

well (e.g. to protect intellectual property). Therefore, both authentication and encryption shall be enabled.

The Zynq MPSoC architecture provides two secure boot modes called *Encryption only* and *Root of Trust*. For the abovementioned reason, the *Root of Trust* mode shall be enabled. This mode uses RSA to ensure authenticity. This means that the private part of an RSA key pair is used to sign software and the public part of the key pair can be used to verify the signature.

On the Zynq MPSoC device, the *Root of Trust* mode can be activated by writing the hash value of the **PPK** into a particular eFuse register. This operation is irreversible. Once the register is written, the **CSU** firmware enforces the authentication of the **FSBL** before executing it. Equally, the **FSBL** must enforce authentication for any further software component it executes. Since both the **CSU** firmware and the **PPK** hash in the eFuse register cannot be manipulated, they form the *Root of Trust*.

To protect the **PSK** from being compromised, secondary keys shall be used. Every software component contained in the boot image shall be signed with an individual **SSK**. The corresponding **SPKs** shall be signed with the **PSK**. During boot, the signature of every software component shall be verified using the corresponding **SPK**. Then, the **SPK** shall be verified using the **PPK**. Finally, the **PPK** shall be verified by calculating its hash value and comparing it to the hash value in the eFuse register.

To be able to authenticate every software component and thus build up a chain of trust, additional information like the signatures, the **SPKs** and the **PPK** must be included in the boot image. This will be discussed in section 3.6. Furthermore, the hash of the **PPKs** must be written to the eFuse register, such that it cannot be exchanged. However, this is not suitable for development purposes. Thus, the **CSU** firmware provides a mode in which authentication is enabled but the comparison with the eFuse register is skipped. This mode shall be used instead of writing the eFuse.

Confidentiality shall be ensured during the boot process by activating AES encryption for every software component (= partition) contained in the boot image. Again, multiple modes can be used to accomplish this:

- **Single key mode:**

All partitions are encrypted with the same key. This key is contained in the *Secure Header (SH)* of every encrypted partition. The secure headers can be decrypted using the *Device Key*.

- **Multiple keys mode:**

Every partition is encrypted with its own unique key. The unique key of every partition is

stored in the [SH](#) of that partition. All Secure Headers can be decrypted using the *Device Key*.

- **Operational key mode:**

The *Device Key* is only used to decrypt the [SH](#) of the [FSBL](#) (384 bits), which contains the *Operational Key*. The *Operational Key* can be used to decrypt the first partition. All following partitions are decrypted using their own key, which is saved in the previous partition. This minimises the use of the *Device Key* and limits its exposure to side-channel attacks.

- **Rolling key mode:**

Every partition is split into smaller blocks, each one encrypted with its own unique key. The initial key is the *Device Key*, the key for each successive block is stored in the previous block. The block size can be defined by the user. This limits the use of every single key and improves resistance against Differential Power Attacks (DPA), which require measurements from multiple cryptographic operations using the same key. This mode can be combined with the operational key mode. Then, the initial key for every partition is not the *Device Key*.

To reduce the risk of attacks against the AES encryption, the operational key mode shall be used. Due to the changing number of required keys, the rolling key mode may cause difficulties during development. Therefore, it does not have to be used. However, instructions shall be provided on how to use the rolling key mode in combination with the operational key mode.

All of the above methods require a *Device Key* which is used to do the first step of decryption. Therefore, it must be stored on the target device. Since it is a symmetric key, it must be protected, such that it cannot be read out. In section 2.4 it has already been established that the [BBRAM](#) is the most suitable place for storing AES keys on the device. [BBRAM](#) does only support *red* keys. The key could also be included in the boot image as an alternative. However, this only works for *grey* or *black* keys and therefore requires much additional effort. Hence, the *Device Key* shall be stored in [BBRAM](#) as a *red* key.

Authentication and encryption must be activated for every part of the boot chain from the [FSBL](#) to the OP-TEE kernel. It is essential for the [TEE](#) that the chain of trust is continuous. However, for the Linux kernel, this is optional, as it is considered non-secure anyway.

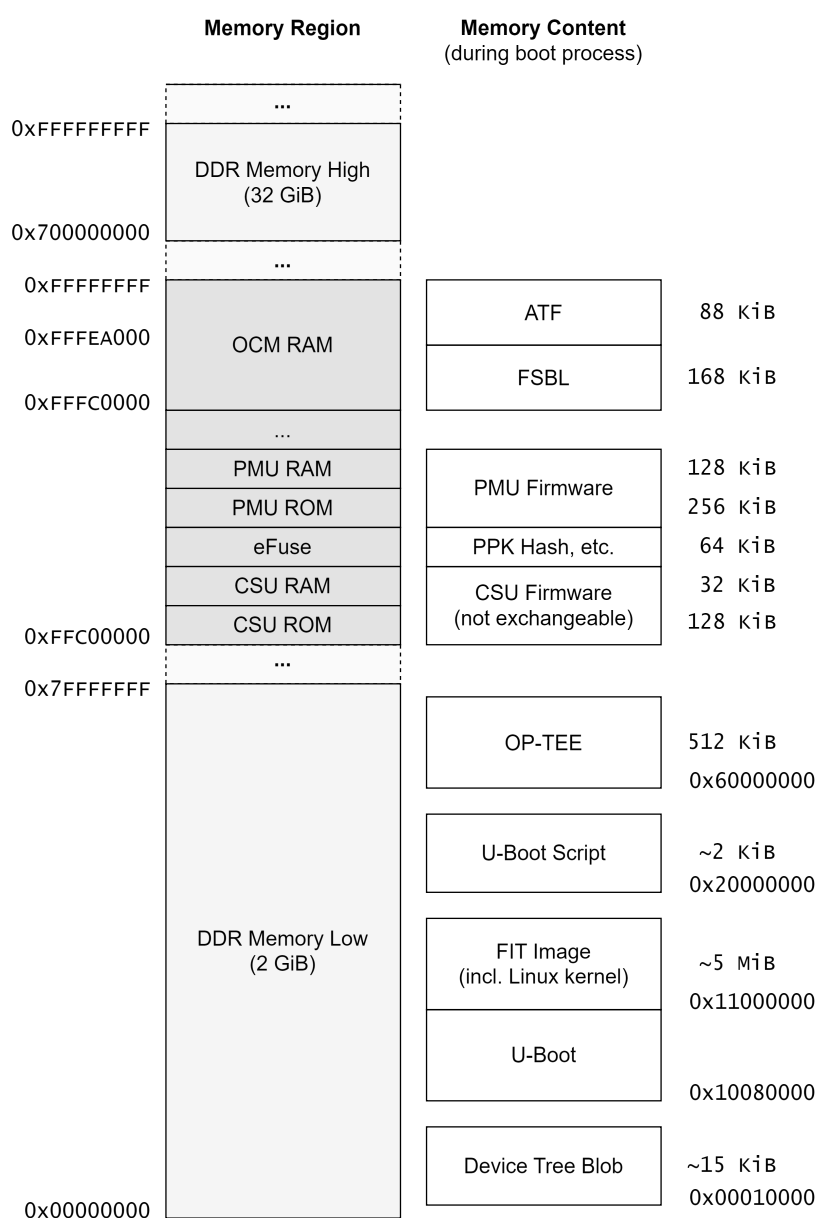
### 3.5. Memory Map

Only the 32-bit address space shall be used to avoid problems with incompatible software. Everything relevant is available within the 32-bit address space. Linux applications will be able to access the full memory space.



The memory map for the boot process has been defined as shown in Fig. 3.2. The assignments in the DDR memory region are just a matter of definition. The software components could be loaded to other addresses, as long as all dependent software components are configured accordingly.

Secure and non-secure address ranges can be defined via the **XMPU** and **XPPU** configuration registers. However, the default settings shall be kept, as this would exceed the scope of this project.



**Figure 3.2.:** Memory mapping during the boot process

### 3.6. Boot Image

To make it possible to verify the software components and thus allow the system to boot securely, additional information must be provided for each component. For example, the system needs to know which keys to use for verification and decryption. The signatures must be provided as well. Therefore, this information must be contained in the boot image.

The Zynq MPSoC architecture expects the boot image to be in a certain format. This format is specified in [12, p. 18 ff].

If a partition in the boot image is encrypted, a Secure Header (SH) must be appended to the partition. This header is encrypted as well and contains the key that is necessary to decrypt the partition. If a partition in the boot image is signed, an Authentication Certificate (AC) must be appended to the partition. This certificate contains the PPK, the SPK, the signature, and some other information. Additional metadata is contained in a header section at the beginning of the boot image. This header section is signed as well and has its own AC.

The resulting boot image for this project should look similar to the one shown in Fig. 3.3.

Boot Header			
Register Initialisation Table			
Image Header Table			
Image Header 1 (FSBL)			
Image Header 2 (Bitstream)			
Image Header 3 (ATF)			
Image Header 4 (DTB)			
Image Header 5 (U-Boot)			
Image Header 6 (OP-TEE OS)			
Partition Header 1.1 (FSBL)			
Partition Header 2.1 (Bitstream)			
Partition Header 3.1 (ATF)			
Partition Header 4.1 (DTB)			
Partition Header 5.1 (U-Boot)			
Partition Header 6.1 (OP-TEE OS)			
Header AC			
Partition 1 (FSBL)	PMU FW	SH	AC
Partition 2 (FPGA)		SH	AC
Partition 3 (ATF)		SH	AC
Partition 4 (DTB)		SH	AC
Partition 5 (U-Boot)		SH	AC
Partition 6 (OP-TEE OS)		SH	AC

**Figure 3.3.:** Structure of the resulting boot image

## 4. System Implementation

This chapter describes how the system defined in chapter 3 can be implemented. It provides a step-by-step tutorial which can be followed to reproduce the project. However, the resulting project can also be found on GitHub<sup>1</sup>.

It must be noted that this tutorial has only been tested with the tools and versions stated in section 4.1. It does not work with earlier versions and it is not guaranteed to work with later versions of these tools.

### 4.1. Tools

#### Vitis Embedded

Vitis Embedded is a package provided by AMD for software development. In earlier versions it was called 'Vitis IDE', not to be confused with 'Vitis HLS'. Vitis Embedded provides an IDE and toolchains for creating, building and debugging software applications for Xilinx processors. With Vitis Embedded, it is possible to develop Linux applications as well as bare-metal applications for any processing unit in Zynq MPSoC devices.

Due to some bugs in the latest versions, both versions 2023.1 and 2023.2 are required for this project.

#### Yocto Project

The Yocto Project is a collaborative open-source project by the Linux Foundation and multiple other organisations (including AMD) that provides a set of tools helping to create custom Embedded Linux distributions. It allows developers to build Linux-based embedded systems with an OS that is tailored to meet the respective requirements. Thus, developers can benefit from the proven Linux kernel and the large range of already available Linux applications and drivers, while at the same time reducing the overhead to a minimum. Furthermore, these customisations can be done independently from the hardware architecture of the target system, making it easy to migrate to other hardware platforms and to collaborate with other developers.

This is achieved by employing a layer model: Multiple layers can be built around the Linux kernel, each of which can add new aspects or override aspects of a lower layer. Thus, layers can be used to encapsulate independent customisations, which improves maintainability and reusability. Every layer consists of one or multiple *recipes* telling the build system what to do. More details can be found in the Yocto Project documentation<sup>2</sup>.

---

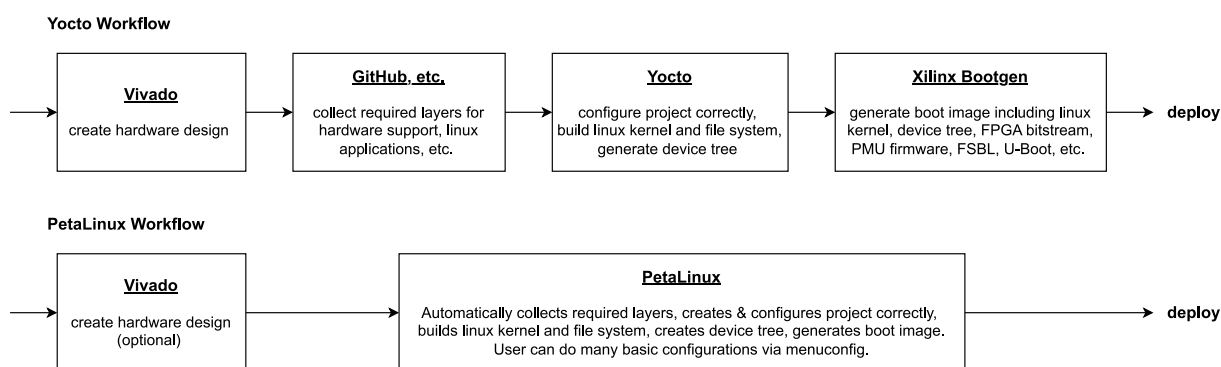
<sup>1</sup>[https://github.com/j-schacht/xilinx\\_zcu102\\_trustzone\\_demo](https://github.com/j-schacht/xilinx_zcu102_trustzone_demo) (Accessed: 2024-01-28)

<sup>2</sup><https://docs.yoctoproject.org/index.html> (Accessed: 2024-01-28)

In this project, Yocto will be used to build a Linux distribution for the ZCU102 board (which will act as the **REE**) and adapt it to support the execution of **TAs** in the **TEE**. Being familiar with the Yocto principles, terminology and project structure will help in understanding the following steps.

## PetaLinux Tools

PetaLinux is the name of a toolkit provided by AMD that can be used to build Linux boot images for Xilinx products. It is based upon Yocto and includes some additional Xilinx-specific tools. It provides its own command line interface and therefore acts as an abstraction layer to the Yocto build system. This makes it a lot easier to set up a project and obtain a bootable Linux image, as can be seen in Fig. 4.1.



**Figure 4.1.:** Workflow to obtain a bootable Linux image with Yocto vs. PetaLinux

PetaLinux internally creates a Yocto project with just some additional files, which means that the project can be customised like any other Yocto project. PetaLinux helps to set up a project easily, while still providing the full flexibility and portability of Yocto. Any existing Yocto layers can be used and custom recipes can be written in the same way. This, however, requires a certain knowledge of Yocto to make more advanced customisations.

In this project, PetaLinux 2023.2 is being used which, according to the Xilinx Wiki<sup>3</sup>, is based on Yocto 4.1 'Langdale'.

## Summary

Tab. 4.1 shows the tools and versions that are being used.

<sup>3</sup><https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841883/> (Accessed: 2024-01-28)

Tool	Version
AMD Vitis Embedded	2023.1 and 2023.2
AMD PetaLinux Tools	2023.2
Yocto Project	4.1 'Langdale' (included in PetaLinux Tools)

**Table 4.1.:** Used tools and versions

## 4.2. Environment Setup

### Operating System

The development system used for this project is running Debian 12. Unless stated otherwise, the tools are installed directly on this system. However, Debian is not officially supported by the Xilinx tools. It is recommended to use Ubuntu if available, which will cause the least problems. When using Debian or something else, a Virtual Machine (VM) running Ubuntu may be necessary to get some of the tools running.

### Installing Vitis Embedded

The installers for both required versions of Vitis Embedded can be downloaded from the Vitis section of the Xilinx Download website<sup>4</sup>. Select the correct version and download the 'Vitis Embedded Installer'. An AMD account is required.

It is possible to install Vitis Embedded on Debian. However, at the latest when trying to build a project, there will be errors due to incompatible packages. The easiest way to get it running is to install it in a Ubuntu VM. In this case, Ubuntu 22.04 LTS has been used. Refer to the user guide [13] for detailed installation instructions.

When running Vitis Embedded in a VM and Vitis needs direct access to the ZCU102 board (e.g. when using the debugger), it may be required to install the Xilinx hardware server on the host system and adapt the VM's network settings, such that it can access the hardware server on the host system.

Due to some bugs in the Vitis Embedded software, two versions are required. In version 2023.1 it is not possible to create boot images (e.g. for programming the BBRAM of the ZCU102). This feature is missing, but it has been added to version 2023.2. However, in version 2023.2 debugging Linux applications is not possible, as the connection to the TCF agent on the ZCU102 fails. This feature works in version 2023.1.

<sup>4</sup><https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis.html>  
(Accessed: 2024-01-28)

## Installing PetaLinux Tools

Installing PetaLinux in a **VM** is not recommended for performance reasons. Fortunately, PetaLinux can be installed on Debian, although it is not officially supported (it outputs a warning, but it works fine). The PetaLinux installer can be downloaded from the PetaLinux section of the Xilinx Download website<sup>5</sup>. Select the correct version and download the 'PetaLinux Installer' as well as the Board Support Package (**BSP**) for the ZCU102 board. An AMD account is required. Detailed installation instructions can be found in the user guide [14].

PetaLinux cannot be installed as root user. To ensure that there are no issues with file permissions, the installation directory should ideally be somewhere in the user's home directory. Here, `~/petalinux/2023.2/` has been used.

```
$ mkdir -p ~/petalinux/2023.2/
$ cd ~/Downloads/
$ chmod +x ./petalinux-v2023.2-10121855-installer.run
$ ./petalinux-v2023.2-10121855-installer.run -d ~/petalinux/2023.2 -p
  aarch64
$ source ~/petalinux/2023.2/settings.sh
```

The installation may not be successful on the first try or PetaLinux may fail when using it for the first time. Possible issues are:

- Missing packages:

This can happen especially when a not supported **OS** is used. In this case, one can check the release notes<sup>6</sup> of the respective PetaLinux version. There, a spreadsheet file is provided, describing all dependencies and a command to install all required packages at once. However, this list may not be complete (e.g. version 2023.2 also requires *libtinfo5*, which had to be installed manually).

- Wrong default system shell:

According to the user guide, the default system shell must be changed from *dash* to *bash*. However, the `sudo dpkg-reconfigure dash` command may not work, depending on the **OS** (even if it does not return an error message). There is an alternative way to change this (be careful, this could break your system!)<sup>7</sup>:

```
$ sudo ln -s bash /bin/sh.bash
$ sudo mv /bin/sh.bash /bin/sh
```

<sup>5</sup><https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html> (Accessed: 2024-01-28)

<sup>6</sup>[https://support.xilinx.com/s/article/000035572?language=en\\_US](https://support.xilinx.com/s/article/000035572?language=en_US) (Accessed: 2024-01-28)

<sup>7</sup><https://unix.stackexchange.com/questions/442510/how-to-use-bash-for-sh-in-ubuntu/442518#442518> (Accessed: 2024-01-28)

This can be reversed as follows:

```
$ sudo ln -s dash /bin/sh.dash
$ sudo mv /bin/sh.dash /bin/sh
```

- Wrong locale settings:

PetaLinux may output a warning *'cannot change locale (en\_US.UTF-8)'*, which causes some operations to fail. This may happen if the **OS** does not have American English installed as a locale. It can be fixed by executing `sudo dpkg-reconfigure locales`. In the upcoming dialog, select `en_US.UTF-8` (and any other locale if desired). In the following step, select a default locale. This does not have to be `en_US.UTF-8`.

The `~/petalinux/` directory will serve as the working directory in the following. Next to the PetaLinux installation, it will also be used for project files and everything else that is needed for this project. Therefore, the **BSP** should be placed here as well:

```
$ cd ~/Downloads/
$ cp ./xilinx-zcu102-v2023.2-10140544.bsp ~/petalinux/.
```

## Setting up Bootgen

Bootgen is a tool that is required for generating, signing and encrypting boot images for Xilinx devices. It does not need to be installed, as it is already included in PetaLinux. In fact, the `petalinux-package` command uses Bootgen internally. However, the `bootgen` command may not be available directly from the command line.

In the PetaLinux installation directory, Bootgen is located here:

```
./components/yocto/buildtools/sysroots/x86_64-petalinux-linux/usr/bin/
bootgen
```

One possible way to make it available from the command line is to create a link in a directory that is part of the search path:

```
$ cd ~/petalinux/2023.2/
$ ln ./components/yocto/buildtools/sysroots/x86_64-petalinux-linux/usr/
    bin/bootgen ./tools/common/petalinux/bin/
```

Now, the `bootgen` command should be available.

### 4.3. Preparing the SD Card

The ZCU102 board shall be booting from SD card. However, it will only be able to boot if the SD card has been partitioned correctly. There are some instructions provided in the PetaLinux user guide [14, appx. I]. Unfortunately, these instructions were not correct or incomplete at the time of testing. Therefore, a complete guide is provided in this section.

Connect an SD card to the development system. A capacity of 8 GB is sufficient. It is recommended to use an external card reader connected via USB, as inbuilt card readers can cause issues. List the available memory devices:

```
$ lsblk
```

Identify the SD card (e.g. `/dev/sdd`). Unmount all mounted partitions of that SD card:

```
$ sudo umount /dev/sdd?*
```

Now, `fdisk` can be used to create the required partitioning scheme:

```
$ sudo fdisk /dev/sdd
```

In the `fdisk` console, follow these steps:

- List existing partitions with **p**.
- Delete all existing partitions with **d**.
- Create a new empty DOS partition table with **o**. (It is important to use DOS. The Xilinx instructions use GPT, which does not work.)
- Create the first partition. This partition will be used for the boot images. Therefore, it should be large enough for all boot images (e.g. 512 MB).
  - Use **n** to create a new partition.
  - Partition type: primary
  - Partition number: 1
  - First sector: default (just hit enter)
  - Last sector: +512M
  - If `fdisk` asks to remove a signature: Yes
- Create the second partition. This partition will be used for the Linux root file system. It should be sufficiently large to contain the file system and have additional space for applications and user data. Since there is no third partition required, it can take up the remaining space of the SD card.



- Use **n** to create a new partition.
- Partition type: primary
- Partition number: 2
- First sector: default (just hit enter)
- Last sector: default (for maximum)
- If fdisk asks to remove a signature: Yes
- Write the changes to the SD card with **w**.

Exit fdisk. Make sure that all cached changes are written to the SD card. Then, list all partitions.

```
$ sync
$ lsblk
```

Identify the first and second partitions of the SD card (e.g. `/dev/sdd1` and `/dev/sdd2`). Create a FAT file system on the first partition and an ext4 file system on the second partition:

```
$ sudo mkfs.vfat -n BOOT /dev/sdd1
$ sudo mkfs.ext4 -L ROOT /dev/sdd2
$ sync
```

Now, the SD card is prepared.

## 4.4. Creating PetaLinux Project

So far, PetaLinux has been installed to `~/petalinux/2023.2/` and the [BSP](#) has been copied to `~/petalinux/`. Before using the PetaLinux commands, the environment must be set up. This must be done for every new Linux shell session.

```
$ source ~/petalinux/2023.2/settings.sh
```

Now, a new project can be created based on the [BSP](#). The project is called 'trustzone\_demo' and stored in `~/petalinux/projects/`.

```
$ mkdir -p ~/petalinux/projects/
$ cd ~/petalinux/projects/
$ petalinux-create -t project -n trustzone_demo -s ~/petalinux/xilinx-
  zcu102-v2023.2-10140544.bsp
```

This will create a new subdirectory 'trustzone\_demo' with all project files.

Optionally, to avoid a warning, change the project configuration:

```
$ cd ~/petalinux/projects/trustzone_demo/
$ petalinux-config
```

In the upcoming menu, open '*Image Packaging Configuration*' and disable '*Copy final images to tftpboot*'. Exit the menu and save the configuration.

## 4.5. Building & Booting a Basic Linux

### First Build

Before modifying the created project, it can be built and tested with the default settings to make sure that everything is set up correctly so far:

```
$ cd ~/petalinux/projects/trustzone_demo/  
$ petalinux-build
```

The build process will take a while (about 30 minutes). This highly depends on the performance of the development system and internet connection. PetaLinux creates a cache, which means that the first build may take long, but any subsequent builds will be faster. After the first build, the project can be adapted step by step in the way it is needed. After every change, the project can be rebuilt (using the same command) and tested.

If any errors occur during the first build, it may help to simply run `petalinux-build` again.

### Packaging the Boot Image

The output files of the build process can be found in `./images/linux/`. For now, the following files are relevant for booting the system:

- *zynqmp\_fsbl.elf* (FSBL executable)
- *pmufw.elf* (PMU firmware executable)
- *system.bit* (FPGA bitstream)
- *u-boot.elf* (U-Boot executable)
- *boot.scr* (U-Boot script)
- *image.ub* (FIT image<sup>8</sup> containing the Linux kernel, device tree blob and initramfs)
- *rootfs.tar.gz* (archive containing the root file system)

Unless a custom *.xsa* file (containing hardware description and bitstream) is provided, PetaLinux outputs a default bitstream, which corresponds to a blank hardware design. For the scope of this work, the default bitstream is sufficient.

Now, the set of individual boot chain components (FSBL, U-Boot, PMU firmware, FPGA bitstream) must be bundled into a single bootable image:

---

<sup>8</sup>[https://www.gibbard.me/linux\\_fit\\_images/](https://www.gibbard.me/linux_fit_images/) (Accessed: 2024-01-28)

```
$ cd ./images/linux/  
$ petalinux-package --boot --fsbl zynqmp_fsbl.elf --fpga system.bit  
  --u-boot --pmufw pmufw.elf
```

This generates a *BOOT.BIN* file in the current directory. When taking a closer look at the boot image, one would find that the `petalinux-package` command automatically includes the Arm Trusted Firmware (*bl31.elf*) and the device tree blob (*system.dtb*). Both are necessary for the device to boot. The structure of the *BOOT.BIN* file and of the FIT image can be inspected using the following commands:

```
$ bootgen -read ./images/linux/BOOT.BIN  
$ dumpimage -l ./images/linux/image.ub
```

## Copying Files to SD Card

All files required for the first test have been created. Now, they must be copied to the prepared SD card. After mounting both partitions of the SD card, copy the following files to the boot partition:

- *BOOT.BIN*
- *boot.scr*
- *image.ub*

Extract the root file system to the root partition of the SD card:

```
$ sudo tar -xf rootfs.tar.gz -C <path to root partition mount point>
```

Now, the SD card is ready to be used as a boot medium. Since the files on the SD card must be deleted and the above steps must be repeated after every rebuild, it may be helpful to automate the process of deleting, packaging, copying and extracting the files by writing a small script. Also, before removing the SD card, `sync` should be called and both partitions should be unmounted. Otherwise, the files may become corrupted and the boot process would fail.

## First Boot

On the ZCU102 board, the boot target can be selected using the DIP switch labelled 'SW6'. Change the DIP switch position to select the SD card as the boot target according to Tab. 4.2

Boot Mode	DIP Switch Position			
	1	2	3	4
JTAG	ON	ON	ON	ON
QSPI	ON	OFF	ON	ON
SD Card	ON	OFF	OFF	OFF

**Table 4.2.:** Boot mode selection using DIP switch 'SW6'

Now, insert the SD card into the board. Connect the UART USB port (J83) to the development system, which should automatically register four additional virtual serial ports. The bootloaders and the Linux kernel will output information via UART0 by default. Use a terminal software (e.g. minicom) to open the serial port with the lowest number (corresponds to UART0 on the ZCU102). The parameters can be found in Tab. 4.3. Finally, power up the ZCU102 board. If everything works as expected, it should be possible to watch the boot process in the serial terminal and eventually access the Linux command line. The default user with root privileges is 'petalinux'. A password must be chosen on the first login.

<b>Baudrate</b>	115200
<b>Data bits</b>	8
<b>Stop bits</b>	1
<b>Parity</b>	None
<b>Flow control</b>	None

**Table 4.3.:** Default parameters for UART communication

## 4.6. Helpful Customisations

This step is optional but recommended. Four customisations are made to make things easier:

- Include a text editor (*vim*),
- change the username of the default user,
- set a default password for the default user,
- configure static IP for access via SSH.

A text editor is a very useful tool, but PetaLinux does not include one by default. Fortunately, *vim* can be included easily:

```
$ cd ~/petalinux/projects/trustzone_demo
$ petalinux-config -c rootfs
```

In the upcoming menu, open '*Filesystem Packages*' --> '*console*' --> '*utils*' --> '*vim*'. Make sure that '*vim*' is enabled. Then, go back to the first page of the menu (do not leave yet).

By default, PetaLinux creates one Linux user named '*petalinux*'. When logging in for the first time, a password must be chosen for this user. Since the boot image and root file system will be recreated very often during this project, this can be time-consuming. Therefore, the username shall be shortened and a password shall be set by default.

On the first page of the menu, open '*PetaLinux RootFS Settings*'. Change the settings as shown in Fig. 4.2. This will create a user '*admin*' with root privileges and a default password '*admin*'. Finally, leave the configuration menu and save the changes.

```
(root:root;admin:admin;) Add Extra Users
(aie;) Create new Groups
(admin:audio,video,aie,input;) Add Users to Groups
(admin) Add Users to Sudo users
```

**Figure 4.2.:** PetaLinux RootFS settings recommended for development

Accessing the Linux shell via UART can be inconvenient, especially when using tools like *vim*. Setting up the Ethernet port with a static IP configuration and establishing a direct Ethernet connection between the ZCU102 and the development system makes it possible to connect to the system via SSH.

```
$ cd ~/petalinux/projects/trustzone_demo
$ petalinux-config
```

In the upcoming menu, open '*Subsystem AUTO Hardware Settings*' --> '*Ethernet Settings*'. Change the settings as shown in Fig. 4.3. Then, leave the configuration menu and save the changes.

```
Primary Ethernet (psu_ethernet_3) --->
[ ] Randomise MAC address
(ff:ff:ff:ff:ff:ff) Ethernet MAC address
[ ] Obtain IP address automatically
(192.168.6.2) Static IP address
(255.255.255.0) Static IP netmask
(192.168.6.1) Static IP gateway
```

**Figure 4.3.:** PetaLinux static IP settings recommended for development

After a rebuild, test the new configuration. Connect the ZCU102 to the development system via Ethernet. Apply the same static IP settings on the development system, but use a different IP address (e.g. 192.168.6.3). It may be necessary to adapt the firewall settings as well. Finally, connect to the system via SSH and try *vim*:

```
# login as 'admin' using the new password 'admin'
$ ssh -o UserKnownHostsFile=/dev/null admin@192.168.6.2

# try vim on the zcu102
admin@xilinx-zcu102:~$ vim ~/hello.txt
```

## 4.7. Secure Boot – Authentication

As specified in section 3.4, the system shall authenticate all software components during the boot process. RSA shall be used to sign and later verify each component. A non-tamperable hash value of the **PPK** shall act as the Root of Trust. Each individual software component shall be signed with a unique **SSK**. All **SPKs** shall be signed with the **PSK**.

Signing all components and compiling a boot image containing all required metadata and keys would be a lot of work if done by hand. Fortunately, there is a tool included in PetaLinux, which does this job: Bootgen can be used to create a boot image based on a Boot Image Format (**BIF**) file, which describes the desired boot image. A **BIF** file is a text file in which all relevant properties of the boot image can be defined, for example:

- Contained *partitions* (i.e. included files; not to be confused with SD card partitions),
- where a partition shall be placed,
- by which processor a partition shall be executed (if it contains software),
- partition signatures,
- which key to use for encryption/decryption,
- which key to use for signature generation/verification,
- etc.

Based on this information, Bootgen automatically builds a boot image in the format that is expected by the Zynq device – and handles all cryptographic operations (like signing and encrypting) in the background. Refer to the user guide [12] for more details.

Till now, the boot image has been generated using the `petalinux-package` command, which is easy to use but limited in possibilities. Instead, Bootgen can be called directly to get access to its full set of features. This is required to generate a boot image suitable for secure boot.

### Creating a BIF File

In the first step, a basic **BIF** file should be created, which makes Bootgen generate the same boot image as before, instead of `petalinux-package`. This **BIF** file can then be modified to enable authentication and encryption, making the boot process secure.

Since `petalinux-package` calls Bootgen internally, it also generates a **BIF** file, which is located here:

```
~/petalinux/projects/trustzone_demo/images/linux/bootgen.bif
```

Based on this file, a custom **BIF** file can be created. A well-formatted and commented example can be found in appendix A. The file can for example be stored here:

```
~/petalinux/projects/trustzone_demo/boot.bif
```

Note that the order of the partitions in the **BIF** file matters (even though the user guide does not mention this)! The following order must be used, otherwise, the system will not boot:

- **FSBL**
- **PMU** firmware
- **FPGA** bitstream
- **ATF**
- **Device Tree Blob (DTB)**
- **U-Boot**

### Building a Boot Image from the BIF File

Now, the boot image described by the created **BIF** file can be generated using the following command:

```
$ cd ~/petalinux/projects/trustzone_demo/  
$ bootgen -arch zynqmp -w -o images/linux/BOOT.BIN -image ./boot.bif
```

To test the new *BOOT.BIN* file, copy all required files to the SD card in the same way as before. If the **BIF** file is correct, the system should be able to boot and run Linux.

### Adding Authentication to the Boot Image

In this step, authentication will be enabled for all components included in the boot image. It is a good idea to make a copy of the **BIF** file and modify the copy (e.g. *boot\_auth.bif*):

```
$ cd ~/petalinux/projects/trustzone_demo/  
$ cp ./boot.bif ./boot_auth.bif
```

Now, some additional properties must be added to the **BIF** file:

In the global area, the path to the primary key file must be defined. The path to the secondary key file that is used to sign the boot header must be provided as well. It must be the same secondary key as for the **FSBL**. Furthermore, some parameters related to the key revocation feature must be provided. Here, the default values can be used, since key revocation will not be considered in this project.

```
...  
[auth_params] ppk_select=0; spk_select=spk-efuse; spk_id=0x00000000  
[pskfile]      keys/primary.pem  
[sskfile]      keys/secondary_fsbl.pem  
...
```

For each partition of the **BIF** file authentication must be enabled separately by adding four parameters to the partition; the authentication algorithm, the path to the respective secondary key file, and two parameters related to key revocation. The **PMU** firmware partition must not contain these parameters, as the **PMU** firmware is signed along with the **FSBL**.

```
...  
authentication = rsa,  
spk_select      = spk-efuse,  
spk_id          = 0x00000000,  
sskfile         = <path-to-secondary-key>  
...
```

To establish a Root of Trust, the hash value of the **PPK** must be written to the eFuse register of the **SoC**, such that it cannot be exchanged. Once the register has been written, authentication during boot is activated (and mandatory). For now, nothing shall be written to the eFuse register, as this is an irreversible operation. By using the following option in the **BIF** file, the authentication process can still be enabled, but the comparison between the calculated **PPK** hash and the eFuse value will be skipped:

```
[fsbl_config] bh_auth_enable
```

Be aware, that this option is for development purposes only. In production systems, this option must be removed from the **BIF** file and the eFuse must be written. Otherwise, the system is not secure!

The fully functioning **BIF** file can be found in appendix B. The comments provide more detailed information about the used parameters.

The last step before the boot image can be built is generating the required keys. When using the same secondary key for every partition (i.e. there are only two key pairs – the primary and the secondary key) Bootgen will automatically generate a new key if the respective key file does not exist. In this case, however, a unique secondary key pair shall be used for every partition. Bootgen cannot generate multiple secondary keys automatically. Therefore, it must be done manually. The easiest way is using *OpenSSL*, which is installed on most Linux systems by default. The keys must be 4096 bits long.

```
$ cd ~/petalinux/projects/trustzone_demo
```



```
$ mkdir keys
$ cd keys
$ openssl genrsa -out primary.pem 4096
$ openssl genrsa -out secondary_fsbl.pem 4096
$ openssl genrsa -out secondary_bit.pem 4096
$ openssl genrsa -out secondary_atf.pem 4096
$ openssl genrsa -out secondary_dtb.pem 4096
$ openssl genrsa -out secondary_ub.pem 4096
```

Finally, Bootgen can be called to generate the boot image:

```
$ cd ~/petalinux/projects/trustzone_demo/
$ bootgen -efuseppkbits images/linux/efuseppkhash.txt -arch zynqmp -w -o
  ./images/linux/BOOT.BIN -image ./boot_auth.bif
```

Bootgen will automatically generate the signature for each partition and include it in the header, together with the corresponding **SPK**. It will also automatically generate signatures for the **SPKs** by using the primary key. The above command will result in a bootable image and a text file *efuseppkhash.txt* containing the hash value of the **PPK**. This will be required if it needs to be written to the eFuse register.

## Testing Authentication

Copy the new boot image and the other required files to the SD card and boot the system. During the boot process, U-Boot outputs information about authentication and encryption. Without authentication, the U-Boot output looks like Fig. 4.4. Now, the output should look like Fig. 4.5.

```
U-Boot 2023.01 (Mar 29 2023 - 13:08:40 +0000)

CPU:   ZynqMP
Silicon: v3
Chip:   zu9eg
Model:  ZynqMP ZCU102 Rev1.0
Board:  Xilinx ZynqMP
DRAM:   2 GiB (effective 4 GiB)
PMUFW:  v1.1
PMUFW:  No permission to change config object
Xilinx I2C Legacy format at nvmem0:
Board name:  zcu102
Board rev:   1.0
Board SN:    881425371803-80024
Ethernet mac: 00:0a:35:04:aa:70
EL Level:    EL2
Secure Boot: not authenticated, not encrypted
Core:  77 devices, 31 uclasses, devicetree: board
NAND:   0 MiB
MMC:    mmc@ff170000: 0
```

**Figure 4.4.:** U-Boot output without authentication

```
U-Boot 2023.01 (Mar 29 2023 - 13:08:40 +0000)

CPU:   ZynqMP
Silicon: v3
Chip:   zu9eg
Model:  ZynqMP ZCU102 Rev1.0
Board:  Xilinx ZynqMP
DRAM:   2 GiB (effective 4 GiB)
PMUFW:  v1.1
PMUFW:  No permission to change config object
Xilinx I2C Legacy format at nvmem0:
Board name:  zcu102
Board rev:   1.0
Board SN:    881425371803-80024
Ethernet mac: 00:0a:35:04:aa:70
EL Level:    EL2
Secure Boot: authenticated, not encrypted
Core:  77 devices, 31 uclasses, devicetree: board
NAND:   0 MiB
MMC:    mmc@ff170000: 0
```

**Figure 4.5.:** U-Boot output with authentication

## Authenticating the Linux Kernel

*Note:* Securing Linux is not the primary goal of this project. Therefore, this step is optional.

So far, all software components included in the *BOOT.BIN* file are being authenticated during the boot process. However, the Linux kernel itself is contained in *image.ub* (the FIT image), which is a separate file. Once U-Boot is running, it reads the *image.ub* file directly from the SD card. Then, it runs integrity checks on the file content using hash values, but it does not authenticate the content (which would not even be possible at the moment, because the file is not signed). Therefore, the Linux kernel is still vulnerable to malicious manipulation. To avoid this, a partition for *image.ub* can be added to the *BIF* file, such that it is signed by Bootgen and included in the *BOOT.BIN* file<sup>9</sup>. It is possible to make the *FSBL* check the signature of *image.ub* during the boot process and, on success, load it to a well-defined address in RAM. Then, U-Boot only needs to boot from the corresponding address. This can be achieved by adding the following partition to the *BIF* file:

```
[
    destination_cpu      = a53-0,
    partition_owner      = fsbl,
    load                 = 0x11000000,
    authentication       = rsa,
    spk_select           = spk-efuse,
    spk_id               = 0x00000000,
    sskfile              = keys/secondary_fit.pem
] images/linux/image.ub
```

Now, the behaviour of U-Boot must be changed, such that it looks for the FIT image at the specified address in RAM, rather than on the SD card. If the boot target is set to *SD card*, U-Boot will scan the SD card for a file called *boot.scr*. If it finds the file, it will execute the script. The *boot.scr* script generated by PetaLinux tells U-Boot to load the *image.ub* file from the SD card into RAM and run the Linux kernel from there (see Fig. 4.6).

At first, a custom U-Boot script must be created to adapt the behaviour. The existing *boot.scr* can be used as a base:

```
$ cd ~/petalinux/projects/trustzone_demo/
$ tail -c+73 < ./images/linux/boot.scr > ./boot.cmd      # remove header
```

Now, *boot.cmd* can be changed to achieve the desired behaviour: When booting from an SD card, U-Boot shall not load a file from the storage device and boot directly from address 0x11000000

<sup>9</sup>as suggested here: [https://support.xilinx.com/s/question/0D54U00006MniTxSAJ/how-to-pack-imageub-and-bootscr-into-bootbin-in-petalinux-20221?language=en\\_US](https://support.xilinx.com/s/question/0D54U00006MniTxSAJ/how-to-pack-imageub-and-bootscr-into-bootbin-in-petalinux-20221?language=en_US) (Accessed: 2024-01-28)

instead. The adapted script can be found in appendix F. The new *boot.scr* can be generated with the following command (`mkimage` is part of PetaLinux):

```
$ cd ~/petalinux/projects/trustzone_demo/
$ mkimage -c none -A arm -T script -d ./boot.cmd ./boot.scr
```

To protect the resulting *boot.scr* file from manipulation another partition must be added to the BIF file, such that *boot.scr* is signed by Bootgen and authenticated by the FSBL before being executed by U-Boot:

```
[
    destination_cpu      = a53-0,
    partition_owner      = fsbl,
    load                 = 0x20000000,
    authentication       = rsa,
    spk_select           = spk-efuse,
    spk_id               = 0x00000000,
    sskfile              = keys/secondary_scr.pem
] boot.scr
```

Since a unique secondary key shall be used for each partition, keys for the new partitions must be generated as described previously. The complete **BIF** file can be found in appendix C.

Lastly, U-Boot must be modified to only load a script from the specified address in RAM. By default, U-Boot looks for a script at address `0x20000000`, if it cannot find the *boot.scr* file on the SD card. However, this is not sufficient, as it would allow manipulating the behaviour of U-Boot by simply adding a *boot.scr* file to the SD card. U-Boot should not check the SD card at all. This can be achieved by editing the following file:

```
~/petalinux/projects/trustzone_demo/project-spec/meta-user/recipes-bsp/u-boot/files/platform-top.h
```

Insert code to redefine `BOOTENV_DEV_MMC` as shown below. This manipulates U-Boot such that it behaves the same way as in JTAG mode, if the boot mode is set to SD card; it loads the script from the default script address `0x20000000`, without checking the SD card before.

```
#if defined(CONFIG_ARCH_ZYNQMP)
#include <configs/xilinx_zynqmp.h> // <-- insert below this line
#define BOOTENV_DEV_MMC_CUSTOM(devtypeu, devtypel, instance) \
    "bootcmd_" #devtypel #instance "=" \
    "echo mmc" #instance ": Trying to boot script at ${scriptaddr} " \
    "&& source ${scriptaddr}; " \
    "echo mmc" #instance ": SCRIPT FAILED: continuing...;\0"
#undef BOOTENV_DEV_MMC
#define BOOTENV_DEV_MMC BOOTENV_DEV_MMC_CUSTOM
#endif // <-- insert above this line
```

U-Boot must be recompiled to apply these changes. Then, the new boot image can be created using Bootgen:

```
cd ~/petalinux/projects/trustzone_demo/
petalinux-build
bootgen -efuseppkbits ./images/linux/efuseppkhash.txt -arch zynqmp -w -o
./images/linux/BOOT.BIN -image ./boot_auth.bif
```

In addition to a reduced attack surface, these additional steps provide another benefit: From now on, only the *BOOT.BIN* file needs to be copied to the boot partition of the SD card, as it includes *image.ub* and *boot.scr*. During the boot process, it can now be seen that U-Boot does not load these files from the SD card anymore. Instead, it assumes that everything has already been loaded to the specified addresses in RAM and boots directly from there (see Fig. 4.7).

```
scanning bus usb@fe200000 for devices... 1 USB Device(s) found
scanning usb for storage devices... 0 Storage Device(s) found
Hit any key to stop autoboot: 0
switch to partitions #0, OK
mmc0 is current device
Scanning mmc 0:1...
Found U-Boot script /boot.scr
3017 bytes read in 13 ms (226.6 KiB/s)
## Executing script at 20000000
Trying to load boot images from mmc0
15299488 bytes read in 1026 ms (14.2 MiB/s)
## Loading kernel from FIT Image at 10000000 ...
Using 'conf-system-top.dtb' configuration
Trying 'kernel-1' kernel subimage
Description: Linux kernel
Created: 2023-04-21 7:47:58 UTC
Type: Kernel Image
Compression: gzip compressed
```

**Figure 4.6.:** U-Boot output before modification

```
scanning bus usb@fe200000 for devices... 1 USB Device(s) found
scanning usb for storage devices... 0 Storage Device(s) found
Hit any key to stop autoboot: 0
mmc0: Trying to boot script at 20000000
## Executing script at 20000000
MMC/USB boot: Assuming that Fit image has been loaded to RAM by FSBL. Not loading anything from storage device.
Booting Fit image from address 0x11000000
## Loading kernel from FIT Image at 11000000 ...
Using 'conf-system-top.dtb' configuration
Trying 'kernel-1' kernel subimage
Description: Linux kernel
Created: 2023-04-21 7:47:58 UTC
Type: Kernel Image
Compression: gzip compressed
Data Start: 0x11000108
Data Size: 9777492 Bytes = 9.3 MiB
Architecture: AAarch64
OS: Linux
```

**Figure 4.7.:** U-Boot output after modification

Now, only the Linux root file system is not protected from manipulation. This problem could be addressed, for example, by encrypting the partition. However, this is a purely Linux-related topic and not the subject of this work. In this context, where Linux is intended to function as [REE](#), it is assumed to be insecure anyway.

## 4.8. Secure Boot – Encryption

At this point, the authenticity of all components in the boot image is ensured. The next step is to ensure confidentiality by encrypting the individual components. As specified in section 3.4, the encryption shall be done using AES and the operational key mode. This means that every partition shall be encrypted with its designated key. The individual key of every partition shall be stored in the previous partition. The first partition (the **FSBL**) is encrypted with the operational key, which is stored in the **FSBL** Secure Header. Only the **FSBL** Secure Header is encrypted with the device key. The device key shall be stored in the **BBRAM** as plain text.

### Adding Encryption to the Boot Image

Encrypting every partition and generating the required headers manually would be a big effort, but Bootgen can do this job automatically. This requires a few additions to the **BIF** file. It is a good idea to make a copy of the current **BIF** file and modify the copy (e.g. *boot\_auth\_encr.bif*):

```
$ cd ~/petalinux/projects/trustzone_demo/  
$ cp ./boot_auth.bif ./boot_auth_encr.bif
```

Now, some additional properties must be added to the **BIF** file:

In the global area, it must be specified where the device key is stored (here: Red key in **BBRAM**) and that the operational key mode shall be used:

```
...  
[keysrc_encryption] bbram_red_key  
[fsbl_config] bh_auth_enable, opt_key  
...
```

In every partition to be encrypted, two options must be added to activate the encryption and specify the path of the key file. The path of the key file should be unique since an individual key shall be used for every partition. Again, the **PMU** firmware partition must not contain these parameters.

```
...  
encryption = aes,  
aeskeyfile = keys/aes_fsbl.nky  
...
```

The adapted **BIF** file can be found in appendix D. Now, when calling Bootgen with this **BIF** file, it requires an additional parameter `-p` to specify the part name. It does not matter what is used as the part name. Bootgen requires it only to add it at the top of each AES key file. Using the

unique ID of the Zynq device may be a good choice. If the Vivado Design Suite is available, the ID can be obtained via the Vivado Hardware Manager by executing the following command:

```
report_property [current_hw_device] IDCODE_HEX
```

With the resulting ID, Bootgen can be called:

```
$ cd ~/petalinux/projects/trustzone_demo/
$ bootgen -efuseppkbits ./images/linux/efuseppkhash.txt -arch zynqmp -p
  "UID_??" -w -o ./images/linux/BOOT.BIN -image ./boot_auth_encr.bif
```

Bootgen will automatically generate all AES key files (\*.nky) that do not exist. These are text files containing the part name, the device key (*Key 0*), the partition key (*Key 1*) and the respective initialisation vectors (*IV 0*, *IV 1*). If an operational key is used, it will be included as well (*Key Opt*). A key file looks like this:

```
Device  UID_??.;

Key 0    24615FE579A92082FC8DB5CBCEEE2611D86A90E3A05FED9682757C683D677951;
IV 0     0C69AE3D51A529EE115324CF;

Key 1    AC87855DC26DC95016575AEE1598474E9D938D70B1924A35F195A7CD5381F40B;
IV 1     433D1D6BB3CAEEB24CD24E29;

Key Opt  A9B96EE6F160F4A88BD7FB0BE60732D015308F0D1A875E5EDDA8CCF9FB90F033;
```

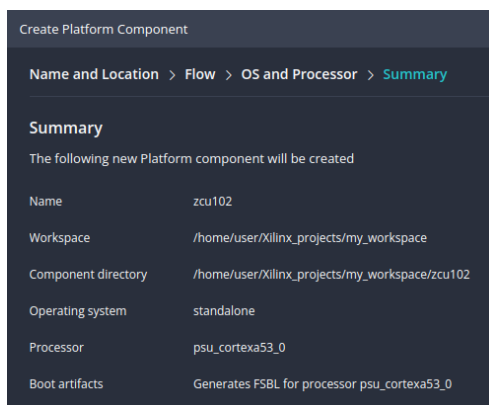
*Key 0* and *Key Opt* should be the same in all generated files. *Key 1* is the unique key of the respective partition and should be different in all key files.

## Writing the Device Key to BBRAM

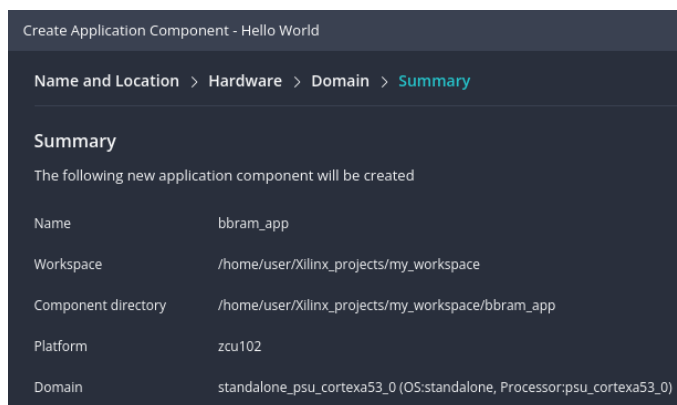
The device key can be specified via the .nky file that is used for the **FSBL**. If this file exists, Bootgen will read *Key 0* from there and use it as the device key. If this file does not exist, Bootgen will create it, which means that a new device key is generated.

If a new device key has been generated, it must somehow be applied to the target device. In this case, it shall be written to the **BBRAM**. AMD provides a guide which describes how the **BBRAM** and eFuse registers can be programmed [15]. Unfortunately, this guide does not work with the current version of Vitis Embedded. Therefore, an updated guide is provided in this section. Note that version 2023.2 of Vitis Embedded must be used. Version 2023.1 will not work.

1. Open Vitis Embedded. Select 'Open Workspace' and choose an empty directory (e.g. ~/petalinux/projects/write\_bbram/).



**Figure 4.8.:** Desired platform summary for BBRAM programming

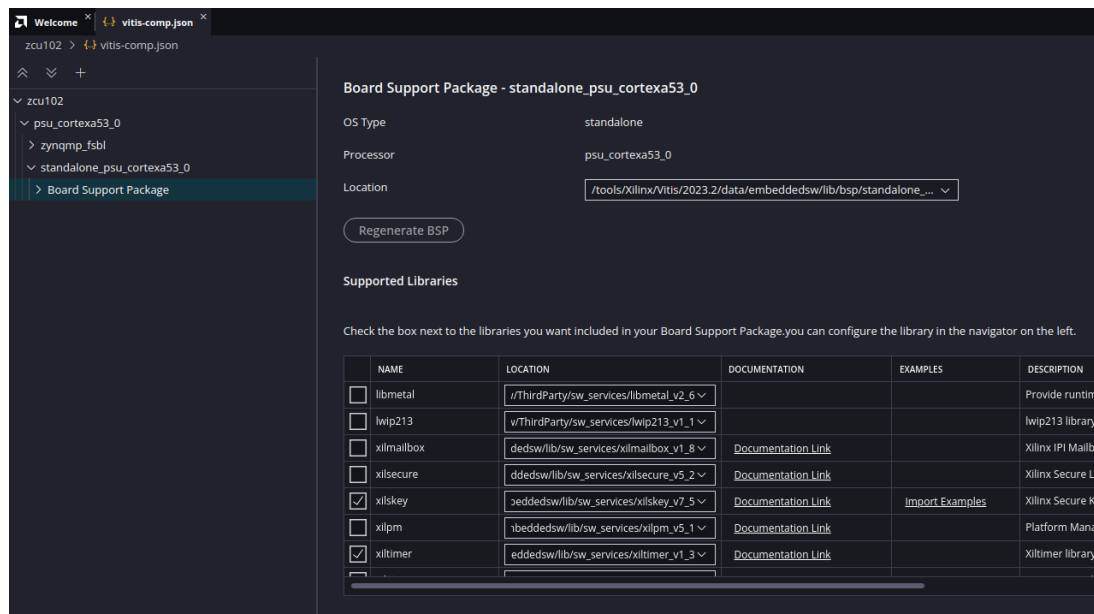


**Figure 4.9.:** Desired application summary for BBRAM programming

2. With the new workspace opened, select '*Create Platform Component*'.
  - Enter component name (e.g. '*zcu102*') and click '*Next*'.
  - Select '*Hardware Design*' and select '*zcu102*'. Click '*Next*'.
  - Select operating system '*standalone*' and processor '*psu\_cortexa53\_0*'. Activate '*Generate Boot artifacts*'. Select target processor to generate FSBL: '*psu\_cortexa53\_0*'. Click '*Next*'.
  - The summary should look like shown in Fig. 4.8.
  - Click '*Finish*'.
3. Once the platform has been created (which may take a while), click on '*Build*' in the '*FLOW*' section on the left side of the main window. Wait until the platform has been built.
4. The platform settings (*vitis-comp.json*) should open in the editor section automatically. Navigate to '*zcu102*' --> '*psu\_cortexa53\_0*' --> '*standalone\_psu\_cortexa53\_0*' --> '*Board Support Package*'. Activate the '*xilsky*' and '*xiltimer*' libraries as shown in Fig. 4.10. Click '*Regenerate BSP*' and wait until the process is done.
5. Build the platform again.
6. Unfortunately, the BBRAM example project provided by AMD cannot be selected from the examples in the IDE anymore. Therefore, it must be created manually. The '*Hello World*' example project is used as a base.
 

Click '*File*' --> '*New Component*' --> '*From Examples*'. Select the '*Hello World*' example and click '*Create Application Component from Template*'.

- Enter component name (e.g. '*bbam\_app*') and click '*Next*'.



**Figure 4.10.:** Desired BSP configuration for BBRAM programming

- Select the platform 'zcu102' and click 'Next'.
- Select domain 'standalone\_psu\_cortexa53\_0' and click 'Next'.
- The summary should look like shown in Fig. 4.9.
- Click 'Finish'.

7. Search for the example code in the Vitis installation directory, for example:

```
/tools/Xilinx/Vitis/2023.2/data/embeddedsw/lib/sw_services/
xilkey_v7_5/examples/xilkey_bbramps_zynqmp_example.c
```

Open the file and copy the code.

- Go back to Vitis. In the workspace explorer on the left, navigate to 'bb Bram\_app' --> 'Sources' --> 'src' --> 'helloworld.c'. Open the file and replace the content with the copied code.
- Open the .nky file that Bootgen generated for the FSBL, for example:

```
~/petalinux/projects/trustzone_demo/keys/aes_fsbl.nky
```

Copy the device key (Key 0) from this file. It should be encoded as a HEX string.

- Go back to Vitis. In the *helloworld.c* file, search for:

```
#define XSK_ZYNQMP_BBRAMPS_AES_KEY "000..."
```

Replace the string with the key copied from the .nky file (see Fig. 4.11).



11. Save the *helloworld.c* file. In the 'FLOW' section, select the '*bb Bram\_app*' component and click 'Build'.
12. Once the app has been built, click 'Create Boot Image', which is located underneath the 'Build' button.
  - Select 'Create a new BIF file'.
  - Select paths for the BIF file and boot image.
  - The dialog should look like shown in Fig. 4.12.
  - Click 'Create Image'.
13. The boot image (*BOOT.BIN*) will be generated and placed at the selected location. Copy the boot image to the boot partition of the SD card.
14. Finally, boot the ZCU102 board with this image. If the output via the UART interface looks like shown in Fig. 4.13, the key has been successfully written to the BBRAM.

```
Zynq MP First Stage Boot Loader
Release 2023.1   Nov 16 2023   - 18:33:10
PMU-FW is not running, certain applications may not be supported.
BBRAM example for ZynqMP
Successfully ran ZynqMP BBRAM example...
```

**Figure 4.13.:** Output message confirming successful BBRAM programming

*Note:* AMD also provides example code for writing the eFuse registers. It can be used in the same way as described above.

## Testing Encryption

Finally, with the device key programmed to BBRAM, the encrypted boot image can be tested. Copy the boot image and root file system generated in section '[Adding Encryption to the Boot Image](#)' to the SD card and boot the system. During the boot process, U-Boot outputs information about authentication and encryption. With just authentication, the U-Boot output looks like Fig. 4.5. Now, the output should look like Fig. 4.14.

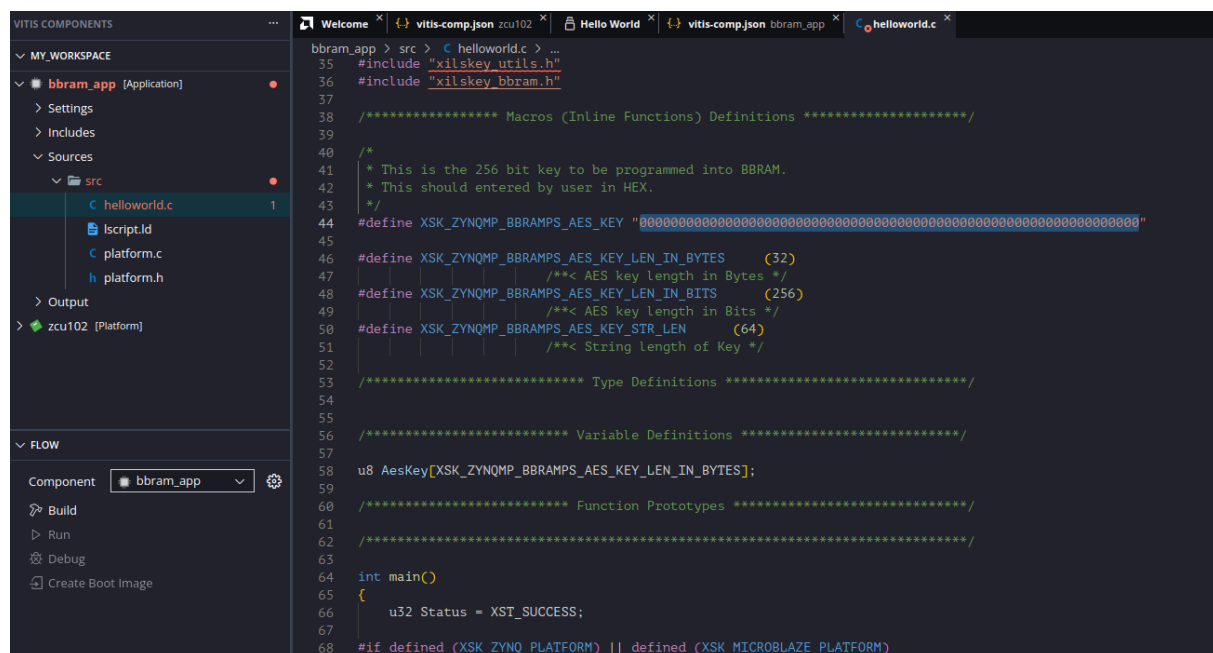


Figure 4.11.: Defining the key to be programmed to BBRAM

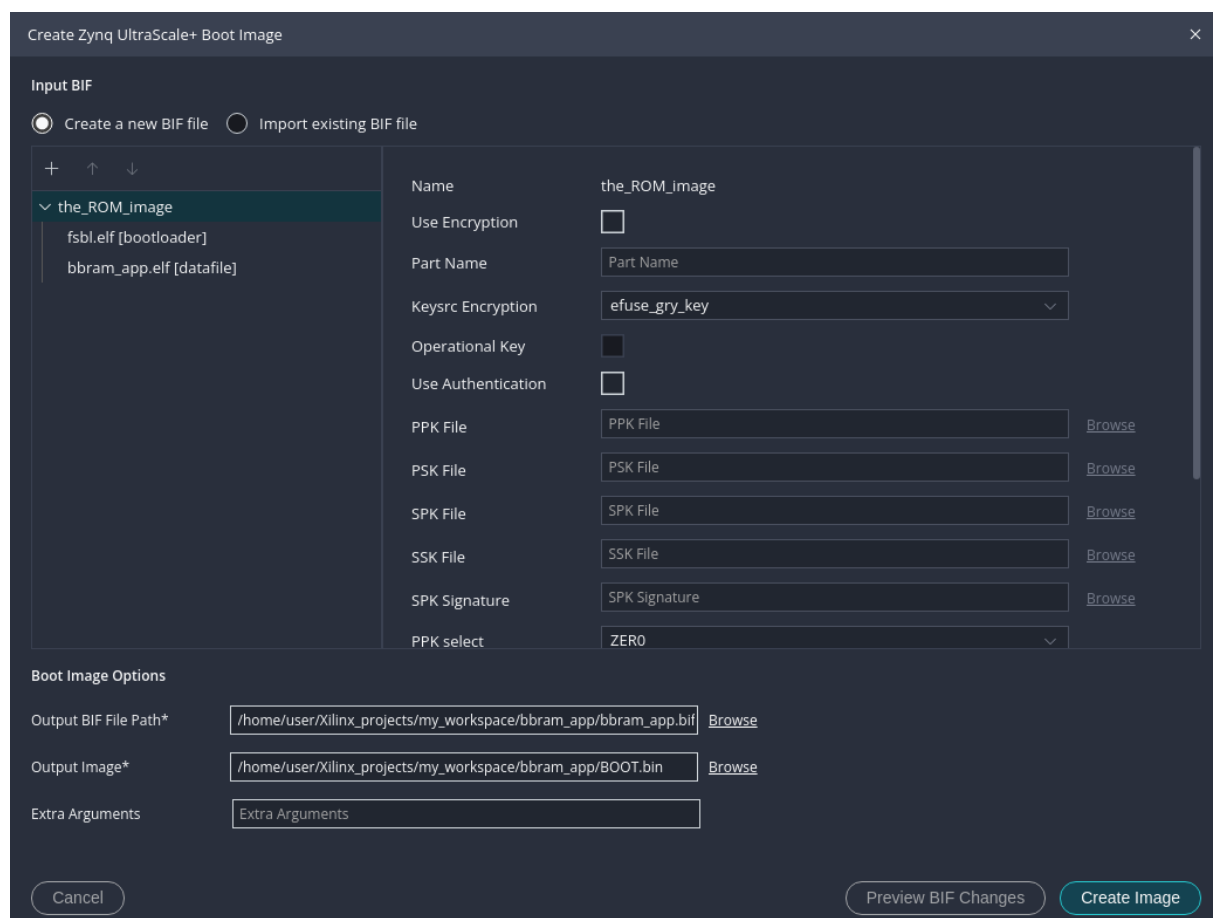


Figure 4.12.: Desired settings in the boot image creation dialog for BBRAM programming

```

U-Boot 2023.01 (Mar 29 2023 - 13:08:40 +0000)

CPU:   ZynqMP
Silicon: v3
Chip:   zu9eg
Model:  ZynqMP ZCU102 Rev1.0
Board:  Xilinx ZynqMP
DRAM:   2 GiB (effective 4 GiB)
PMUFW:  v1.1
PMUFW:  No permission to change config object
Xilinx I2C Legacy format at nvmem0:
Board name:  zcu102
Board rev:   1.0
Board SN:    881425371803-80024
Ethernet mac: 00:0a:35:04:aa:70
EL Level:   EL2
Secure Boot: authenticated, encrypted
Core:  77 devices, 31 uclasses, devicetree: board
NAND:   0 MiB
MMC:    mmc@ff170000: 0

```

**Figure 4.14.:** U-Boot output with authentication and encryption

## Rolling Key Mode

As explained in section 3.4, it would be best practice to use rolling keys for large partitions of the boot image (e.g. the Linux kernel). However, if the size of a partition changes, the number of required keys might not match the number of keys in the respective key file, which will cause Bootgen to fail. This can be fixed by deleting the key file, such that Bootgen generates new keys. To avoid this issue, rolling keys are not recommended to be used during development. The feature can be activated later by adding a single line to the respective partition in the [BIF](#) file, as shown here:

```

[
    ...
    encryption      = aes,
    aeskeyfile       = keys/aes_fit.nky,
    blocks           = 4096 (*)           # add this line
    ...
] images/linux/image.ub

```

This, for example, will make Bootgen split the *image.ub* partition into blocks of 4096 bytes and generate as many keys as needed for the number of blocks. These keys will all be written to the *aes\_fit.nky* file.

## 4.9. Setting up the TEE

Now that a secure boot chain has been established and the [REE](#) is set up, the next step is setting up the [TEE](#) based on the Arm TrustZone technology. First of all, a secure monitor must be installed, which is responsible for context switching between [REE](#) and [TEE](#) and for managing the communication between both environments. Without a secure monitor, TrustZone cannot be used. As stated in section 3.3, the Arm Trusted Firmware (more specifically: [TF-A](#)) shall be

used as the secure monitor. Within the secure world, instead of directly executing a bare-metal application, OP-TEE OS shall be used as the supervisor. Finally, the OP-TEE client shall be installed in the REE, such that TAs can be loaded from the REE file system and then verified and executed by OP-TEE OS in the TrustZone.

### OP-TEE Support for Zynq MPSoC Architecture

Different approaches can be found about how to get OP-TEE running on the Zynq MPSoC architecture. However, none of them works with current versions of PetaLinux.

Originally, the OP-TEE developers provided an example PetaLinux project for Zynq MPSoC. In their Wiki, they describe how to set up this example project<sup>10</sup>. Unfortunately, there are multiple problems with this approach:

- The example project has been made for OP-TEE 3.14.0 and relies on PetaLinux 2018.2. Neither is it compatible with newer versions of OP-TEE nor with newer versions of PetaLinux. Downgrading to PetaLinux 2018.2 is not an option, because a requirement of this project is the use of current software versions. Besides, PetaLinux 2018.2 does not work with current versions of Ubuntu or Debian (and is not compatible with current versions of Vivado either).
- Even with PetaLinux 2018.2, the provided example does not work when following the instructions in the OP-TEE Wiki. It seems that some changes have been made to the OP-TEE Git repositories afterwards, which are causing problems. In [8, ch. 5.8.2], the authors describe a workaround that makes it possible to compile the example project with PetaLinux 2018.2.

Via the corresponding Git repository, the authors of [8] published updated instructions afterwards, describing how OP-TEE can be built with PetaLinux 2019.2. Instead of using the example project, they are adding OP-TEE to an existing project by including the *meta-linaro* layer. However, as of 2021, OP-TEE support has been removed from the *meta-linaro* layer. It is now available in the *meta-arm* layer. Therefore, these instructions do not work anymore.

For OP-TEE versions later than 3.14.0 the Wiki provides instructions on how to build OP-TEE for Zynq MPSoC. However, the build system has been changed in these newer versions and there are no instructions on how to include the build results into an existing PetaLinux project. For including OP-TEE in a PetaLinux project, the Wiki refers to a Makefile from version 3.14.0, which has been written for PetaLinux 2020.2. Unfortunately, this Makefile no longer works; one reason being outdated URLs.

In summary, none of these approaches work with current versions of the tools in use. Nonetheless, they provide useful hints on how OP-TEE can be integrated into a PetaLinux project. On this ground, the following approach has been found, which makes it possible to build and include version 3.18.0 of OP-TEE into the existing project.

<sup>10</sup><https://optee.readthedocs.io/en/3.14.0/building/devices/zynqmp.html> (Accessed: 2024-01-28)

## Fetching Sources

Files from three different Git repositories are required. Therefore, these repositories must be downloaded:

```
$ cd ~/petalinux
$ git clone https://git.yoctoproject.org/meta-arm/
$ cd meta-arm/
$ git checkout langdale
$ cd ~/petalinux
$ git clone https://github.com/openembedded/openembedded-core.git
$ cd openembedded-core/
$ git checkout langdale
$ mkdir ~/petalinux/optee/
$ cd ~/petalinux/optee/
$ git clone https://github.com/OP-TEE/build.git
$ cd build/
$ git checkout tags/3.14.0
```

For the *meta-arm* and *openembedded-core* repository, the branch corresponding to the used version of Yocto (here: 'Langdale') must be checked out. For the OP-TEE *build* repository, version 3.14.0 must be checked out, even though a different version of OP-TEE shall be built. This is necessary because the required files are missing in newer versions.

To ensure reproducibility, Tab. 4.4 shows the identifiers of the commits that have been used here.

Repository	URL	Commit ID
meta-arm	<a href="https://git.yoctoproject.org/meta-arm/">https://git.yoctoproject.org/meta-arm/</a>	4a9d140e4e50c8d91e5c3cccdc1081004b34b2b0
openembedded-core	<a href="https://github.com/openembedded/openembedded-core.git">https://github.com/openembedded/openembedded-core.git</a>	78211cda40eb018a3aa535c75b61e87337236628
OP-TEE build	<a href="https://github.com/OP-TEE/build.git">https://github.com/OP-TEE/build.git</a>	4a5b6176a6150ffce8951b31ea60f3c8195fe970

**Table 4.4.:** Used versions of the required Git repositories

## Adding OP-TEE Recipes to the PetaLinux Project

The *meta-arm* layer provides recipes for building the OP-TEE OS and the OP-TEE client with Yocto. In addition, it also provides a test suite and example applications (which are part of the OP-TEE project). Services from the Trusted Services project<sup>11</sup> are available as well, but not used here.

Copy the recipes from the *meta-arm* layer into the user layer of the PetaLinux project:

<sup>11</sup><https://trusted-services.readthedocs.io/> (Accessed: 2024-01-28)

```
$ cd ~/petalinux
$ cp -r ./meta-arm/meta-arm/recipes-security ./projects/trustzone_demo/
  project-spec/meta-user/recipes-security
```

An additional option must be added to four of the recipes. It tells Yocto that these recipes are compatible with the ZCU102 platform. This can be done by creating a *.bbappend* file for each recipe (or appending to the file if it already exists):

```
$ cd ~/petalinux/projects/trustzone_demo/project-spec/meta-user/recipes-
  security/optee
$ echo 'COMPATIBLE_MACHINE:zynqmp = "zynqmp-generic|xilinx-zcu102"' >>
  ./optee-client_%.bbappend
$ echo 'COMPATIBLE_MACHINE:zynqmp = "zynqmp-generic|xilinx-zcu102"' >>
  ./optee-test_%.bbappend
$ echo 'COMPATIBLE_MACHINE:zynqmp = "zynqmp-generic|xilinx-zcu102"' >>
  ./optee-examples_%.bbappend
$ echo 'COMPATIBLE_MACHINE:zynqmp = "zynqmp-generic|xilinx-zcu102"' >>
  ./optee-os-taddevkit_%.bbappend
```

Append the following code to the *optee-os\_%.bbappend* file in the same directory. The code originates from *~/petalinux/optee/build/zynqmp/optee/optee-os\_%.bbappend*, but the *COMPATIBLE\_MACHINE* and *OPTEEOUTPUTMACHINE* options have been modified and the syntax has been adapted to the newer version of Yocto:

```
# append to ~/petalinux/projects/trustzone_demo/project-spec/meta-user/
  recipes-security/optee/optee-os_%.bbappend:

OPTEEMACHINE = "zynqmp-zcu102"
OPTEEOUTPUTMACHINE = "zynqmp-zcu102"
COMPATIBLE_MACHINE:zynqmp = "zynqmp-generic|xilinx-zcu102"
EXTRA_OEMAKE:append = " CFG_TEE_CORE_LOG_LEVEL=2"
PLNX_DEPLOY_DIR ?= "${TOPDIR}/images/linux"

do_compile:append() {
    ${S}/scripts/gen_tee_bin.py --input ${B}/core/tee.elf \
        --out_tee_raw_bin ${B}/core/tee_raw.bin
}

do_install:append() {
    install -m 644 ${B}/core/tee.elf \
        ${D}${nonarch_base_libdir}/firmware/tee.elf
}

deploy_optee() {
    install -d ${PLNX_DEPLOY_DIR}
    install -m 644 ${DEPLOYDIR}/optee/tee_raw.bin \
        ${PLNX_DEPLOY_DIR}/tee_raw.bin
    install -m 644 ${DEPLOYDIR}/optee/tee.elf ${PLNX_DEPLOY_DIR}/bl32.elf
}
```

```
do_deploy[postfuncs] += " deploy_optee"  
do_deploy_setscene[postfuncs] += " deploy_optee"
```

Now, modify the *optee.inc* file in the same directory as shown below:

```
# ~/petalinux/projects/trustzone_demo/project-spec/meta-user/recipes-  
security/optee/optee.inc:  
  
# replace this line:  
OPTEEMACHINE ?= "${MACHINE}"  
  
# by this line:  
OPTEEMACHINE = "zynqmp-zcu102"
```

Finally, add the recipes for the OP-TEE client and OP-TEE OS to the project's base set of packages, such that PetaLinux builds them automatically:

```
$ echo 'IMAGE_INSTALL:append = " optee-os optee-client"' >> ~/petalinux/  
projects/trustzone_demo/project-spec/meta-user/conf/petalinuxbsp.conf
```

The *optee-test* package contains the test suite, which can be used to test if OP-TEE is working correctly. The *optee-example* package provides some example TAs. Both packages are optional. Therefore, it is recommended to add them to the list of packages that can optionally be included in the root file system:

```
$ echo 'CONFIG_optee-test' >> ~/petalinux/projects/trustzone_demo/project  
-spec/meta-user/conf/user-rootfsconfig  
$ echo 'CONFIG_optee-examples' >> ~/petalinux/projects/trustzone_demo/  
project-spec/meta-user/conf/user-rootfsconfig
```

Now, the packages can be activated via the PetaLinux configuration menu:

```
$ cd ~/petalinux/projects/trustzone_demo/  
$ petalinux-config -c rootfs
```

In the upcoming menu, open '*user packages*' and enable both packages for now. They can be disabled later if not required. Exit the menu and save the changes.

## Resolving Dependencies

The OP-TEE recipes rely on some Python packages:

- python3-pyelftools,
- python3-pycryptodome,
- python3-pycryptography.

The recipes for these packages can be taken from the *openembedded-core* layer:

```
$ mkdir -p ~/petalinux/projects/trustzone_demo/project-spec/meta-user/
  recipes-devtools/python
$ cd ~/petalinux/openembedded-core/
$ cp ./meta/recipes-devtools/python/python3-pyelftools_0.29.bb ~/
  petalinux/projects/trustzone_demo/project-spec/meta-user/recipes-
  devtools/python/.
$ cp ./meta/recipes-devtools/python/python3-pycryptodome_3.15.0.bb ~/
  petalinux/projects/trustzone_demo/project-spec/meta-user/recipes-
  devtools/python/.
$ cp ./meta/recipes-devtools/python/python3-cryptography_37.0.4.bb ~/
  petalinux/projects/trustzone_demo/project-spec/meta-user/recipes-
  devtools/python/.
$ cp ./meta/recipes-devtools/python/python-pycryptodome.inc ~/petalinux/
  projects/trustzone_demo/project-spec/meta-user/recipes-devtools/
  python/.
$ cp -r ./meta/recipes-devtools/python/python3-cryptography/ ~/petalinux/
  projects/trustzone_demo/project-spec/meta-user/recipes-devtools/
  python/.
```

### Setting up the Secure Monitor (ATF)

As already mentioned in 'Packaging the Boot Image', the ATF is already included in the boot image, because the ZCU102 board cannot boot without a secure monitor. Even if the TrustZone feature is not used, the secure monitor is still needed to execute the OS or bare-metal application in the non-secure world. Therefore, PetaLinux always generates a **TF-A** executable (*bl31.elf*). However, it is necessary to make some changes to the ATF, such that it will execute OP-TEE OS. This can be done by adding another *.bbappend* file to the user layer:

```
$ mkdir -p ~/petalinux/projects/trustzone_demo/project-spec/meta-user/
  recipes-bsp/arm-trusted-firmware
$ touch ~/petalinux/projects/trustzone_demo/project-spec/meta-user/
  recipes-bsp/arm-trusted-firmware/arm-trusted-firmware_%.bbappend
```

Add the following build options to the newly created file:

```
EXTRA_OEMAKE:append = " DEBUG=0"
EXTRA_OEMAKE:append = " LOG_LEVEL=LOG_LEVEL_NOTICE"
EXTRA_OEMAKE:append = " RESET_TO_BL31=1"
EXTRA_OEMAKE:append = " NEED_BL32=1"
EXTRA_OEMAKE:append = " SPD=opteed"
EXTRA_OEMAKE:append = " ZYNQMP_BL32_MEM_BASE=0x60000000"
EXTRA_OEMAKE:append = " ZYNQMP_BL32_MEM_SIZE=0x80000"
```

With these options, the CPU reset vector is set to the ATF (*BL31*) entry point. The secure payload (*BL32*), which will be OP-TEE OS in this case, is defined to be an external executable placed at address `0x60000000` with a maximum size of `0x80000` bytes. Furthermore, the OP-TEE



dispatcher is selected as the [SPD](#), which will delegate the Secure Monitor Calls to OP-TEE OS. It is necessary to build the [ATF](#) in release configuration with a log level not higher than 'notice'. Otherwise, the executable would be too large for the designated [OCM](#) area.

All other [ATF](#) configurations can be made via PetaLinux:

```
$ cd ~/petalinux/projects/trustzone_demo/  
$ petalinux-config
```

In the upcoming menu, open '*Trusted Firmware ARM (TF-A) Configuration*'. Then, enable '*TF-A memory settings*' and apply the following changes:

- Set '*TF-A MEM SIZE*' to 0x16001. This is the maximum size of the [ATF](#) executable.
- Set '*TF-A MEM BASE*' to 0xFFFEA000. This is the address of the [ATF](#) executable in [OCM](#).
- Set '*PRELOADED BL33 BASE*' to 0x10080000. This is the address of the U-Boot executable in DDR RAM.
- Disable '*TF-A debug*'.

Close the menu and save the changes.

## Modifying the Linux Kernel Configuration

To allow the OP-TEE client to communicate with OP-TEE OS via the [ATF](#), it is necessary to enable the corresponding device driver. This can be done via PetaLinux:

```
$ cd ~/petalinux/projects/trustzone_demo/  
$ petalinux-config -c kernel
```

In the upcoming menu, open '*Device Drivers*', enable and open '*Trusted Execution Environment support*' and activate '*OP-TEE*'. Close the menu and save the changes.

Now, extend the device tree with the OP-TEE firmware device. This device serves as an interface to the [TEE](#). Open the device tree file in the user layer:

```
~/petalinux/projects/trustzone_demo/project-spec/meta-user/recipes-bsp/  
device-tree/files/system-user.dtsi
```

Insert an entry for OP-TEE as shown below:

```

/include/ "system-conf.dtsi"
/ {
    firmware {
        optee {
            compatible = "linaro,optee-tz";
            method = "smc";
        };
    };
};

```

## Extending the BIF File

With the previous modifications, PetaLinux will create a new file *tee\_raw.bin* during the build process, which contains the compiled OP-TEE OS. Since OP-TEE OS is loaded and initialised during the start procedure of the ATF, it must be included in the boot image. Furthermore, authentication during the boot process is crucial to ensure a continuous chain of trust to create a TEE. Therefore, the BIF file must be extended, such that Bootgen signs and encrypts the OP-TEE OS binary and finally includes it in the boot image.

As before, it is recommended to make a copy of the current BIF file and modify the copy (e.g. *boot\_auth\_encr\_tee.bif*):

```

$ cd ~/petalinux/projects/trustzone_demo/
$ cp ./boot_auth_encr.bif ./boot_auth_encr_tee.bif

```

Now, the following partition must be added to the BIF file:

```

[
    trustzone      = secure,
    destination_cpu = a53-0,
    exception_level = el-1,
    load           = 0x600000000,
    startup        = 0x600000000,
    authentication = rsa,
    spk_select     = spk-efuse,
    spk_id         = 0x000000000,
    sskfile        = keys/secondary_tee.pem,
    encryption     = aes,
    aeskeyfile     = keys/aes_tee.nky
] images/linux/tee_raw.bin

```

Most of the parameters are identical to those of the other partitions. However, it must be specified that OP-TEE OS must only be executed by the CPU in secure mode (i.e. in the TrustZone) and the exception level must be set to supervisor mode. In addition, the load and entry point addresses must be defined, such that the FSBL loads the binary to the correct location in RAM and the ATF starts execution at the correct address. The final BIF file can be found in appendix E.

Again, a new RSA key must be generated for the partition. The AES key will be generated automatically.

```
$ cd ~/petalinux/projects/trustzone_demo/
$ openssl genrsa -out keys/secondary_tee.pem 4096
```

## Build & Test

Finally, the modified project can be built. Afterwards, Bootgen can be called with the new **BIF** file to generate the boot image:

```
$ cd ~/petalinux/projects/trustzone_demo/
$ petalinux-build
$ bootgen -efuseppkbits ./images/linux/efuseppkhash.txt -arch zynqmp -p
  "UID_??" -w -o ./images/linux/BOOT.BIN -image ./boot_auth_encr_tee.
  bif
```

Copy the boot image and the root file system to the SD card and power up the ZCU102. The boot log should now look similar to Fig. 4.15 (check output via the other UART interfaces if this is not the case).

```
Zynq MP First Stage Boot Loader
Release 2023.2 Oct 12 2023 - 15:51:06
NOTICE: BL31: Secure code at 0x60000000
NOTICE: BL31: Non secure code at 0x80000000
NOTICE: BL31: v2.8(release):xlnx_rebase_v2.8_2023.2_ksb_sep
NOTICE: BL31: Built : 12:21:43, Aug 31 2023
I/TC:
I/TC: OP-TEE version: 3.18.0-dev (gcc version 12.2.0 (GCC)) #1 Fri Jul 15 09:06:00 UTC 2022 aarch64
I/TC: WARNING: This OP-TEE configuration might be insecure!
I/TC: WARNING: Please check https://optee.readthedocs.io/en/latest/architecture/porting_guidelines.html
I/TC: Primary CPU initializing
I/TC: Primary CPU switching to normal world boot

U-Boot 2023.01 (Sep 21 2023 - 11:02:37 +0000)
:
[ 1.627907] optee: probing for conduit method.
[ 1.632005] optee: revision 3.18 (1ee64703)
[ 1.632164] optee: dynamic shared memory is enabled
[ 1.641230] optee: initialized driver
:
[ OK ] Started TEE Suppllicant.
```

**Figure 4.15.:** Boot log of FSBL, ATF, OP-TEE OS and Linux kernel (excerpts)

After the **FSBL** has been executed, the **ATF (BL31)** prints some messages which, among other information, contain the addresses of the non-secure code (U-Boot) and the secure code (OP-TEE OS). Then, the **ATF** hands off to OP-TEE OS, which outputs its version identifier and a warning message, initialises the CPU, and finally switches back to the non-secure world to continue the

boot process. Finally, U-Boot is executed and initiates the Linux kernel startup. In the Linux kernel log it can be seen that the OP-TEE driver is initialised and the TEE supplicant is started. The warning message printed by OP-TEE OS can be ignored for now. It is a reminder to ensure that the porting guidelines have been followed and all possible vulnerabilities have been eliminated before a system goes into production. This refers, for example, to the exchange of pre-generated keys (see section 4.12).

From the Linux shell, the test suite can be executed to verify that OP-TEE is functioning correctly. Run the test using the following command:

```
$ sudo xtest
```

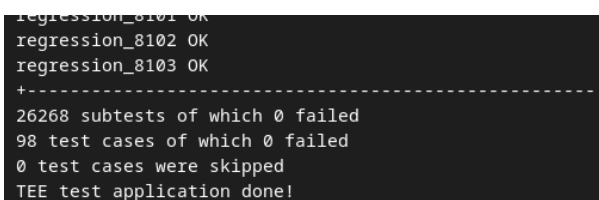
The test takes a while and prints a summary once finished (see Fig. 4.16). If the test fails, it may be necessary to manually start the TEE supplicant daemon and run the test again:

```
$ sudo tee-supplciant -d  
$ sudo xtest
```

If the test is successful, the example applications can be tested, for example:

```
$ sudo optee_example_hello_world
```

Refer to the OP-TEE Wiki for further details about the test suite<sup>12</sup> and the example TAs<sup>13</sup>.



```
regression_8101 OK  
regression_8102 OK  
regression_8103 OK  
+-----  
26268 subtests of which 0 failed  
98 test cases of which 0 failed  
0 test cases were skipped  
TEE test application done!
```

**Figure 4.16.:** OP-TEE test suite output if the TEE is working properly

## 4.10. Developing Linux Applications

Now that both environments – REE and TEE – are running, applications can be developed. The easiest way to develop Linux applications is by using Vitis Embedded, which includes the required toolchains for the Zynq MPSoC platform. The Xilinx Embedded Design Tutorial<sup>14</sup> explains how to build and debug Linux applications using Vitis IDE 2022.2. With Vitis

<sup>12</sup>[https://optee.readthedocs.io/en/3.18.0/building/gits/optee\\_test.html](https://optee.readthedocs.io/en/3.18.0/building/gits/optee_test.html) (Accessed: 2024-01-28)

<sup>13</sup>[https://optee.readthedocs.io/en/3.18.0/building/gits/optee\\_examples/optee\\_examples.html](https://optee.readthedocs.io/en/3.18.0/building/gits/optee_examples/optee_examples.html)  
(Accessed: 2024-01-28)

<sup>14</sup><https://xilinx.github.io/Embedded-Design-Tutorials/docs/2022.2/build/html/docs/Introduction/ZynqMPSoC-EDT/6-build-linux-sw-for-ps.html> (Accessed: 2024-01-28)

Embedded the same steps can be followed, except that the GUI is different and some menus and dialogs may have a different structure. The included example projects are a good starting point for custom projects.

*Note:* The debugging session is established via a TCF agent on the target system, which is included by default and started automatically. A network connection between the development system and the ZCU102 is used for communication between Vitis Embedded and the TCF agent. However, this feature seems to have a bug in Vitis Embedded 2023.2 causing the connection to fail. Therefore, version 2023.1 should be used here!

## 4.11. Developing Trusted Applications

Developing [TAs](#) is not as straightforward as developing Linux applications. [TAs](#) commonly consist of two parts – a Linux userspace application (*'host application'*) and the application that is executed in the [TEE](#). Both parts require separate toolchains that the developer has to build beforehand.

A guide on how to download and build the required toolchains and how to compile a [TA](#) is available in [8, ch. 5.8.2]. After cloning the OP-TEE [OS](#) and OP-TEE client repository as described in the guide, make sure to check out the correct version (3.18.0) in each repository before calling `make`. Once the toolchains are in place, the OP-TEE examples repository provides a good starting point for the development of own [TAs](#).

[TAs](#) must be signed before they can be executed by OP-TEE [OS](#). Executing only authenticated applications ensures that the [TEE](#) is not compromised with malicious software. Pay attention to the information on this in section [4.12](#).

## 4.12. Notes on Key Management

In production, keys must be generated in a secure environment with proper random number generation. Furthermore, secure storage with strict access restrictions must be ensured. It is important to note that these aspects have not been considered in the previous steps. Depending on the specific development and production environment, additional steps may be necessary to ensure that keys (particularly those forming the Root of Trust) are not compromised. Bootgen provides some features to help protect those keys:

With Bootgen it is possible to create the boot image without having to provide the private RSA keys. Instead of providing the full RSA key pairs, it is possible to provide only the public key and the signature of each partition. This requires that the signatures have been generated before (which requires the private key). Thus, the signatures could be generated by the secure system

that stores the keys, but the boot image could be created by a different, less secure system. More details can be found in [8, ch. 5.2.1] and the Bootgen user guide [12].

With the AES keys used for encryption, this is not possible in the same way. Since AES is a symmetric-key algorithm, the same key must be used for encryption and decryption. This means that the AES key is not only required by Bootgen to encrypt the sections. It must also be deployed to the target device, such that the data can be decrypted during boot. However, using the operational key mode it is possible to protect the device key in a development environment to a certain degree, as explained in [16, p. 101 ff].

OP-TEE uses RSA to verify **TA**s loaded from the **REE** file system. The public key needed for signature verification is compiled into OP-TEE **OS**. Every **TA** must be signed with the corresponding private key. Therefore, the **TA** signatures could be generated by the secure system that stores the private key, and only the public key gets exposed to developers, such that OP-TEE **OS** can be built and deployed. Unfortunately, OP-TEE does not offer as sophisticated key management as the Zynq architecture. Only in version 3.20.0, which does not support Zynq MPSoC, secondary keys have been introduced<sup>15</sup>.

Be aware that OP-TEE comes with a default RSA key pair. When following this guide, the default public key will be included in OP-TEE **OS**. This means that OP-TEE **OS** would accept any **TA** that has been signed with the publicly available private key, making the **TEE** obsolete. In a production environment, a new key pair must be generated and stored securely, as it is part of the Root of Trust. Refer to the OP-TEE Wiki<sup>16</sup> for more details.

---

<sup>15</sup><https://optee.readthedocs.io/en/3.20.0/architecture/subkeys.html> (Accessed: 2024-01-28)

<sup>16</sup>[https://optee.readthedocs.io/en/3.18.0/building/trusted\\_applications.html#signing-of-tas](https://optee.readthedocs.io/en/3.18.0/building/trusted_applications.html#signing-of-tas) (Accessed: 2024-01-28)

## 5. Results & Discussion

### 5.1. Achievements

This work showed how a hybrid system consisting of a **REE** and a **TEE** can be set up on an AMD Zynq UltraScale+ MPSoC device – more specifically the ZCU102 Evaluation Kit. The Arm TrustZone technology has been leveraged to isolate the secure world from the non-secure world within a single CPU.

As a mandatory foundation for the **TEE**, the boot process has been secured, such that authenticity and confidentiality are ensured for all relevant software components, thus building up a continuous chain of trust. Linux has been chosen as **OS** for the **REE**, whereas the **TEE** is based on OP-TEE **OS**. The corresponding OP-TEE client and driver have been included in the Linux kernel. A test suite and a set of example applications have been included in the Linux user space. By running the test suite, it has been validated that the system is functioning correctly. Finally, the toolchains required to develop **TAs** have been built and installed.

As a result, the development system is now ready to build custom **TAs** and the ZCU102 is ready to execute of such **TAs**.

Design decisions have been discussed and justified in the **System Design** chapter. The **System Implementation** chapter has been written in the style of a tutorial, making it possible to easily reproduce the project for the ZCU102 board. However, instructions were not just given but also explained in detail. Background information about the tools in use has been provided as well. Thereby, this work conveys a certain knowledge about the tools, processes and architectures, making it possible to adapt the given instructions to other target devices of the Zynq MPSoC family and also to transfer the concepts to different architectures.

The project files are publicly available on GitHub<sup>1</sup>, such that this project can be used as a foundation for other projects.

### 5.2. Future Work

Even though the **TEE** with OP-TEE **OS** is installed and appears to be fully functioning, there are still some questions that have not been answered in this work but are relevant for the development of certain **TAs**. For example, it needs to be investigated how memory regions and peripherals can be defined as either secure or non-secure and to what extent this can be done dynamically during runtime. Furthermore, it must be understood how OP-TEE's memory management works, how access to peripherals is possible through OP-TEE **OS** and how the system handles illegal memory access attempts.

---

<sup>1</sup>[https://github.com/j-schacht/xilinx\\_zcu102\\_trustzone\\_demo](https://github.com/j-schacht/xilinx_zcu102_trustzone_demo) (Accessed: 2024-01-28)

Based on this, it could be experimented with custom hardware designs. So far, only the default bitstream created by PetaLinux has been used. The possibility of including external hardware designs allows for the implementation of custom peripherals in the **PL**. In this context, it must be figured out how to make these peripherals TrustZone-compatible, such that they can be isolated and used by **TAs**.

Coming back to the primary motivation of this project, the next milestone would be the development of a **TA** that performs a (partial) reconfiguration of the **PL** and also ensures that the configuration memory can only be read and written from the **TEE**. This functionality would be an elementary step towards porting the *trusted shell* of the *TruFPGA* protocol [5] to a TrustZone-based **TEE**.

It is important to note that no full security analysis has been conducted as part of this project. There are known attacks against the Arm TrustZone architecture in general and against the Zynq MPSoC architecture in particular, as shown in [17]. Furthermore, several attack paths exist for the Zynq 7000 series, as shown in [18]. Zynq MPSoC devices may be still vulnerable to some of these methods. Hence, it must be analysed in which ways the **TEE** configuration presented in this work could be attacked and which countermeasures can be taken.

Lastly, a long-term challenge will be to keep this project updated to work with future versions of the tools. Experience has shown that backwards compatibility is not always given with AMD's tools. The dependencies between many different projects aggravate this problem.



## List of Acronyms

<b>AC</b>	Authentication Certificate
<b>APU</b>	Application Processing Unit
<b>ATF</b>	Arm Trusted Firmware
<b>BBRAM</b>	Battery Backed RAM
<b>BIF</b>	Boot Image Format
<b>BSP</b>	Board Support Package
<b>CSP</b>	Cloud Service Provider
<b>CSU</b>	Configuration and Security Unit
<b>DTB</b>	Device Tree Blob
<b>FSBL</b>	First Stage Bootloader
<b>OCM</b>	On-Chip Memory
<b>OS</b>	Operating System
<b>PL</b>	Programmable Logic
<b>PMU</b>	Platform Management Unit
<b>PPK</b>	Primary Public Key
<b>PS</b>	Processing System
<b>PSK</b>	Primary Secret Key
<b>PUF</b>	Physical Unclonable Function
<b>REE</b>	Rich Execution Environment
<b>RPU</b>	Real-Time Processing Unit
<b>SH</b>	Secure Header
<b>SMC</b>	Secure Monitor Call
<b>SoC</b>	System-on-Chip
<b>SPD</b>	Secure Payload Dispatcher
<b>SPK</b>	Secondary Public Key
<b>SSK</b>	Secondary Secret Key
<b>SVC</b>	Supervisor Call
<b>TA</b>	Trusted Application
<b>TEE</b>	Trusted Execution Environment
<b>TF-A</b>	Trusted Firmware-A
<b>VM</b>	Virtual Machine
<b>XMPU</b>	Xilinx Memory Protection Unit
<b>XPPU</b>	Xilinx Peripheral Protection Unit

## Bibliography

- [1] A. Paju, M. O. Javed, J. Nurmi, J. Savimäki, B. McGillion, and B. B. Brumley, “SoK: A Systematic Review of TEE Usage for Developing Trusted Applications,” in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, no. 34. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1–15, [Online]. Available: <https://doi.org/10.1145/3600160.3600169>. Accessed: 2024-02-07.
- [2] K. Xia, Y. Luo, X. Xu, and S. Wei, “SGX-FPGA: Trusted Execution Environment for CPU-FPGA Heterogeneous Architecture,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 301–306, [Online]. Available: <https://doi.org/10.1109/DAC18074.2021.9586207>. Accessed: 2024-02-07.
- [3] T. Nakai, D. Suzuki, and T. Fujino, “Towards Isolated AI Accelerators with OP-TEE on SoC-FPGAs,” in *Applied Cryptography and Network Security Workshops*, J. Zhou et al., Ed. Cham: Springer International Publishing, Sep. 2022, pp. 200–217, [Online]. Available: [https://doi.org/10.1007/978-3-031-16815-4\\_12](https://doi.org/10.1007/978-3-031-16815-4_12). Accessed: 2024-02-07.
- [4] D. Dik and M. S. Berger, “Control Plane Isolation of Network Security Protocols using FPGA-SoC Trusted Execution Environment,” in *2023 IEEE Nordic Circuits and Systems Conference (NorCAS)*, 2023, pp. 1–6, [Online]. Available: <https://doi.org/10.1109/NorCAS58970.2023.10305445>. Accessed: 2024-02-07.
- [5] S. Zeitouni, J. Vliegen, T. Frassetto, D. Koch, A.-R. Sadeghi, and N. Mentens, “Trusted Configuration in Cloud FPGAs,” in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 233–241, [Online]. Available: <https://doi.org/10.1109/FCCM51124.2021.00036>. Accessed: 2024-01-04.
- [6] T. La, K. Pham, J. Powell, and D. Koch, “Denial-of-Service on FPGA-based Cloud Infrastructures – Attack and Defense,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 3, p. 441–464, Jul. 2021, [Online]. Available: <https://doi.org/10.46586/tches.v2021.i3.441-464>. Accessed: 2024-02-07.
- [7] T. M. La, K. Matas, N. Grunchevski, K. D. Pham, and D. Koch, “FPGADefender: Malicious Self-oscillator Scanning for Xilinx UltraScale + FPGAs,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 3, article 15, pp. 1–31, Sep. 2020, [Online]. Available: <https://doi.org/10.1145/3402937>. Accessed: 2024-02-07.
- [8] T. Vögeli and T. Delafontaine, “Secure Boot for System on Chip,” Bachelor’s thesis, Institute of Embedded Systems, Zurich University of Applied Sciences, Winterthur, Switzerland, 2020, [Online]. Available: [https://github.com/InES-HPMM/ZYNQ\\_USplus\\_secure\\_boot\\_reference\\_design/blob/master/BA20\\_rosn\\_02\\_Secure\\_Boot\\_final.pdf](https://github.com/InES-HPMM/ZYNQ_USplus_secure_boot_reference_design/blob/master/BA20_rosn_02_Secure_Boot_final.pdf). Accessed: 2024-01-04.
- [9] Advanced Micro Devices Inc., “Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891, v1.10),” Nov. 2022, [Online]. Available: <https://docs.xilinx.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview>. Accessed: 2024-02-01.

- [10] Advanced Micro Devices Inc., “ZCU102 Evaluation Board: User Guide (UG1182, v1.7),” Feb. 2023, [Online]. Available: [https://www.xilinx.com/support/documents/boards\\_and\\_kits/zcu102/ug1182-zcu102-eval-bd.pdf](https://www.xilinx.com/support/documents/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf). Accessed: 2024-01-29.
- [11] Advanced Micro Devices Inc., “Zynq UltraScale+ Device: Technical Reference Manual (UG1085 v2.4),” Dec. 2023, [Online]. Available: <https://docs.xilinx.com/viewer/book-attachment/dqE2tE0k~iMhpEDoQwXIKg/11~gTSJf1Bn~faUQSW8eVw>. Accessed: 2024-02-11.
- [12] Advanced Micro Devices Inc., “Bootgen User Guide (UG1283, v2023.2),” Oct. 2023, [Online]. Available: <https://docs.xilinx.com/viewer/book-attachment/8bGQIqhQAeVmRLAdguryjQ/~fT24P0xKE58nt~k4mgyzg>. Accessed: 2024-01-29.
- [13] Advanced Micro Devices Inc., “Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400, v2023.2),” Dec. 2023, [Online]. Available: <https://docs.xilinx.com/viewer/book-attachment/X1QD5u5YoN5xSktsiEhkDA/zld1saCKysgBEBLkAJe75A>. Accessed: 2024-01-29.
- [14] Advanced Micro Devices Inc., “PetaLinux Tools Documentation: Reference Guide (UG1144, v2023.2),” Oct. 2023, [Online]. Available: [https://docs.xilinx.com/viewer/book-attachment/e0BYEp0v1FOLLZbAK0f81Q/6aaU6yhYNCmJwcb4Q\\_5nnQ](https://docs.xilinx.com/viewer/book-attachment/e0BYEp0v1FOLLZbAK0f81Q/6aaU6yhYNCmJwcb4Q_5nnQ). Accessed: 2024-01-29.
- [15] Xilinx Inc., “Programming BBRAM and eFUSES (XAPP1319, v2.1),” Nov. 2020, [Online]. Available: <https://docs.xilinx.com/v/u/en-US/xapp1319-zynq-usp-prog-nvm>. Accessed: 2024-01-29.
- [16] Advanced Micro Devices Inc., “Zynq UltraScale+ MPSoC Software Developer Guide (UG1137, v2023.2),” Nov. 2023, [Online]. Available: <https://docs.xilinx.com/viewer/book-attachment/tYXNk70H1W6QCEQEMDYPXQ/Muo10AFCUt7fcqYyiroSTw>. Accessed: 2024-01-29.
- [17] M. Gross, N. Jacob, A. Zankl, and G. Sigl, “Breaking TrustZone memory isolation and secure boot through malicious hardware on a modern FPGA-SoC,” *Journal of Cryptographic Engineering*, vol. 12, pp. 181–196, Sep. 2022, [Online]. Available: <https://doi.org/10.1007/s13389-021-00273-8>. Accessed: 2024-01-04.
- [18] E. M. Benhani, L. Bossuet, and A. Aubert, “The Security of ARM TrustZone in a FPGA-Based SoC,” *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1238–1248, 2019, [Online]. Available: <https://doi.org/10.1109/TC.2019.2900235>. Accessed: 2024-01-04.

## A. Basic BIF-File

This BIF file creates a simple boot image without authentication or encryption.

```
the_ROM_image:
{
    /* --- first stage bootloader settings --- */
    [
        bootloader,          /* mark this as the FSBL */
        destination_cpu      = a53-0, /* shall be executed on the Cortex-A53 CPU */
        exception_level      = el-3    /* exception level 3 = secure monitor mode */
    ] images/linux/zynqmp_fsbl.elf    /* path to the FSBL executable */

    /* --- PMU firmware settings --- */
    [
        pmufw_image          /* mark this as the PMU firmware */
    ] images/linux/pmufw.elf        /* path to the PMU firmware executable */

    /* --- FPGA bitstream settings --- */
    [
        destination_device   = pl      /* shall be applied to the FPGA */
    ] images/linux/system.bit      /* path to the FPGA bitstream */

    /* --- Arm Trusted Firmware settings --- */
    [
        /* ATF is an implementation of a secure monitor, which manages the TrustZone. */

        trustzone            = secure, /* must only be executed in secure mode */
        destination_cpu      = a53-0, /* shall be executed on the Cortex-A53 CPU */
        exception_level      = el-3    /* exception level 3 = secure monitor mode */
    ] images/linux/bl31.elf        /* path to the ATF executable */

    /* --- device tree blob settings --- */
    [
        destination_cpu      = a53-0, /* shall be available on the Cortex-A53 CPU */
        load                 = 0x00100000 /* load partition to this memory address */
    ] images/linux/system.dtb      /* path to the dtb file */

    /* --- u-boot settings --- */
    [
        destination_cpu      = a53-0, /* shall be executed on the Cortex-A53 CPU */
        exception_level      = el-2    /* exception level 2 = hypervisor mode */
    ] images/linux/u-boot.elf      /* path to the u-boot executable */
}
```

## B. BIF-File with Authentication

This BIF file creates a boot image with authentication but without encryption.

```
the_ROM_image:
{

/* --- general authentication settings --- */

/* use first primary public key;
 * use spk-efuse register to check if secondary public key has been revoked;
 * use SPK ID = 0, since no secondary public key has been revoked yet */
[auth_params] ppk_select=0; spk_select=spk-efuse; spk_id=0x00000000

/* path to primary secret file. only the public key will be included in the resulting image. */
[pskfile] keys/primary.pem

/* path to secondary secret file. this will be used to sign and verify the boot header.
 * only the public key will be included in the resulting image. for some reason, the boot
 * header must use the same key pair as the FSBL. */
[sskfile] keys/secondary_fsbl.pem

/* enable authentication, but do not check the PPK with the hash stored in the eFuse. this
 * makes it possible to test authentication without having to blow the RSA_EN fuse. */
[fsbl_config] bh_auth_enable

/* --- first stage bootloader settings --- */
[
    bootloader,                                /* mark this as the FSBL */
    destination_cpu    = a53-0,                /* shall be executed on the Cortex-A53 CPU */
    exception_level    = el-3,                  /* exception level 3 = secure monitor mode */

    authentication    = rsa,                    /* must be verified using RSA */
    spk_select         = spk-efuse,              /* use spk-efuse register to check for SPK revocation */
    spk_id              = 0x00000000,            /* same SPK ID used for all SPKs, so all can be revoked at once */
    sskfile             = keys/secondary_fsbl.pem /* FSBL must use the same key pair as the boot header */

] images/linux/zynqmp_fsbl.elf                /* path to the FSBL executable */

/* --- PMU firmware settings --- */
[
    pmufw_image                                /* mark this as the PMU firmware */

    /* pmufw_image does not support authentication. however, it will be signed along with
     * the FSBL, if authentication is enabled for the FSBL. */

] images/linux/pmufw.elf                      /* path to the PMU firmware executable */

/* --- FPGA bitstream settings --- */
[
    destination_device = pl,                    /* shall be applied to the FPGA */

    authentication    = rsa,                    /* must be verified using RSA */
    spk_select         = spk-efuse,              /* use spk-efuse register to check for SPK revocation */
    spk_id              = 0x00000000,            /* same SPK ID used for all SPKs, so all can be revoked at once */
    sskfile             = keys/secondary_bit.pem /* use unique key pair for the FPGA bitstream */

] images/linux/system.bit                     /* path to the FPGA bitstream */

/* --- Arm Trusted Firmware settings --- */
[
    /* ATF is an implementation of a secure monitor, which manages the TrustZone. */

    trustzone          = secure,                 /* must only be executed in secure mode */
    destination_cpu    = a53-0,                 /* shall be executed on the Cortex-A53 CPU */
    exception_level    = el-3,                  /* exception level 3 = secure monitor mode */

    authentication    = rsa,                    /* must be verified using RSA */
    spk_select         = spk-efuse,              /* use spk-efuse register to check for SPK revocation */
    spk_id              = 0x00000000,            /* same SPK ID used for all SPKs, so all can be revoked at once */
    sskfile             = keys/secondary_atf.pem /* use unique key pair for the ATF */

] images/linux/bl31.elf                      /* path to the ATF executable */

/* --- device tree blob settings --- */
[
    destination_cpu    = a53-0,                 /* shall be available on the Cortex-A53 CPU */
    load               = 0x00100000,            /* load partition to this memory address */

    authentication    = rsa,                    /* must be verified using RSA */
    spk_select         = spk-efuse,              /* use spk-efuse register to check for SPK revocation */
    spk_id              = 0x00000000,            /* same SPK ID used for all SPKs, so all can be revoked at once */
    sskfile             = keys/secondary_dtb.pem /* use unique key pair for device tree blob */

] images/linux/system.dtb                    /* path to the dtb file */
```

```
/* --- u-boot settings --- */
[
    destination_cpu    = a53-0,           /* shall be executed on the Cortex-A53 CPU */
    exception_level    = el-2,           /* exception level 2 = hypervisor mode */

    authentication     = rsa,            /* must be verified using RSA */
    spk_select          = spk-efuse,     /* use spk-efuse register to check for SPK revocation */
    spk_id              = 0x00000000,    /* same SPK ID used for all SPKs, so all can be revoked at once */
    sskfile             = keys/secondary_ub.pem /* use unique key pair for u-boot */

] images/linux/u-boot.elf             /* path to the u-boot executable */
}
```

## C. BIF-File with Authentication (incl. Linux kernel)

This BIF file creates a boot image with authentication (including the U-Boot script and the Linux kernel) but without encryption.

```
the_ROM_image:
{
    /* --- general authentication settings --- */
    /* use first primary public key;
     * use spk-efuse register to check if secondary public key has been revoked;
     * use SPK ID = 0, since no secondary public key has been revoked yet */
    [auth_params] ppk_select=0; spk_select=spk-efuse; spk_id=0x00000000

    /* path to primary secret file. only the public key will be included in the resulting image. */
    [pskfile] keys/primary.pem

    /* path to secondary secret file. this will be used to sign and verify the boot header.
     * only the public key will be included in the resulting image. for some reason, the boot
     * header must use the same key pair as the FSBL. */
    [sskfile] keys/secondary_fsbl.pem

    /* enable authentication, but do not check the PPK with the hash stored in the eFuse. this
     * makes it possible to test authentication without having to blow the RSA_EN fuse. */
    [fsbl_config] bh_auth_enable

    /* --- first stage bootloader settings --- */
    [
        bootloader,
        destination_cpu    = a53-0,
        exception_level    = el-3,
        authentication     = rsa,
        spk_select         = spk-efuse,
        spk_id             = 0x00000000,
        sskfile            = keys/secondary_fsbl.pem
    ] images/linux/zynqmp_fsbl.elf

    /* mark this as the FSBL */
    /* shall be executed on the Cortex-A53 CPU */
    /* exception level 3 = secure monitor mode */

    /* must be verified using RSA */
    /* use spk-efuse register to check for SPK revocation */
    /* same SPK ID used for all SPKs, so all can be revoked at once */
    /* FSBL must use the same key pair as the boot header */

    /* path to the FSBL executable */

    /* --- PMU firmware settings --- */
    [
        pmufw_image
    ] images/linux/pmufw.elf

    /* mark this as the PMU firmware */

    /* pmufw_image does not support authentication. however, it will be signed along with
     * the FSBL, if authentication is enabled for the FSBL. */

    /* path to the PMU firmware executable */

    /* --- FPGA bitstream settings --- */
    [
        destination_device = pl,
        authentication     = rsa,
        spk_select         = spk-efuse,
        spk_id             = 0x00000000,
        sskfile            = keys/secondary_bit.pem
    ] images/linux/system.bit

    /* shall be applied to the FPGA */

    /* must be verified using RSA */
    /* use spk-efuse register to check for SPK revocation */
    /* same SPK ID used for all SPKs, so all can be revoked at once */
    /* use unique key pair for the FPGA bitstream */

    /* path to the FPGA bitstream */

    /* --- Arm Trusted Firmware settings --- */
    [
        /* ATF is an implementation of a secure monitor, which manages the TrustZone. */
        trustzone          = secure,
        destination_cpu    = a53-0,
        exception_level    = el-3,
        authentication     = rsa,
        spk_select         = spk-efuse,
        spk_id             = 0x00000000,
        sskfile            = keys/secondary_atf.pem
    ] images/linux/bl31.elf

    /* must only be executed in secure mode */
    /* shall be executed on the Cortex-A53 CPU */
    /* exception level 3 = secure monitor mode */

    /* must be verified using RSA */
    /* use spk-efuse register to check for SPK revocation */
    /* same SPK ID used for all SPKs, so all can be revoked at once */
    /* use unique key pair for the ATF */

    /* path to the ATF executable */

    /* --- device tree blob settings --- */
    [
        destination_cpu    = a53-0,
        load               = 0x00100000,
        authentication     = rsa,
        spk_select         = spk-efuse,
        spk_id             = 0x00000000,
        sskfile            = keys/secondary_dtb.pem
    ] images/linux/system.dtb

    /* shall be available on the Cortex-A53 CPU */
    /* load partition to this memory address */

    /* must be verified using RSA */
    /* use spk-efuse register to check for SPK revocation */
    /* same SPK ID used for all SPKs, so all can be revoked at once */
    /* use unique key pair for device tree blob */

    /* path to the dtb file */
}
```

[illegible]



## D. BIF-File with Authentication & Encryption

This BIF file creates a boot image with authentication and encryption.

```
the_ROM_image:
{
    /* --- general authentication settings --- */
    /* use first primary public key;
     * use spk-efuse register to check if secondary public key has been revoked;
     * use SPK ID = 0, since no secondary public key has been revoked yet */
    [auth_params] ppk_select=0; spk_select=spk-efuse; spk_id=0x00000000

    /* path to primary secret file. only the public key will be included in the resulting image. */
    [pskfile] keys/primary.pem

    /* path to secondary secret file. this will be used to sign and verify the boot header.
     * only the public key will be included in the resulting image. for some reason, the boot
     * header must use the same key pair as the FSBL. */
    [sskfile] keys/secondary_fsbl.pem

    /* --- general encryption settings --- */

    /* use red key from BBRAM (battery-backed RAM) for decryption */
    [keysrc_encryption] bbram_red_key

    /* bh_auth_enable: enable authentication, but do not check the PPK with the hash stored in the
     * eFuse. this makes it possible to test authentication without having to blow the RSA_EN fuse.
     * opt_key: use operational key for encryption/decryption. */
    [fsbl_config] bh_auth_enable, opt_key

    /* --- first stage bootloader settings --- */
    [
        bootloader,                /* mark this as the FSBL */
        destination_cpu            /* shall be executed on the Cortex-A53 CPU */
        exception_level            /* exception level 3 = secure monitor mode */
        authentication             /* must be verified using RSA */
        spk_select                 /* use spk-efuse register to check for SPK revocation */
        spk_id                     /* same SPK ID used for all SPKs, so all can be revoked at once */
        sskfile                    /* FSBL must use the same key pair as the boot header */
        encryption                 /* activate AES encryption */
        aeskeyfile                 /* use unique AES key for the FSBL */

    ] images/linux/zynqmp_fsbl.elf /* path to the FSBL executable */

    /* --- PMU firmware settings --- */
    [
        pmufw_image                /* mark this as the PMU firmware */

        /* pmufw_image does not support authentication. however, it will be signed along with
         * the FSBL, if authentication is enabled for the FSBL. Same applied to encryption. */

    ] images/linux/pmufw.elf /* path to the PMU firmware executable */

    /* --- FPGA bitstream settings --- */
    [
        destination_device          /* shall be applied to the FPGA */
        authentication             /* must be verified using RSA */
        spk_select                 /* use spk-efuse register to check for SPK revocation */
        spk_id                     /* same SPK ID used for all SPKs, so all can be revoked at once */
        sskfile                    /* use unique RSA key pair for the FPGA bitstream */
        encryption                 /* activate AES encryption */
        aeskeyfile                 /* use unique AES key for the FPGA bitstream */

    ] images/linux/system.bit /* path to the FPGA bitstream */

    /* --- Arm Trusted Firmware settings --- */
    [
        /* ATF is an implementation of a secure monitor, which manages the TrustZone. */
        trustzone                  /* must only be executed in secure mode */
        destination_cpu            /* shall be executed on the Cortex-A53 CPU */
        exception_level            /* exception level 3 = secure monitor mode */
        authentication             /* must be verified using RSA */
        spk_select                 /* use spk-efuse register to check for SPK revocation */
        spk_id                     /* same SPK ID used for all SPKs, so all can be revoked at once */
        sskfile                    /* use unique RSA key pair for the ATF */
        encryption                 /* activate AES encryption */
        aeskeyfile                 /* use unique AES key for the ATF */

    ] images/linux/bl31.elf /* path to the ATF executable */
}
```

```

/* --- device tree blob settings --- */
[
    destination_cpu    = a53-0,                /* shall be available on the Cortex-A53 CPU */
    load               = 0x00100000,           /* load partition to this memory address */

    authentication     = rsa,                  /* must be verified using RSA */
    spk_select          = spk-efuse,           /* use spk-efuse register to check for SPK revocation */
    spk_id              = 0x00000000,           /* same SPK ID used for all SPKs, so all can be revoked at once */
    sskfile             = keys/secondary_dtb.pem, /* use unique RSA key pair for device tree blob */

    encryption         = aes,                  /* activate AES encryption */
    aeskeyfile          = keys/aes_dtb.nky      /* use unique AES key for the device tree blob */

] images/linux/system.dtb /* path to the dtb file */

/* --- u-boot settings --- */
[
    destination_cpu    = a53-0,                /* shall be executed on the Cortex-A53 CPU */
    exception_level    = el-2,                 /* exception level 2 = hypervisor mode */

    authentication     = rsa,                  /* must be verified using RSA */
    spk_select          = spk-efuse,           /* use spk-efuse register to check for SPK revocation */
    spk_id              = 0x00000000,           /* same SPK ID used for all SPKs, so all can be revoked at once */
    sskfile             = keys/secondary_ub.pem, /* use unique RSA key pair for u-boot */

    encryption         = aes,                  /* activate AES encryption */
    aeskeyfile          = keys/aes_ub.nky      /* use unique AES key for u-boot */

] images/linux/u-boot.elf /* path to the u-boot executable */

/* --- FIT image settings --- */
[
    /* The FIT image (image.ub) contains the linux kernel, device tree blob and initramfs and
     * will be used by u-boot to get linux running. The FSBL shall load the image to a well-
     * defined address in RAM. The u-boot script (boot.scr) tells u-boot where to find the image. */

    destination_cpu    = a53-0,                /* shall be available on the Cortex-A53 CPU */
    partition_owner     = fsbl,                 /* FSBL has to load this partition to RAM */
    load               = 0x10000000,           /* load partition to this memory address
                                           * (will be called from boot.scr) */

    authentication     = rsa,                  /* must be verified using RSA */
    spk_select          = spk-efuse,           /* use spk-efuse register to check for SPK revocation */
    spk_id              = 0x00000000,           /* same SPK ID used for all SPKs, so all can be revoked at once */
    sskfile             = keys/secondary_fit.pem, /* use unique RSA key pair for FIT image */

    encryption         = aes,                  /* activate AES encryption */
    aeskeyfile          = keys/aes_fit.nky      /* use unique AES key for FIT image */

] images/linux/image.ub /* path to the FIT image */

/* --- u-boot script settings --- */
[
    /* The u-boot script (boot.scr) shall be loaded to address 0x20000000 in RAM by the FSBL.
     * This is the default address where u-boot will look for a script. Then, the script makes
     * u-boot load the FIT image from a well-defined address. */

    destination_cpu    = a53-0,                /* shall be available on the Cortex-A53 CPU */
    partition_owner     = fsbl,                 /* FSBL has to load this partition to RAM */
    load               = 0x20000000,           /* load partition to this memory address
                                           * (u-boot will look here for a script) */

    authentication     = rsa,                  /* must be verified using RSA */
    spk_select          = spk-efuse,           /* use spk-efuse register to check for SPK revocation */
    spk_id              = 0x00000000,           /* same SPK ID used for all SPKs, so all can be revoked at once */
    sskfile             = keys/secondary_scr.pem, /* use unique RSA key pair for u-boot script */

    encryption         = aes,                  /* activate AES encryption */
    aeskeyfile          = keys/aes_scr.nky      /* use unique AES key for u-boot script */

] boot.scr /* path to the u-boot script */
]

```

## E. BIF-File with Authentication & Encryption

This BIF file creates a boot image with authentication and encryption, including the OP-TEE OS binary.

```
the_ROM_image:
{
    /* --- general authentication settings --- */
    /* use first primary public key;
     * use spk-efuse register to check if secondary public key has been revoked;
     * use SPK ID = 0, since no secondary public key has been revoked yet */
    [auth_params] ppk_select=0; spk_select=spk-efuse; spk_id=0x00000000

    /* path to primary secret file. only the public key will be included in the resulting image. */
    [pskfile] keys/primary.pem

    /* path to secondary secret file. this will be used to sign and verify the boot header.
     * only the public key will be included in the resulting image. for some reason, the boot
     * header must use the same key pair as the FSBL. */
    [sskfile] keys/secondary_fsbl.pem

    /* --- general encryption settings --- */

    /* use red key from BBRAM (battery-backed RAM) for decryption */
    [keysrc_encryption] bbRam_red_key

    /* bh_auth_enable: enable authentication, but do not check the PPK with the hash stored in the
     * eFuse. this makes it possible to test authentication without having to blow the RSA_EN fuse.
     * opt_key: use operational key for encryption/decryption. */
    [fsbl_config] bh_auth_enable, opt_key

    /* --- first stage bootloader settings --- */
    [
        bootloader,
        destination_cpu    = a53-0,
        exception_level    = el-3,
        authentication     = rsa,
        spk_select         = spk-efuse,
        spk_id             = 0x00000000,
        sskfile            = keys/secondary_fsbl.pem,
        encryption         = aes,
        aeskeyfile         = keys/aes_fsbl.nky,
    ] images/linux/zynqmp_fsbl.elf

    /* --- PMU firmware settings --- */
    [
        pmufw_image
    ] images/linux/pmufw.elf

    /* --- FPGA bitstream settings --- */
    [
        destination_device = pl,
        authentication     = rsa,
        spk_select         = spk-efuse,
        spk_id             = 0x00000000,
        sskfile            = keys/secondary_bit.pem,
        encryption         = aes,
        aeskeyfile         = keys/aes_bit.nky,
    ] images/linux/system.bit

    /* --- Arm Trusted Firmware settings --- */
    [
        /* ATF is an implementation of a secure monitor, which manages the TrustZone. */
        trustzone          = secure,
        destination_cpu     = a53-0,
        exception_level     = el-3,
        authentication     = rsa,
        spk_select         = spk-efuse,
        spk_id             = 0x00000000,
        sskfile            = keys/secondary_atf.pem,
        encryption         = aes,
        aeskeyfile         = keys/aes_atf.nky,
    ] images/linux/bl31.elf
```

[illegible]

## F. U-Boot Script

```
#
# This is a boot script for U-Boot. It is a customized version of the PetaLinux default script:
#
# If the system boots from SD card or USB stick, it will not look for the image.ub or any other file
# on the storage device. Instead, it will try to boot from address 0x11000000. This allows to include
# the image.ub (FIT image) into the BOOT.BIN file using bootgen. This way, it is possible to sign and
# encrypt the linux image as well. The FSBL will check the signature and decrypt the FIT image. Then,
# it will load the image to memory address 0x11000000 and U-Boot only needs to boot from there.
#
# Generate boot.scr:
# mkimage -c none -A arm -T script -d boot.cmd boot.scr
#

for boot_target in ${boot_targets};
do
    if test "${boot_target}" = "mmc0" || test "${boot_target}" = "mmc1" || test "${boot_target}" = "usb0" || test "${boot_target}" = "usb1";
    then
        # The FIT image is signed, encrypted and included in the BOOT.BIN. The FSBL will place it
        # at memory address 0x11000000 before calling U-Boot. Therefore, we can simply try to boot
        # from this address. fatload is not necessary.
        echo "MMC/USB boot: Assuming that Fit image has been loaded to RAM by FSBL. Not loading anything from storage device."
        echo "Booting Fit image from address 0x11000000"
        bootm 0x11000000;
        echo "Booting Fit image failed"
    else
        echo "Trying to load boot images from ${boot_target}"
    fi

    if test "${boot_target}" = "jtag" ;
    then
        booti 0x00200000 0x04000000 0x00100000
    fi

    if test "${boot_target}" = "xspi0" || test "${boot_target}" = "qspi" || test "${boot_target}" = "qspi0";
    then
        sf probe 0 0 0;
        sf read 0x10000000 0xF40000 0x6400000
        bootm 0x10000000;
        echo "Booting using Fit image failed"

        sf read 0x00200000 0xF00000 0x1D00000
        sf read 0x04000000 0x4000000 0x4000000
        booti 0x00200000 0x04000000 0x00100000;
        echo "Booting using Separate images failed"
    fi

    if test "${boot_target}" = "nand" || test "${boot_target}" = "nand0";
    then
        nand info;
        nand read 0x10000000 0x4180000 0x6400000
        bootm 0x10000000;
        echo "Booting using Fit image failed"

        nand read 0x00200000 0x4100000 0x3200000
        nand read 0x04000000 0x7800000 0x3200000
        booti 0x00200000 0x04000000 0x00100000;
        echo "Booting using Separate images failed"
    fi
done
```