POLITECNICO DI TORINO

Master degree course in Ingegneria Informatica

Master Degree Thesis

Post-Quantum Firmware Integrity Verification for Xilinx Zynq UltraScale+ MPSoC

Supervisor
Prof. Antonio Lioy
Ing. Silvia Sisinni
Ing. Grazia D'Onghia

Candidate
Giacomo Daniel Biondo

December 2024

To my family, who always supported me

Summary

In today's fast-paced technological environment, cybersecurity is facing critical challenges, especially with the rise of quantum computing. This advancement threatens traditional cryptographic systems, such as RSA, which could soon be vulnerable to powerful quantum-based attacks. To mitigate this risk, the field of Post-Quantum Cryptography (PQC) aims to develop algorithms resilient to quantum capabilities. Despite progress in PQC, incorporating these algorithms into existing systems requires careful adaptation to maintain performance and usability. One key area where PQC solutions are essential is in embedded devices' secure and measured boot processes. These processes ensure that only trusted software loads at startup, protecting unauthorised access and potential threats.

This work focuses on developing secure and measured boot procedures for embedded devices that resist quantum computing threats. The target platform was the Xilinx ZCU104 evaluation board, designed to showcase the Xilinx Zynq UltraScale+ MPSoC's capabilities. This board features a quad-core ARM Cortex-A53 processor and a dual-core ARM Cortex-R5 real-time processor with programmable logic. The Cortex-A53 cores support ARM TrustZone technology, enabling secure and non-secure execution modes to enhance embedded security.

The main contribution of this thesis was implementing a measured boot process within the firmware of the Zynq UltraScale+ MPSoC platform. This included creating a measurement log compatible with the "TCG EFI Platform Specification", enabling the collection of integrity measurements on the code executed during the system's startup. These measurements allow for the verification of memory contents' integrity. The generated log can be accessed by an fTPM (Firmware Trusted Platform Module) in the secure execution environment of the ARM Cortex-A53, which initialises the Platform Configuration Registers (PCRs) of the fTPM with boot-acquired measurements. This setup supports remote attestation, allowing external entities to verify the system's trusted boot status. To improve security against quantum threats, hash algorithms with stronger security than SHA-256 are recommended. In this work, a hardware-accelerated SHA3-384 implementation was used, providing enhanced resistance in verifying boot integrity.

Additionally, this work investigated implementing a quantum-resistant secure boot for the Zynq UltraScale+ MPSoC, which currently relies on RSA-4096 signatures for authentication. This approach leaves the platform exposed to quantum computing threats. A quantum-secure design based on Leighton-Micali Hash-Based Signatures (LMS) was proposed, compliant with recommendations from standardisation bodies, to create a robust secure boot process.

Acknowledgements

I cordially thank Prof. Antonio Lioy for entrusting me with this work, which has allowed me to enrich my professional knowledge.

I also sincerely thank Ing. Silvia Sisinni and Ing. Grazia D'Onghia for guiding and supporting me with their precious advice in achieving this work.

Contents

Chapter 1

Introduction

1.1 Firmware Security

In today's technological landscape, ensuring firmware security is essential to maintaining the integrity and reliability of computing systems, particularly in high-performance, embedded platforms like the Zynq UltraScale+ (ZU+) MPSoC. Firmware, as low-level software embedded within hardware devices, plays a critical role in managing foundational functions and acts as a bridge between the device's hardware and operating system. This positioning makes firmware a prime target for attacks, making its protection central to the security of embedded systems and, by extension, the entire ecosystem of connected devices, networks, and data they support. In the context of the ZU+ MPSoC, firmware is integral to core processes, initiating from the moment a device powers on. It enables the initialisation of the processing cores, FPGA logic, and essential hardware peripherals. It also manages the loading of higher-level software, including operating systems, and facilitates communication between hardware and software. The high level of access and control granted to firmware over the MPSoC's complex architecture means that a firmware compromise could result in unauthorised control over nearly every aspect of the device. Firmware vulnerabilities, therefore, represent a significant risk: if exploited, they could enable attackers to gain persistent access, bypassing conventional software security measures and potentially compromising an entire system even through reboots or reinstallation.

Such persistent and deep access makes firmware an attractive target for advanced persistent threats (APTs), which may exploit vulnerabilities to maintain long-term control over systems. This concern is further compounded by supply chain risks, where malicious code can be introduced during manufacturing or distribution, affecting devices before they reach end-users. On platforms like the ZU+ MPSoC, the risks are heightened as firmware updates, if not carefully managed, could potentially "brick" a device, leaving it inoperable, due to the non-volatile memory firmware resides in. Furthermore, many embedded systems, including those with resource-constrained designs in IoT, may lack the memory or processing capacity to implement extensive security protocols, underscoring the need for efficient, secure solutions adapted to the unique constraints of these systems.

The diversity in firmware environments across embedded platforms also presents a significant challenge for developing a universal security solution. Each platform's distinct hardware, operating systems, and architecture require tailored approaches to firmware security. This work addresses these challenges on the ZU+ MPSoC platform, focusing on a comprehensive security framework that incorporates post-quantum cryptographic algorithms alongside robust mechanisms such as secure and measured boot, event logging, and remote attestation procedures. Integrating post-quantum techniques into the secure boot process provides resilience against potential quantum computing threats. Additionally, measured boot and event logging allow for detailed integrity verification and accountability throughout the boot process, while remote attestation enables verification of the device's trustworthiness by external entities. Together, these mechanisms form a layered security solution, enhancing the platform's protection against evolving cybersecurity risks.

1.1.1 Zynq Ultrascale+ MPSoC

The ZU+ MPSoC by Xilinx is a powerful and flexible platform tailored for advanced embedded applications. It combines programmable FPGA logic with a high-performance ARM-based processing system in a single chip. Its architecture integrates a multi-core ARM Cortex-A53 CPU cluster for general-purpose processing, a Cortex-R5 for real-time tasks, and a GPU, allowing it to tackle varied computing needs from data processing to graphics. The chip also features programmable FPGA logic, enabling developers to customise hardware functions and create accelerators to offload demanding tasks, achieving a balance between software flexibility and hardware efficiency. Furthermore, its extensive connectivity options and secure boot and cryptographic capabilities make it suitable for applications demanding high performance and robust security, such as in automotive, industrial automation, telecommunications, and defence sectors.

The ZCU104 Evaluation Board is a development board designed to showcase and prototype applications using the ZU+ MPSoC. With its rich set of interfaces, including USB, Ethernet, HDMI, DisplayPort, and PCIe, the ZCU104 enables a wide range of connectivity and expansion possibilities, making it suitable for diverse applications. Additionally, the board is equipped with DDR4 memory, ample storage, and multiple debugging interfaces, which simplify performance monitoring and optimisation during development. Its compatibility with Xilinx's development tools, Vitis and Vivado, streamlines design, simulation, and implementation, allowing engineers to leverage pre-built IP cores and libraries.

Together, the ZU+ MPSoC and the ZCU104 board offer a powerful and adaptable foundation for high-performance embedded systems, bridging the gap between hardware flexibility and processing efficiency in complex, multipurpose applications.

1.1.2 Key Firmware Integrity Techniques

Firmware integrity in embedded systems is safeguarded through a set of complementary mechanisms, secure boot, measured boot, event logging, and remote attestation, that collectively ensure only authorised and unaltered firmware is executed while enabling detection, recording, and external verification of any security breaches.

1. **Secure Boot**: secure boot acts as the foundational layer of firmware protection. This mechanism is designed to authenticate each stage of the boot process using cryptographic signatures, ensuring that only firmware with a valid signature can execute. In secure boot, the system's boot process is divided into sequential stages, with each stage verifying the next before it can proceed. If any component fails verification, the boot process halts, effectively preventing unauthorised or potentially malicious code from running. On platforms like the Xilinx ZU+ MPSoC, secure boot plays a critical role in verifying the integrity of all firmware stages, from the initial bootloader to the operating system, establishing a trusted baseline for subsequent operations.

2. **Measured Boot**: measure boot builds upon secure boot by generating cryptographic records, or "measurements", of each boot component. Unlike secure boot, which simply allows or denies execution based on signature validation, measured boot records the cryptographic hash of each stage, even if the boot sequence proceeds. These hashes are stored in secure Platform Configuration Registers (PCRs) managed by a Trusted Platform Module (TPM) or a firmware-based TPM (fTPM) such as the one implemented with OP-TEE on the ZU+ MPSoC. These measurements allow the system to detect any unauthorised changes in boot components and to verify the integrity of the entire boot sequence. They also enable the concept of remote attestation, whereby a verifier can examine these measurements to confirm the device's integrity.

3. **Event Logging**: it complements secure and measured boot by creating a chronological record of key boot events, including integrity measurements and validation checks. Structured according to Trusted Computing Group (TCG) standards, the event log provides an auditable, transparent record of each significant event during the boot sequence. This log

is essential for embedded systems in untrusted or distributed environments, as it allows for detailed tracking and analysis of any unauthorised modifications. Event logs can also be referenced in remote attestation, providing external verifiers with the ability to review a comprehensive history of the device's boot process and identify any irregularities or tampering attempts.

4. **Remote Attestation**: it is a procedure that allows a device to prove its integrity to an external verifier by securely transmitting its recorded measurements and event logs. In this process, the device sends its PCR values and event logs to a remote party, which then cross-checks the received data against trusted baselines. This process enables the remote verifier to confirm the integrity of the device's firmware and its boot sequence, even if the device operates in a location where it could be tampered with. For embedded systems deployed in sensitive or untrusted environments, remote attestation is invaluable, as it provides an additional layer of verification that extends beyond local checks, reinforcing the trustworthiness of the device.

Together, secure boot, measured boot, event logging, and remote attestation establish a multi-layered security framework that not only prevents unauthorised firmware execution but also documents the device's operational history and provides a means for external verification. This comprehensive approach strengthens firmware integrity, protects against a wide range of threats, and is essential for maintaining the security of embedded systems in complex and distributed environments.

1.1.3 Quantum Threats and the Necessity of Post-Quantum Cryptography

Quantum computing poses a significant future threat to current cryptographic algorithms, particularly those like RSA and ECC, which rely on the difficulty of factoring large numbers or computing discrete logarithms-problems that can be solved efficiently by quantum algorithms such as Shor's algorithm. To address this threat, the field of Post-Quantum Cryptography (PQC) is developing cryptographic algorithms that remain secure even against quantum attacks. For firmware integrity, PQC represents a vital shift, ensuring that critical security functions like signature verification, encryption, and authentication remain robust in the face of quantum threats. Implementing PQC algorithms within firmware security protocols not only secures systems for the future but also provides a seamless transition for devices that need to operate reliably across both classical and post-quantum environments.

This research focuses on exploring post-quantum firmware integrity verification in the Xilinx ZU+ MPSoC. As an advanced platform that combines ARM multi-core processors with FPGA programmable logic, it offers a versatile environment for implementing innovative security protocols. The aim is to integrate post-quantum secure boot, measured boot, and event logging mechanisms, ensuring that only authorised, quantum-resilient firmware can execute.

The secure boot process in traditional embedded systems, such as the Xilinx ZU+ MPSoC, typically relies on RSA-4096 digital signatures to verify firmware integrity and authenticity during the boot sequence. However, given the anticipated ability of quantum computers to break RSA-4096 using algorithms like Shor's, there is a clear need for transitioning to post-quantum secure solutions. In this post-quantum approach, the secure boot process is redefined by replacing RSA-4096 with post-quantum signature algorithms, such as LMS (Leighton-Micali Signature) or XMSS (eXtended Merkle Signature Scheme). With these algorithms, firmware partitions are signed and verified using quantum-resistant keys, safeguarding the firmware against potential quantum attacks. Specifically, the process involves generating digital signatures for each critical firmware partition using a post-quantum algorithm before deployment. During boot, the secure bootloader on the ZU+ MPSoC verifies each partition's signature against an embedded post-quantum public key, ensuring that only authenticated, untampered firmware components are allowed to execute. If verification fails, the boot process is halted, preventing any unauthorised or maliciously altered firmware from executing. This ensures firmware integrity and authenticity even in a post-quantum era, strengthening the secure boot's resilience.

The measured boot process remains based on the existing ZU+ MPSoC hardware capabilities, specifically utilising the SHA3-384 hash core of the CSU (Configuration Security Unit) to compute integrity measurements for each boot stage. The SHA3-384 hash function is particularly relevant in quantum computing due to its resilience to known quantum attacks. The Grover algorithm, a quantum algorithm capable of reducing the effective security of cryptographic hashes, theoretically halves the computational complexity of finding a preimage or collision. For instance, while a classical brute-force attack against a 384-bit hash requires 2^{384} operations, Grover reduces this to 2^{192}. However, 2^{192} remains far beyond the reach of even the most optimistic projections for quantum computing capabilities, ensuring that SHA3-384 is robustly quantum-resistant for the measured boot process. This level of security is consistent with the requirements for cryptographic hash functions in trusted computing applications. These measurements, considered quantum-safe, taken directly by the CSU, record the integrity of each component in sequence, without other modifications to accommodate post-quantum cryptography in this part of the process. The CSU continues to store these measurements as cryptographic hashes in the PCRs within a firmware-based TPM (fTPM). As each boot stage is loaded, its hash is recorded, allowing for a sequential, hardware-enforced integrity record consistent with the TCG standards. This measured boot process does not employ post-quantum algorithms for the measurement itself, as it relies on the SHA3-384 hash function natively provided by the ZU+ architecture, which remains quantum-resistant for this purpose.

The event log, generated in line with TCG standards, captures all key events during the boot process, creating a chronological record of each integrity measurement. This event log is essential for enabling remote attestation, where an external verifier can inspect and confirm the device's boot history, ensuring that no tampering has occurred. Since the event logging process is based on SHA3-384 hash measurements from the CSU, it does not require any additional post-quantum modifications. This standard SHA3-384 hashing approach within the event logging process remains secure against quantum attacks due to the inherent quantum resistance of SHA-3. The log can be securely transmitted to a remote verifier, who inspects the integrity measurements to verify the device's firmware authenticity, enabling robust integrity confirmation in scenarios where embedded systems are deployed in untrusted or remote environments.

The key enhancement in this post-quantum framework lies within the secure boot process, where RSA-4096 signatures are replaced by post-quantum signature algorithms to secure the firmware's authenticity against quantum threats. The measured boot and event logging processes remain based on the CSU's SHA3-384 capabilities, preserving existing, quantum-resistant mechanisms for integrity measurement and attestation without further modification. Together, these elements establish a comprehensive post-quantum secure boot framework, enhancing firmware integrity verification for the ZU+ MPSoC in a post-quantum context.

1.1.4 Practical Implementation on ZU+ MPSoC

The ZU+ MPSoC's architecture, with its combination of ARM cores and FPGA logic, is particularly well-suited for the integration of advanced cryptographic protocols. By using OP-TEE as the secure environment, the MPSoC can handle fTPM functionality within the TrustZone, creating a reliable space for measurements and event logging. Each critical boot partition, such as the First Stage Boot Loader (FSBL), the ARM Trusted Firmware (BL31), U-Boot, and OP-TEE, is measured and logged. These measurements create a comprehensive integrity log for each partition, which can be inspected or attested by an external verifier. This setup allows for the integration of PQC algorithms, facilitating a secure transition to quantum-resistant security measures within the existing boot process.

This work aims to advance the development of resilient firmware integrity solutions by demonstrating a post-quantum firmware verification framework on the ZU+ MPSoC. By addressing the challenges posed by quantum computing, this approach provides a foundation for future firmware security standards that can withstand both current and emerging threats. The convergence of secure boot, measured boot, and post-quantum cryptographic techniques within the ZU+ MPSoC form a comprehensive solution for securing firmware integrity. This approach not only enhances the immediate resilience of embedded systems but also future-proofs them against the evolving

threat landscape. As quantum computing progresses, the deployment of PQC in firmware integrity verification will become essential, ensuring that systems, data, and networks remain secure and reliable well into the future.

Chapter 2

Post-quantum Cryptography

2.1 Introduction to Post-Quantum Realities

The advent of quantum computing represents a significant milestone in the field of cryptography and cybersecurity. Continued progress in the development of quantum computers foreshadows a potentially disruptive cryptographic transition. Currently, the most widely used public-key cryptographic algorithms are theoretically vulnerable to attacks based on Shor's algorithm, which relies on operations achievable only by a large-scale quantum computer. When practical quantum computing becomes available to cyber adversaries, the security of nearly all modern public-key cryptographic systems will be compromised. This entails the vulnerability of secret symmetric keys and private asymmetric keys currently safeguarded by existing public-key algorithms, along with the information protected under those keys. Recorded communications and other stored information protected by these cryptographic algorithms will be at risk of exposure. Any data still considered private or sensitive will be vulnerable to exposure and undetected modification. Once the practical exploitation of Shor's algorithm becomes feasible, protecting stored keys and data will necessitate re-encrypting them with a quantum-resistant algorithm and either deleting or physically securing "old" copies (e.g., backups).

Information integrity and sources will become unreliable unless processed or encapsulated (e.g., re-signed or timestamped) using mechanisms immune to quantum computing-based attacks. Confidentiality of previously stored encrypted material by adversaries will remain unattainable. Fortunately, many cryptographic researchers have contributed to the development of algorithms immune to Shor's algorithm or other known quantum computing algorithms. These algorithms sometimes referred to as quantum-resistant, are designed for a world with practical quantum computing and are termed post-quantum algorithms. However, our understanding of quantum computing capabilities remains incomplete [1].

Challenges in Transitioning Existing Algorithms

Unfortunately, the adoption of PQ public-key standards is likely to pose more challenges than the integration of new classical cryptographic algorithms. Without thorough implementation planning, it could take decades before most of the currently vulnerable public-key systems are replaced. Key establishment, which involves the secure generation, acquisition, and management of keys, along with digital signature applications, represent the most crucial functions currently reliant on public-key cryptography. It would be optimal to have readily available replacements for quantum-vulnerable algorithms like RSA and Diffie-Hellman for these purposes. However, each class of potential PQC solutions presents challenges for secure implementation. For instance, some candidates suffer from issues such as excessively large signature sizes, high processing requirements, or the need for very large public and/or private keys. Secure implementation may also need to address concerns like public-key validation, potential public-key reuse, and decryption failure, depending on the algorithm and its usage. Even when secure operation is achievable, performance and scalability concerns may necessitate significant modifications to existing protocols

and infrastructures. On the other hand, existing protocols might need to be modified to handle larger signatures or key sizes. Implementations of new applications will need to accommodate the demands of PQC and allow the new schemes to adapt to them [1].

2.2 Hash-Based Signature Algorithms

As already mentioned, PQC algorithms have been developed as a response to the growing concerns about the security of traditional cryptographic technologies, such as RSA, ECC, and DH, which might be vulnerable to attacks from quantum computers. Among these algorithms, one of the most promising approaches is represented by Hash-based Signature (HBS) algorithms. In a typical HBS scheme, the signing of a message involves several steps. Initially, the message is processed using a cryptographic hash function, which transforms the message into a fixed-size hash value. This hash value is then signed using the private key associated with the entity generating the signature. The result is a digital signature that can be verified using the corresponding public key. These algorithms offer robust security even in the presence of advanced quantum computers, as the signature generation relies on mathematical properties of cryptographic hash functions that are not vulnerable to quantum search algorithms. The security of such algorithms is based on the computational hardness of problems associated with the cryptanalysis of cryptographic hash functions, making it difficult for an adversary to derive the private key even with the assistance of a quantum computer. In this context, two distinct approaches emerge: stateful and stateless schemes.

Stateful schemes maintain an internal state that evolves during the signing process. This state may include information such as counters or random values, which contribute to the generation of signatures. On the other hand, stateless schemes do not maintain any internal state between signatures, making each signature completely deterministic and dependent only on the message and the private key. While stateful schemes offer flexibility and can be adapted to specific security requirements, they require careful management of the state to avoid collisions or consistency losses. In contrast, stateless schemes are easier to implement and less prone to state consistency issues, making their management more straightforward and less error-prone.

2.3 Stateful HBS Algorithms

Stateful HBS algorithms offer better key and signature sizes than stateless HBS algorithms, and the underlying cryptographic building blocks are generally considered well-understood. However, a critical consideration is the management of the state, which is a fundamental aspect of security. This is especially critical when signing data over extended periods using the same key pair; i.e., the resulting signatures will be validated with the same public key over a long period [2].

Stateful hash-based signature schemes are secure against the development of quantum computers, but they are not suitable for general use because their security depends on careful state management. They are most appropriate for applications in which the use of the private key may be carefully controlled and where there is a need to transition to a post-quantum secure digital signature scheme before the PQC standardization process has been completed. This is because stateful schemes can be implemented with the flexibility to adapt to specific security needs without relying entirely on defined standards. Stateful HBS schemes are primarily intended for applications with the following characteristics: it is necessary to implement a digital signature scheme in the near future; the implementation will have a long lifetime; and it would not be practical to transition to a different digital signature scheme once the implementation has been deployed. An application that may fit this profile is the authentication of firmware updates for constrained devices. Some constrained devices that will be deployed soon will be in use for decades. These devices will need to have a secure mechanism for receiving firmware updates, and it may not be practical to change the code for verifying signatures on updates once the devices have been deployed.

In a stateful HBS scheme, an HBS private key consists of a large set of one-time signature (OTS) private keys. The signer needs to ensure that no individual OTS key is ever used to sign

more than one message. If an attacker were able to obtain digital signatures for two different messages that were created using the same OTS key, then it would become computationally feasible for that attacker to forge signatures on arbitrary messages. Therefore, when a stateful HBS scheme is implemented, extreme care needs to be taken in order to ensure that no OTS key is ever reused. To obtain assurance that OTS keys are not reused, the signing process should be performed in a highly controlled environment [3].

2.3.1 LMS and XMSS

At a high level, eXtended Merkle Signature Scheme (XMSS) and Leighton-Micali Signature (LMS) are very similar. They each consist of two components: a one-time signature (OTS) scheme and a method for creating a single, long-term public key from a large set of OTS public keys.

One-Time Signature Systems

Both LMS and XMSS make use of variants of the Winternitz signature scheme (Winternitz OTS - WOTS). In the Winternitz signature scheme, the message to be signed is hashed to create a digest; the digest is encoded as a base b number and then each digit of the digest is signed using a hash chain, as follows. A hash chain is created by first randomly generating a secret value, x, which is the private key. The size of x should generally correspond to the targeted security strength of the scheme. So, for the parameter sets approved by this recommendation, x will be either 192 or 256 bits in length. The public key, pub, is then created by applying the hash function, H, to the secret b - 1 times, $H^{b-1}(x)$.

A sample Winternitz chain for b = 4:

$x_k \rightarrow \boxed{\text{H}} \rightarrow H(x_k) \rightarrow \boxed{\text{H}} \rightarrow H(H(x_k)) \rightarrow \boxed{\text{H}} \rightarrow pub_k = H(H(H(x_k)))$

Note: The base b is referred to as the Winternitz parameter. The term "Winternitz parameter", also denoted by w, refers to a different but related quantity: the number of bits of the digest that is encoded by b. RFC 8554 specifies that w could be 1, 2, 4, or 8, which corresponds to a b of 2, 4, 16, or 256, respectively [4].

The kth digit of the digest, Nk, is signed by applying the hash function, H, to the private key N_k times, $H^{N_k}(x_k)$. When N_k is 1 the signature is $s_k = H^1(x_k) = H(x_k)$.

The hash function, H, is applied to sk twice (b=4 in this case), and if the resulting value is the same as the public key, pub_k, then the signature is valid. In general, the signature for the kth digit of a digest can be verified by checking that $pub_k = H^{b-1-N_k}(s_k)$.

A sample Winternitz signature generation and verification (b=4, $N_k = 1$)

$x_k \rightarrow \boxed{\text{H}} \rightarrow s_k = H(x_k)$

$s_k \rightarrow \boxed{\text{H}} \rightarrow H(s_k) \rightarrow \boxed{\text{H}} \rightarrow pub_k = H(H(s_k))$

To protect against attacks, the Winternitz signature scheme computes a checksum of the message digest and signs the checksum along with the digest. For an n-digit message digest, the checksum is computed as $\sum_{k=0}^{n-1}(b - 1 - N_k)$. The checksum is designed so that the value is non-negative, and any increase in a digit in the message digest will result in the checksum becoming smaller. This prevents an attacker from creating an effective forgery from a message signature since the attacker can only increase values within the message digest and cannot decrease values within the checksum 2.1.

Merkel Trees

While a single, long-term public key could be created from a large set of OTS public keys by simply concatenating the keys together, the resulting public key would be unacceptably large. XMSS and LMS instead use Merkle hash trees, which allow for the long-term public key to be very short in exchange for requiring a small amount of additional information to be provided with

	Digest								Checksum	
Digest	6	3	F	1	E	9	0	B	3	D
Private Key	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
Signature	$H^6(x_0)$	$H^3(x_1)$	$H^{15}(x_2)$	$H(x_3)$	$H^{14}(x_4)$	$H^9(x_5)$	x_6	$H^{11}(x_7)$	$H^3(x_8)$	$H^{13}(x_9)$
Public Key	$H^{15}(x_0)$	$H^{15}(x_1)$	$H^{15}(x_2)$	$H^{15}(x_3)$	$H^{15}(x_4)$	$H^{15}(x_5)$	$H^{15}(x_6)$	$H^{15}(x_7)$	$H^{15}(x_8)$	$H^{15}(x_9)$

Table 2.1. A sample Winternitz signature for a 32-bit message digest using b = 16 (the digest is written as eight hexadecimal digits) [3]

each OTS key. To create a hash tree, the OTS public keys are hashed once to form the leaves of the tree, and these hashes are then hashed together in pairs to form the next level up. Those hash values are then hashed together in pairs, the resulting hash values are hashed together, and so on until all of the public keys have been used to generate a single hash value (the root of the tree), which will be used as the long-term public key 2.1.

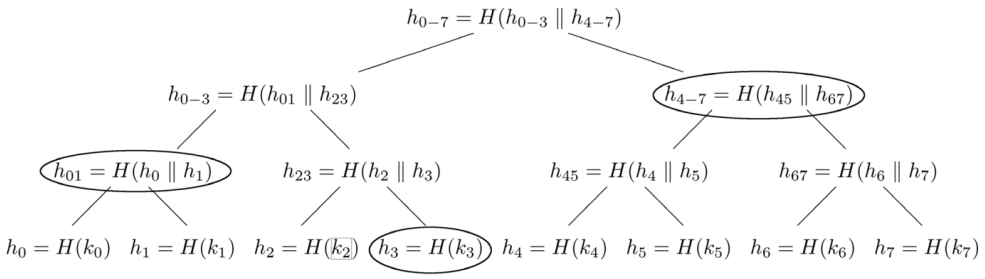

Figure 2.1. A Merkle Hash Tree [3]

XMSS and LMS - Main Differences

While they share some fundamental characteristics, they differ in several key aspects that influence their applications and implementations.

XMSS uses a more complex Merkle tree structure, allowing for greater flexibility in the use of cryptographic hash functions and providing enhanced security through the randomization of Merkle tree leaves. This makes XMSS suitable for applications requiring a high level of security, although this comes at the cost of increased implementation complexity and resource usage.

LMS, on the other hand, adopts a simpler Merkle tree structure, making it easier to implement and less demanding in terms of computational power and memory. This simplicity makes LMS ideal for applications where simplicity and efficiency are critical, even though it might not offer the same advanced security features as XMSS.

2.4 Stateless HBS Algorithms - SPHINCS+

SPHINCS was designed as a stateless hash-based signature scheme and was the first signature scheme to propose parameters to resist quantum cryptanalysis. SPHINCS uses many components from XMSS but works with larger keys and signatures to eliminate the state. At a high level, SPHINCS+ works like SPHINCS. The basic idea is to authenticate a huge number of few-time signature (FTS) key pairs using a so-called hypertree. It is a stateless hash-based signature algorithm and is considered safe against attacks by quantum computers. The advantage of this algorithm is that the state problem is resolved as part of the algorithm. However, the tradeoff is that signature sizes are often an order of magnitude larger than XMSS or LMS. This may make deploying these algorithms on constrained devices infeasible [2].

Few-Time Signature Scheme

In contrast to the OTS scheme, an FTS scheme allows the reuse of a key pair a few times. The FTS scheme is only used in the stateless HBS scheme SPHINCS+. As a result, the total tree height of SPHINCS+ can be reduced significantly, making it applicable in practice [5]. For each new message, a (pseudo)random FTS key pair is chosen to sign the message. The signature consists of the FTS signature and the authentication information for that FTS key pair. The authentication information is roughly a hyper-tree signature, i.e. a signature using a certification tree of Merkle tree signatures. Every leaf node in the Merkle tree corresponds to a single hashed FTS public key. The tree's root node corresponds to the Merkle Signature Scheme (MSS) public key, which is used to authenticate the FTS public keys. A Merkle tree with a tree height h authenticates 2^h FTS key pairs. More specifically, a hyper-tree is a tree of hash-based many-time signatures (MTS). These MTS allow a key pair to sign a fixed number N of messages, for SPHINCS+ N is a power of 2, for example, 256. The MTS key pairs are organized in an N-ary tree with d layers.

Note: Each MTS key pair can sign N messages. The tree structure is N-ary, meaning each node (MTS key pair) has N children.

On the top layer d - 1 there is a single MTS key pair which is used to sign the public keys of N MTS key pairs that form layer d - 2. Each of these N MTS key pairs is used to sign another N MTS public keys forming layer d - 3. This goes on down to the N^{d-1} key pairs on the bottom layer which are used to sign N FTS public keys, each, leading to a total number of N^d authenticated FTS key pairs. The authentication information for an FTS key pair consists of the d MTS signatures that build a path from the FTS key pair to the top MTS tree. An MTS signature is just a classical Merkle-tree signature in the case of SPHINCS+. It consists of an OTS on the given message plus the authentication path in the binary hash tree, authenticating the N OTS key pairs of one MTS key pair. The public key of SPHINCS+ is essentially the public key of the top-level MTS which is just the root node of its binary hash tree and hence, a single hash value. However, the SPHINCS+ key structure leverages both secret and public seeds to create a secure and efficient signing mechanism. The secret key is primarily a single secret seed, supplemented by an additional secret value (which is the same size as the secret seed, this additional value adds another layer of security) and a copy of the public key. This seed is the cornerstone for generating all OTS and FTS keys in a pseudorandom manner, defining the complete virtual structure of the key pair. The public key, on the other hand, includes the root node of the top-level Merkle tree and a public seed, which ensures that the verifier can accurately regenerate the required keys for authentication and verification purposes. This design provides a robust and scalable method for secure digital signatures in the SPHINCS+ scheme [6].

Example Process

Consider an example with three layers in the hyper-tree and a binary structure, meaning each node has two children (d=3, N=2). At the topmost layer (layer 2), there is a single MTS key pair. This key pair is responsible for signing the public keys of the two MTS key pairs in the middle layer (layer 1). Moving down, each MTS key pair in layer 1 signs the public keys of the MTS key pairs in the bottom layer (layer 0). Specifically, each of the two MTS key pairs in layer 1 signs the public keys of two MTS key pairs in layer 0, resulting in a total of four MTS key pairs at this layer. At the bottom layer, each of the four MTS key pairs is used to sign the public keys of two FTS key pairs. When a new message needs to be signed, one of these FTS key pairs from the bottom layer is chosen pseudo-randomly. This FTS key pair then signs the message. To ensure the authenticity of the FTS public key used for signing, an authentication path is constructed, starting from the bottom layer up to the top MTS key pair in layer 2. This path includes MTS signatures, each comprising an OTS signature of a message and an authentication path within the Merkle tree that verifies the OTS public key.

The verifier uses the top-level MTS public key, which is the root hash of the Merkle tree at the top layer, to authenticate the entire hierarchical structure. By following the path of MTS signatures from the FTS key pair up through the layers, the verifier can confirm the authenticity of the FTS public key. Once authenticated, the verifier can then proceed to verify the signature on the message itself.

Final Considerations On Different Key Pairs

In the SPHINCS+ scheme, three types of key pairs play distinct roles: OTS, FTS, and MTS. OTS key pairs are used within MTS signatures to sign messages securely. FTS key pairs, chosen pseudo-randomly for each new message, are used to sign the actual messages, but their public keys require authentication. MTS key pairs facilitate this authentication through a hierarchical structure, signing the public keys of other MTS or FTS key pairs. The hyper-tree structure of SPHINCS+ consists of multiple layers, where the topmost layer contains a single MTS key pair that signs the public keys of MTS key pairs in the subsequent layer. This process continues down to the bottom layer, where each MTS key pair signs several FTS public keys. When an FTS key pair is used to sign a message, its public key is authenticated via a path of MTS signatures leading back to the top MTS key pair. The verifier uses the root hash of the top layer's Merkle tree to confirm the authenticity of the entire structure, thereby ensuring the integrity of the message signature. This hierarchical approach allows SPHINCS+ to efficiently and securely manage the authentication of FTS key pairs, ensuring robust security for each signed message despite the frequent generation of new key pairs.

2.5 Choice of Hash-Based Signature Parameters

LMS, XMSS and SPHINCS+ have support for both SHA-2 and SHA-3. Due to the widespread use of SHA-2, the SHA-256 hash function with an output size of 32 bytes is selected. This choice guarantees a level of security equivalent to an exhaustive key search for AES-256, thus reaching NIST's highest security level five. If Grover's attack is feasible, this equals a level of 128 bits in a pre-quantum world. For ECC, a 256-bit curve is selected, and for RSA 3072-bit integers for comparison with commonly used asymmetric algorithms 2.2.

Algorithm	Parameter		Signature size	Public key size	NIST security level
LMS	h=15	w=4	4.7KiB	32B	5
	h=15	w=16	2.7KiB	32B	5
	h=15	w=256	1.6KiB	32B	5
XMSS	h=16	w=16	2.6KiB	32B	5
SPX+	256s		29KiB	64B	5
RSA	3072		384B	384B	!
ECC	P-256		64B	64B	!

Table 2.2. Signature and public key sizes and the reached NIST security level, in comparison to RSA and ECC, for LMS, XMSS and SPHINCS+ with SHA-256 as underlying hash function and an output size of 32 bytes [5]

Due to the nature of stateful HBS, the ability to sign firmware images is limited to a predetermined count, defined at the time of key generation. For the secure boot use case, the required number of firmware updates is estimated, including a security margin. Considering the maximum lifetime of a security controller to be 40 years and assuming up to two updates per day, the total number of required signatures is estimated to be up to 29200, with one signature used for each firmware update. From this estimation, the required tree height parameter for LMS is derived to be $h = 15$ and XMSS to be $h = 16$. For XMSS, the Winternitz parameter w is limited to $w = 16$. For LMS, the Winternitz parameter is $W \in [1, 2, 4, 8]$, which maps to $w \in [2, 4, 16, 256]$. The notation of w is used for SPHINCS+ and XMSS, so we will use this notation as well for LMS. The Winternitz parameter allows a trade-off between signature size and overall performance. A higher Winternitz parameter generates smaller signatures but has the drawback of worse performance. This is not the case for $w = 2$, as for both $w = 2$ and $w = 4$ the required hash compress calls add up to the same count, while the signature for $w = 2$ is larger. Therefore, the original Winternitz parameter set can be limited to $w \in [4, 16, 256]$. The resulting signature sizes of the selected parameters for LMS and XMSS are listed in 2.2.

In contrast to XMSS and LMS, the SPHINCS+ parameters are described with certain sets of parameter combinations. This is due to the fact, that the stateless property of SPHINCS+

is a result of carefully combining parameters. The available list of parameters can be split into two variants, "small" and "fast", which are denoted with "s" and "f", respectively. The "small" variant has the drawbacks of slower key generation and signing. However, it achieves smaller signature sizes and faster verification. As this is desirable for the secure boot scenario, we select the "small" variant. The respective signature and public key size for the selected parameter set are listed in 2.2. The "simple" and "robust" construction that can be used in SPHINCS+ only influences the security proof and runtime but not signature or public key sizes. They are referred to as SPX+-s and SPX+-r, respectively [5].

In conclusion, the choice of the "small" variant of SPHINCS+ for secure boot applications is motivated by the need for efficient verification and compact signature storage. While it entails slower key generation and signing processes, these are acceptable trade-offs given the infrequency of these operations compared to verification. The distinctions between the simple and robust constructions provide flexibility in balancing security-proof strength with runtime performance, further tailoring the scheme to specific application needs.

2.6 Lattice Based Cryptography

Among the various post-quantum techniques, such as multivariate polynomial cryptography, code-based cryptography, and hash-based cryptography, lattice-based cryptography stands out as one of the most promising. Lattices are mathematical structures that can be visualized as infinite grids of points in n-dimensional space. The mathematical properties of these structures make the associated problems extremely difficult to solve, even for quantum computers [7].

2.6.1 Computational Problems in Lattices

The main computational problems on which the security of lattice-based cryptography is based include:

Shortest Vector Problem (SVP): given a lattice, find the shortest non-zero vector. This problem is extremely difficult to solve, and its complexity forms a solid foundation for cryptographic security.

Closest Vector Problem (CVP): given a point in space and a lattice, find the lattice vector closest to that point. This problem is known to be hard and is used as a basis for cryptographic security.

Learning With Errors (LWE): involves solving systems of linear equations that are slightly perturbed by errors. This problem, introduced by Oded Regev, underpins many modern cryptographic constructions. The difficulty of removing the errors makes the LWE problem resistant to attacks, including quantum ones.

Short Integer Solution (SIS): requires finding a short vector x such that $A \cdot x{=}0$ modulo q where A is a given matrix. The difficulty in finding such short vectors is the basis for the security of many lattice-based digital signature schemes.

Module Learning With Errors (MLWE): a variant of LWE where operations are performed on modules instead of vectors. This problem retains the difficulty of LWE but allows for more efficient and flexible constructions suitable for various applications.

Nth Degree Truncated Polynomial Ring (NTRU) Problem: based on polynomial rings, involves finding specific polynomials that solve particular lattice equations. This problem underlies the NTRU encryption scheme, known for its efficiency and security.

2.6.2 Advantages of Lattice-Based Cryptography

Lattice-based cryptography offers several advantages over other post-quantum cryptographic techniques. One of the primary benefits is its efficiency. Arithmetic operations within lattice structures

can be performed with high efficiency, making lattice-based cryptographic systems more practical for real-world applications compared to many other post-quantum techniques. This efficiency translates into faster computation times and lower resource consumption, which are critical factors for the deployment of cryptographic systems in various technological environments. Another significant advantage is the flexibility of lattice-based cryptographic primitives. These primitives can be utilized to construct advanced cryptographic schemes such as homomorphic encryption and fully homomorphic encryption (FHE). Homomorphic encryption allows computations to be performed on encrypted data without decrypting it first, which has profound implications for data security and privacy in cloud computing and other areas where data needs to be processed while remaining confidential.

The security of lattice-based cryptography is also noteworthy. The underlying computational problems, such as SVP, CVP, LWE, SIS, MLWE, and NTRU, are not only hard to solve for classical computers but also resistant to quantum attacks. This resistance makes lattice-based systems a robust choice for ensuring long-term security in a future where quantum computing could break traditional cryptographic schemes.

2.6.3 Lattice-Based Algorithms: FALCON and CRYSTALS-Dilithium

Two of the leading lattice-based cryptographic algorithms proposed as standards for post-quantum cryptography are FALCON and CRYSTALS-Dilithium. These algorithms exemplify the practical application of lattice-based principles to create secure and efficient digital signature schemes.

FALCON (Fast Fourier Lattice-based Compact Signatures over NTRU)

FALCON is a digital signature algorithm that utilizes the NTRU structure. It is designed to be highly efficient and compact while maintaining a high level of security. FALCON leverages the Fast Fourier Transform (FFT) to accelerate polynomial arithmetic operations, making the signing and verification processes very fast. This efficiency is crucial for applications that require rapid cryptographic operations. Security in FALCON is based on hard lattice problems, ensuring resistance to both classical and quantum attacks. The algorithm is designed to be compact, with relatively small public keys, private keys, and signatures. This compactness makes FALCON well-suited for applications where space efficiency is a priority. The use of efficient polynomial operations and FFT ensures fast execution times for both key generation and signature processes, contributing to its practicality.

CRYSTALS-Dilithium

CRYSTALS-Dilithium is another lattice-based digital signature algorithm designed to be both practical and secure in the post-quantum context. It uses a lattice structure called Module-LWE (Learning With Errors) to ensure security. One of the strengths of CRYSTALS-Dilithium is its simplicity, which makes it easy to implement and reduces the risk of implementation errors that could compromise security. Similar to FALCON, CRYSTALS-Dilithium is highly efficient, offering fast signing and verification times and reasonably compact signatures. The algorithm is designed to balance security with practicality, featuring relatively compact public keys, private keys, and signatures. This balance ensures that the algorithm is both secure and efficient, suitable for a wide range of applications. The straightforward lattice-based operations employed by CRYSTALS-Dilithium ensure both efficiency and robustness in practical implementations.

Lattice-based cryptography stands out as one of the most promising areas in post-quantum cryptographic research. Its efficiency, flexibility, and robust security make it an ideal candidate for replacing current cryptographic schemes that are vulnerable to quantum computers. Algorithms like FALCON and CRYSTALS-Dilithium demonstrate how lattice theory can be applied to create practical and secure digital signature schemes, ensuring robust protection for the future of digital communications. These advancements underscore the importance of continued research and development in lattice-based cryptography as we move toward a quantum-resistant digital world.

2.7 Cryptography in UEFI Specification

The Unified Extensible Firmware Interface (UEFI) specification serves as the modern standard firmware interface for booting operating systems and initializing hardware during the system startup process. UEFI replaces the traditional BIOS (Basic Input/Output System) and offers several advantages, including support for larger disk capacities, faster boot times, and enhanced security features such as Secure Boot. As technology continues to advance and threat actors evolve, the complexity of computational tasks within the UEFI environment increases. One of the key challenges faced by the UEFI specification is known as "crypto agility". This term refers to the necessity for cryptographic mechanisms to be adaptable and responsive to changing security requirements and computational capabilities. Traditionally, the UEFI specification hardcoded specific key lengths and algorithms for cryptographic operations. However, with the rapid progress in computational capabilities, these fixed standards have become inadequate in ensuring robust security. There is a growing recognition within the industry that separating cryptographic requirements from the primary UEFI specification is essential for maintaining flexibility and adaptability. Separating cryptographic specifications from the core UEFI specification allows for a more focused approach to tracking the evolving cryptographic landscape.

By decoupling cryptographic standards from interface and data structure specifications, the UEFI community can better address the dynamic needs of the business ecosystem and respond effectively to emerging security threats. Presently, there is an ongoing industry-wide discussion regarding the segregation of cryptographic specifications from the UEFI specification. The objective is to enhance the standard's responsiveness to the evolving computational capabilities and security requirements of modern computing environments. The primary aim of this initiative is to achieve "crypto agility" within the UEFI specification. Crypto agility ensures that cryptographic standards can adeptly adapt to evolving security needs and computational capabilities. By enabling flexible and adaptable cryptographic mechanisms, UEFI can effectively mitigate emerging threats and maintain the integrity and security of the boot process [8].

The integration of crypto agility principles into the UEFI specification is imperative for ensuring the resilience and security of modern computing systems. By separating cryptographic specifications and focusing on adaptability and responsiveness, the UEFI community can effectively address the evolving threat landscape and maintain the trustworthiness of the boot process in an increasingly interconnected and dynamic digital environment.

2.7.1 Current Security Strength

Regarding current security strength, the choice of key lengths is crucial. For instance, with SHA, smaller sizes are vulnerable to pre-image attacks. Similarly, RSA key lengths need to be sufficient to resist prime factoring. Guidance from organizations like the NSA (National Security Agency) recommends specific key lengths for different algorithms to withstand attacks from contemporary computers. These recommended lengths are often larger than what was previously standard, highlighting the need for cryptographic agility and adaptation to evolving threats. Asymmetric cryptography, like RSA and SHA, plays a significant role in system firmware. In the mid-90s, Shor's algorithm postulated that quantum computers could break RSA and SHA by exploring the entire state space rather than iteratively guessing primes. While quantum computing was initially theoretical, recent advancements have brought it closer to reality, posing a potential challenge to current cryptographic standards. In a world with practical quantum computers, asymmetric algorithms would become ineffective, necessitating larger hash functions for sufficient security. This presents a dilemma, particularly concerning asymmetric cryptography's role in signing images and encrypting network key exchanges. The industry is actively discussing this problem exploring approaches to address it, including Mosca's theorem.

Mosca's theorem outlines metrics such as the time required to make encryption secure (X) and the duration products need to remain secure (Y). Currently, efforts are focused on the "Y" phase, working towards cryptographic agility that includes resilience to potential quantum-based attacks. The ultimate goal is to support products that can remain secure within their operational lifetimes (X) while navigating the race toward achieving viable quantum computers (Z). NIST initiated

the Post Quantum Cryptography (PQC) project in 2016 to develop cryptographic systems secure against both quantum and classical computers. This project primarily focuses on public-key-based cryptography, with ongoing evaluations and rounds of standardization expected to be finalized by 2024.

In summary, the potential impact of quantum computing on cryptography underscores the need for proactive measures, including the exploration of post-quantum cryptography solutions, to ensure the security of UEFI-based systems and beyond [8].

2.7.2 Open Quantum Safe (OQS) Project

In addition to the NIST PQC project, the industry is also preparing for the post-quantum era through initiatives like the Open Quantum Safe (OQS) project. This project, initiated in 2016, aims to support the development and prototyping of quantum-resistant cryptography [9].

The OQS project consists of two main components:

1. Liboqs is an open-source C library for quantum-resistant cryptography. It offers a collection of open-source implementations for key encapsulation mechanisms (KEM) and digital signature algorithms resistant to quantum computing attacks. Additionally, liboqs provides a unified API for utilizing these algorithms and includes a test environment and benchmarking procedures for evaluating performance and robustness;

2. The integration of prototypes into existing protocols and applications, such as integrating liboqs into OpenSSH.

The project is released under the MIT license and offers language bindings for various programming languages like Go, Java, .NET, Python, and Rust, making it versatile and usable in a wide range of applications. Liboqs can integrate into existing crypto libraries like OpenSSL, BoringSSL, and OpenSSH. The project has made significant progress, with the latest release being version 0.6, supporting most of the round three algorithms, except for a few like GEMSS (Group-Enhanced Merkle Signature Scheme) 2.2.

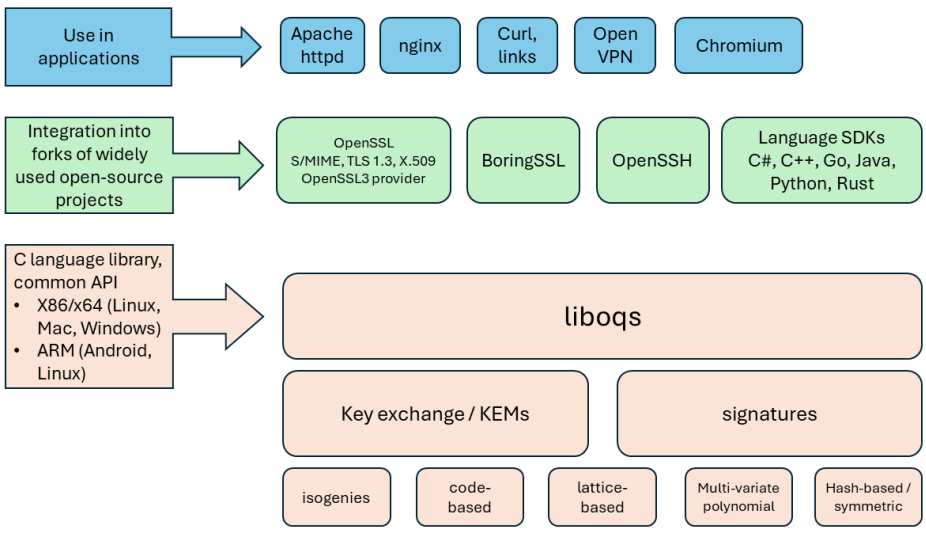

Figure 2.2. OQS Project, Intagration of liboqs [8]

2.7.3 Transition Plan

Hybrid Mode

Regarding the key sizes of PQC algorithms, some algorithms have key sizes larger than 64 KB, with some exceeding one megabyte, which may impact deployment and implementation. As for transitioning to PQC, the industry is considering a hybrid approach. This approach involves combining PQC algorithms with NIST-approved algorithms to maintain security while adapting to the post-quantum threat landscape. For instance, combining PQC key establishment algorithms with EC-DHE for key exchange or using PQC digital signatures alongside ECDSA for authentication purposes. This hybrid mode increases the complexity for attackers, as breaking both current modern cryptography and PQC simultaneously would be challenging.

Stateful hash-based Cryptography

Apart from the hybrid mode, another transition plan uses stateful hash-based cryptography. This plan emphasises the importance of preparing for the quantum era while ensuring compatibility and security in the interim period. The NIST HBS scheme is so named because its private key comprises a large set of one-time signature private keys. The signer must carefully manage these keys' states to ensure that each one-time key is used only once for signing a message. If an attacker were to obtain the digital signatures for two different messages that were created using the same one-time key, it would become possible for them to forge signatures for arbitrary messages. Therefore, extreme care must be taken to prevent the reuse of one-time keys when implementing an HBS scheme.

One specific use case for stateful HBS is firmware image authentication, where signing a large number of messages with limited-time keys is required. The number of messages that can be signed with one specific public-private key pair in stateful HBS is limited, depending on the parameter h, which represents the height of the hash tree. For example, with h set to 10, the key can sign around 1700 messages, assuming one release image needs to be signed every day for three years. If a developer needs to sign multiple debug images daily, a larger h value, such as 20, would be necessary to accommodate the increased workload. The public key size is irrelevant to the parameter h; only the signature size is related to h. The larger the number of signed messages required, the larger the value of h needed, resulting in larger signature sizes.

2.7.4 Potential PQC usage in UEFI

In the context of UEFI, there are two potential categories for PQC usage:

1. General-purpose PQC algorithms, which can be used for key establishment during session creation or digital signatures for runtime challenge-response scenarios like identity authentication;
2. Stateful hash-based signatures, which are suitable for special use cases such as signing firmware capsule updates, secure image verification, and variable authentication during different UEFI phases.

To cover the general PQC usage, we constructed "LibOQS" in EDKII. We find "LibOQS" to be a suitable library as it defines a common interface for all round-three algorithms. For example, "OQS_Signature_New" is for digital signature algorithms, while "OQS_KEM_New" is for key establishment algorithms. If a program needs to switch to a new algorithm, it can simply change the algorithm name. EDKII, short for EFI Development Kit II, is an open-source software development environment designed to facilitate the development of firmware based on UEFI. It represents an updated and enhanced version of the previous EFI Development Kit (EDK), initially developed by Intel. EDKII offers a wide range of tools, libraries, and resources for UEFI firmware development, enabling developers to create, test, and customize system firmware more efficiently. It encompasses essential components such as firmware cores, device drivers, protocols, and user

interfaces, along with tools for firmware compilation, debugging, and simulation. Thanks to its open-source nature, EDKII is widely embraced by the developer community for creating UEFI firmware across various devices and platforms, including PCs, servers, embedded devices, and IoT systems. Its flexibility and modularity make it a valuable tool for supporting cutting-edge technologies and implementing new features in UEFI firmware.

Some implementations use a large stack, up to 4MB, which can cause stack overflow into the heap area in UEFI, leading to allocation failures. To detect stack or heap usage in different PQC implementations, we need to carefully manage resources and adjust stack sizes as necessary to avoid overflow issues. For the signature algorithms, such as Rainbow and SPHINCS+, we noticed that they also demand substantial stack space, with Rainbow needing a particularly large stack. In addition to LibOQS, we enabled two HBS algorithms, LMS and XMSS, in EDKII. Unlike general signature algorithms, these only require a verification function in EDKII, as key generation and signing occur during the manufacturing phase for secure boot or firmware updates. After calculating the stack and heap sizes for both LMS and XMSS reference codes, we found the usage to be acceptable. XMSS consumes slightly more stack space than LMS, which is reasonable given the implementation's use of maximum-sized fixed arrays instead of variable-sized ones. If smaller parameter sizes are viable, we should aim to reduce stack usage accordingly.

Beyond UEFI, other industry standards are also impacted. For instance, BIOS may enable HTTPS, requiring a network TLS. TLS includes public certificates, and digital signature key exchange data and researchers have published papers prototyping post-quantum and hybrid key exchange and authentication for TLS. Similarly, BIOS may use the SPDM (Security Protocol and Data Model) protocol to authenticate devices and obtain signed measurements or establish secure sessions. SPDM, akin to TLS, includes public key certificates, signature key exchange data, etc. However, the SPDM protocol has limitations, such as a maximum certificate chain size of 164 KB and a session message size limitation of 64 KB. Regarding TPM, it's another area of research for PQC. Prototypes have been developed, but limitations have been identified, such as default I/O buffer size and longer stateful HBS key generation times. To address these limitations, larger TPM RAM and cache may be needed.

Chapter 3

ARM TrustZone

3.1 ARM Trusted Firmware

The ARM Trusted Firmware (ATF) is a critical software component designed to provide a secure and reliable environment on ARM processors, especially on platforms based on the ARMv8-A architecture. ATF is a fundamental element in the security ecosystem for devices powered by ARM processors, offering a foundation for secure boot, cryptographic key management, workload isolation, and much more. Its modular architecture allows for flexible configuration, adaptable to the specific needs of the system and security policies. ATF tightly integrates with the Trusted Execution Environment (TEE) and other security technologies, such as TrustZone, to provide a trustworthy environment for executing sensitive security code. Additionally, ATF supports the ARM Trusted Firmware Interface specifications, which define a standard API for interfacing trusted firmware with higher-level system software. This facilitates interoperability and portability of the firmware across different ARM platforms.

ATF is a crucial element in building secure devices based on the ARMv8-A architecture. It ensures secure boot and the management of critical system resources in a reliable environment. TF-A, or Trusted Firmware-A, is a critical component in the system boot process, operating before the main operating system loads. Its primary objective is to uphold system integrity and security through various essential functionalities. At the forefront of TF-A's capabilities is Secure Boot. Acting as the initial gatekeeper, TF-A meticulously verifies the integrity and authenticity of subsequent boot images, particularly when employed as the First or Second Stage Boot Loader (FSBL or SSBL). By scrutinizing boot images before their execution, TF-A effectively thwarts unauthorised or tampered software from infiltrating the system, thus bolstering overall security. Another vital responsibility of TF-A is managing TrustZone modes within the ARM processor architecture. Through seamless transition between secure and non-secure modes, TF-A ensures the proper segregation of secure and non-secure operations. This segregation is fundamental for safeguarding sensitive data and critical system functions from potential security breaches or unauthorized access.

TF-A also facilitates the Secure Monitor Interface, providing a secure pathway for communication with the Secure Monitor residing within the secure world. This interface enables Secure Monitor Calls (SMC) to be handled securely and under controlled conditions, reinforcing the system's overall security posture. Furthermore, TF-A undertakes the management of hardware interrupts and timers, crucial for maintaining system reliability and stability. By efficiently managing interrupts and providing timer functionalities, TF-A contributes to the seamless operation of the system, ensuring timely execution of tasks and effective response to external events. In brief, TF-A's multifaceted role in the boot process extends beyond mere initialisation. It serves as a bastion of security, orchestrating Secure Boot mechanisms, managing TrustZone modes, providing a secure interface for system calls, and ensuring reliable interrupt and timer management. Through these key functionalities, TF-A lays the groundwork for a secure, robust, and trustworthy system environment.

TF-A can be used as the Secondary Program Loader (SSBL):

- TF-A BL2 → BL33 (U-Boot, Barebox)

In this configuration, U-Boot or Barebox is used to load and boot the operating system (such as Linux) or other software execution environments after TF-A has completed its initial boot phase. They can handle advanced initialization operations, device configuration, and system booting.

Alternatively, TF-A can be loaded later in the boot process, for example:

- U-Boot SPL (Secondary Program Loader) → TF-A → U-Boot
- Barebox PBL (Pre Boot Loader) → TF-A → Barebox

U-Boot and Barebox are both bootloaders primarily used in embedded systems and devices running Linux as the operating system.

Note: U-Boot SPL and Barebox SPL are reduced versions of U-Boot and Barebox designed to run during the early stages of the boot process.

3.1.1 TF-A Services

Power Management

The Power State Coordination Interface (PSCI) is an interface designed to manage power management requests within a computer system. Specifically, PSCI coordinates power management requests originating from the NS-worl (Non-secure world) and informs the S-world (Secure world) of such requests.

This process may involve operations such as:

- core idling management;
- CPU activation/deactivation;
- system power on/off control.

An important feature of PSCI is its ability to forward power management requests to the Secure Monitor, which is a critical component of the operating system responsible for executing low-level system operations and security. The Secure Monitor, operating at firmware level running at the highest privilege level (EL3), receives requests through an interface called Secure Monitor Call (SMC). Essentially, PSCI acts as a bridge between power management requests from various parts of the system and low-level system firmware responsible for executing such operations. This helps ensure that power management operations are performed securely and efficiently, contributing to system stability and overall performance. PSCI is widely used in embedded systems and devices with advanced security architectures, such as devices based on ARM architectures with TrustZone. This is because secure power management is critical to ensuring the security of data and operations within such devices 3.1.

System Control and Management Interface (SCMI) driver

The System Control and Management Interface (SCMI) driver represents a fundamental component for managing power, performance, and resources within a System-on-Chip (SoC). This standard interface offers a consistent and uniform way to access and control these functionalities, regardless of the specifics of the hardware device. To function properly, the SCMI driver requires a power controller, which regulates the power and operating modes of the SoC based on requests from the operating system or other system components. Additionally, the SCMI driver facilitates interaction with the Arm System Control Processor (SCP), which plays a crucial role in system management. The SCP allows for the delegation of specific management tasks, ensuring efficient and reliable control of system operations. A key aspect of the SCMI driver is its ability to provide platform-agnostic Application Processor (AP) firmware. This means that the firmware can be implemented on different platforms without needing modification, simplifying the development and maintenance process of the system 3.2.

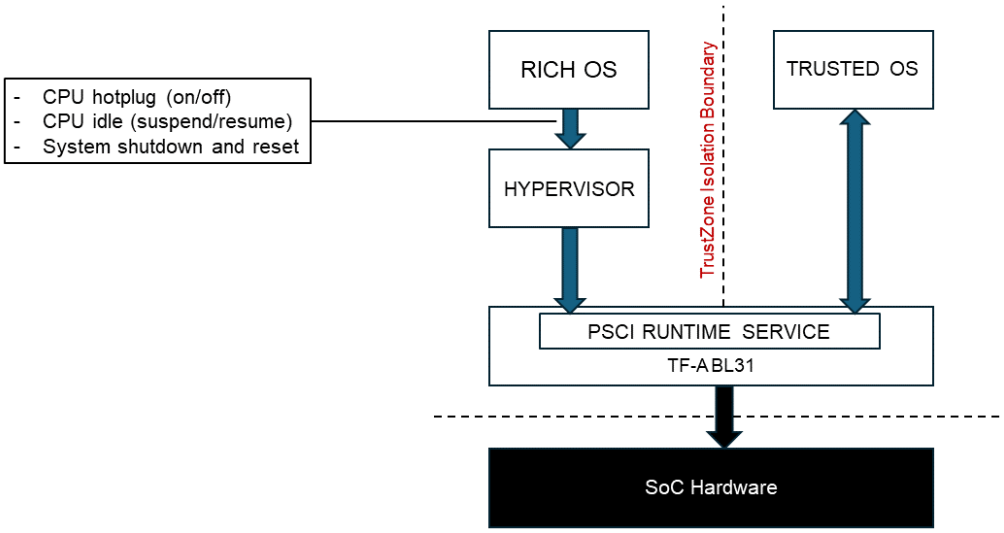

Figure 3.1. Power State Coordination Interface

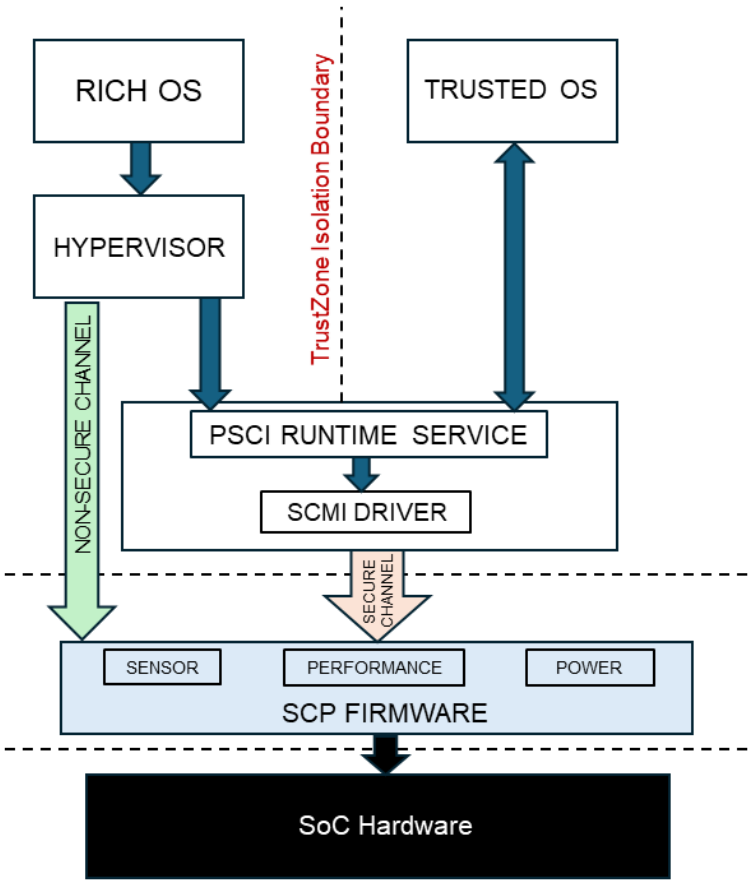

Figure 3.2. SCMI Driver

Exception Handling

The Software Delegated Exception Interface (SDEI) is a standard forum for delivering critical system events. The Trusted Firmware provides a compliant reference implementation of this interface, managed through SMC calls. These exceptions could stem from critical system events such

as hardware failures, error conditions, or debug events (both hardware and software-related). The operating system or hypervisor can register callbacks with the SDEI to manage these exceptions efficiently. The purpose of these callbacks is to ensure that critical system events are handled promptly and appropriately. SDEI operates with two levels of exception priority: normal and critical. This distinction allows for the prioritisation of exception handling based on the severity of the event. Moreover, in situations where the operating system or hypervisor requires elevated privileges or sensitive operations to be performed, SDEI facilitates communication through Secure Monitor Calls (SMC). This ensures that the handling of such exceptions maintains the necessary security measures and access controls. In the current implementation of TF-A, platform error handling through RAS (Reliability, Availability, and Serviceability) routes mass events up into the normal world to be handled 3.3.

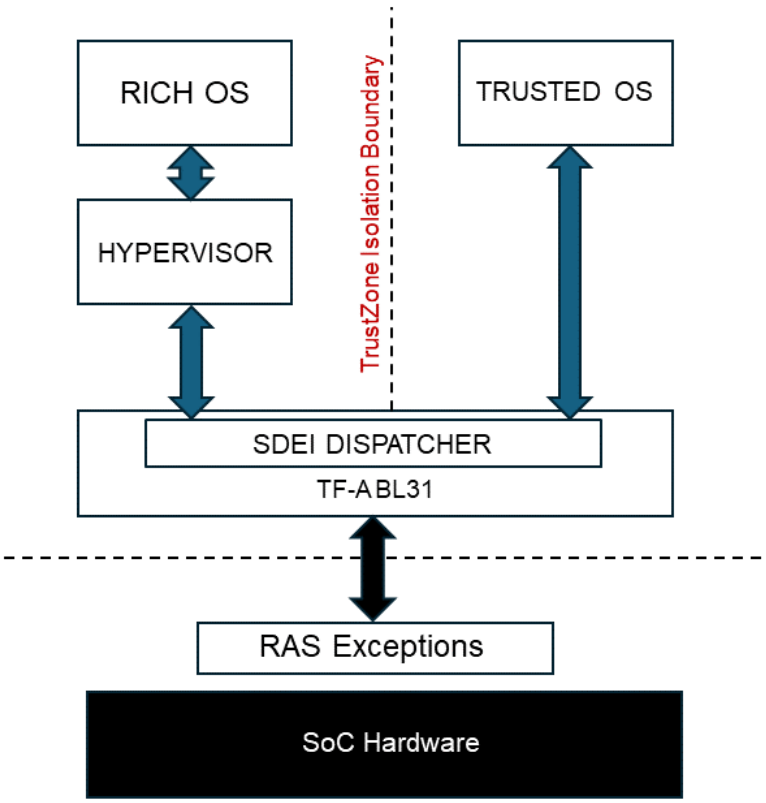

Figure 3.3. Software Delegated Exception Interface

3.2 ARM TrustZone Technology

ARM TrustZone is a security technology developed by ARM Holdings that provides a secure and isolated environment within ARM processors, enabling the execution of sensitive code and data in a protected context. It is designed to address security challenges in mobile devices, IoT, and other embedded systems. The TrustZone architecture divides the ARM processor into two "worlds": the Secure World (SeW) and the Normal World (NoW). The SeW is isolated and protected from the NoW through dedicated hardware and software. This separation allows sensitive operations, such as key management, authentication, and access control, to be performed in a secure and controlled environment. ARM TrustZone represents a critical technology in ensuring the security of devices powered by ARM processors, providing a robust framework for establishing and maintaining secure computing environments. At its core, TrustZone leverages dedicated hardware to enforce hardware isolation between the SeW and the NoW. This separation ensures that sensitive resources and data remain shielded from unauthorised access, bolstering the overall security posture of the system.

Central to the architecture is the TrustZone Monitor (TZM), also known as Secure Monitor, a vital software component responsible for managing the transition between the SeW and NoW. TZM plays a pivotal role in orchestrating secure interrupts, configuring secure and non-secure contexts, and providing essential security services that underpin the integrity of the system. This technology offers a suite of Secure APIs, facilitating controlled and secure communication between software residing in the NoW and the SeW. This enables the development of secure applications capable of accessing protected resources while adhering to stringent security protocols. Another key point is the establishment of Roots of Trust (RoT), comprising trusted hardware and software components that serve as the bedrock for system security. These RoT form a foundation upon which additional security measures can be built, ensuring the integrity and reliability of the overall system.

ARM TrustZone represents a comprehensive approach to security, enabling sensitive operations to be executed within an isolated and protected environment. By providing hardware-enforced isolation, essential security services through the TZM, Secure APIs for controlled communication, and a foundation for RoT, TrustZone serves as a cornerstone technology for safeguarding devices and data in today's interconnected world.

3.2.1 ARM Cortex-A Processor

In processors such as the ARM Cortex-A series, software execution occurs within either a secure or non-secure state 3.4. The privileged software, known as the Secure Monitor (SM), is responsible for implementing mechanisms for secure context switching between these states, ensuring the integrity and confidentiality of sensitive operations and data [10].

The determination of the current execution state of the processor is governed by a single bit known as the Non-Secure (NS) bit. This bit's value is indicative of whether the processor is operating in the secure or non-secure world. The Secure Configuration Register (SCR) holds the value of this bit, which is propagated throughout the system, including memory buses and peripherals, influencing the security context of all software execution. This architecture enables the establishment of a secure environment, where critical operations such as cryptographic key handling, secure boot, and authentication take place. In contrast, the NoW accommodates regular application execution and general system tasks.

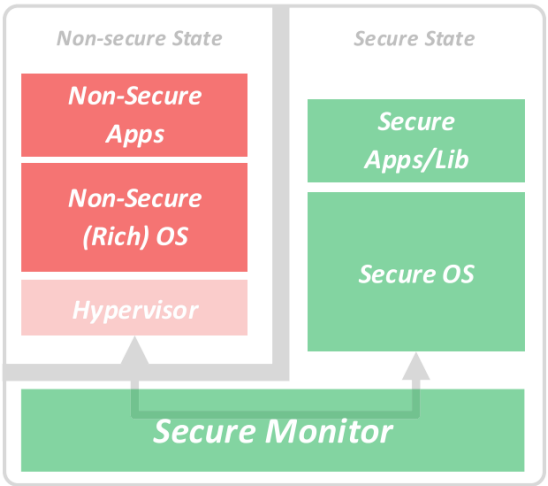

Figure 3.4. TrustZone for Cortex-A [10]

By leveraging the SeW and NoW segregation, ARM processors can ensure robust security measures while maintaining the flexibility and performance required for a wide range of computing applications. TrustZone technology, coupled with SM implementations and the NS bit mechanism, forms a foundation for building secure systems across various domains, including mobile devices, IoT endpoints, automotive systems, and more.

3.2.2 Monitor Mode

The Monitor mode provides a secure environment for executing sensitive operations, such as accessing system resources or managing peripherals. This protected environment is crucial for ensuring the security of critical operations performed on the device. When the processor is in Monitor mode, it waits for service requests from the SM. The SM is a critical software component responsible for handling requests from the NoW and providing the requested services securely and in a controlled manner.

The processor can enter Monitor mode in two main ways:

1. **Secure Monitor Call (SMC)**: the processor can enter Monitor mode by executing a privileged instruction called SMC. This instruction allows the software running in the NoW to request services from the SM. The SM responds to these requests by performing the necessary actions securely and then returning to the previous context.
2. **Exception and Interrupt Configuration**: the processor can also enter Monitor Mode through the appropriate configuration of exceptions, Interrupt Request (IRQ), and Fast Interrupt Request (FIQ) handled in the SeW. When an exception or interrupt handled in the SeW occurs, the processor automatically switches to Monitor Mode to handle the event securely and appropriately.

Monitor Mode is essential for ensuring the security and reliable management of critical operations on ARM-based devices, allowing the SM to provide protected and controlled services for applications in the NoW.

3.2.3 TEE and REE

ARM TrustZone enables the utilization of a TEE, a standard supported by GlobalPlatform for the isolated execution of software [11]. The TEE operates within an isolated execution environment, running concurrently with a standard operating system, which operates within a so-called Rich Execution Environment (REE). Unlike the REE, only security-critical, authenticated, and unaltered software is intended to be executed within a TEE. This ensures that the trusted computing base remains as small as possible. Software in the NoW can interact with Software in the SeW through a TEE driver, accessible from the user space via the Client TEE API. Trusted Applications (TAs) running within the SeW cannot directly access functionalities provided by a Trusted Operating System (TOS). Access to such functionalities is only possible through the Internal Core TEE API.

Additionally, the TEE provides a secure environment for sensitive operations, such as cryptographic key management, secure storage, and secure communication channels. It ensures the confidentiality, integrity, and availability of data and services within the trusted environment. Furthermore, the TEE is responsible for enforcing access control policies, authenticating users and devices, and maintaining the security posture of the system. It acts as a TEE for critical security functions, protecting against various threats, including malware, unauthorised access, and physical attacks. Overall it plays a pivotal role in establishing a secure foundation for computing platforms, enabling the execution of sensitive operations and ensuring the security and integrity of the system as a whole.

fTPM and TEE/TPM drivers

An fTPM, or Firmware-based Trusted Platform Module, is an implementation of a TPM that utilizes system firmware to provide TPM functionality. It operates by emulating the functions of a hardware TPM using system firmware. In the context of a TEE like OP-TEE, the fTPM can be implemented as a service within the TEE itself. This means that the fTPM runs inside the TEE and utilises the resources and protections of the TEE to provide a secure environment for TPM operations. Communication between the fTPM and the TEE occurs through a defined

interface, which may include specific API calls for TPM and mechanisms for secure communication between the fTPM and the TEE. TPM drivers and TEE drivers in the NoW are necessary to enable software in the NoW to interact with the fTPM and the TEE 3.5.

TPM drivers allow software in the NoW to communicate with the fTPM, sending requests and receiving responses through the appropriate interface. TEE drivers, on the other hand, allow software in the NoW to communicate with the TEE itself, sending requests and receiving responses through the TEE's interface. These drivers act as a bridge between software in the NoW and the TEE, enabling software in the NoW to leverage the secure and protected functionalities provided by the TEE, including access to the fTPM for TPM operations.

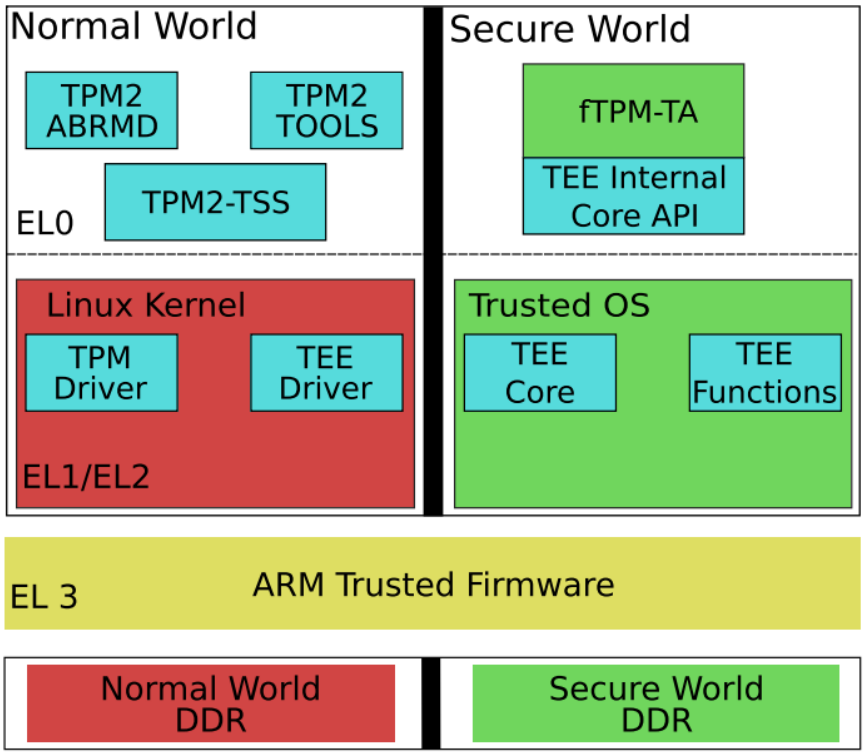

Figure 3.5. Architectural representation of Microsft's fTPM running inside an ARM TrustZone based Trusted Execution Environment [11]

TEE API

Within the TEE, two distinct sets of programming interfaces, or APIs, serve essential functions in facilitating communication and managing operations. Firstly, the Internal Core TEE API stands as a fundamental component, enabling software residing within the TEE itself to access internal functionalities and services. These APIs are integral for the TEE kernel and other internal TEE services to execute critical operations, administer system resources, and deliver fundamental functionalities necessary for the TEE's operation. Given the elevated privilege level associated with operations conducted via the Internal Core TEE API, they necessitate special permissions within the TEE.

In contrast, the Client TEE API serves as a conduit for communication between software external to the TEE, such as applications in the NoW and the TEE. This set of APIs enables software in the NoW to initiate requests to the TEE, access TEE services and functionalities, and receive responses or results from TEE operations. Although operations conducted through the Client TEE API are at a lower privilege level compared to those executed via the Internal Core TEE API, they require validation and authorisation by the TEE before execution. In essence, the Internal Core TEE API empowers software within the TEE to conduct critical operations and manage system resources, while the Client TEE API facilitates secure and controlled

communication and interaction between external software and the TEE. Together, these APIs play complementary roles in bolstering the security and efficacy of the TEE within the operating environment.

Secure Boot within the TEE

In the context of Secure Boot within the TEE, the RoT serves as the initial boot point where the integrity of critical software, such as the bootloader, firmware, and TEE code itself, is verified. Secure Boot within the TEE ensures that only authorised and verified software can be executed within the secure environment of the TEE, thereby maintaining the integrity of the Trusted Computing Base (TCB). This protects sensitive data and cryptographic operations managed within the TEE from malicious attacks or unauthorised modifications to the software. Additionally, Secure Boot in the TEE establishes a chain of trust from the RoT to subsequent layers of software, ensuring that each component in the boot process is verified and authenticated before being executed. This helps prevent the compromise of the system by malicious actors or unauthorised modifications to critical software components. In the TEE, only code that has been appropriately authorised and whose authorisation has been verified by other authorised codes is accepted for execution. This authorisation process encompasses all code executed after the ROM boot (with the understanding that the ROM code is authorised by its presence).

TAs are only permitted to directly access their own data resources. No method allows a TA to directly access the resources of other TAs or other components of the TEE. This strict access control policy ensures that each TA operates within its designated boundaries and cannot interfere with the operation or access the data of other TAs or components within the TEE. It enhances the security and isolation of the TEE environment, protecting sensitive data and critical operations from unauthorised access or interference.

3.2.4 Exception Levels

In the ARM TrustZone architecture, the concept of "exception levels" serves as a cornerstone, delineating various levels of privilege and isolation within the processor. These exception levels denoted as EL0, EL1, EL2, and EL3, each embody distinct features and privileges crucial for system operation.

- The lowest tier is EL0, representing the user level. Here, user code executes, albeit without direct access to hardware or privileged instructions.
- EL1 stands above EL0 and operates as the kernel or supervisor level. It is at this level that the operating system kernel operates, endowed with access to all hardware resources and privileged instructions essential for managing the operating system and providing services to the user level (EL0).
- Advancing further, EL2 surpasses EL1 and serves as the Trusted OS Monitor level. EL2 is designated for running a trusted monitor or hypervisor of the operating system, equipped with full access to hardware resources and privileged instructions, including control over lower levels.
- EL3 occupies the apex as the highest exception level and the epitome of security. Reserved for secure system boot and execution of the SM, EL3 enjoys full access to all hardware resources and wields control over all other exception levels.

Each exception level furnishes an isolated and privileged environment for code execution, characterised by varying levels of authority and access to system resources. Leveraging these levels facilitates the implementation of sophisticated security policies, ensuring robust isolation between different system components. Consequently, this architecture contributes significantly to furnishing a secure and reliable platform conducive to diverse computing environments.

Chapter 4

Zync Ultrascale+ MPSoC

4.1 ZU+ architecture

In embedded systems, ensuring security and flexibility is paramount, particularly as the landscape of cybersecurity threats continues to evolve. The Zynq UltraScale+ MPSoC, a family of System-on-Chip (SoC) devices developed by Xilinx, now a part of Advanced Micro Devices (AMD), emerges as a pivotal solution in addressing these dual imperatives. This thesis explores the integration of the ZU+ MPSoC into embedded systems to enhance security and flexibility, thereby enabling robust protection against emerging threats while facilitating adaptability to diverse application requirements. At the heart of the ZU+ MPSoC lies a fusion of high-performance ARM multi-core processors and the exceptional flexibility and parallel processing capabilities of programmable FPGA technology. This unique amalgamation empowers developers to harness the processing power of ARM Cortex-A53 and Cortex-R5 cores while leveraging the customizable nature of FPGAs to tailor the system to specific application needs 4.1.

4.1.1 ZU+ components

The Zynq UltraScale+ MPSoC features are as follows [12]:

- Cortex-R5F dual-core real-time processor unit (RPU)
- Arm Cortex-A53 64-bit quad/dual-core processor unit (APU)
- Mali-400 MP2 graphic processing unit (GPU)
- External memory interfaces: DDR4, LPDDR4, DDR3, DDR3L, LPDDR3, 2x Quad-SPI, and NAND
- General connectivity: 2x USB 3.0, 2x SD/SDIO, 2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x1GE, and GPIO
- Security: Advanced Encryption Standard (AES), RSA public key encryption algorithm, and Secure Hash Algorithm-3 (SHA-3)
- AMS system monitor: 10-bit, 1 MSPS ADC, temperature, voltage, and current monitor
- The processor subsystem (PS) has five high-speed serial I/O (HSSIO) interfaces supporting the protocols:
  - PCIe: base specification, version 2.1 compliant, and Gen2x4
  - SATA 3.0
  - DisplayPort: Implements a DisplayPort source-only interface with video resolution up to 4k x 2k

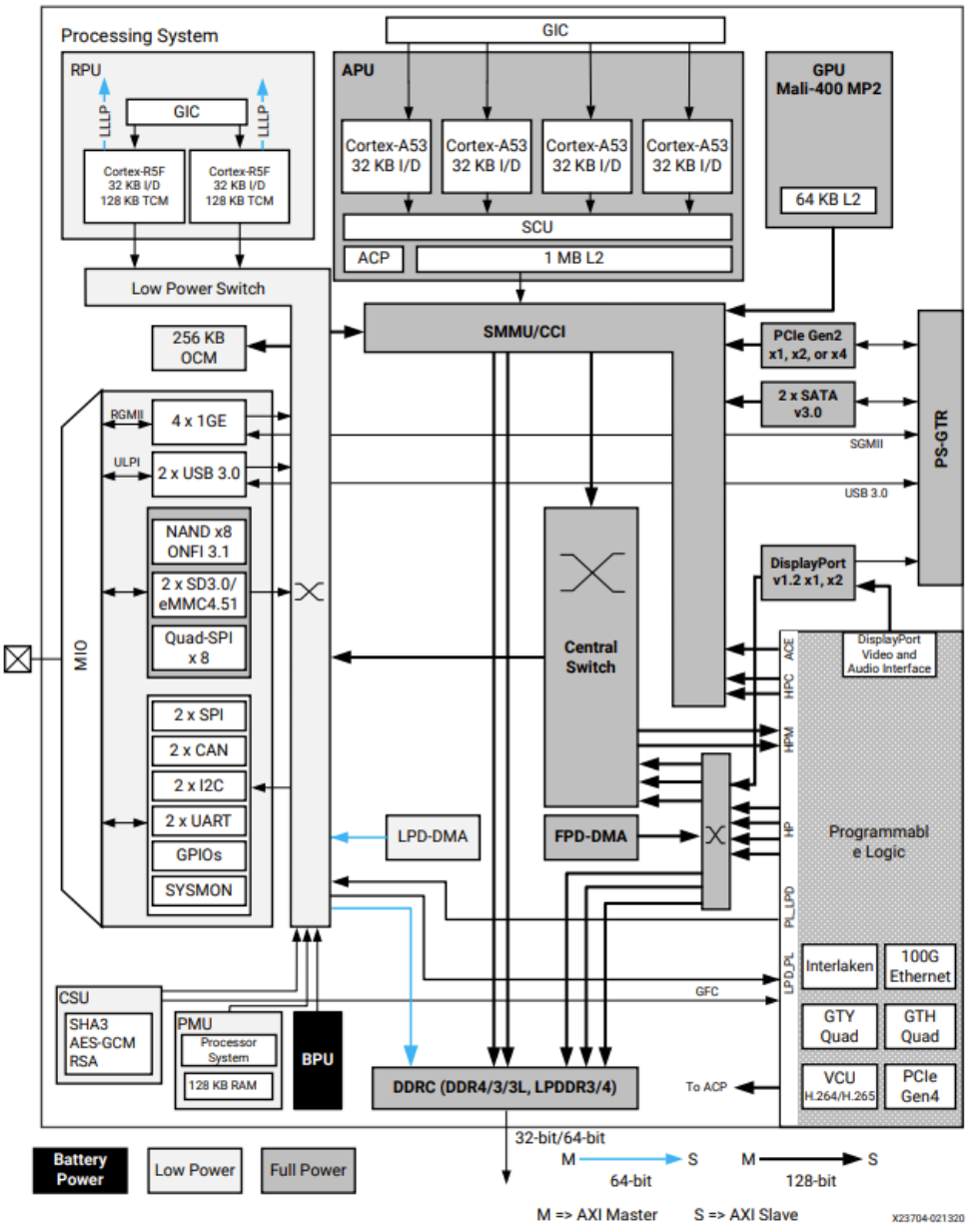

Figure 4.1. ZU+ Device Hardware Architecture [12]

  - USB 3.0: Compliant to USB 3.0 specification implementing a 5 Gb/s line rate
  - Serial GMII: Supports a 1 Gb/s SGMII interface

- Platform Management Unit (PMU) for power sequencing, safety, security, and debug functions.

4.1.2 Application Processing Unit (APU)

The main role of the Application Processing Unit (APU) is to execute the first and second-stage bootloaders, and finally Linux [13].

The APU has the following specifications:

- Quad-core ARM Cortex-A53 processor

- CPU frequency up to 1.5 GHz
- Aarch64 architecture (also known as ARM64)
- 32 kB L1 cache per processor and a shared L2 cache (1 MB)
- Floating-point Unit (FPU) and cryptographic extension

The APU has two levels of cache. Each core has a local L1 cache. Other cores cannot access this cache. The L1 cache is divided into an I-cache for instructions and a D-cache for data. Additionally, there is a shared L2 cache among the cores. This cache has more memory but is slower than the L1 cache. The System Control Unit (SCU) in the APU handles cache coherence and connects the two levels of cache 4.2.

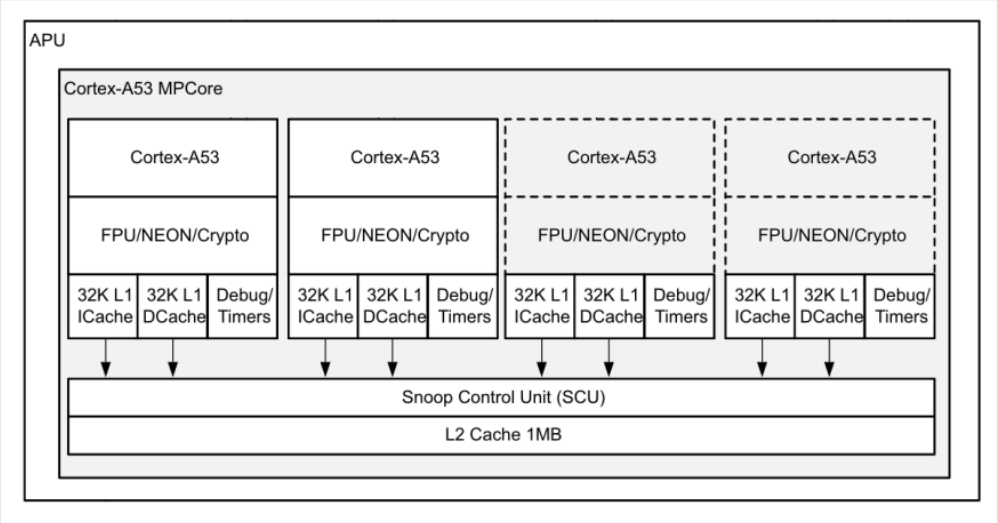

Figure 4.2. APU Architecture [13]

The APU is connected to DDR memory through the System Memory Management Unit (SMMU). The SMMU performs translations from virtual memory addresses to physical memory addresses. It also ensures that only one processor can take control of the memory bus at a time (memory arbitration). It performs memory protection so that each processor can only access the memory allocated to it. The APU can access other parts of the chip through the central switch. It also utilises the low-power switch to access I/O peripherals, and memory integrated into the chip, CSU, and PMU. Subsequently, when discussing Secure Boot, we will delve into the other key components, including the PMU and the CSU.

4.1.3 I/O connectivity

I/O High-Speed Connectivity (main):

- DisplayPort: it is primarily used for high-performance video processing applications. It enables developers to output and test video signals, display real-time results and prototype advanced graphical systems.
- USB 3.0: it is an interface standard for external devices that offers much higher data transfer speeds compared to previous versions of USB. It is widely used for external storage devices, high-resolution digital cameras, video capture devices, and more.
- SATA 3.1: it is an interface standard for high-speed storage devices such as hard disk drives and solid-state drives (SSDs). SATA 3.1 offers data transfer speeds of up to 6 Gbps.

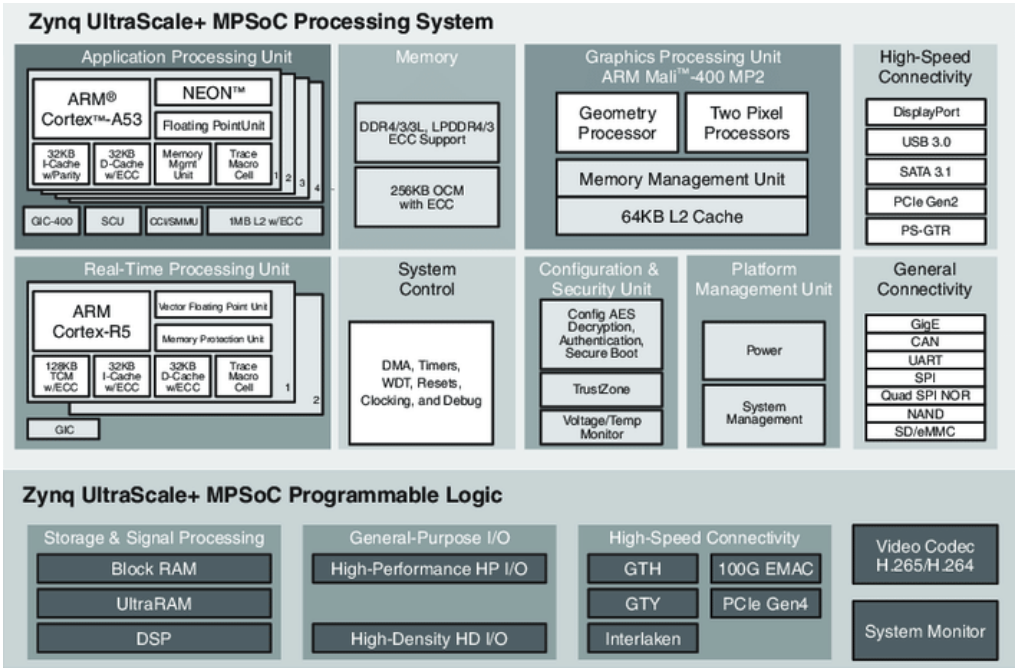

Figure 4.3. ZU+ Device Hardware Architecture (semplified)

- PCIe Gen2: PCIe (Peripheral Component Interconnect Express) is an interface standard for connecting hardware components. PCIe Gen2 is the second generation of PCIe, which offers improved performance compared to the first generation.
- Serial GMII (Serial Gigabit Media Independent Interface): is a network interface that enables connection between a network device and a physical transmitter/receiver (PHY) serially. It is a serial version of the Gigabit Media Independent Interface (GMII), which is a physical layer interface used to connect an Ethernet controller to a PHY in parallel.

I/O General Connectivity (main):

- GigE: GigE (Gigabit Ethernet) is a network connection standard that offers data transfer speeds of up to 1 Gbps. It is widely used for connecting network devices such as computers, switches, and routers.
- CAN (Controller Area Network): is a serial communication standard primarily designed for use in automotive environments but is also used in other industrial applications.
- UART (Universal Asynchronous Receiver/Transmitter): is a hardware component that manages the transmission and reception of data asynchronously through a serial interface.
- SPI (Serial Peripheral Interface): is a synchronous serial communication interface used to connect peripheral devices to a microcontroller or microprocessor.
- QUAD SPI NOR: it is a 4-channel version of SPI primarily used to connect NOR Flash non-volatile memory devices.
- NAND: NAND Flash is a type of flash memory used for long-term data storage in devices such as memory cards, SSDs, and embedded devices.
- SD/eMMC: SD (Secure Digital) and eMMC (embedded MultiMediaCard) are two types of flash memories used primarily for data storage in embedded devices.
- GPIOs (General Purpose Input/Output): These are general-purpose input/output pins that can be programmed to perform a variety of functions within an embedded system.

4.2 Security Features and Root-of-Trust Establishment

Critical to establishing a secure foundation is the ZU+'s robust suite of security features, particularly in the boot process. By providing authentication mechanisms for both software images and bitstreams, starting from the First Stage Boot Loader, the device ensures the integrity and authenticity of the code, mitigating the risk of unauthorised modifications. Furthermore, the device goes beyond authentication to safeguard confidentiality, protecting against threats such as cloning and reverse engineering. The Secure Boot process is a cornerstone in ensuring the integrity and authenticity of embedded systems, particularly in the context of the ZU+ MPSoC. As the foundation of system boot-up, this process orchestrates a series of critical operations aimed at verifying and validating the system's integrity before executing any user code. This thesis delves into the intricacies of the Secure Boot process within the ZU+ MPSoC, shedding light on its underlying mechanisms and functionalities.

4.2.1 The Secure Boot Sequence

The role of the hardware-based finite state machine

At the core of the Secure Boot process in the Zynq UltraScale+ MPSoC lies a hardware-based finite state machine (FSM), intricately woven into the chip's architecture. This FSM, meticulously crafted at the hardware level, serves as the linchpin of the boot sequence, executing a series of critical functions essential for ensuring the robustness and security of the system's startup procedure. Tasked with initiating the boot sequence, the hardware FSM within the ZU+ MPSoC embarks on a journey encompassing several pivotal functions, each meticulously designed to fortify the system's security posture and uphold the integrity of the boot process.

First and foremost, the FSM enforces Test Interface Lockdown, a vital measure aimed at restricting access to the test interface. By fortifying this access point, the FSM mitigates the risk of unauthorised manipulation or tampering with the system's components, thereby bolstering its overall security. Subsequently, the FSM undertakes the crucial task of PMU Register Zeroisation, ensuring a clean slate by resetting the PMU registers. This preemptive measure minimises the likelihood of residual data interfering with the boot process, fostering a pristine environment for system initialisation.

The FSM optionally executes a Built-In Self-Test (BIST) on the programmable FPGA logic, validating its functionality and integrity. While optional, this step is highly recommended to verify the readiness of the FPGA logic for operation, enhancing the reliability of the system. Furthermore, leveraging the cryptographic capabilities of the CSU, the FSM performs a SHA-3/384 Integrity Check on the PMU ROM. By subjecting the boot code to rigorous cryptographic verification, the FSM establishes trust in its authenticity, safeguarding against tampering or corruption.

Lastly, upon successful completion of the preceding checks, the FSM releases the reset signal to the PMU, signalling its readiness to proceed with the execution of the BootROM code and initiate the subsequent boot-up sequence. In essence, the hardware FSM within the ZU+ orchestrates a meticulously choreographed ballet of security measures, each aimed at fortifying the system's integrity and resilience during the critical boot process. Through its diligent execution of essential functions, the FSM lays the foundation for a secure and reliable system startup, safeguarding against potential threats and ensuring the continuity of operations.

The role of the Platform Management Unit (PMU)

Operating as a highly robust triple-redundant MicroBlaze processor, the PMU assumes a critical role in the Secure Boot process of the ZU+ MPSoC. Following the completion of its checks and functions, the PMU undertakes an integrity check of the CSU ROM, ensuring the trustworthiness of the boot code before proceeding further. If the integrity check yields a positive result, the PMU releases the reset signal to the CSU, enabling it to initiate the execution of its own BootROM code.

The architecture of the PMU is designed to efficiently control power management within the Low Power Domain (LPD) of a system. At its core, the PMU includes a dedicated, fault-tolerant triple-redundant processor that ensures high reliability. It features ROM storage for PMU boot code and routines that manage power-up and power-down requests and handle interrupts. Additionally, the PMU has 128 KB RAM with error correction capabilities (ECC) used for code and data storage.

The PMU architecture comprises local registers accessible only to the PMU and global registers accessible both by the PMU processor and other system masters, which include power, isolation and reset request registers. A 32-bit AXI slave interface allows access to the PMU RAM and global registers from other components outside the PMU. To manage system signals, the PMU includes an interrupt controller for 23 interrupts, four of which come from the inter-processor interconnect (IPI). General-purpose input (GPI) and output (GPO) registers facilitate communication between the PMU, the PS (processing system), the PL (programmable logic), and other resources within the system. These registers support a range of signals, such as power and reset controls, memory built-in self-test (MBIST) status, error reporting, and direct reset commands. The PMU also incorporates a Microprocessor Debug Module (MDM) controller accessible through the PS JTAG interface, enabling debugging operations. Together, these components make the PMU architecture a robust and reliable framework for managing power, signal communication, and error handling across the system 4.4.

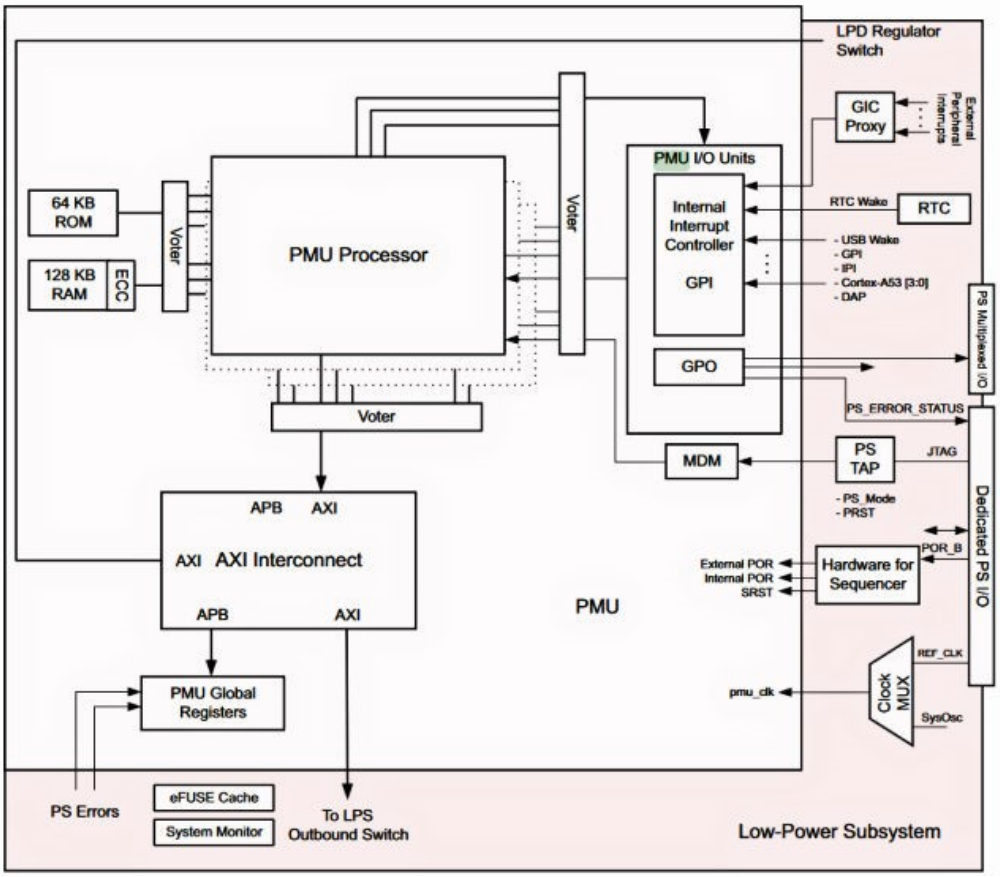

Figure 4.4. PMU architecture [13]

One of the primary tasks undertaken by the PMU is the optional zeroisation of registers within the LPD and Full Power Domain (FPD). This process wipes these registers clean, eliminating any residual data that could potentially interfere with system operations, thereby enhancing the overall security of the boot process. Additionally, the PMU resets its internal RAM, effectively clearing any sensitive information stored within. This further fortifies the security posture of the boot process by preventing unauthorised access to potentially sensitive data.

Another critical function performed by the PMU is the verification of voltages across various domains, including LPD, Auxiliary (AUX), and Input/Output (I/O). By ensuring that the system operates within specified power parameters, the PMU safeguards against potential voltage-related issues that could compromise system stability and reliability. Optionally, the PMU can also zeroise memories within the Central Security Unit (CSU), LPD, and FPD domains. This additional security measure reduces the risk of data leakage or unauthorised access, further bolstering the overall security of the boot process.

Furthermore, leveraging the cryptographic capabilities of the CSU, the PMU conducts an integrity check on the CSU ROM using the SHA-3/384 hashing algorithm. This validation step verifies the authenticity and integrity of the boot code stored within the CSU ROM, ensuring that only trusted code is executed during the boot process. Upon successful completion of the integrity check, the PMU releases the reset signal to the CSU, allowing it to initiate the execution of its BootROM code and subsequent boot-up procedures. This final step marks the transition from the preparatory phase to the actual execution of the boot process, setting the stage for the device to become fully operational.

The additional functionalities of the PMU are managed by the PMU firmware. This firmware is divided into multiple blocks, which consist of APIs and modules. These modules utilise the APIs provided by the base PMU firmware to perform tasks and functions. The core APIs of the PMU firmware are essential for the modules, providing functions for system initialisation, power monitoring, system error management, timer control, and so on (with Inter-Processor Interrupts managing interrupts sent between processing units) 4.5.

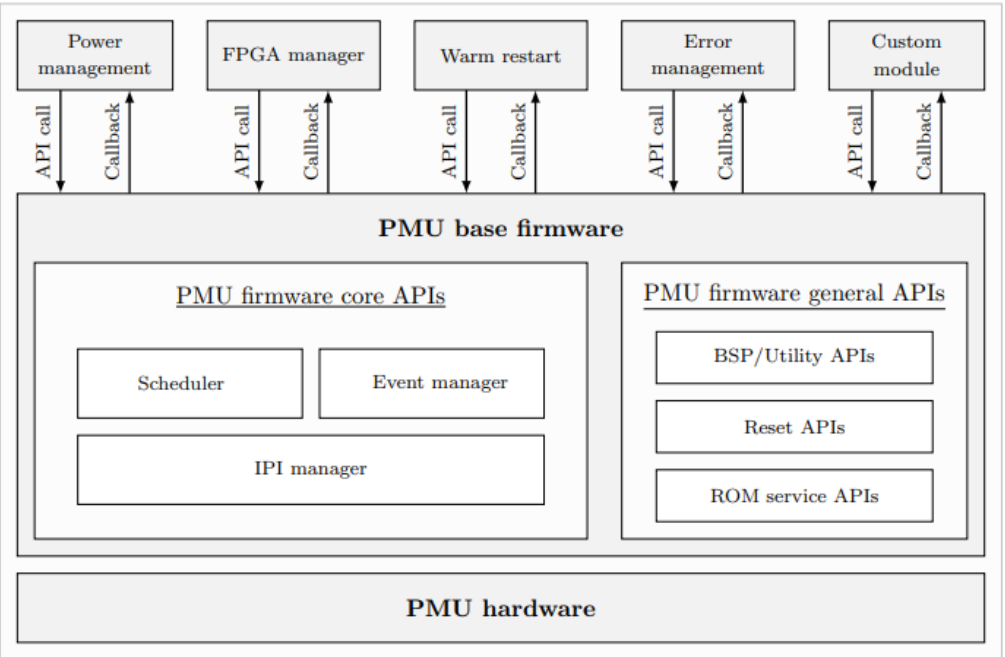

Figure 4.5. PMU base firmware [12]

This modular structure enhances the versatility and scalability of the PMU firmware, allowing it to adapt to various system requirements and configurations. By leveraging the core APIs, the modules can efficiently execute their designated tasks, contributing to the overall efficiency and reliability of the system. In essence, the PMU firmware serves as the backbone of system management, orchestrating a synchronised effort between its constituent blocks to ensure optimal performance and stability in computing devices. Through its modular design and robust API framework, it empowers devices to navigate the complexities of modern computing environments with resilience and efficiency.

Triple-Redundant MicroBlaze Processor

The inclusion of a triple-redundant MicroBlaze processor architecture within the ZU+ MPSoC enhances the system's fault tolerance and reliability, albeit at an increased computational cost. This configuration operates on the principles of parallel execution, voting, and fault tolerance, ensuring robustness even in the face of hardware failures or transient errors. The triple-redundant MicroBlaze architecture enables parallel execution, with three instances of the processor operating simultaneously. Each of these instances executes the same set of instructions and processes data independently, effectively increasing the computational throughput of the system. Upon completion of each operation, the results produced by the different processor instances are compared. The majority outcome is then selected as the final result, ensuring precision and reliability in determining the system's response, even in the presence of potential discrepancies or errors in individual processor instances.

The primary advantage of this triple-redundant architecture lies in its fault-tolerance capabilities. In the event of a hardware failure or transient errors affecting one of the processor instances, the remaining two continue to operate seamlessly, preserving the functionality and integrity of the system. This fault tolerance mechanism enhances the system's reliability and availability, particularly crucial in mission-critical applications. It is important to acknowledge that the adoption of this architecture incurs increased computational costs compared to a single processor configuration, due to the need to execute and compare results across all three instances. However, this additional cost is often justified in applications requiring high reliability and fault tolerance.

The role of the Configuration Security Unit (CSU)

The ZU+ device features an independent CSU responsible for enabling the Secure Boot process. Situated within the LPD, the CSU oversees security-related functions and boot operations crucial for establishing a trusted boot environment. The CSU comprises two primary blocks. On the left is the Secure Processor Block (SPB), housing the highly robust triple-redundant MicroBlaze processor for boot operation control. Additionally, it includes an associated ROM, a small private RAM, and necessary control/status registers to support all secure operations. On the right side lies the Cryptographic Interface Block (CIB), housing core components such as AES-GCM, CSU DMA (Direct Memory Access), SHA-3 core, RSA accelerator, and Processor Configuration Access Port (PCAP). Post-boot, the CSU serves for tamper detection, while the cryptographic blocks can be utilised by an application running in both the Processing System (PS) and Programmable Logic (PL).

It executes a series of critical functions, each designed to establish and maintain a secure foundation for system operation. First and foremost, when RSA authentication is enabled, the CSU applies the Hardware Root-of-Trust (HW RoT). This process verifies the authenticity of hardware components using RSA authentication, laying the groundwork for a secure boot process. Additionally, the CSU enforces the Secure Boot mode, ensuring that only trusted software components are permitted to execute during system startup. This measure prevents the execution of unauthorised or compromised software, thereby bolstering the system's overall security posture. The CSU validate the integrity of user-provided public keys. By verifying the integrity of these keys, the CSU ensures that only valid and unaltered keys are accepted for authentication purposes, safeguarding against potential security breaches. Furthermore, the CSU enables the revocation of compromised or unauthorised public keys, mitigating the risk posed by unauthorised access attempts. This capability adds a layer of security by preventing the use of compromised keys in authentication processes. In the boot process, the CSU is responsible for loading essential components such as the First Stage Boot Loader (FSBL) and PMU firmware into memory for execution. These components are vital for initialising essential system components and facilitating the boot process.

Additionally, the CSU authenticates and/or decrypts critical firmware or data during the boot process, ensuring its integrity and confidentiality. This authentication and decryption process further enhances the security of the system by verifying the integrity of essential components. After processing, including during fallback scenarios, the CSU securely erases sensitive data stored in

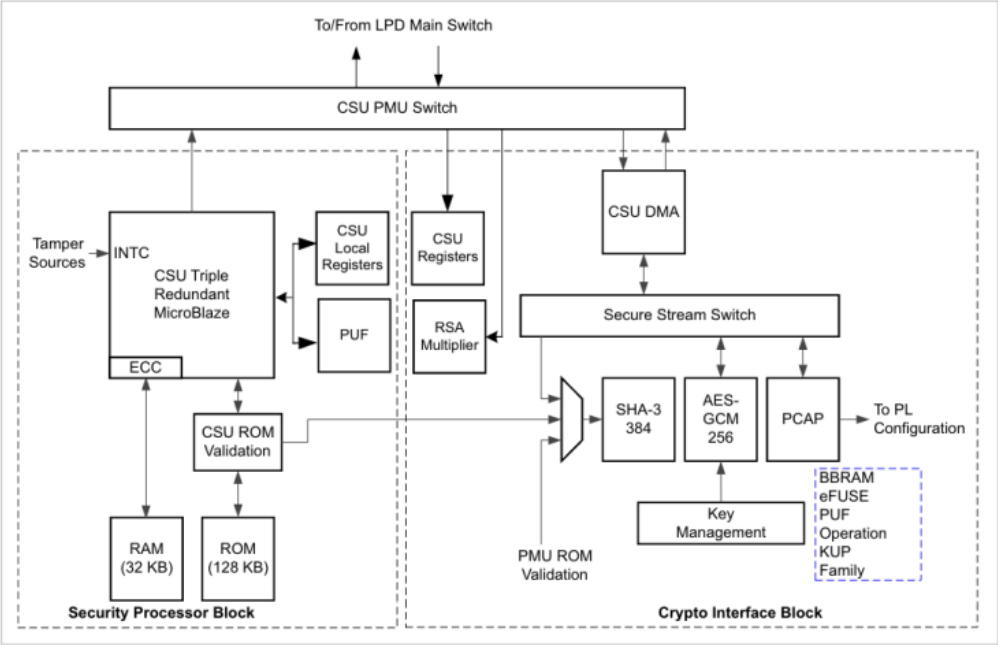

Figure 4.6. CSU architecture [13]

storage elements through zeroisation. This measure prevents unauthorised access to sensitive information, enhancing the overall security of the system. Through its execution of critical functions such as HW RoT application, Secure Boot enforcement, key validation, firmware loading, authentication, and data zeroisation, the CSU ensures that the system operates securely and reliably, safeguarding against potential threats and vulnerabilities.

4.2.2 Boot Modes and Boot Image Structure

The ZU+ MPSoC on the ZCU104 development board supports booting from various sources like QSPI, SD, USB, eMMC, NAND, or JTAG [12]. JTAG serves as an interface for testing, diagnosis, and programming of electronic devices like microcontrollers, FPGAs, and SoCs. To boot a board via JTAG, the following process is followed: to begin the process, the board is connected to the computer through a JTAG cable, establishing a direct link for communication. Once connected, the programming software on the computer identifies the device present on the board. This recognition is crucial for ensuring compatibility and accurate configuration. Subsequently, the programmable logic configuration, encapsulated within a bitstream file, is transmitted to the device via the established JTAG connection. This bitstream contains the specific instructions and settings required for the device to operate according to its intended functionalities. Following the successful loading of the bitstream, the device undergoes the boot-up sequence. During this phase, it utilises the information contained within the loaded bitstream to initialise its internal components and execute the programmed functionalities. This boot-up process is essential for transitioning the device from a powered-off state to an operational state, ready to perform its designated tasks effectively. Thus, through these sequential steps of connection, identification, bitstream loading, and boot-up, the device is prepared and configured for executing its specified functionalities.

QSPI, SD, USB, eMMC, and NAND are flash memory devices used for data storage, including bitstreams, boot programs, firmware, and other data used by embedded devices like microcontrollers, FPGA processors, and SoCs. QSPI (Quad Serial Peripheral Interface) flash memory stands out for its high-speed data transfer capabilities, making it an ideal choice for quickly loading large boot images. During the boot process, the system's bootloader retrieves the initial boot image stored in the QSPI flash, ensuring a swift startup.

SD cards, known for their flexibility and removability, offer another common boot medium. They typically store boot programs within a FAT32 file system. The bootloader accesses and reads the boot image from the SD card, subsequently transferring control to the operating system or firmware. This makes SD cards particularly useful in development and prototyping scenarios due to their ease of use and availability.

USB devices provide a versatile option for booting, leveraging the system's USB interface. The bootloader is configured to detect and read the boot image from a USB flash drive or other USB storage devices. This method is often employed for system recovery or firmware updates, highlighting the convenience of USB booting for systems that require external storage capabilities.

Embedded MultiMediaCard (eMMC) storage integrates non-volatile memory directly onto the system's Printed Circuit Board (PCB). The boot process involves reading the boot image from the eMMC storage, typically from a designated boot partition. eMMC is favoured in consumer electronics, smartphones, and tablets due to its compact form factor and reliable performance.

NAND flash memory, known for its large storage capacity, also serves as a robust boot medium. The boot process starts with a bootloader stored in a small boot block area, which then loads the main boot image. NAND flash is particularly suitable for applications requiring substantial storage space, such as SSDs and industrial devices, though it necessitates management techniques like wear levelling and bad block management.

The typical boot sequence begins with hardware initialisation following a reset. The system's CSU or bootloader selects the boot source based on predefined settings, such as jumpers, fuses, or configuration bits. The bootloader is then executed, initialising additional system components and loading further stages of the boot process if necessary. Finally, the main boot image is read into the system's memory, and control is handed over to this image, which proceeds with initialising the operating system or application. These processes collectively ensure that the system initialises correctly and securely, leveraging the distinct advantages of each storage medium to meet the specific needs of the application.

Golden Image Search Mechanism

FSBL and PMU firmware are stored in a boot image. The image (golden image) has a predefined format containing a boot header, partition header, and one or more partitions. The boot header describes boot parameters, features, and details of the boot image, being the first data the CSU searches for. The partition header describes how partitions are defined in the image. There is always a partition for the FSBL image. Other images, like PMU firmware, are optional 4.7.

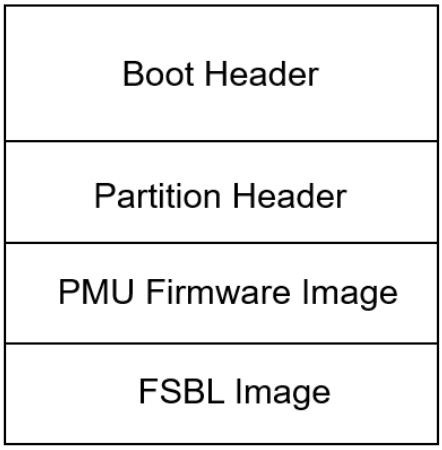

Figure 4.7. Boot Image Format [13]

The CSU looks for the boot device, described in the boot mode register, to find the boot image header and utilises the golden image search mechanism: in storage memory, multiple boot images

can be stored, with each boot image located every 32 kB. When the CSU attempts to read the boot header, it first looks for the "XLNX" identification string. If it fails to find this string at the initial address, it adjusts the read address by 32 kB and retries the process. In the case of booting from an SD card, the CSU utilises the offset value in the MULTI BOOT register. This offset value is converted into a string and then appended to the BOOT.BIN file name to create a new file name for booting. Once the identification string is found, the CSU proceeds to validate the boot header checksum. Following successful validation, it initiates the initialisation of the PS device responsible for running the FSBL, typically the Arm Cortex-A53 Application Processing Unit (APU). Finally, the CSU loads the FSBL image into the on-chip memory (OCM) for further execution.

4.2.3 Secure Boot Configuration and Image/Bitstream Confidentiality and Authentication

The Zynq UltraScale+ device allows for secure booting, where the boot image can be encrypted, authenticated, or both, depending on the desired security level 4.8.

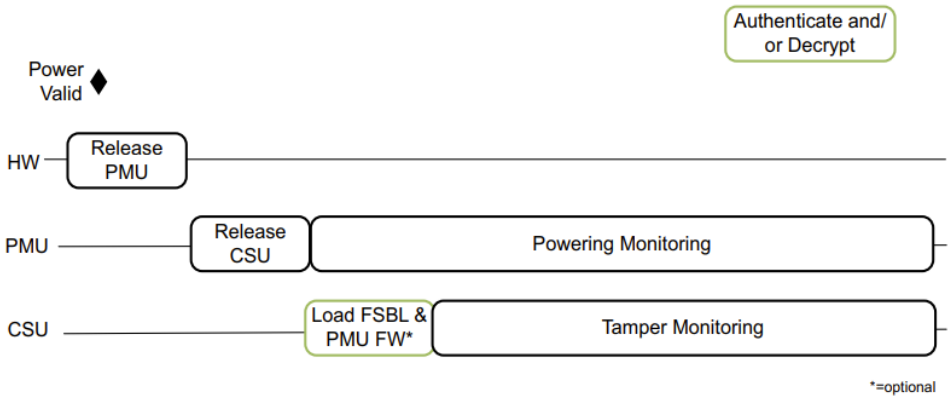

Figure 4.8. Secure Boot Timeline [14]

One of the following combinations can be implemented:

- Encrypted boot image
- Authenticated boot image
- Both encrypted and authenticated boot image for the highest security level

Symmetric image/bitstream confidentiality and authentication

Storing an encrypted image/bitstream in an external flash memory (or other medium) and decrypting it during the Secure Boot of the ZU+ device provides a high level of confidentiality. This ensures that the information contained in the image/bitstream is accessible only to those who share the same secret key (symmetric) [14]. Encryption and decryption of the image/bitstream provide confidentiality while the system is at rest and during secure boot. Xilinx strongly recommends that the externally stored image/bitstream always be encrypted. To leverage this security feature, the image/bitstream must first be encrypted using the BootGen software. BootGen is integrated into the Xilinx Software Development Kit (SDK) but is also available as a standalone tool. BootGen software uses a key provided by the user to perform encryption. If an AES key is not provided, BootGen software also offers the option to automatically generate one. The key is then loaded into the Battery-Backed RAM (BBRAM) or eFUSE on the ZU+ device through an application running on the PS. When decryption of the image/bitstream is enabled, symmetric authentication is automatically enabled.

AES-GCM combines counter mode for confidentiality with a universal hash function-based authentication mechanism (authentication tag). Therefore, AES-GCM provides not only confidentiality but also integrity and authentication simultaneously. AES-GCM (Galois/Counter Mode) is an encryption algorithm that combines two functions to provide confidentiality, integrity, and authentication simultaneously. It uses AES in counter mode (CTR) to encrypt the plaintext, ensuring confidentiality. It generates an authentication tag using a universal hash function (GHASH) over the ciphertext and additional authenticated data (AAD). This tag verifies the integrity and authenticity of the data. Thus, AES-GCM encrypts the data while also ensuring it has not been tampered with and confirming its source. This cryptographically strong authentication scheme ensures that any attempt to modify the image/bitstream (even just a single bit) prevents the device from booting. If the authentication check is successful, the device then begins normal operation, and boot commands are executed. Confirmation of passing the authentication step can be seen in the AES_STATUS register with the GCM_TAG_PASS bit set to 1.

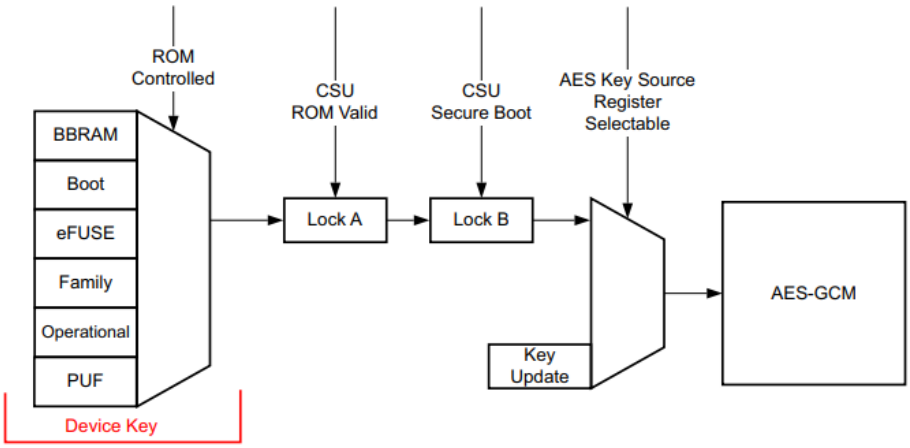

Figure 4.9. Key Management [14]

The device key is selected by the CSU ROM during boot based on the boot header. After CSU boot-up, the source of the device key cannot be changed until the next Power-On Reset (POR). A device key is available only when an AES-GCM engine is used in the secure boot or when authentication is enabled. No device key is available in the non-secure boot (i.e. when neither authentication nor encryption is enabled).

Here is a list of the various device keys:

- BBRAM: the BBRAM key is stored in plain text format in a 256-bit RAM array;
- Boot: Tte boot key register retains the decrypted key while the key is in use;
- eFUSE: it can be in plain text form, obfuscated (i.e., encrypted with the family key), or encrypted with the PUF KEK (see below);
- Family: it is a constant AES key value hard-coded into the devices. The same key is used across all devices in the ZU+ MPSoC family. This key is used only by the CSU ROM to decrypt an obfuscated key. The obfuscated key can be stored in the eFUSE or in the authenticated boot header;
- Key Update Register (KUP): after boot-up, any key can be loaded into this register via the Advanced Peripheral Bus (APB) by software running on the PS (Processing System). A 256-bit Key Update (KUP) key is stored in the eight AES key update registers;
- Operational: the Operational Key (OP) is obtained by decrypting the secure header using a plain text key obtained from other device key sources. For secure boot, this key is optional. The use of the OP key is specified in the boot header and minimizes the use of the device key, thereby limiting its exposure;

- Physical Unclonable Function Key Encryption Key (PUF KEK): it is a key-encryption key generated by the PUF. The PUF generates a unique cryptographic key for each device based on variations in voltage, temperature, etc., making it extremely difficult for an attacker to clone or reproduce the cryptographic key. Since the generated key is inherently tied to the device itself and cannot be replicated, the PUF offers a high level of security.

Asymmetric image/bitstream authentication

ZU+ devices possess the capability to authenticate the entire encrypted or unencrypted image/bitstream before sending it to the on-chip decryption engine (i.e., authenticate-then-decrypt) [14]. If the image/bitstream has been altered in any way (including just a change of a single bit), the device's asymmetric authentication function detects these modifications. Not only does it disable the decryption engine, but it also prevents the device from booting by entering a secure lockout mode. Only an authorised image/bitstream can configure the ZU+ device. This method utilises the asymmetric digital signature algorithm RSA-4096 (authentication) and, therefore, does not require the device to contain a secret to perform this authentication task. Instead, the asymmetric authentication function utilises user-defined public key information. Due to limited space, the function stores an SHA-3 hash of 384 bits of the 4096-bit public key in the device's eFUSE bits. It is up to the user to define the private and public key pairs for this operation. Several open-source and commercial products can be used to generate these key pairs (such as OpenSSL and SafeNet).

4.2.4 First-Stage Bootloader (FSBL)

The first-stage bootloader (FSBL) initiates by authenticating the remainder of the boot image (the second-stage bootloader (U-Boot) and subsequent system components). If it detects any corruption within the image, it applies an offset to the boot header search address of the CSU BootROM by modifying the CSU's MultiBoot register. Subsequently, it triggers a software reset. The next time the CSU BootROM is executed, it utilizes this offset to search for another boot header. This mechanism is termed MultiBoot [12]. To load U-Boot, the FSBL undergoes four phases.

The boot process commences with the FSBL orchestrating crucial initialisation tasks to prepare the system for operation. Initially, the FSBL undertakes hardware initialization, a pivotal step where it configures essential components such as the programmable logic (PL/FPGA), the processor, and the DDR memory. This ensures that the hardware environment is set up correctly for subsequent boot stages. Following hardware initialisation, the FSBL proceeds to initialise the boot device. By reading the boot mode register, it determines the primary boot device. Subsequently, it validates and interprets the boot image header, essential for understanding the structure and contents of the boot image. Additionally, the FSBL sets initialisation parameters for the ARM Trusted Firmware (ATF), laying the groundwork for the subsequent boot stages. A critical aspect of the boot process is partition copy validation. The FSBL rigorously validates the partition header before proceeding to copy each partition into memory. The PMU firmware partition is directly copied into PMU RAM, ensuring efficient access to critical firmware functionalities. Simultaneously, the ATF is copied into DDR memory, optimising its accessibility and performance. Moreover, the U-Boot image, serving as the secondary bootloader, is copied into DDR memory, facilitating efficient execution and control flow during subsequent boot stages. The culmination of the FSBL's efforts is the handoff phase, where control is transitioned to U-Boot, the next stage in the boot process.

Before relinquishing control, the FSBL ensures the proper initialisation of the ATF, enhancing the security and reliability of subsequent boot operations. Finally, the program counter is updated to enable U-Boot to assume control seamlessly, initiating further system initialisation and facilitating user interaction. Through these coordinated phases, the FSBL orchestrates a smooth and reliable boot sequence, laying the foundation for the system's operation. This structured approach ensures a seamless and secure boot process, facilitating the transition from the initial hardware initialisation to the execution of the subsequent boot stages.

4.2.5 ARM Trusted Firmware (ATF)

After the FSBL initialises the hardware and sets up the basic system environment, the next component in the boot sequence of the Zynq MPSoC platform is the ARM Trusted Firmware (ATF). Unlike the FSBL, which focuses on low-level hardware initialisation, the ATF serves as a higher-level software layer responsible for managing critical system settings and enforcing security policies. One of the key functions of the ATF is to act as a proxy for modifying critical system settings that are inaccessible to the Linux operating system directly. Linux, considered inherently insecure due to its monolithic kernel design, lacks the privilege to access or modify certain system configurations directly. Instead, it relies on the ATF to perform these privileged operations on its behalf. To facilitate communication between the Linux operating system and the ATF, SMC are utilised. SMC instructions enable the OS to invoke specific services provided by the ATF securely. Upon receiving an SMC request from the OS, the ATF acts as an intermediary, leveraging functionalities provided by hardware components such as the CSU and the PMU to handle these requests securely and in compliance with system security policies.

The CSU is essential to enforcing security policies and access controls within the system. It manages cryptographic keys, access permissions, and other security-related configurations, ensuring that only authorised entities can access sensitive resources and that data integrity and confidentiality are preserved. On the other hand, the PMU oversees system management tasks such as power management, system monitoring, and performance optimisation. It provides essential functionalities that enable the ATF to manage system resources efficiently while ensuring compliance with power and performance requirements. In summary, the ATF serves as a vital component in the Zynq MPSoC's security architecture, acting as a liaison between the Linux operating system and critical system configurations. Through SMC, the ATF enables Linux to access privileged functionalities securely, leveraging the capabilities of the CSU and PMU to enforce system security policies and manage system resources effectively.

This integration of the ATF and hardware-level security features exemplifies the robust security framework provided by TrustZone technology in the Zynq MPSoC platform. In the previous chapter, we delved deeper into the workings of ARM Trusted Firmware and TrustZone technology.

4.2.6 Second-Stage Bootloader (U-Boot)

U-Boot, short for Universal Bootloader, is a versatile bootloader commonly used in embedded systems like the Zynq MPSoC. Its primary responsibility is to initialise the hardware and boot the Linux operating system. However, it offers a range of additional functionalities that make it a powerful tool in the embedded developer's arsenal. One notable feature of U-Boot is its Command-Line Interface (CLI), accessible through the serial port of the Zynq MPSoC. This CLI provides a user-friendly interface for executing commands related to various tasks, such as reading and writing to flash memory, manipulating the device tree, downloading files over the network, and interfacing with hardware components. This flexibility allows developers to perform a wide array of system management and debugging tasks directly from the bootloader environment. In terms of booting Linux, U-Boot offers multiple configuration options.

By default, it boots the Linux kernel from a local storage device, such as an SD card, where the kernel image, device-tree blob (DTB), and root filesystem are stored. However, U-Boot also supports network booting, enabling the retrieval of the kernel image and DTB from a TFTP server. This network boot capability is particularly useful in scenarios where local storage may be limited or unavailable, providing an alternative method for bootstrapping the system. Regardless of the chosen boot method, U-Boot ensures that the kernel image and DTB are loaded into memory before passing control to the Linux kernel. Additionally, it facilitates the transmission of boot arguments to the kernel, allowing for customisation and configuration of the boot process according to specific requirements. Overall, U-Boot serves as a crucial component in the boot sequence of Zynq MPSoC-based systems, offering flexibility, functionality, and reliability in the initialisation and booting of Linux-based operating systems. Its extensive feature set and customisable nature make it an indispensable tool for embedded developers working with Zynq platforms.

4.2.7 Kernel Boot

After the kernel assumes control during the boot process of the Zynq MPSoC platform, it begins a series of crucial operations to prepare the system for operation. Firstly, the kernel relies on the device tree to identify and configure the hardware components present in the system. This data structure provides essential information about devices such as memory regions, peripheral controllers, and interrupt controllers, enabling the kernel to initialise and interact with the on-chip hardware effectively. Next, one of the kernel's primary tasks is to mount the root filesystem. This filesystem contains the essential files and directories required for the functioning of the operating system. Whether the root filesystem is stored on an SD card, eMMC, or accessed via NFS, mounting it grants the kernel access to user-space programs, configuration files, and system libraries necessary for system operation.

Following the root filesystem mounting, the kernel searches for the initialisation process (init) with process ID 1 (PID1). 'Init' is the first user-space process started by the kernel and serves as the parent process for all other user-space processes. It manages system startup, initialising system services, and handling system shutdown procedures. 'Init' sets up the system environment, launches essential system daemons, and coordinates the startup sequence. Once the 'Init' process is located and executed, it hands over control to the initialisation system, often 'systemd' in modern Linux distributions. 'Systemd' is a comprehensive system and service manager that orchestrates the boot process, manages system services, and coordinates the execution of user-space processes. It replaces the traditional 'SysV' init system and offers advanced features such as parallel service startup, dependency management, and logging. 'Systemd' initialises the system environment, starts all configured services, and sets up the user session, ensuring a smooth transition to the fully operational state.

In short, the boot process of the Zynq MPSoC platform involves a coordinated sequence of operations after the kernel assumes control. These include device tree utilisation for hardware configuration, root filesystem mounting for access to essential system files, initialization process execution for system setup, and handover to the initialization system (e.g., systemd) for comprehensive system management and service startup.

4.2.8 Pre-boot Failure and Possible Fallbacks

In the event of pre-boot failures involving the PMU and CSU BootROMs, two fallback solutions can be implemented to ensure system reliability and facilitate troubleshooting. These solutions leverage the golden image search mechanism and CSU error code registers to address and manage boot failures effectively.

Utilising the CSU BootROM's Golden Image Search Mechanism

The CSU BootROM includes a golden image search mechanism designed to enhance the robustness of the boot process. This mechanism is particularly useful when the default boot image is corrupted or fails to load properly. Here's how it works:

Multiple Boot Images: for this fallback solution to function, multiple boot images must be pre-stored on the boot device (e.g., QSPI flash, SD card, eMMC).

Search Process: upon detecting a failure with the initial boot image, the CSU BootROM automatically initiates a search for an alternative boot image. This search is conducted according to a predefined order or list of possible boot image locations.

Loading Alternative Image: once a viable alternative boot image is found, the BootROM attempts to load it, thus providing a secondary path to successful system initialisation.

This approach significantly increases the resilience of the system by ensuring that a single point of failure (the primary boot image) does not prevent the device from booting. It is particularly beneficial in environments where uptime and reliability are critical.

Implementing a Linux Service for Monitoring CSU Error Code Registers

A complementary solution involves the implementation of a Linux-based service that monitors the CSU error code registers. This solution is aimed at post-boot diagnostics and user notification:

Error Code Register Monitoring: the CSU maintains a set of error code registers that log specific error conditions encountered during the boot process. These registers provide valuable insights into what went wrong if a boot attempt fails.

Linux Service Development: a Linux service can be developed to read these error code registers once the system has successfully booted. This service would periodically check the registers and log or display error information.

User Notification: the service can be configured to alert the user or system administrator when a previous boot attempt was unsuccessful. Notifications can be delivered through various means, such as system logs, alerts, or user interface messages.

This service does not currently exist and would need to be created as part of the system's software suite. Its development would involve writing software that interacts with the CSU registers, parses the error codes, and integrates with the system's notification infrastructure.

Benefits and Considerations

The implementation of both fallback solutions enhances the robustness of the boot process by providing complementary layers of resilience. The golden image search mechanism furnishes immediate redundancy throughout the boot sequence, offering a safety net in case of glitches with the default boot image. Meanwhile, the Linux service steps in post-boot, delivering diagnostics and feedback to users and administrators.

This combined approach ensures that the system can swiftly recover from boot failures while keeping stakeholders informed about any issues that require attention. Ultimately, the adoption of these fallback mechanisms bolsters the system's reliability and resilience, enabling proactive troubleshooting and maintenance. However, their implementation demands careful planning and development efforts. The golden image mechanism relies on the presence of multiple boot images and a meticulously crafted search strategy, while the Linux service necessitates custom software development and seamless integration with existing system components. These fallback mechanisms represent indispensable strategies for managing and mitigating pre-boot failures in embedded systems. They facilitate both immediate recovery and informed diagnostics, contributing to overall system stability and performance.

Chapter 5

Design of Post-Quantum Secure and Measured Boot on Zynq UltraScale+ MPSoC

5.1 Bootflow Design

This boot flow diagram shows the detailed stages of the boot process for a Zynq UltraScale+ MPSoC. The FSBL is configured to load critical boot images into the device in this sample design 5.1.

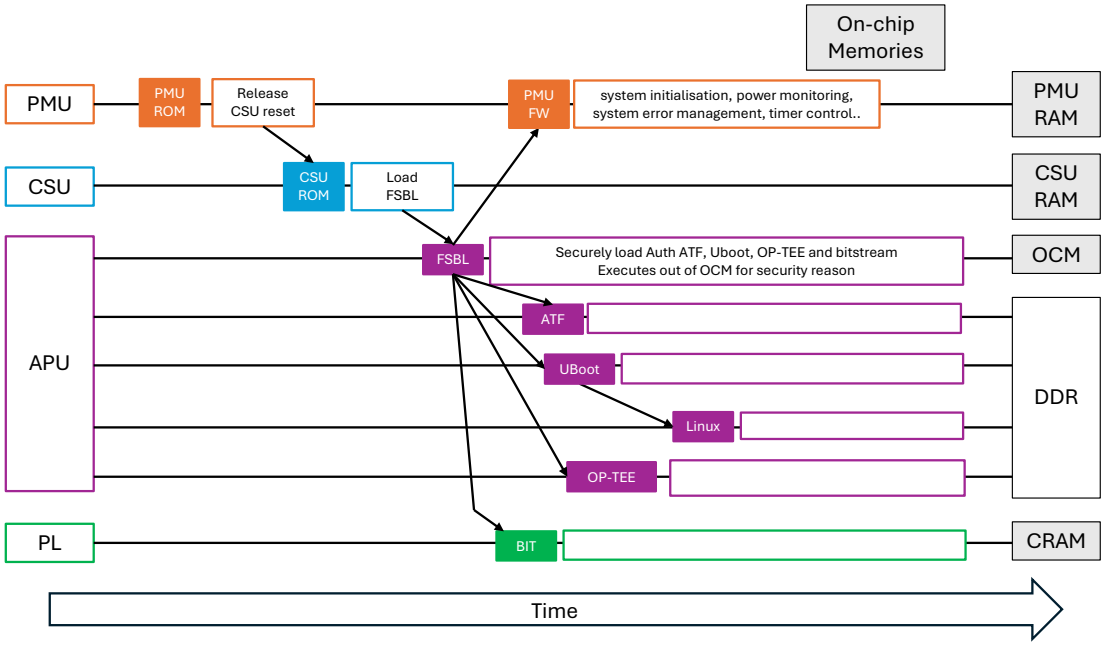

Figure 5.1. Bootflow design - ZynqUltrascale+ MPSoC [15]

This process involves several key steps, including authentication, loading into memory, and preparing each component for execution. Here's a step-by-step breakdown of what happens, summarising in a less detailed way what was described in the previous sections.

The boot process begins with the PMU ROM, which contains the essential boot code for the PMU. The PMU's initial role is to ensure that the system's secure initialisation can begin

smoothly. It performs an important check on the CSU boot ROM code, verifying its integrity to confirm that the CSU can be trusted. Once this check is successful, the PMU releases the reset signal on the CSU, effectively "waking it up" and enabling it to proceed with the next stages of booting.

With the CSU now active, the CSU ROM takes over. The CSU's main job is to handle the secure handoff from the PMU to the higher-level boot components. CSU ROM loads the FSBL into memory, allowing it to take control of the boot process.

After loading, the FSBL becomes the central component of the boot process. The FSBL is responsible for securely loading and authenticating the subsequent system partitions in a precise order.

In this design, the bitstream should be the first component loaded to configure the Programmable Logic (PL) of the FPGA (Field-Programmable Gate Array). The bitstream is a file containing the configuration data that defines how the PL is set up, including which hardware resources to activate and how they should be interconnected. Typically, bitstreams are generated from hardware description language (HDL) designs, such as VHDL or Verilog, after synthesis and implementation, and they are loaded into the FPGA to configure its behaviour at power-up or during operation. After the bitstream is loaded, the PMUFW is initialised to manage system resources. However, in this implementation, the FSBL does not load any partition containing an image for the bitstream.

FSBL loads the PMUFW, which will handle important tasks such as system initialisation, power management, error monitoring, and control over system timers. The PMUFW thus plays a vital role in maintaining the system's operational stability.

Next, the FSBL loads the ATF, which is placed out of On-Chip Memory (OCM) for security reasons. The ATF is a key component of the SeW and it is responsible for setting up the security environment by initialising the SeW environment and handling any future secure calls that may be required. Loading the ATF early ensures that the SeW is ready for any security-sensitive operations as soon as they're needed.

Following the ATF, the FSBL moves on to loading U-Boot into DDR memory. U-Boot acts as the second-stage bootloader and provides a versatile environment for additional configuration and setup tasks. From U-Boot, the system can load further operating systems or perform various diagnostic or network-related tasks. By loading U-Boot, the FSBL enables the system to bridge from initial secure boot processes to more complex, customisable configurations.

The FSBL also loads OP-TEE, which establishes a TEE in a secure memory section. OP-TEE is designed to provide a secure environment isolated from the main operating system, allowing for the execution of trusted applications. This TEE is crucial for running sensitive operations, such as cryptographic functions, in a way that's safe from potential interference by non-secure applications running on the device.

U-boot loads Linux into DDR. Linux serves as the main operating system and is executed in the NoW on the APU. Once loaded, Linux takes over as the primary runtime environment, managing hardware resources, running applications, and handling standard operating system functions. By the time Linux is launched, all prior security steps have ensured that it is running in a trusted, authenticated environment, isolated from the SeW.

5.1.1 OP-TEE: Overview and Purpose

OP-TEE is a TEE specifically designed to function alongside a non-secure Linux kernel running on Arm Cortex-A cores, leveraging the TrustZone technology for hardware isolation. It implements the TEE Internal Core API v1.3.1, which is the interface exposed to Trusted Applications (TAs), and the TEE Client API v1.0, which describes how to communicate with the TEE. The non-secure operating system is referred to as the REE in TEE specifications. This REE is commonly a flavor of Linux, such as a GNU/Linux distribution, or the Android Open Source Project (AOSP). OP-TEE is primarily built upon the Arm TrustZone technology as its core hardware isolation mechanism. However, it is also structured to support compatibility with other isolation

technologies suitable for TEE objectives, including operation as a virtual machine or on dedicated CPU cores.

The main design goals for OP-TEE include:

1. **Isolation**: the TEE ensures strong isolation from the non-secure OS, safeguarding the loaded TAs from one another through underlying hardware support. This isolation helps prevent unauthorised access to sensitive data and secure operations.
2. **Small Footprint**: OP-TEE is designed to have a minimal resource requirement, enabling it to reside within the limited on-chip memory available on Arm-based systems. This compact design allows for efficient resource usage without sacrificing performance.
3. **Portability**: OP-TEE aims to be easily integrated into various architectures and hardware configurations. It supports different setups, including multiple client operating systems and the ability to run alongside multiple TEEs, enhancing its versatility for different applications and environments.

By focusing on these goals, OP-TEE provides a robust and flexible platform for secure application execution, catering to the growing demand for trusted environments in a variety of computing scenarios [16].

5.2 TPM and Measured Boot Design

A Trusted Platform Module (TPM) is a specialised hardware component designed to enhance the security of computing platforms by providing a range of functions focused on trust and integrity. Among its primary roles, the TPM securely stores cryptographic keys, certificates, and platform measurements, which are crucial for establishing a trusted computing environment. The TPM ensures that sensitive data is protected from unauthorised access and modification by relying on secure hardware mechanisms. One of the essential functions of a TPM is to store and quote platform measurements, which help to confirm that the platform remains trustworthy. During the boot process, critical system components, such as the UEFI/BIOS, operating system kernel, boot loader, and Secure Boot police, are measured before execution. These measurements are typically represented as cryptographic hashes, which provide a unique fingerprint of each component's state. The TPM uses its Platform Configuration Registers (PCRs) to store these measurements securely. When a measurement is made, it is combined with the current value of its corresponding PCR through a process known as "extension". This process involves concatenating the previous PCR value with the new measurement and then hashing the result. This creates a chain of trust, where each new measurement builds upon the previous one. Consequently, the value of a PCR reflects the integrity of all components measured within it, providing a reliable indicator of the platform's overall security posture. In the following figure, we provide a specific example of this process as implemented in our design, demonstrating how each component's integrity contributes to the platform's trustworthiness 5.2.

The Measured Boot concept leverages the TPM's capabilities to ensure that a platform boots in a trusted state. In a measured boot process, each component executed during the boot sequence is measured, and these measurements are recorded in the PCRs. If any component has been altered or compromised, the corresponding measurement will differ from the expected value, triggering alerts or preventing the system from booting entirely. Once the boot process is complete, the TPM can generate a signed quote based on the measurements stored in its PCRs. This signed quote proves the platform's integrity and can be shared with a remote attestation verifier, such as Intel Trust Authority. The verifier can check the TPM quote against a predefined policy that specifies the expected measurements for a trusted boot. If the measurements match, it indicates that the platform has booted securely and is in a trustworthy state.

Overall, the combination of TPM and Measured Boot establishes a robust security framework that helps to protect computing environments against unauthorised access and tampering. By ensuring that only trusted components are executed, organisations can significantly enhance the integrity of their systems and safeguard sensitive information [17].

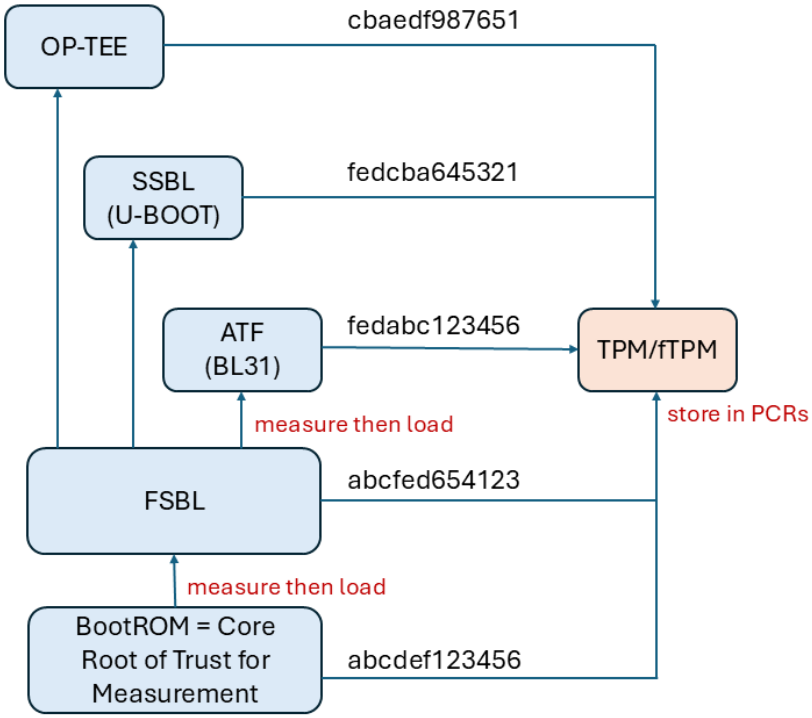

Figure 5.2. Measured Boot - Store measure in PCRs

5.2.1 TPM Event Logs

TPMs are critical for maintaining a system's integrity by using PCRs to store hash measurements of various system events. These events can happen during the boot process, such as those recorded by UEFI, or at runtime, as in the case of IMA (Integrity Measurement Architecture) events. For boot-time attestation, TPMs rely on TCG Event Logs, specifically UEFI event logs, which can be validated or replayed against the PCRs. Meanwhile, IMA logs measure runtime integrity, extending the PCRs accordingly. Each of the TPM's 24 PCRs receives multiple extensions with measurement events. TPMs typically allocate PCRs for specific purposes. For instance, PCRs 0-7 are often designated for BIOS/UEFI and boot measurements, while PCRs 8-15 are generally used by the operating system. This predefined structure supports different phases of the system's lifecycle, from pre-boot to OS runtime, providing a way to isolate measurements according to system components. The pcr_extend process, which concatenates the existing PCR value with a new measurement hash and rehashes the combination, ensures that PCRs act as cryptographic evidence of all measurements accumulated within them. Since PCR values are updated through hashing, it's impossible to reverse the final PCR hash to extract individual measurement events.

While PCRs provide strong indicators of system integrity, they also come with challenges. The final PCR values obscure details about individual events, making it challenging to derive a human-readable record of each step leading to the PCR's state. This can complicate the development of precise attestation policies or identify specific causes of mismatches between expected and actual PCR values. For example, PCR 7 includes measurements of UEFI configuration settings, such as Secure Boot status, boot manager code, configuration settings, and option ROM details. A policy could verify the integrity of PCR 7 to ensure these components are unaltered. However, if a mismatch occurs, the PCR alone cannot indicate which specific component, such as Secure Boot, was altered. Moreover, if a policy only needs to validate the Secure Boot status, it cannot do so without validating the entire PCR 7. Some PCRs also capture what are known as "fragile" measurements, prone to variability across boots or between identical systems. This variability can complicate attestation since it is challenging to isolate a stable component for verification based on the PCR value alone.

To address these limitations, attestation can incorporate TPM event logs, which provide a detailed record of the events measured for each PCR. By replaying an event log and confirming that it matches the PCR value, we can verify the log's integrity. Once validated, the event log enables us to assess specific system elements individually. For example, instead of requiring a strict match for PCR 7, an attestation policy can specify that Secure Boot must be enabled, and verified by the event log replay. If the log replay aligns with the PCR, individual entries within the event log can be compared against the policy to confirm Secure Boot status. If the replay fails, an error will indicate a mismatch, directing attention to the source of the discrepancy [17].

Using TPM event logs in place of raw PCR values presents several key benefits:

1. **Targeted Verification**: Event logs enable verification of specific system elements through policies. This allows precise attestation of critical features, such as confirming Secure Boot status, while disregarding unrelated measurements.
2. **Improved Troubleshooting**: when an attestation fails, event logs provide specific details on which system component changed, streamlining troubleshooting and accelerating problem resolution.
3. **Readable and Accessible Information**: compared to PCR values, event logs present measurements in a more human-readable format, making it easier to interpret the integrity status of the system.
4. **Addressing PCR Fragility**: Event logs help mitigate the "fragility" of PCRs by allowing selective verification of only relevant events. By focusing on specific event data instead of an exact PCR value, attestation becomes more reliable and less susceptible to the issues of variable PCR measurements.

5.2.2 fTPM: A Software-Based Approach

A Firmware Trusted Platform Module (fTPM) is a software-based implementation of TPM that runs on the system's main processor rather than as a discrete hardware component. It is designed to provide similar security functions as a traditional TPM, but with the advantage of being more flexible and easier to integrate into existing systems. The fTPM can operate within TEE such as OP-TEE, allowing it to leverage the secure resources provided by the TEE while running as a trusted application.

Key Features of fTPM are:

1. **Integration with Trusted Execution Environments**: the fTPM operates on top of TEEs like OP-TEE, which utilises the Arm TrustZone technology for hardware isolation. This allows the fTPM to maintain a secure environment for sensitive operations while still being integrated with the system's software stack.
2. **Secure Key Management**: just like a traditional TPM, the fTPM is responsible for generating, storing, and managing cryptographic keys. It can perform secure operations such as signing, encryption, and decryption using these keys, ensuring that sensitive information is protected against unauthorised access.
3. **Platform Integrity Measurements**: the fTPM can facilitate the same measurement processes as a traditional TPM, recording the integrity of critical system components during boot and runtime. It supports the same concepts of Measured Boot, where components are hashed and extended into PCRs, ensuring that the platform's state can be trusted.
4. **Remote Attestation**: the fTPM can generate signed quotes of the PCR values, which can be used for remote attestation. This allows external verifiers to check whether the platform is in a trustworthy state based on the measurements stored in the fTPM. This process is essential for establishing trust in remote interactions and can be particularly useful in cloud environments or IoT devices.

5. **Flexibility and Cost-Effectiveness**: since the fTPM operates in firmware, it eliminates the need for a dedicated hardware TPM chip. This can reduce costs and complexity in system design, particularly for applications where space and resources are limited.

The fTPM, as a trusted application running on OP-TEE, benefits from the secure execution environment provided by the TEE. OP-TEE ensures that the fTPM operates in a secure context, protecting it from interference by non-secure applications and providing access to the secure hardware resources available on the platform. This integration allows for a more robust security architecture, as OP-TEE can handle secure boot processes, cryptographic operations, and isolation of the fTPM from REE. Given this, the fTPM serves as a flexible, firmware-based alternative to traditional TPMs, providing essential security functionalities within a TEE like OP-TEE. Its ability to manage cryptographic keys, perform integrity measurements, and facilitate remote attestation while leveraging the secure environment of the TEE makes it an attractive solution for modern security requirements in various computing platforms.

5.3 Secure Boot Design

The Zynq UltraScale+ MPSoC offers two distinct modes for secure booting: Hardware Root of Trust (HRoT) and Encrypt-Only. These modes allow the system to verify and protect boot and configuration files against unauthorised modification or access.

In HRoT mode (the only one used in this design), the device applies asymmetric cryptography for authentication, optionally paired with encryption, ensuring the confidentiality, integrity, and authenticity of all files involved in the boot process. This approach leverages RSA-4096 for robust authentication, working with SHA-3/384 for hashing, providing a solid foundation for secure initialisation.

In contrast, the Encrypt-Only mode relies on encryption without using asymmetric methods. In this setup, all configuration data must be encrypted and authenticated using AES-GCM (Galois/Counter Mode), but it omits the asymmetric signature checks. Thus, this mode protects the content's confidentiality but does not validate its authenticity to the same extent as the HRoT.

Central to both secure boot processes is the CSU, which manages secure boot operations and enforces the selected security protocols. It also upholds the device's security state by strictly controlling transitions between secure and non-secure modes. Importantly, the CSU prevents any changes from a secure to an insecure state (and vice versa) without a full Power-On Reset (POR), ensuring that once a secure boot starts, it cannot be bypassed or tampered with mid-process. When the FSBL and, if applicable, the PMUFW have securely loaded, the CSU zeroises or clears any sensitive data from the cryptographic modules. It then releases the reset of the chosen processing unit, either the Application Processing Unit (APU) or the Real-Time Processing Unit (RPU), allowing it to start execution.

The HRoT of Trust process in the ZU+ design relies on two critical key pairs, each serving a unique role: the Primary Public Key (PPK) and the Secondary Public Key (SPK). Two PPKs can be configured for each system, and these keys are safeguarded in a hybrid storage mechanism. The full PPK resides in external memory, while a SHA-3/384 hash of the PPK is permanently stored in the device's eFUSE memory, adding a layer of immutability. During boot, the CSU validates the integrity of the PPK by comparing the external PPK with the hash in the eFUSE, ensuring that only authorised keys are trusted. Additionally, PPKs are revocable, allowing for security updates if key rotation is necessary. The PPK's primary role is to authenticate the SPK, which, in turn, authorises all other components.

The SPK system is more flexible and supports varying configurations depending on the revocation method selected. For the FSBL, up to 32 SPKs can be used, while other boot components may use up to 256 SPKs if the enhanced revocation method is enabled. Since the SPK is included within the authenticated boot image, it is protected from unauthorised modification. Like the PPK, SPKs are revocable, allowing for selective updates to system authentication keys as part of a dynamic security policy. The authenticated SPK subsequently verifies all other partitions, creating a secure, multi-stage chain of trust throughout the boot process. These secure boot

mechanisms in the ZU+ MPSoC establish a resilient environment that prevents unauthorised code execution and protects system integrity from the very first boot stages [13].

5.3.1 Authentication Certificate

Authentication Certificate (AC) play a critical role in the secure boot process, enabling strong validation of each partition's integrity before loading or execution. Each AC is essentially a structured data block containing all the necessary components to verify the authenticity of a partition, including public keys and signatures. These certificates allow the system's BootROM or FSBL to validate each partition independently, establishing a secure chain of trust throughout the boot sequence. Each AC contains an Authentication Header that specifies essential parameters, such as key sizes and the cryptographic algorithms used for signing, along with the public keys associated with each stage of the verification process. These details guide BootROM and FSBL in verifying the integrity and authenticity of the partition by detailing the cryptographic framework required for each partition 5.3.

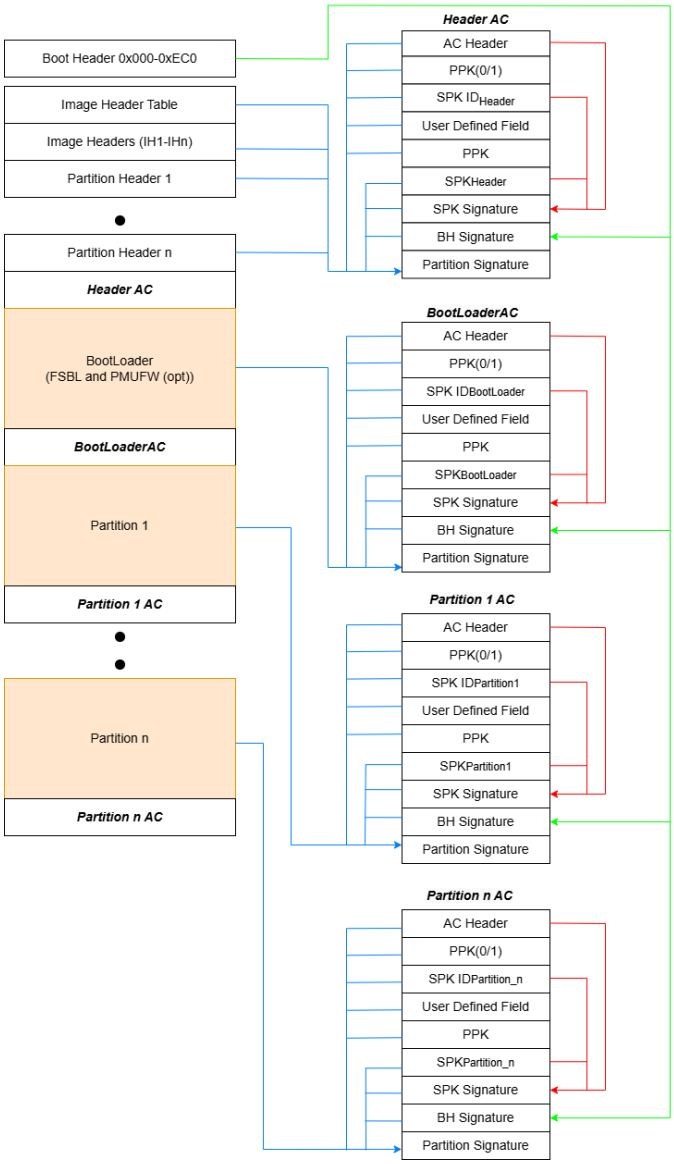

Figure 5.3. Secure Boot Image [13]

In the ZU+ architecture, RSA-4096 is the chosen asymmetric algorithm for secure boot authentication. Both the PPK and the SPK have a length of 4096 bits, a configuration that offers a high level of security due to the robustness of the RSA-4096 bit key size. By adhering to this standard, ZU+ guarantees strong, verifiable signatures on all authenticated partitions, preventing tampering or unauthorized code from entering the secure boot process. When authentication is enabled for a partition, an AC is attached to it. This certificate holds both the signing keys and the necessary signatures to enable the system to check whether the partition content has remained unmodified. The certificate also serves as a binding layer, linking the partition with verified cryptographic credentials that are checked during boot. If any mismatch or unauthorised modification is detected in the signatures, the boot process halts, preventing potential threats from compromising the device.

Additionally, the Header Table, a critical structure that organises metadata for each partition, must also be authenticated when secure boot is enabled. This is achieved through a separate Header Table Authentication Certificate, appended to the end of the header content. By authenticating the Header Table, ZU+ ensures that the layout and details of each partition, as well as their load addresses and sizes, have not been tampered with before execution. Authentication Certificates underpin the secure boot process by enabling a trusted validation chain that authenticates every critical part of the boot image before execution. This hierarchical system of certificates and signatures builds a secure boot environment where each component in the boot sequence is verified by cryptographic proof. In essence, each partition's certificate, and especially the Header Table certificate, establishes a foundation of integrity that protects the boot process against intrusion, malware injection, and unauthorized changes. The integration of RSA-4096 with structured AC in the ZU+ MPSoC thus provides a robust solution for secure boot, making it highly resistant to tampering and ensuring the device's trustworthiness at every boot stage [18].

5.3.2 Secure Boot Authentication: Signing and Verification with Primary and Secondary Keys

The signing and verification processes in the secure boot for ZU+ MPSoC are essential for ensuring that every partition loaded into the device has not been tampered with and is verified as authentic. The process involves generating cryptographic signatures at a secure facility, which are then used by the boot components on the device to verify the integrity of the partitions during boot.

Signing Process

1. **Preparation of Public Keys (PPK and SPK)**: in the signing process, the PPK and SPK are first prepared. These keys are stored in the AC, which accompanies each partition. The PPK serves as the primary anchor in the chain of trust, while the SPK is used to authenticate specific partitions.

2. **Signing the SPK**: the SPK is signed using the PSK, resulting in the SPK signature. This signature verifies that the SPK has been generated by an authorised source. The SPK signature is also stored in the AC, further embedding the SPK's authenticity within the secure boot chain.

3. **Partition Signing**: each partition intended for secure boot is signed using the SSK. This signature becomes the Partition Signature and is also included in the AC. With this step, each partition is bound to a unique cryptographic signature, allowing it to be verified during the boot process on the device.

4. **Appending the Authentication Certificate**: once the keys and partition signatures are prepared, the AC is either appended or prepended to each partition. This configuration depends on the device setup and ensures that the boot components on the device can locate the certificate when verifying partitions.

5. **Storing the PPK Hash in eFUSE**: the final step of the signing process is to store a cryptographic hash of the PPK in an eFUSE on the device. The eFUSE storage is permanent

and highly secure, ensuring that the hash cannot be altered or removed. This hash serves as the ultimate reference for verifying the PPK during the device boot.

A detailed diagram illustrating the signing process is provided in the following figure 5.4.

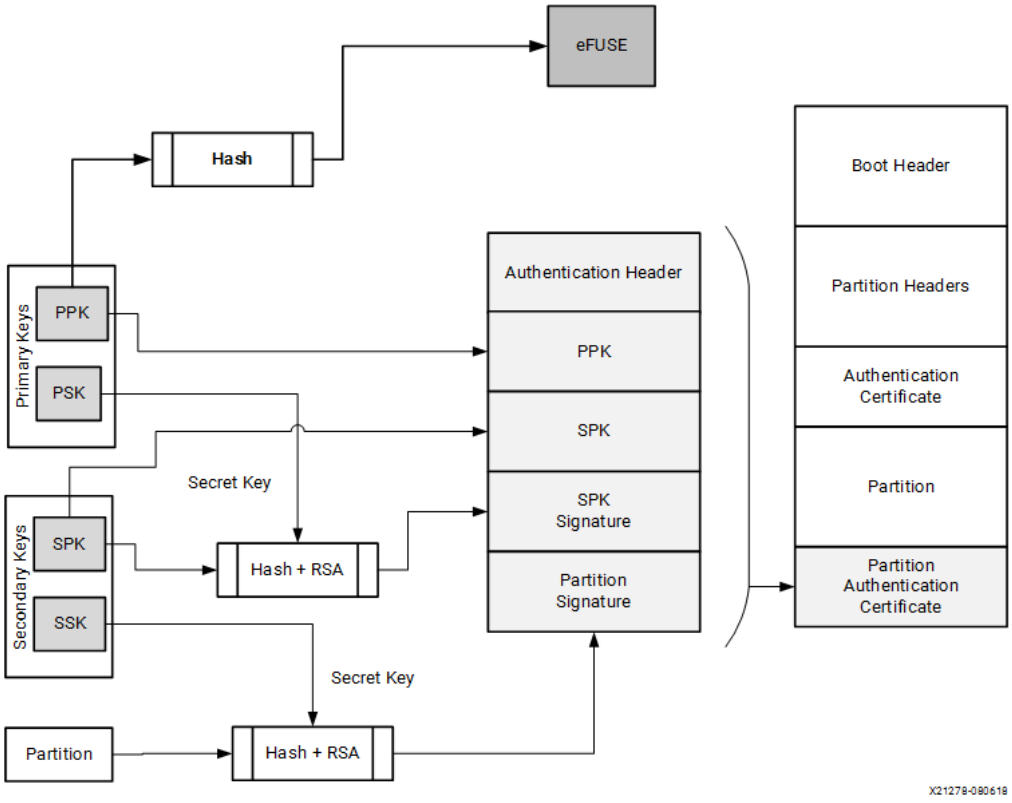

Figure 5.4. Signing process [18]

Verification Process

During the device boot, the BootROM and later stages such as the FSBL or U-Boot perform a series of verification steps to authenticate the partitions before loading them.

1. **Verification of the Primary Public Key (PPK)**: the BootROM retrieves the PPK from the AC attached to the boot image. It then generates a hash of this PPK, which it compares with the PPK hash stored in the device's eFUSE. If the hashes match, the BootROM can trust the PPK; otherwise, the boot fails, as this indicates a potential security breach. Trusting the PPK is essential as it serves as the root of the trust chain for all subsequent keys and partitions.

2. **Verification of the Secondary Public Key (SPK)**: with a trusted PPK, the BootROM proceeds to verify the SPK. It retrieves the SPK from the AC and generates a hash of this SPK. Then, using the PPK, the device verifies the SPK signature stored in the AC. This check ensures the SPK's integrity and authenticity, confirming that it was signed by a trusted source. If the hashes match, the SPK is trusted; otherwise, the secure boot fails. Since the SPK is used to verify the actual partitions, this step is crucial for the overall boot security.

3. **Verification of Partitions**: after validating the SPK, the device can begin verifying individual partitions. Each partition is read from the boot image, and its hash is generated. The BootROM or FSBL then uses the SPK to verify the Partition Signature found in the

AC. By comparing the hash generated from the partition with the hash verified from the certificate, the system can determine if the partition is genuine. If the hashes match, the partition is considered trustworthy and can be loaded. If they do not match, the device will halt the boot process, preventing the potentially unsafe partition from loading.

A detailed diagram illustrating the verification process is provided in the following figure 5.5.

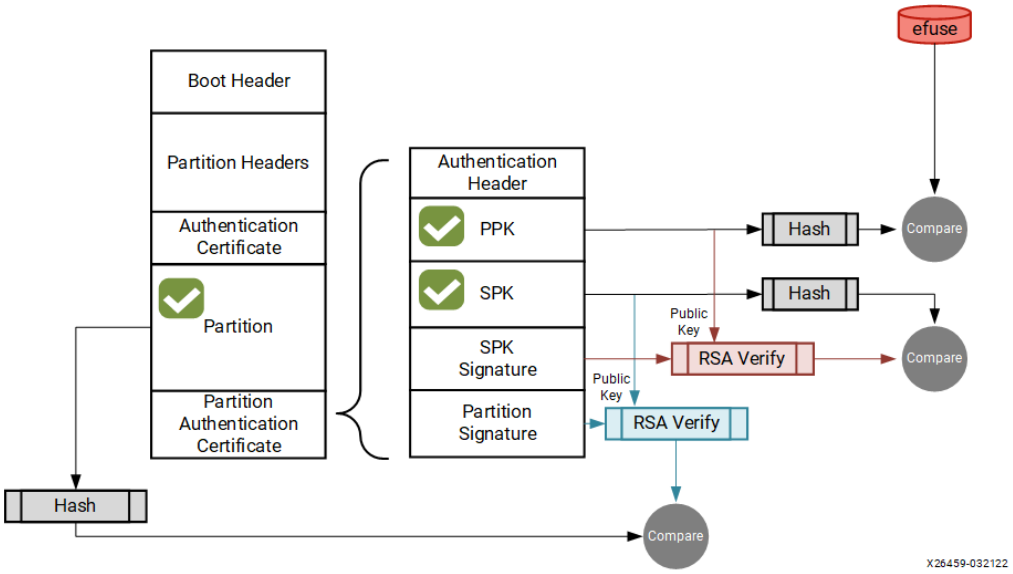

Figure 5.5. Verification process [18]

The secure boot process relies on the structured relationship between these keys and signatures. By layering the PPK, SPK, and Partition Signatures in this order, the device establishes a cryptographic chain of trust that effectively secures the boot process. Each layer of verification depends on the previous one, and any attempt to tamper with a partition, key, or certificate would cause the boot process to fail, preserving the integrity of the device. Through these steps, ZU+ MPSoC ensures that only verified, untampered software can execute, establishing a robust security foundation from the initial boot stages onward.

Enhancing Authentication Security with Post-Quantum Algorithms

The CSU we are using includes a 4096-bit RSA core dedicated to signing and verifying operations, ensuring a relatively robust level of authentication under classical cryptographic assumptions. However, the advent of quantum computing poses a significant challenge to RSA's security, particularly due to Shor's algorithm, which can efficiently factor large integers and render RSA vulnerable. To mitigate this potential risk, it would be prudent to adopt a post-quantum algorithm such as LMS (Leighton-Micali Signature scheme), which is a stateful, hash-based approach. LMS provides strong security guarantees that remain resilient even in the presence of quantum computing capabilities, thereby enhancing the overall robustness of the authentication process.

Chapter 6

Secure and Measured Boot implementation on Zynq UltraScale+ MPSoC

6.1 Post-Quantum Measured Boot Implementation

When embedded systems are network-connected, the implementation of Measured Boot is highly recommended. Network connectivity allows these systems to perform firmware and application updates, which is critical for maintaining functionality and security over time. However, network access also broadens the system's attack surface, exposing it to potential cyber threats. Hackers and malicious actors may exploit this increased exposure to compromise system integrity, necessitating robust security measures during both boot and runtime. One effective solution for mitigating network-related vulnerabilities in embedded systems is remote attestation. Remote attestation verifies the integrity of the system during boot-up and can continue monitoring during operation. This approach leverages both secure boot and measured boot, as these two mechanisms work in tandem to safeguard the system. In Secure Boot, files and partitions are authenticated, ensuring only trusted code is executed. Measured boot, on the other hand, records integrity measurements, typically SHA3 digests, of files and system states, which provides a record of the system's state over time.

In a reference design, the implementation of Measured Boot is often enhanced by a TPM. The TPM adds a layer of hardware or firmware-based security by calculating and storing cryptographic measurements in its PCRs. For each image or component measured, the system calculates a SHA3 digest and sends this value as an event to the TPM, which uses SHA2 to log the event into a PCR. Each PCR holds a history of the events that have modified it, creating a continuous record of the system's configuration over time. This structure is essential for creating an auditable chain of trust. During Measured Boot, the TPM uses cryptographic signatures, often RSA-based, to validate the integrity of its PCR values when requested. This allows a remote attestation server to request the PCR digests from a device and verify them to ensure they match the expected values. The server can then perform runtime integrity checks on clients by reviewing the sequence of recorded measurements. These checks provide real-time insights into whether the device has been tampered with or is running in a secure state. Based on this attestation, the server can apply specific policies, such as limiting network access or triggering an alert if discrepancies are detected.

Importantly, neither secure boot nor measured boot relies on the device's Programmable Logic (PL) resources, which means that using these mechanisms does not impact the unit cost of the Zynq UltraScale+ device or other similar platforms. Thus, implementing these security protocols is both cost-effective and beneficial, enhancing system security without requiring additional hardware resources [19].

Note: in this implementation, we do not utilise a physical TPM or fTPM, nor do we have any PL partitions to load. Instead, only non-PL partitions, specifically the PMU firmware (pmufw), BL31 (the ARM Trusted Firmware runtime), U-Boot, and OP-TEE, are loaded. The integrity measurements for these components are taken and stored in an Event Log. This approach ensures that the measurements are readily available and can be accessed if, in the future, a TPM or fTPM is added to the system to perform attestation. By logging these measurements in a structured Event Log, we maintain an audit trail of system integrity similar to what would be stored in a TPM's PCRs. This Event Log contains SHA3-384 digests of each of these critical non-PL partitions during the boot process, capturing the system's state at every stage. Even without a TPM, the system can retain a history of these boot measurements, which can later be leveraged for security verification or remote attestation if a TPM is introduced. This design choice allows for flexibility and forward compatibility with enhanced security modules in future implementations.

6.1.1 Measurements Performed by FSBL

The CSU ROM measurement relies on a SHA3-384 digest generated by the CSU during the initial boot stages. This digest is stored in specific registers, and, upon boot, the FSBL records this SHA3-384 digest in the Event Log. This initial measurement provides a record of the system state from the start of boot, allowing it to be verifiable in future attestation processes.

SHA3-384 is particularly suited for secure boot processes as it is resistant to attacks from both classical and quantum computers. Unlike earlier cryptographic hash functions, SHA3-384 is based on the Keccak sponge construction, which offers strong collision resistance and robustness. Importantly, SHA3-384 is quantum-safe because it is resistant to Grover's algorithm, a quantum algorithm that could otherwise provide a quadratic speedup in finding hash collisions or preimages. Despite this speedup, Grover's algorithm would still require approximately $2^(192)$ operations to break SHA3-384, maintaining a security level equivalent to 192 classical bits. This makes it a future-proof choice in the context of post-quantum cryptography, ensuring that the CSU ROM measurements remain secure against advances in quantum computing.

For the FSBL itself, the measurement process is conducted after the FSBL begins executing. Only an authenticated FSBL is permitted to load, and once running, it calculates a SHA3-384 digest over a specific range in the On-Chip Memory (OCM) as specified by the boot image's partition header table. However, because the FSBL has already started executing, certain data structures within this OCM range, including the partition header table, may have been altered. This means that the FSBL measurement captures the state of both the FSBL and the partition header table, resulting in a SHA3-384 digest that reflects any modifications to the partition header. This digest is then recorded in the Event Log, representing a combined measurement of the FSBL and the partition header.

Non-PL partitions are measured based on the memory locations specified by their respective partition headers. This measurement occurs after authentication and decryption, provided those processes are defined in the partition header table. A SHA3-384 digest of the specified memory range is computed and stored in the Event Log, capturing the post-authentication and/or decryption state of each partition. For PL images, measurements depend on whether the images are authenticated and/or encrypted. If a PL image is authenticated, the system reuses the SHA3-384 digest generated during the authentication process, which is then recorded in the Event Log. For unauthenticated PL images, a SHA3-384 digest is calculated on the data as it streams to the PCAP (Processor Configuration Access Port) over the secure stream switch, and this digest is recorded in the Event Log. This process ensures that even unauthenticated images are integrity-checked based on their transmission data.

These measurement processes collectively build a comprehensive integrity record in the Event Log, enabling the system to have a detailed history of all integrity checks conducted during boot. This record can later be used for remote attestation or verification of the system's secure state, even in the absence of a TPM.

6.1.2 FSBL Changes to Support Measured Boot

By default, Xilinx's FSBL does not support measured boot or communication with a TPM. To enable measured boot functionality, several new files have been added to the FSBL, providing support for measurements and event logging. These files are designed to capture and store SHA3-384 digests for different system components, enabling secure boot measurements that could be used in the future by a TPM or fTPM. Below is an overview of these new files and their purposes:

1. **Fsbl_measured_boot.c**: contains the subroutines needed to measure CSU ROM, FSBL, and all non-PL partitions. This file is critical for implementing measured boot functionality by recording the initial boot stages and various system partitions.

2. **Fsbl_measured_boot.h**: this is the header file for "fsbl_measured_boot.c", defining the structures and function prototypes needed for performing measurements on the system's non-PL components.

3. **Fsbl_measured_pl.c**: contains the subroutines needed to measure PL partitions, facilitating the measurement of integrity for any programmable logic data loaded into the system.

4. **Fsbl_measured_pl.h**: the header file for "fsbl_measured_pl.c", containing declarations and necessary structures for executing PL partition measurements.

5. **Fsbl_measured_utils.c**: contains essential low-level subroutines for performing measurements, including functions like Tpm_ReadPcr, Tpm_Event, and SHA3 cryptographic routines. We are not using this file in our implementation because we do not have a TPM; instead, we save the measurements in an event log.

6. **Fsbl_measured_utils.h**: this header file supports "fsbl_measured_utils.c" by defining function prototypes and required data structures for lower-level measurement and cryptographic operations.

In addition to these new files, certain existing FSBL files have been modified to integrate the measured boot process:

1. **Xfsbl_config.h**: detailed debugging has been enabled in this configuration file, allowing the observation of SHA3-384 digests and PCR events during measured boot, which aids in testing and verification.

2. **Xfsbl_initialization.c**: the global variable *Iv* has been renamed to *Global_FsblIv* to prevent conflicts with an identically named variable in the "xilsecure" library, avoiding unexpected behaviour. CSU ROM measurement has been added early in the initialisation process, providing an initial record of system integrity. The FSBL measurement is added after the partition header table has been loaded, ensuring that the measurement includes any modifications to the header.

3. **Xfsbl_partition_load.c**: the variable *Iv* is renamed to *Global_FsblIv* to maintain consistency and avoid conflicts. New calls have been added to measure all non-PL partitions as well as PL partitions, ensuring comprehensive measurement coverage for each loaded partition. The partition number has also been added to the PL parameter structure to support measurements tied to specific partitions.

4. **Xfsbl_plpartition_valid.c**: includes measurement functionality for authenticated PL partitions, recording their integrity for later verification.

5. **Xfsbl_plpartition_valid.h**: adds the partition number to the PL parameter data structure, ensuring that each PL partition's measurement is accurately associated with its specific partition.

These modifications collectively enable the FSBL to perform measured boot. By systematically measuring each loaded component and recording the SHA3-384 digests, the system can build an Event Log that captures the integrity of each stage of the boot process. This log can be used for future attestation, offering a robust foundation for enhanced security measures and compatibility with TPM or fTPM in future implementations. This implementation provides a flexible, forward-compatible solution that lays the groundwork for remote attestation and system verification, even in the absence of a physical TPM.

6.1.3 FSBL Changes to Support Event Log

The creation of the Event Log in this implementation was achieved by adapting specific files from the Arm Trusted Firmware project. Originally, these files were designed to create and print an Event Log for an FVP (Fixed Virtual Platform), a virtual environment used to emulate complex hardware systems for testing and development. Adapting these files for the ZU+ MPSoC allowed us to capture essential boot-time measurements in our Event Log, even without a physical TPM present. In setting up the Event Log, we followed the guidelines outlined in the 'TCG PC Client-Specific Implementation Specification'. This specification, published by the Trusted Computing Group (TCG), provides detailed instructions on how to initialise and measure platform events during boot, in compliance with TCG standards. It is primarily designed for platforms that use the Extensible Firmware Interface (EFI), detailing how boot events should be measured and stored in the TPM's PCRs and how these events should be recorded in the Event Log.

Here are the files added to integrate the Event Log functionality within the FSBL code:

1. **tcg.h**: this file defines macros and data structures for the TPM event log, based on the specifications outlined by the TCG. It serves as a foundation for logging events in a format compatible with TPM systems.
2. **event_print.c**: this module handles the printing of the event log. It takes the memory address of the event log as input and outputs the stored events in a human-readable format, facilitating debugging and analysis of the event log content.
3. **event_log.c**: this file is responsible for populating the Event Log data structure. It collects and organises events that occur during the boot process, ensuring that they are correctly stored and prepared for logging.
4. **event_log.h**: serving as the header file for "event_log.c", this file declares the functions and data structures used to manage and manipulate the Event Log, providing the necessary interface for other parts of the system to interact with the log.

TCG PC Client Specific Implementation Specification

The TCG PC Client-Specific Implementation Specification is a critical document in the realm of trusted computing, guiding how to achieve secure measurements and logging during system boot. It defines a set of processes to ensure the integrity of the system as it initialises and loads an operating system, placing particular emphasis on the recording of measurements in a TPM [20].

Key components of the specification include:

1. **Event Measurement and PCR Logging**: this aspect of the specification dictates that certain boot events must be measured and recorded into specific PCRs within the TPM. These measurements may include firmware components, bootloaders, and the operating system itself. Each PCR entry provides a cryptographic digest, which reflects the state of the system at various stages of the boot process. If any component is altered or tampered with, the digest will change, signalling a potential security issue.
2. **Event Log Creation**: the specification also defines the structure of the Event Log, which contains detailed entries describing each measured boot event. Each entry typically includes

metadata about the event, the measured digest, and descriptive information to aid in event identification. This log serves as an audit trail that remote attestation servers can analyse to verify the integrity of the system, either at boot or later.

3. **Compatibility with EFI**: the specification is particularly focused on EFI-based platforms, laying out how EFI events and components should be measured and recorded. This process ensures that secure measurements are carried over to the operating system, creating a chain of trust from firmware to OS. For systems not using EFI directly, like our custom implementation, the principles and structure of the Event Log remain highly applicable, even if they require adaptation.

In our implementation, we adapted these specifications to meet the needs of the ZU+ MPSoC. Since we do not have a physical TPM, our system is set up to save SHA3-384 digests in an Event Log, emulating what would be sent to a TPM's PCRs in a complete TCG-compliant environment. This approach allows us to log boot-time measurements securely and consistently, and it provides a clear, structured Event Log that could potentially be read by a TPM or fTPM if one is integrated in the future.

In the context of the TCG and TPM, several important fields are defined to maintain security and integrity in systems. These fields are part of the event log and are used to record various events related to the system's security, particularly during boot processes. In figure 6.1, we can observe the first two events: TCG_EfiSpecIDEvent and the first PCR_Event2. These preliminary events provide the initial information related to the platform and its authentication status. Following these initial events, the log continues with measurements of the system's critical components, such as ROM, FSBL, BL31, U-Boot, and OP-TEE. These later events represent the measurements of each component as part of the boot process. They ensure the integrity of the system by securely recording the hashes of these elements to verify their authenticity during boot.

```
TCG_EfiSpecIDEvent:
  PCRIndex           : 0
  EventType          : 3
  Digest             : 00
                     : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                     : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                     : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 EventSize          : 33
  Signature          : Spec ID Event03
  PlatformClass      : 0
  SpecVersion        : 2.0.2
  UintnSize          : 1
  NumberOfAlgorithms : 1
  DigestSizes        :
    #0 AlgorithmId   : SHA384
       DigestSize    : 48
  VendorInfoSize     : 0
PCR_Event2:
  PCRIndex           : 0
  EventType          : 3
  Digests Count      : 1
    #0 AlgorithmId   : SHA384
       Digest        : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                     : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                     : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  EventSize          : 17
  Signature          : StartupLocality
  StartupLocality    : 0
```

Figure 6.1. Event Log

Below is an explanation of the key fields and their roles:

1. **TCG_EfiSpecIDEvent**: this event type provides information about the specifications and format of the events recorded in the TPM. It helps in identifying how the event is structured and what it represents.

- PCRIndex refers to a specific PCR (Platform Configuration Register) in the TPM where the measurements of system components are stored.
- EventType: 3 corresponds to EV_NO_ACTION, which is used for logging informational or status events that do not signify critical changes to system security. These events are essentially "neutral", indicating that nothing significant has occurred about system integrity.
- Digest represents the cryptographic hash value of the event being logged. This value is calculated by applying a cryptographic hash function (SHA3/384) to the relevant data.
- EventSize is the size of the event. This field helps determine the total size of the event in the log, considering the event structure and any associated algorithm sizes.
- Signature: Spec ID Event03 is used to represent a specific signature linked to an event that identifies the TPM's role in the security process. It signifies a unique signature that is validated within the TPM's operations.
- SpecVersion: 2.0.2 refers to the version of the TCG specification for TPM 2.0, ensuring compatibility and understanding between systems that follow this standard.
- UintnSize specifies the size of the UINTN fields used in various data structures used in TCG EFI Specification. 0x01 indicates UINT32 and 0x02 indicates UINT64.
- NumberOfAlgorithms indicates the number of cryptographic hash algorithms used in the event log.
- SHA3/384 is the cryptographic hash algorithm used to generate a 48-byte digest of the event data. It provides the highest level of security achievable with the SHA core of the CSU. The digest is used to verify that no tampering has occurred with the event log data. It ensures that the integrity of the system is maintained by comparing the computed digest with a known good value.

2. **PCR_Event2**: this is an updated version of the original PCR event structure defined in the TPM specifications. The PCR stores measurements of critical components, and the event logs these measurements as part of the system's security.
   - EventType: 1 corresponds to EV_POST_CODE, which is used to log events that contain measurements of critical components. These measurements help verify the integrity of key system components, such as the firmware or the OS.
   - StartupLocality: the event signature "StartupLocality" in the TPM event log records the locality of the system's startup. This helps track where the system is initialised, which could be useful for systems with multiple processing locations or configurations.
   - The other fields of this event have already been described previously.

These fields are all part of a system that works together to ensure secure boot and event logging in a trusted computing environment. By leveraging these fields, the system can track integrity from boot to runtime, verifying that no unauthorised changes have been made to critical software and firmware components.

6.1.4 Compilation and Integration of Firmware Components

Following the modifications made to the FSBL to enable Measured Boot functionality, the next step was to compile the necessary firmware components for the Xilinx ZU+ MPSoC using Vitis Unified IDE. Vitis is a comprehensive integrated development environment provided by Xilinx that supports a wide range of workflows for embedded systems, including firmware compilation and deployment.

Using Vitis, I compiled two essential firmware files critical to the boot process and the operation of the ZU+ MPSoC. The first file, pmufw.elf, is the firmware for the PMU. The PMU firmware plays a crucial role in the runtime management of the platform, coordinating power-saving operations, handling system power states, and ensuring efficient energy usage. It ensures

that the platform operates within defined power and performance parameters, which is especially important for embedded systems with strict power requirements.

The second file, fsbl.elf, is the FSBL, which is the first component executed during the boot process. The FSBL is responsible for initializing the system, configuring essential hardware settings, and preparing the environment for subsequent boot stages. It loads and verifies the integrity of other critical boot components, including the PMU firmware, ATF, U-Boot, and OP-TEE. Together, these two firmware components ensure that the ZU+ MPSoC initializes correctly and efficiently, progressing through the boot stages with the required security measures. By compiling the FSBL with Vitis, I incorporated the modified code, enabling it to generate SHA3-384 digests for measured boot and log them in the Event Log, following the TCG specifications.

The Xilinx Zynq UltraScale+ MPSoC requires these two essential firmware images to ensure proper system initialisation and runtime management (FSBL and PMUFW). However, the OP-TEE build Makefile does not include the necessary steps for compiling these two firmware components. As a result, pre-built binaries for both the FSBL and PMUFW are required to generate a valid boot image. These pre-built images can be found on the Xilinx wiki page [21].

Following this, I proceeded with the process of integrating OP-TEE into the system. To do so, I followed the OP-TEE documentation [16] and downloaded the OP-TEE repository, specifically version 4.3.0, which contains not only the OP-TEE OS, but also other essential firmware components such as ARM Trusted Firmware, U-Boot-Xlnx, Linux-Xlnx, OP-TEE Client, OP-TEE Test, OP-TEE Examples and others repositories available in the OP-TEE GitHub project provide useful resources to support integration on the ZU+ MPSoC, ensuring that each component is configured and compatible for the board's specific environment. These components are integral to setting up a secure boot environment with OP-TEE on the ZCU104 platform.

After obtaining the repository, I compiled the required firmware components using the proper command. The build command creates the 'BOOT.bin' file. It is generated using the Bootgen tool, which takes as input a .bif (Boot Image Format) file. This file describes the structure and sequence of components that are included in the boot image. Specifically, the .bif file combines the following pre-built firmware components: PMU Firmware, FSBL, ARM Trusted Firmware (BL31), U-Boot, OP-TEE. Bootgen reads the .bif file and packages these firmware components into a single 'BOOT.bin' image. This image is then used to securely boot the ZCU104 system, ensuring the correct sequence and integrity of the boot process. The use of a .bif file allows precise control over the order and inclusion of firmware components, essential for initialising the system securely. The content of the .bif file used in our implementation is as follows:

```
all:
{
    [pmufw_image] pmufw.elf
    [bootloader, destination_cpu = a53-0] fsbl.elf
    [destination_cpu = a53-0, exception_level = el-3, trustzone] bl31.elf
    [destination_cpu = a53-0, exception_level = el-2] u-boot.elf
    [destination_cpu = a53-0, load=0x60000000, startup=0x60000000,
    exception_level = el-1, trustzone] tee_raw.bin
}
```

Additionally, the 'zynqmp-zcu104.ub' image is created, a Flattened Image Tree (FIT) image that contains the Linux kernel, device tree blob (DTB), and the root filesystem. This image is required to boot Linux on the ZCU104, as it provides the kernel, hardware configuration, and the initial file system environment needed for the system to function properly. Both files must be loaded onto the same partition of the SD card, formatted as FAT32, to enable booting 6.2.

6.2 Booting in SD Card Mode

To set up the ZCU104 board for SD boot mode, follow these steps to configure the hardware, make the necessary connections, and start a terminal session to monitor the boot process.

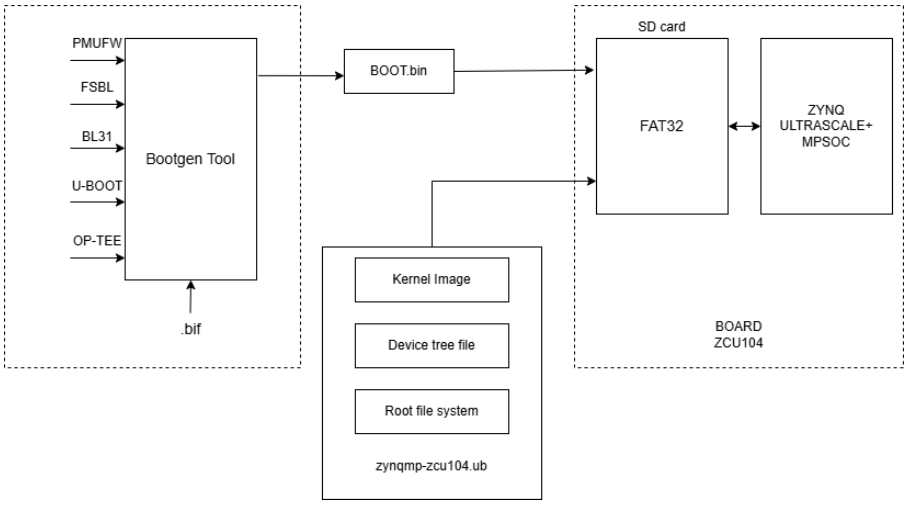

Figure 6.2. Booting in SD Mode - Diagram

1. **Insert the SD Card**: place the SD card, containing boot files like BOOT.bin and zynqmp-zcu104.ub, into the J100 SD card slot on the ZCU104 board. This slot is specifically designated for SD boot mode, allowing the system to boot from the SD card.
2. **Connect the USB Cable**: connect a Micro USB cable from the Micro USB port J83 on the ZCU104 to an available USB port on your host computer. This connection enables a UART interface that allows serial communication between the host and the board.
3. **Set the Board to SD Boot Mode**: adjust the SW6 switch configuration to set the ZCU104 to boot from the SD card, as outlined in the following figure 6.3. This step ensures the board prioritises the SD card as the primary boot source.
4. **Power the Board**: plug in a 12V power supply to the ZCU104's 6-pin Molex power connector, which powers up the board and initiates the boot sequence from the SD card.
5. **Start a Terminal Session**: open a terminal program on your host machine to monitor and interact with the board's boot process. On Linux, you can use PuTTY or Minicom, both of which are widely used for serial communication. PuTTY is an open-source terminal emulator available on multiple platforms, including Linux. It supports various network protocols, including serial communication, and is commonly used for embedded systems like the ZCU104. Minicom is a text-based serial communication tool for Linux that provides an interface to connect to embedded devices, enabling monitoring of the boot process and debugging. Configure your terminal program to connect to the ZCU104's COM port (usually accessible through /dev/ttyUSB on Linux systems) at a 115200 bps baud rate, the default UART setting for ZCU104.

These steps prepare the ZCU104 for monitoring and control during the boot process, allowing you to observe system logs and interact with the system in real-time, which is particularly useful for debugging and issuing commands during boot.

6.3 Secure Boot Implementation

To implement Secure Boot on the ZCU104 board, we need to modify the boot image process to ensure that each boot component is authenticated before execution. The goal of Secure Boot is to prevent unauthorised or tampered firmware from being loaded during the boot process, and this is achieved through cryptographic authentication of the firmware components.

In this case, Secure Boot utilises RSA encryption with RSA-4096 keys. Two types of keys are used: the Primary Secret Key (PSK0) and the Secondary Secret Key (SSK0). These keys are

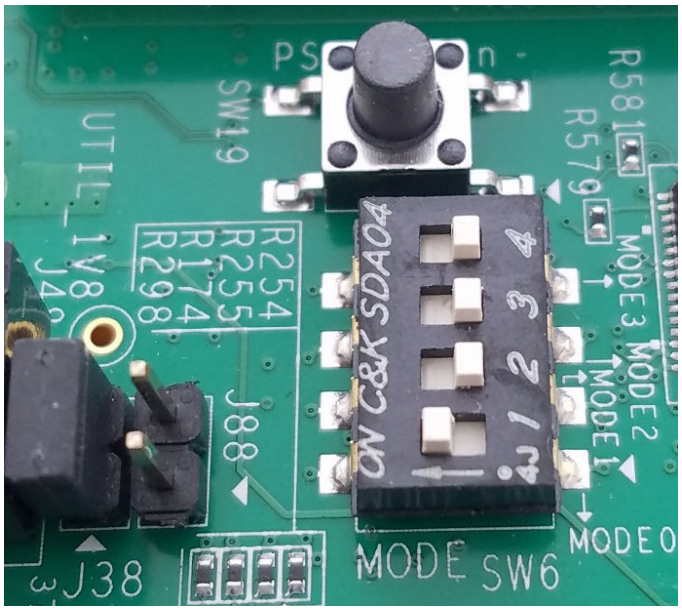

Figure 6.3. Boot mode pins SW6 [15]

used to sign the firmware components, and during boot, their corresponding public keys are used to verify the authenticity of the signed components. If the signature check fails, the boot process is halted, ensuring that only trusted and verified firmware is loaded.

The .bif file, which is used by the Bootgen tool to generate the BOOT.bin image, is the key element in configuring the boot image and enabling Secure Boot. In the modified .bif file, the authentication attribute is applied to each component to ensure that the firmware is verified using the RSA keys. Specifically, the components like FSBL, ARM Trusted Firmware (BL31), U-Boot, and OP-TEE are all signed and verified. For example, the FSBL is configured with RSA authentication and a specific destination_cpu, indicating that the component should be verified before it is executed.

Furthermore, the fsbl_config section in the .bif file includes the auth_enable setting, which activates the authentication process for the FSBL. This ensures that not only the FSBL itself is authenticated, but it also checks the integrity of the subsequent boot components it loads. By enabling authentication at this stage, we guarantee that the rest of the boot chain, comprising BL31, U-Boot, and OP-TEE, is also validated before execution. The auth_params section of the .bif file specifies parameters such as 'spk_id' and 'ppk_select', which are used to select the appropriate public keys for the authentication process. These keys are crucial in ensuring the integrity and authenticity of the components. If any of the firmware components are tampered with or if an unauthorised module is detected, the boot process will terminate immediately.

The modified .bif file appears as follows:

```
all:
{
    [pskfile] psk0.pem
    [sskfile] ssk0.pem
    [auth_params] spk_id = 0; ppk_select = 0
    [fsbl_config] a53_x64, bh_auth_enable
    [pmufw_image] pmufw.elf
    [bootloader, destination_cpu = a53-0, authentication = rsa] fsbl.elf
    [destination_cpu = a53-0, exception_level = el-3,
    trustzone, authentication = rsa] bl31.elf
    [destination_cpu = a53-0, exception_level = el-2,
    authentication = rsa] u-boot.elf
```

```
    [destination_cpu = a53-0, load=0x60000000, startup=0x60000000,
    exception_level = el1, trustzone, authentication = rsa] tee_raw.bin
}
```

By integrating Secure Boot with the measured boot, the system not only ensures that the boot components have not been tampered with but also logs cryptographic measurements for further security verification. Together, these mechanisms provide a robust defence against unauthorised firmware or software being executed on the ZCU104 board, making it more resistant to potential attacks or malicious modifications.

6.3.1 Implementing Post-Quantum Authentication in the Boot Process

Currently, the partition authentication process relies on RSA signatures, with the verification being handled by the CSU's dedicated 4096-bit RSA core. This ensures compatibility with existing infrastructure and leverages hardware-accelerated cryptographic operations. The modifications implemented so far involve adjustments to the '.bif' files of the OP-TEE project to define the partition authentication structure and maintain alignment with the secure boot flow.

To incorporate post-quantum security, one viable approach would be to integrate a library like 'liboqs' into the Vitis development environment. Liboqs offers a rich collection of PQC algorithms, including LMS (Leighton-Micali Signature scheme), which can be directly utilised to replace RSA signatures in the boot process. This integration would involve the following steps:

1. **Library Integration**: import the 'liboqs' library into the Vitis project, ensuring that it is included in the build system and linked with the existing cryptographic workflow.
2. **Key Generation and Signing**: modify the signing process to generate LMS-based signatures for the boot partitions. This would replace the RSA private key operations currently used to sign partitions.
3. **Verification Update**: update the CSU's verification logic (or software fallback) to utilise LMS-based signature verification. This may involve software-level adaptations if hardware support is unavailable, leveraging the computational efficiency of hash-based algorithms.
4. **Testing and Validation**: thoroughly test the integration to validate the LMS signatures during the boot process and ensure robustness against tampering.

Advantages of LMS Integration

LMS offers significant advantages for securing the boot process in a post-quantum era. As a hash-based algorithm, it is inherently resistant to quantum computer attacks, including those leveraging Shor's algorithm, which makes it an excellent choice for ensuring long-term cryptographic security. Its design relies solely on hash operations, making it highly efficient and well-suited for embedded environments where computational resources are often limited. One of the key benefits of adopting LMS is its compatibility with existing RSA infrastructure. By integrating it through software libraries such as liboqs, LMS can be used alongside current systems, enabling a smooth transition to post-quantum algorithms without requiring immediate overhauls of established processes. Furthermore, LMS aligns with emerging cryptographic standards, being part of the NIST-approved stateful hash-based schemes. This ensures not only enhanced security but also compliance with the latest guidelines for PQC.

Incorporating LMS into the boot process thus provides a robust defence against quantum threats while offering a forward-looking approach to authentication. It ensures that the framework remains adaptable and secure as technological landscapes evolve, maintaining the integrity of the system against both current and future challenges.

Chapter 7

Secure and Measured Boot on Zynq UltraScale+ MPSoC: Evaluation Tests

7.1 Security Tests

In this chapter, we analyse the results of security tests performed on three distinct boot configurations to evaluate the effectiveness of measured and secure boot mechanisms in protecting system integrity. These configurations simulate various scenarios to explore the detection capabilities and resilience of the boot process against unauthorised modifications and attacks. The scenarios examined include a baseline test with a correctly implemented measured and secure boot, a corrupted boot scenario designed to simulate a failure in authentication and a rollback attack simulation. By examining the outcomes of these tests, we gain insights into how the measured and secure boot mechanisms respond to both intended and malicious changes, highlighting the security protocols' strengths and potential vulnerabilities.

7.1.1 Scenario 1: Standard Measured and Secure Boot

The first scenario tests a properly configured measured and secure boot process, serving as the reference model. In this setup, the boot process consists of multiple authenticated partitions, each measured to create a baseline set of cryptographic digests (hashes) stored for integrity verification. Measured boot calculates hashes during the boot process and records them in the Event Log, creating a traceable record of the integrity of each component loaded during startup. These measurements do not involve immediate comparisons but are instead stored for potential verification in the futtyure. Alongside this, the secure boot mechanism employs RSA authentication to verify the digital signatures of each partition, ensuring that only authorised components are loaded into the system. This dual approach prevents the system from executing unverified or altered code, establishing a trusted boot environment. This initial test provides a foundation for evaluating deviations in the following scenarios.

To verify the integrity and authenticity of each boot component, a predefined set of cryptographic digests, or a whitelist, is established as a baseline. This baseline contains the expected hash values of each partition, calculated and stored during the initial trusted configuration of the system. These digests serve as reference points, enabling the boot process to identify unauthorised modifications by comparing the stored values with the digests calculated during runtime. Each digest in this baseline list corresponds to a specific partition in the boot sequence, covering all critical components such as the ROM, FSBL, ATF, U-Boot, and OP-TEE. In a correctly functioning system, the digests generated at boot time should match these baseline values. If any discrepancy is detected, it indicates that a modification has occurred, potentially flagging a

security breach. This whitelist approach ensures that only authorised, unaltered components are executed, adding a layer of security by verifying each step of the boot process.

In our case, we have the following whitelist:

```
EVENT           DIGEST

ROM             : 90 CB 37 29 64 B6 E7 CD 4F 3F CE DD 89 38 5E CF
                : 0E 2D FC A8 4C 3F 44 DC 19 65 CF 5E 4B C9 72 F1
                : E3 B4 27 83 9B E5 BE 7F DB 3B 5A 0B 31 27 04 26

FSBL            : 14 7B 26 3F F0 B9 73 9C 3C 02 AE 3E 19 62 7E 4B
                : 78 ED 44 A6 B0 63 91 6C 11 AC 28 89 D5 7A FB 75
                : 64 B4 94 E1 CF 6D 58 20 B7 7A E5 4C 5C 42 B8 3C

BL31            : E0 77 BE B2 3A F9 BD 30 ED 98 A7 12 D1 52 72 49
                : 9A 03 6B F1 D5 15 24 AE DF D4 AB 98 C4 75 13 1D
                : A9 70 BA 90 89 F6 45 5F 8D E6 2F 61 14 C7 9E D4

U-BOOT          : A7 14 4C C4 12 4D FE B5 2B A0 65 D0 EE 76 70 56
                : F6 A2 F5 18 BE 7B E4 BB DC 30 CF 70 27 D1 0F 07
                : B4 A4 0B 9C BB E1 CB 63 38 DD 1A 98 64 85 2F 95

OP-TEE          : 5E D9 E7 2B 8B 38 63 0F 71 A6 A7 58 46 1A F8 1E
                : 25 A9 6B 02 1D 5A B6 24 40 34 AC DA 5B F4 D4 EA
                : 9E 8B D1 BB 6E C7 46 2A 30 2F 10 6B E0 FE D3 EA
```

7.1.2 Scenario 2: Corrupted Measured and Secure Boot

The second scenario simulates a system integrity failure by introducing deliberate corruption into the boot image. This scenario involves altering a single bit in the final partition within the 'BOOT.bin' file, resulting in a discrepancy between the calculated and stored digest values for that partition. Since the measured boot compares digests at each boot stage, the mismatch indicates a modification, leading the secure boot's RSA authentication to fail when verifying the OP-TEE partition. This failure prevents the secure boot process from authenticating the partition, ultimately halting the boot process due to a perceived security violation. This test demonstrates how even minor, undetectable changes in code or data can lead to a breakdown in the boot process, highlighting the precision and sensitivity of measured and secure boot mechanisms in detecting tampered components.

To modify a single bit in a binary file on Ubuntu, we can use a series of commands to target only the last bit of the last byte, without altering any other data in the file. This process uses the 'dd' command alongside 'printf' to locate, read, and modify the specific bit. Here are the steps and commands to execute this operation:

1. **Identify the File Size**: first, determine the file size in bytes, as we need to know the exact position of the last byte.

   ```
   file_size=$(stat --format=%s filename.bin)
   ```

2. **Read the Last Byte**: next, read the last byte to check its current value. This helps us understand the bit pattern before modification.

   ```
   last_byte=$(dd if=filename.bin bs=1 skip=$((file_size - 1)) count=1
       2>/dev/null | od -An -t u1)
   ```

3. **Flip the Last Bit**: use a bitwise XOR operation to toggle the last bit (bit 0) of the last byte.

```
new_byte=$((last_byte ^ 1))
```

4. **Write the New Byte Back**: finally, write the modified byte back to the last position in the file using 'printf' and 'dd'.

```
printf \\x$(printf "%02x" $new_byte) | dd of=filename.bin bs=1
    seek=$((file_size - 1)) count=1 conv=notrunc
```

This sequence of commands will modify only the last bit of the file's last byte, ensuring that the rest of the file remains unaltered.

7.1.3 Scenario 3: Rollback Attack Simulation

The final scenario examines a rollback attack, a security threat in which an attacker attempts to replace the current software with an older, possibly vulnerable version. In this simulation, additional print statements are introduced in the code, creating a new version with modified outputs but no significant security changes. Despite appearing identical in functionality, this modified code results in a different set of cryptographic digests during measured boot. The stored digests, reflecting an earlier state of the system, now fail to match the newly generated values. This discrepancy, detected by measured boot, flags the system as potentially compromised.

In this case, we have the following hash values:

```
EVENT           DIGEST

ROM             : 90 CB 37 29 64 B6 E7 CD 4F 3F CE DD 89 38 5E CF
                : 0E 2D FC A8 4C 3F 44 DC 19 65 CF 5E 4B C9 72 F1
                : E3 B4 27 83 9B E5 BE 7F DB 3B 5A 0B 31 27 04 26

FSBL            : 14 7B 26 3F F0 B9 73 9C 3C 02 AE 3E 19 62 7E 4B
                : 78 ED 44 A6 B0 63 91 6C 11 AC 28 89 D5 7A FB 75
                : 64 B4 94 E1 CF 6D 58 20 B7 7A E5 4C 5C 42 B8 3C

BL31            : B2 BB 24 71 B7 11 83 84 83 24 21 70 49 19 D6 3F
                : 51 DA 82 56 34 60 3B D9 8F 4D 9A AA 02 50 D7 33
                : BD 66 6A ED 0B E3 6F F8 C4 65 E9 E3 96 AB 03 71

U-BOOT          : E3 AF 99 B9 C7 F8 B6 4C FA 87 FC 02 11 EC 3C 6A
                : 7E 09 19 78 9A D9 0A 07 6F D9 8D 47 12 D9 18 40
                : B0 AC 72 EB 55 34 29 A7 30 70 99 16 E7 0F 75 B6

OP-TEE          : E8 98 74 C3 3B D3 1E 64 41 DB 6A DB C3 02 EC E7
                : BC A2 DC 92 9F 38 8E 61 50 0C EF 65 57 8A A1 66
                : 85 6C C3 19 42 7B 99 9B 79 3D B6 10 57 3D 8A EB
```

In a real-world rollback attack, an attacker could attempt to revert the system to a prior software state, effectively bypassing recent updates or security patches. Rollback attacks can be perilous in embedded systems or IoT devices, where updates address security vulnerabilities. The measured boot process plays a critical role in countering such attacks by verifying that each component matches its latest approved version, effectively preventing unauthorised downgrades.

Through this series of tests, we provide a detailed evaluation of each boot configuration, underscoring the importance of measured and secure boot mechanisms in safeguarding system integrity. Subsequent sections will delve into the specifics of each test, including partition sizes, cryptographic digest values, and detailed measurement data, further illustrating the mechanisms' effectiveness and limitations in different attack scenarios. This analysis provides a comprehensive understanding of the boot process's security layers and their significance in embedded and secure computing.

7.2 Performance Tests

The boot process represents a critical phase in the operation of an embedded system, during which fundamental components are initialised and the operating system is launched. Depending on security and reliability requirements, the boot process can be configured in various ways, each with significant implications in terms of loading times and functionalities. Here, we analyse five primary scenarios: Standard Boot, Measured Boot, Measured and Secure Boot, Corrupted Boot, and Rollback Boot.

Standard Boot

The Standard Boot represents the most basic boot sequence, devoid of security or measurement functionalities. In this scenario, the system simply loads the components without performing any verification or recording measurements. The fsbl.bin file has a size of 107192 bytes, reflecting minimal code necessary for handling the loading process. The time taken to load the partitions, from the initiation of FSBL execution until the completion of the autoboot process, is 1.68 seconds, indicating a swift boot-up attributable to the simplicity of operations.

Measured Boot

In the Measured Boot process, the system records measurements of critical components (e.g., FSBL and subsequent partitions) without authentication checks for each loaded partition. These measurements can later be utilised for integrity analysis or verification purposes. The inclusion of measurement-related code increases the fsbl.bin size to 148128 bytes. Consequently, the partition loading time extends to 2.48 seconds, due to the additional overhead introduced by the measurement operations.

Measured and Secure Boot

The Measured and Secure Boot combines the recording of measurements with cryptographic verification to ensure that each loaded component is authentic and intact. This mode represents the highest level of security, preventing the execution of unauthorised components. The fsbl.bin size remains 148128 bytes, but the loading time significantly increases to 5.46 seconds. This increase is attributable to the cryptographic checks performed on each component before loading, ensuring comprehensive security.

Corrupted Boot

To simulate error scenarios, a Corrupted Boot involves an attempt to boot with a corrupted FSBL or partition. In this case, the secure boot process detects the corruption and halts further boot stages to maintain system security. The fsbl.bin size remains 148128 bytes, while the partition loading time is 3.57 seconds. This reduced time compared to the Measured and Secure Boot indicates that the system detected the corruption early in the process, promptly aborting the boot sequence.

Rollback Boot

Rollback Boot is a security mechanism that prevents the system from loading outdated or untrusted versions of boot components by reverting to a previously verified version if necessary. Similar to the Measured and Secure Boot, the fsbl.bin size is 148128 bytes. The loading time is 5.47 seconds, consistent with the time required for cryptographic verifications and restoration operations. This ensures that only trusted components are executed, safeguarding against rollback attacks where an attacker might attempt to replace secure components with older, potentially vulnerable versions.

Here is the table summarising the boot times, and fsbl.bin size for each scenario 7.1.

Case	Boot Type	fsbl.bin (bytes)	Time to Load Partitions (s)
1	Standard Boot	107192	1.68
2	Measured Only	148128	2.48
3	Measured and Secure	148128	5.46
4	Corrupted	148128	3.57 (Failed)
5	Rollback	148128	5.47

Table 7.1. Boot Cases with fsbl.bin size and Time to Load Partitions

Security Levels and Loading Times

These scenarios highlight a clear trade-off between speed and security. The additional complexity introduced by measurement and verification functionalities significantly affects both the FSBL size and the partition loading time but ensures increasing levels of security. To visually represent these trade-offs, a bar graph can be utilised to illustrate the loading times concerning the security levels of each boot case. The graph should order the boot cases by increasing loading times and differentiating security levels using distinct colours. Additionally, annotations can indicate whether a boot process failed in the case of corruption, providing a comprehensive overview of the relationship between boot performance and security 7.1.

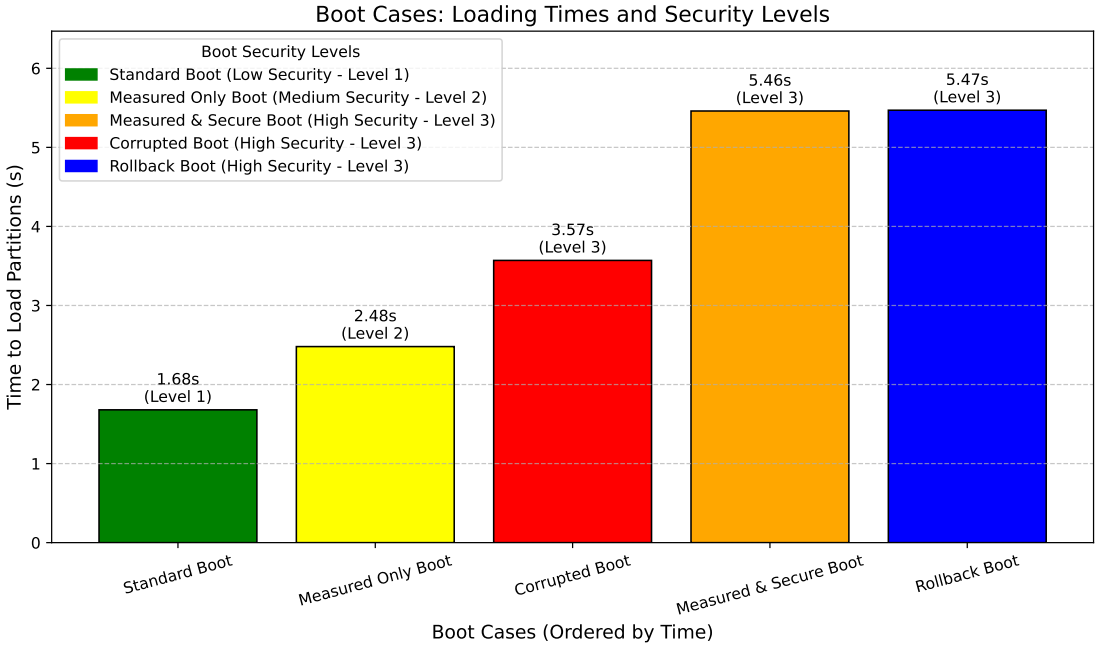

Figure 7.1. Boot Cases Histogram

Security Levels Explained:

- **Level 1 (Low Security)**: minimal or no security features.
  Example: Standard Boot, where no measurements or verifications are performed.
- **Level 2 (Medium Security)**: basic security measures such as recording measurements.
  Example: Measured Only Boot.
- **Level 3 (High Security)**: comprehensive security, including both measurements and integrity/authenticity verification for all boot components.
  Example 1: Measured and Secure Boot, which prevents unauthorised components from loading.

Example 2: Corrupted Boot (failed) demonstrates a scenario where the security measures detect corruption and halt the boot process.

Example 3: Rollback boot ensures the system can revert to a previous, verified version of boot components if an outdated or untrusted version is detected. Rollback Boot

This visualisation highlights the relationship between security levels and performance, showing that increased security generally comes at the cost of longer boot times.

Chapter 8

Conclusions and Future Works

8.1 Key contributions

The initial phase of the research centred on a comprehensive study of the Zynq UltraScale+ MPSoC (ZU+ MPSoC), a highly versatile System-on-Chip (SoC) architecture that combines ARM Cortex-A53 cores, Cortex-R5F real-time processors, and programmable logic. This hybrid structure enables both high-performance computing and real-time control, making it a powerful platform for diverse applications.

Study of Zynq Ultrascale+ MPSoC

A detailed analysis of the boot process was essential to understanding how the system transitions from a powered-off state to a fully operational environment. The boot process on the ZU+ MPSoC is a multi-stage sequence that ensures the proper initialisation and secure loading of all software components. Each stage is executed in a defined order, with strict dependencies, to maintain the integrity and authenticity of the system. The CSU plays a pivotal role in securing the boot process. This hardware module provides essential cryptographic functionalities such as hashing, encryption, and signature verification. It supports algorithms like SHA3/384 for hashing and RSA for digital signature verification, making it the cornerstone of the platform's secure boot and measured boot capabilities. One of the critical tasks of the CSU is to validate the authenticity of boot components at each stage. By verifying digital signatures, the CSU ensures that only trusted and unmodified code is executed, preventing unauthorised modifications or tampering during the boot process. This mechanism forms the basis of the secure boot implementation. The CSU also includes hardware acceleration for cryptographic operations, significantly enhancing performance. For example, its SHA3 core enables the computation of digests with high efficiency, which is crucial for implementing measured boot. This hardware-backed security ensures that the system is protected against common attack vectors such as code injection and firmware corruption.

The ZCU104 development board, equipped with the ZU+ MPSoC, provided the ideal platform for practical experimentation. Its architecture closely reflects the capabilities of the ZU+ MPSoC, offering a balanced combination of processing power, real-time control, and FPGA programmability. Through the ZCU104, it was possible to closely examine each boot stage, including the functionality of the First Stage Boot Loader (FSBL), the execution of the Arm Trusted Firmware (ATF) components such as BL31, and the initialisation of the U-Boot bootloader. This granular analysis highlighted opportunities for integrating advanced security features like measured boot and secure boot.

An essential focus of the study was on the immutable ROM code, which serves as the root of trust for the entire boot process. The ROM performs the initial validation of the FSBL by verifying its digital signature, ensuring that the chain of trust begins with a secure and unalterable starting point. Understanding the role and limitations of this ROM code was crucial for designing enhancements to the measured boot process. Another significant aspect was the investigation of

the hardware interfaces provided by the ZCU104. These include the programmable logic (PL) and processing system (PS) interfaces, which are critical for deploying secure boot strategies. The seamless integration between PL and PS enables the implementation of advanced cryptographic features within the CSU and their efficient usage during boot.

This initial phase of research provided a solid understanding of the ZU+ MPSoC's architecture and its secure boot capabilities. It laid the foundation for the subsequent integration of OP-TEE, the customisation of the FSBL for measured boot, and the implementation of an event log for tracking cryptographic measurements. Each of these components was built upon the insights gained from this detailed analysis.

Integration of OP-TEE 4.3.0

A critical achievement of this work was the successful integration of OP-TEE 4.3.0 into the ZU+ MPSoC platform. OP-TEE, as a Trusted Execution Environment (TEE), introduces a secure world that operates independently from the normal world. This separation is pivotal for executing security-critical operations such as cryptographic calculations, key management, and the enforcement of secure policies. By integrating OP-TEE, the platform was transformed into a robust environment capable of hosting trusted applications alongside conventional operating systems like Linux. This dual-world architecture enhances security by isolating sensitive operations from potential vulnerabilities in the normal world.

The integration process required significant adaptations to ensure compatibility with the Zynq architecture. Unlike generic platforms, the ZU+ MPSoC introduces unique hardware resources, such as its tightly coupled processing system (PS) and configurable programmable logic (PL). Ensuring that OP-TEE could fully utilise these resources while maintaining its performance was a challenging but rewarding aspect of this work. One major challenge was adapting OP-TEE's secure boot sequence to align with the Zynq boot flow. The handoff between the FSBL, Arm Trusted Firmware (ATF), and OP-TEE required precise modifications to maintain the integrity and security of the boot chain. By addressing these challenges, we successfully enabled OP-TEE to operate seamlessly within the Zynq environment.

This integration provided several benefits. It established a foundation for implementing advanced security features such as measured boot and secure storage within the TEE. Trusted applications running in OP-TEE can leverage hardware-accelerated cryptographic functions provided by the CSU, ensuring that operations are both secure and efficient. The inclusion of OP-TEE also allowed for the implementation of a secure communication channel between the normal world and the secure world. This feature is critical for applications that require frequent interaction between untrusted user-space applications and secure services, ensuring that sensitive data remains protected at all times.

Despite these achievements, the integration also highlighted certain limitations. For instance, the dependence on RSA-based authentication in the current implementation introduces potential vulnerabilities in the face of emerging post-quantum threats. While the secure world is well-protected, the authentication mechanisms employed during the boot process could benefit from stronger, quantum-resistant algorithms.

Integration of Measured and Secure Boot

Building upon the secure foundation established by OP-TEE, a significant advancement in this work was the implementation of measured boot functionality through modifications to the FSBL. This feature ensures that every partition loaded during the boot process is subjected to cryptographic measurement, producing digests that can be validated against known good values. By incorporating measured boot, we added a layer of integrity verification to the boot process, crucial for detecting unauthorised changes or tampering. The measurements captured during the boot process include the immutable ROM digest, along with the digests of critical components such as the FSBL, BL31, U-Boot, and OP-TEE. These measurements are computed using the SHA3/384 algorithm, a state-of-the-art cryptographic standard known for its resilience against current cryptanalytic attacks. Leveraging the hardware-accelerated SHA3 core of the CSU ensured efficient

computation of these secure hashes, aligning with the ZU+ MPSoC's design for high-performance security operations.

The use of SHA3/384 ensures a very high level of security, as the algorithm is resistant to known collision and preimage attacks. This makes it ideal for guaranteeing the integrity of the measured components. However, the authentication process, which relies on RSA keys, introduces a potential vulnerability. While RSA remains secure against classical computational methods, its susceptibility to future quantum computing advancements highlights an area for improvement.

Secure boot functionality was also integrated to complement the measured boot mechanism. Secure boot authenticates all partitions loaded during the boot process using digital signatures. By employing asymmetric RSA keys, the system ensures that only authorised and unmodified software is executed. This approach significantly strengthens the system against unauthorised modifications, ensuring that only trusted software can gain control during the critical early stages of execution. Despite its benefits, the reliance on RSA for cryptographic signing represents a limitation. As quantum computing capabilities advance, RSA-based systems may become vulnerable to attacks leveraging quantum algorithms such as Shor's. While currently adequate, this reliance underscores the importance of transitioning to post-quantum cryptographic standards to future-proof the security framework.

Integration of Event Log

An important innovation introduced in this thesis was the development of a structured Event Log designed according to the specifications of the Trusted Computing Group (TCG). This event log plays a crucial role in securely storing the cryptographic digests of each partition measured during the boot process. By capturing these digests and systematically storing them, the event log ensures that the integrity of the system can be verified not only during boot but also post-boot, providing an additional layer of security through remote attestation. The event log facilitates a deeper level of verification by enabling comparisons between the measured digests and pre-computed golden values, which are known to be secure and authentic. This comparison can be conducted at any time after the boot process, allowing for the detection of discrepancies that could indicate tampering or corruption.

The use of a structured log also makes the system resilient to various types of attacks, such as rollback attacks, where an older, trusted image may be maliciously reintroduced into the system. However, rollback attacks are not the only threat that the event log helps mitigate. For example, the event log can also be used to detect downgrade attacks, where an attacker might replace the current, secure version of the software with a lower, less secure one, potentially exploiting vulnerabilities in earlier software versions. Since the event log maintains the integrity of each partition's measurements, any attempt to load a downgraded version of a partition will be detected due to the mismatch in cryptographic digests. Furthermore, the event log is instrumental in defending against injection attacks, where malicious code is introduced into an otherwise legitimate partition. Even if the malicious code is executed within a partition that was not originally compromised, any alteration of the partition's contents will cause a mismatch in its digest, which can be detected during post-boot verification. The event log also plays a key role in enhancing the system's ability to defend against persistent attacks, where an attacker seeks to maintain control over the system across multiple reboots or boot cycles. By periodically comparing the recorded digests in the event log with the expected golden values, the system can ensure that no tampering has occurred between boot sessions, further strengthening its resilience. In addition, the event log can also support forensic analysis. By securely recording every measured event, it creates a comprehensive record of all software components loaded into the system. In the event of a security breach or suspected attack, this log provides valuable insights into what exactly happened during boot, helping security analysts pinpoint the exact point of compromise or identify any irregularities in the boot sequence.

The introduction of the TCG-compliant event log significantly enhances the security framework by providing a transparent and verifiable record of system measurements. This approach not only protects against rollback, downgrade, and injection attacks but also facilitates ongoing integrity checks and supports forensic investigations, making it a key component in ensuring the long-term security and trustworthiness of the platform.

8.1.1 Strengths and Limitations of the Current Approach

1. **Strengths**:

   - **Integration of OP-TEE**: the integration of OP-TEE provides a highly secure environment within the ZU+ MPSoC platform. By leveraging OP-TEE, the system benefits from a separation between the normal world (where the main operating system runs) and the secure world (where security-critical operations occur). This separation ensures that sensitive operations, such as cryptographic key management or secure data handling, are executed in an isolated environment that is protected from potentially untrusted software running in the normal world. The use of OP-TEE not only enhances the security of the boot process but also creates a foundation for further development of secure applications within the platform.
   - **SHA3/384 for Integrity**: the decision to use SHA3/384 for generating cryptographic digests of the system's partitions significantly strengthens the integrity verification process. SHA3/384, a member of the SHA-3 family, offers robust security against various cryptographic attacks, including those posed by emerging quantum technologies. By leveraging the CSU of the ZCU104, which accelerates SHA3 operations in hardware, the system achieves optimal performance and reliability in measuring the integrity of critical components like the FSBL, BL31, U-Boot, and OP-TEE. This choice ensures that the system remains highly resistant to attacks that might compromise its boot process.
   - **Event Log for Post-Boot Analysis**: one of the most notable contributions of this work is the creation of a structured event log that records the cryptographic digests of each partition measured during boot. This event log, based on the TCG specifications, provides a secure and transparent record of all measured boot events. By storing these measurements, the event log facilitates post-boot analysis and verification of system integrity. Furthermore, this log lays the foundation for remote attestation, enabling external systems to verify the trustworthiness of the device by comparing recorded digests against known golden values. This capability enhances the overall security of the system, as it allows for ongoing checks of the system's integrity even after the boot process has been completed.
   - **Measured Boot**: the implementation of the measured boot adds an extra layer of security to the system by ensuring that each partition loaded during the boot process is cryptographically measured. This provides confidence that no unauthorised or malicious modifications have been made to any of the boot components. The combination of the measured boot with the event log and OP-TEE creates a robust security framework that actively detects tampering and ensures that the system operates as expected from a trusted baseline.

2. **Limitations**:

   - **Reliance on RSA for Authentication**: while RSA-based authentication remains a widely used and secure method for ensuring the authenticity of software components, it represents a potential vulnerability in the long term, especially in the face of emerging quantum computing technologies. Quantum computers, once sufficiently advanced, could potentially break RSA encryption by exploiting Shor's algorithm to factor large prime numbers more efficiently than classical computers. This would render RSA-based signatures insecure, allowing attackers to forge cryptographic signatures and bypass authentication mechanisms. Although RSA remains secure for the foreseeable future, this reliance presents a significant weakness in the system that needs to be addressed to future-proof the platform against quantum threats.
   - **Lack of Post-Quantum Cryptography Solutions**: currently, the system does not incorporate any post-quantum cryptography (PQC) solutions that are designed to resist attacks from quantum computers. While RSA keys provide sufficient security at present, the introduction of PQC algorithms, such as those based on lattice-based cryptography or hash-based signatures, would be essential to safeguard the system against quantum-enabled threats. The adoption of PQC solutions would significantly

strengthen the cryptographic backbone of the system, making it more resilient in a future where quantum computing capabilities become mainstream. This is an area for future development and integration to ensure the long-term security of the system.

- **Limited Integration with TPM for Remote Attestation**: the event log implemented in this work follows TCG specifications, which is a step towards establishing a reliable and verifiable system for boot-time integrity checks. However, the event log is not yet fully integrated with a Trusted Platform Module (TPM) or a firmware TPM (fTPM), which would allow for remote attestation. Remote attestation is a mechanism that enables external systems to verify the trustworthiness of the device based on the measurements stored in the event log. While the event log stores valuable integrity information, its effectiveness for remote attestation is limited without a fully functional TPM integration. The lack of such integration means that, although the system can perform local integrity checks, it cannot yet securely report its state to external entities for validation, thus limiting the scope of its trust framework.

8.2 Future Work and Enhancements

The following points outline potential areas for future work and improvements based on the contributions made in this thesis:

1. **Integration of Post-Quantum Cryptography**: as quantum computing technology progresses, traditional cryptographic algorithms, such as RSA, face growing vulnerabilities due to the potential of quantum algorithms like Shor's algorithm, which can efficiently factor large numbers and break RSA encryption. Therefore, integrating PQC algorithms into the secure boot and measured boot process becomes a crucial next step. Among the various PQC candidates, the Leighton-Micali Signature (LMS) algorithm stands out as a promising solution. LMS, as discussed in the PQC chapter, offers robust security against quantum attacks and is highly suitable for resource-constrained environments, such as embedded systems. By replacing RSA with LMS, the system can remain secure even in the face of advancements in quantum computing, ensuring long-term resilience. Future work should focus on integrating LMS into the boot process, enhancing both the integrity verification and authentication mechanisms, making them quantum-resistant and ensuring the continued trustworthiness of the system in a post-quantum world.

2. **Extended TPM Integration**: while the event log in this work was designed with compatibility for TPM or fTPM devices, the full integration of a TPM would significantly enhance the system's security capabilities, particularly in distributed, networked environments. TPMs provide a secure, hardware-backed mechanism for storing cryptographic keys and performing operations like secure boot verification and remote attestation. By integrating a full TPM, the event log could be securely sealed within the TPM's protected storage, ensuring that the integrity measurements are tamper-proof. Moreover, real-time remote attestation, a process where a remote entity can verify the system's trustworthiness by checking its event log, would become a seamless and reliable feature. This integration could also enable networked systems to participate in trusted environments, where devices constantly report their integrity status to a central authority, reducing the risk of compromise in larger, distributed networks.

3. **Extension to Other Platforms**: while this work has primarily focused on the ZCU104 platform, the methodologies and security principles developed here can be extended to other platforms that utilise the ZU+ MPSoC or similar architectures. The modular nature of the OP-TEE integration, the event log, and the boot security measures make them applicable to a wide range of embedded systems and SoCs. Future work could explore the adaptation of these features to other platforms, including those from different manufacturers, with minimal modification. This extension would not only validate the scalability of the approach but also make it easier to implement a standardised, secure boot process across various devices, contributing to the broader adoption of these security mechanisms in the embedded systems industry.

This thesis has demonstrated the feasibility and effectiveness of integrating OP-TEE, measured boot, and secure boot mechanisms into the ZU+ MPSoC, culminating in a highly secure platform capable of protecting against unauthorised modifications, rollback attacks, and other potential threats. By combining hardware-accelerated cryptographic operations with a structured event log, the system achieves a high level of assurance regarding its integrity, ensuring that only trusted code is executed during the boot process. Despite its strengths, the system's reliance on RSA for authentication remains a limitation in the context of future-proofing against quantum threats. The integration of PQC will be essential for ensuring long-term security. Additionally, the extended integration of TPMs for remote attestation offers promising directions for future development.

The work presented here lays a solid foundation for future enhancements in secure boot, measured boot, and post-quantum security, contributing to the ongoing evolution of trusted computing in embedded systems. As both cryptographic techniques and hardware security continue to evolve, future advancements will ensure that the system remains resilient against emerging threats, safeguarding the integrity and confidentiality of critical applications in an increasingly connected world.

Bibliography

[1] W. Barker, W. Polk, and M. Souppaya, "Getting ready for post-quantum cryptography: explore challenges associated with adoption and use of post-quantum cryptographic algorithms", The Publications of NIST Cyber Security White Paper (DRAFT), CSRC, NIST, GOV, vol. 26, April 2021, pp. 1–10, DOI 10.6028/NIST.CSWP.04282021

[2] Post-Quantum Use In Protocols, https://www.ietf.org/archive/id/draft-vaira-pquip-pqc-use-cases-01.html

[3] D. A. Cooper, D. C. Apon, Q. H. Dang, M. S. Davidson, M. J. Dworkin, C. A. Miller, *et al.*, "Recommendation for stateful hash-based signature schemes", NIST Special Publication, vol. 800, October 2020, pp. 1–59, DOI 10.6028/NIST.SP.800-208

[4] D. McGrew, M. Curcio, and S. Fluhrer, "Rfc 8554: Leighton-micali hash-based signatures", RFC-8554, April 2019, DOI 10.17487/RFC8554

[5] A. Wagner, F. Oberhansl, and M. Schink, "To be, or not to be stateful: Post-quantum secure boot using hash-based signatures", Proceedings of the 2022 Workshop on Attacks and Solutions in Hardware Security, New York, NY, USA, 2022, pp. 85–94, DOI 10.1145/3560834.3563831

[6] J.-P. Aumasson, D. J. Bernstein, W. Beullens, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, A. Hülsing, P. Kampanakis, S. Kölbl, *et al.*, "Sphincs+", June 2022, https://sphincs.org/data/sphincs+-r3.1-specification.pdf

[7] J. Howe, T. Pöppelmann, M. O'neill, E. O'sullivan, and T. Güneysu, "Practical lattice-based digital signature schemes", ACM Trans. Embed. Comput. Syst., vol. 14, April 2015, pp. 1–24, DOI 10.1145/2724713

[8] Intel Corporation, https://uefi.org/sites/default/files/resources/Post%20Quantum%20Webinar.pdf

[9] Open Quantum Safe, https://openquantumsafe.org/

[10] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey", ACM Comput. Surv., vol. 51, January 2019, pp. 1–36, DOI 10.1145/3291047

[11] M. Gross, K. Hohentanner, S. Wiehler, and G. Sigl, "Enhancing the security of fpga-socs via the usage of arm trustzone and a hybrid-tpm", ACM Trans. Reconfigurable Technol. Syst., vol. 15, November 2021, pp. 1–26, DOI 10.1145/3472959

[12] Advanced Micro Devices (AMD), "Zynq ultrascale+ mpsoc software developer guide (ug1137)", June 2024, https://docs.amd.com/r/en-US/ug1137-zynq-ultrascale-mpsoc-swdev/About-This-Guide

[13] Advanced Micro Devices (AMD), "Zynq ultrascale+ device technical reference manual (ug1085)", December 2023, https://docs.amd.com/r/en-US/ug1085-zynq-ultrascale-trm/Zynq-UltraScale-Device-Technical-Reference-Manual

[14] Advanced Micro Devices (AMD), "Developing tamper-resistant designs with zynq ultrascale+ devices (xapp-1323)", August 2018, https://docs.amd.com/v/u/en-US/xapp1323-zynq-usp-tamper-resistant-designs

[15] Advanced Micro Devices (AMD), "Zynq ultrascale+ mpsoc: Embedded design tutorial (ug1209)", September 2024, https://docs.amd.com/r/en-US/ug1209-embedded-design-tutorial

[16] OP-TEE Documentation, https://optee.readthedocs.io/en/latest/index.html

[17] Intel Documentation, https://docs.trustauthority.intel.com/main/articles/introduction.html

[18] Advanced Micro Devices (AMD), "Bootgen user guide (ug1209)", May 2024, https://docs.amd.com/r/en-US/ug1283-bootgen-user-guide
[19] Advanced Micro Devices (AMD), "Measured boot of zynq ultrascale+ devices (xapp-1342)", April 2019, https://docs.amd.com/v/u/en-US/xapp1342-measured-boot
[20] Trusting Computing Group (TCG), "Tcg efi platform specification for tpm family 1.1 or 1.2", TCG Published, January 2014, https://trustedcomputinggroup.org/wp-content/uploads/TCG_EFI_Platform_1_22_Final_-v15.pdf
[21] Xilinx Wiki Page, https://xilinx-wiki.atlassian.net/wiki/spaces/A/overview
[22] Xilinx Download Page, https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html
[23] Xilinx SoC Prebuilt Firmware, https://github.com/Xilinx/soc-prebuilt-firmware/tree/xlnx_rel_v2024.1

Appendix A

Users' Manual

A.1 Download Vivado and Vitis

The Xilinx Vivado Design Suite and Vitis Unified Software Platform are key tools for developing, programming, and deploying embedded systems on Xilinx hardware platforms.

Vivado Design Suite is used for hardware design, including creating and managing FPGA configurations, generating bitstreams, and programming FPGA devices. In the context of secure boot, you will use Vivado to configure the hardware and possibly create boot images.

Vitis Unified Software Platform is the software development environment for embedded applications on Xilinx FPGAs and SoCs. It integrates seamlessly with Vivado, allowing you to develop software applications while leveraging Vivado's hardware tools. Vitis also supports features needed for secure and measured boot, making it essential for your project on the ZCU104.

To download Vivado and Vitis, follow these steps:

1. **Step 1**: register for and Log into an Xilinx (AMD) Account Go to the AMD Xilinx website. Click on Sign In in the top right. If you don't already have a Xilinx account, select Create Account and fill out the registration information. After creating your account, log in to access the download resources.
2. **Step 2**: navigate to the Download Page [22]
3. **Step 3**: choose Vitis for Software Developers (SW Developer) Select Vitis under the available options, as it includes both the Vitis platform and Vivado. Vitis installation allows you to include Vivado and other essential tools, simplifying the installation of all necessary components. For a secure boot project on the ZCU104, ensure you select compatible versions that meet the specific requirements of your board.
4. **Step 4**: select Your Desired Version. You will see a list of Vitis versions. For the installation on Windows, I used the "AMD Unified Installer for FPGAs And Adaptive SoCs 2024.1: Windows Self-Extracting Web Installer" to download and install both Vitis and Vivado, along with any other necessary components.
5. **Step 5**: download the Installation File. Click Download next to the installer for your operating system (Windows or Linux). Once the download is complete, locate the installer file and run it.
6. **Step 6**: install Vitis, Vivado, and Additional Tools. During the installation process, the setup wizard will guide you through various options. Select Vivado Design Suite when prompted, along with other software tools relevant to your project, such as the Vitis AI libraries or PetaLinux (if your project requires them). The installer will check for any necessary dependencies and guide you through configuring the installation options based on your selections.

7. **Step 7**: complete the Installation. Follow the remaining instructions provided by the setup wizard, including license management if necessary. Ensure that the installation path meets any system requirements, especially on Linux where specific paths may be needed. After installation, you should have access to both Vivado and Vitis, ready for use with your secure and measured boot demo on the ZCU104. With these tools, you can begin configuring, developing, and testing your project's hardware and software elements.

A.2 Steps to Generate a '.xsa' file for the ZCU104 Board using Vivado

Here's a step-by-step guide on how to generate an '.xsa' (Xilinx Support Archive) file from Vivado for ZCU104 board. This file is essential for integrating hardware and software development in Vitis, as it contains hardware configuration data for the ZCU104.

1. **Open Vivado and Set Up a New Project**
   - **Launch Vivado**: open Vivado on your computer.
   - **Create a New Project**: from the Vivado start page, click on "Create New Project". Choose a project name and select a suitable location to save the project. Ensure the "Project Type" is set to "RTL Project", and tick the box to "Do not specify sources at this time".
   - **Select the ZCU104 Board**: when prompted to select the board, go to the "Boards" tab. In the list, search for and select "ZCU104". This automatically configures the project for the Zynq UltraScale+ MPSoC on the ZCU104.
2. **Define the Block Design**
   - **Create a Block Design**: go to the "Flow Navigator" on the left side of Vivado and select "Create Block Design". Give the block design a name, e.g., "zcu104_design".
   - **Add the Processing System**: in the Block Design view, click on "Add IP" and search for "Zynq UltraScale+ MPSoC". Double-click to add it to the design.
   - **Run Block Automation**: with the Zynq MPSoC block selected, click on "Run Block Automation" in the green bar. Accept the default settings provided by Vivado for the ZCU104. This configures the processing system with default parameters for the ZCU104 board, connecting essential components like DDR memory and clocks.
   - **Customize the Design (Optional)**: if the project has specific hardware requirements (e.g., additional IP cores, custom peripherals), add them to the block design now. Customize the connections as necessary.
3. **Validate and Generate the Design**
   - **Validate the Block Design**: after completing the design, click on "Validate Design" in the toolbar to check for any errors in the configuration. Resolve any warnings or errors that may appear.
   - **Create the HDL Wrapper**: go back to the "Sources" pane, right-click on your block design file, and select "Create HDL Wrapper". Choose the option "Let Vivado manage wrapper and auto-update".
4. **Generate the Bitstream**
   - **Run Synthesis**: from the "Flow Navigator", select "Run Synthesis". This process may take some time.
   - **Run Implementation**: after synthesis completes, select "Run Implementation". This further processes the design and prepares it for bitstream generation.

- **Generate Bitstream**: once implementation is finished, click "Generate Bitstream" in the Flow Navigator. Wait for the process to complete. You should now have a bitstream file generated for your design.

5. **Export the Hardware and Generate the '.xsa' File**

   - **Export Hardware**: after bitstream generation is complete, go to $File \rightarrow Export \rightarrow ExportHardware$. In the dialog box, ensure that the option "Include Bitstream" is selected, so the bitstream is packaged with the hardware design.
   - **Save the '.xsa' File**: choose a location to save your '.xsa' file, which will contain the complete hardware specification. Click "OK" to export the hardware, generating the '.xsa' file.

This '.xsa' file can now be used in Vitis as a hardware platform, enabling you to develop software applications that run on your custom hardware configuration for the ZCU104 board.

Pre-exisiting '.xsa'

Another option for obtaining the '.xsa' file is to use a pre-existing '.xsa' file from another project that matches your hardware and design requirements for the ZCU104. If there is already a project available with a compatible hardware configuration, this '.xsa' file can save you significant time by avoiding the need to set up and configure a new Vivado project from scratch. You simply need to make sure that the '.xsa' file from the other project aligns closely with your current needs, including any custom IP or peripheral configurations you plan to use. Once you have a suitable '.xsa' file, you can directly import it into Vitis as a hardware platform, allowing you to move straight to the software development and testing phases. This approach is particularly useful for rapid prototyping or if your project closely follows an existing setup with minimal modifications.

A.3 Steps to Generate '.elf' files for the ZCU104 Board using Vitis Unified IDE

Here's a detailed explanation of the process for generating 'fsbl.elf' and 'pmufw.elf' in Vitis, using the settings and options specific to your project for the ZCU104 board.

1. **Launch Vitis and Create a New Platform Component**

   - **Open Vitis**: start by launching the Vitis IDE on your computer.
   - **Start a New Platform Component**: go to the top menu, and select $File \rightarrow NewComponent \rightarrow Platform$ to create a new platform component. This component will include essential boot files, including the FSBL and PMU firmware.
   - **Define the Component Name and Location**: enter a "Component Name" that describes your platform (e.g., 'zcu104_platform'). For "Component Location", specify the directory where Vitis will store the platform component. By default, this is typically set to the workspace defined during the installation, but you may specify a custom location if desired. Click "Next" to proceed.

2. **Specify the Hardware Design**

   - **Add the Hardware Design File ('.xsa')**: in the "Hardware Design" step, you'll need to select the '.xsa' file that defines your board's hardware configuration. Click "Browse" and navigate to the location of the '.xsa' file, either the one you generated in Vivado or a pre-existing '.xsa' that matches the ZCU104 board requirements. Select the '.xsa' file and ensure it is correctly loaded into the project. Click "Next" to continue.

3. **Select the Operating System and Processor**

- **Verify OS and Processor Selections**: based on the '.xsa' file, Vitis will automatically select the operating system and processor for the platform. The "Operating System" should automatically be set to "Standalone". The "Processor" field should be set to "psu_cortexa53_0" by default. Confirm these settings before moving on.

4. **Enable Boot Artifacts Generation**

   - **Generate Boot Artifacts**: ensure the "Generate Boot Artifacts" checkbox is selected. This is essential as it enables the automatic generation of the 'fsbl.elf' (First Stage Boot Loader) and 'pmufw.elf' (Platform Management Unit Firmware) files when you build the platform. These files are required for the boot process on the ZCU104 and will be included in the boot image.

5. **Build the Platform**

   - **Build the Platform**: with all settings configured, proceed to build the platform. Right-click on the platform component in the "Explorer" pane and select "Build Project". Vitis will start building the platform, including the generation of the FSBL and PMU firmware.
   - **Verify Generated Files**: if the build completes successfully, Vitis will generate the 'fsbl.el'' and 'pmufw.elf' files and place them in the designated directories within the platform component's build folders: The 'fsbl.elf' file can be found at: "Vitis_Workspace/platform_name/zynqmp_fsbl/build" The 'pmufw.elf' file will be located at: "Vitis_Workspace/platform_name/zynqmp_pmufw/build"

These ELF files are now ready for use in the boot image (BOOT.BIN) for your ZCU104. With these files, you can proceed to create the boot image using either the "Boot Image Creation" tool in Vitis or the "Bootgen" utility if additional configuration is required.

A.4 How to build OP-TEE project

Here's a detailed guide for building OP-TEE with examples, Linux Kernel, and Rootfs for the Xilinx ZCU104 board.

Prerequisities

Ensure you have the following software installed on your Ubuntu machine: git, make, gcc, build-essential, curl, python3, python3-pip, ccache, libssl-dev, libusb-1.0-0-dev, device-tree-compiler, bison, flex, ninja-build, rsync, unzip, git-repo.

You can install the necessary dependencies using:

```
sudo apt-get update
sudo apt-get install -y \
  adb acpica-tools autoconf automake bc bison build-essential \
  ccache cpio cscope curl device-tree-compiler e2tools expect \
  fastboot flex ftp-upload gdisk git libattr1-dev libcap-ng-dev \
  libfdt-dev libftdi-dev libglib2.0-dev libgmp3-dev libhidapi-dev \
  libmpc-dev libncurses5-dev libpixman-1-dev libslirp-dev \
  libssl-dev libtool libusb-1.0-0-dev make mtools ncat \
  ninja-build python3-cryptography python3-pip python3-pyelftools \
  python3-serial python-is-python3 rsync swig unzip uuid-dev \
  wget xdg-utils xterm xz-utils zlib1g-dev
```

Step 1: Set Up the OP-TEE Environment

Install repo tool: repo is a tool used to manage multiple Git repositories. Download and install it as follows:

```
curl https://storage.googleapis.com/git-repo-downloads/repo > /bin/repo
chmod a+x /bin/repo
```

Create the OP-TEE directory: set up a directory for OP-TEE where you will store the source code:

```
mkdir -p /optee
cd /optee
```

Initialize the repo: OP-TEE uses the repo tool to manage its repositories. To initialise the repo, run:

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
repo init -b 4.3.0 -u https://github.com/OP-TEE/manifest.git -m zynqmp.xml
```

Sync the repositories: after initialising the repo, sync it to download the source code:

```
repo sync -j $(nproc)
```

Check out a specific commit: you will need a specific commit for ARM Trusted Firmware (ATF). To do this, run:

```
git -C arm-trusted-firmware checkout 04013814718e870261f27256216cd7da3eda6a5d
```

This ensures you're working with a known, stable commit of the ATF.

Step 2: Build the Toolchains

Build the toolchain: before building OP-TEE itself, you need to compile the required toolchains. This can be done using the following command:

```
make -j 2 toolchains
```

This command will build the toolchain with 2 parallel jobs (-j 2), which should be sufficient for building the toolchains.

Step 3: Build OP-TEE Image

Build the OP-TEE image: now that the toolchains are ready, you can build the OP-TEE image for the ZCU104 board. To build the OP-TEE image for the ZCU104 board, you'll use the following command:

```
MEASURED_BOOT=y DEBUG=1 make -j $(nproc) PLATFORM=zynqmp-zcu104 all image
```

This Makefile command is tailored to build all the essential components needed for OP-TEE to run on the ZCU104 board, including the ARM Trusted Firmware (TF-A), U-Boot bootloader, OP-TEE OS, and the Linux kernel, along with any additional components configured for a measured boot.

Below is a step-by-step description of how this command interacts with the Makefile to complete the build:

1. **Measured Boot and Debug Options**:

- **MEASURED_BOOT=y**: this flag activates measured boot, a critical security feature that verifies boot integrity by generating cryptographic hashes of each boot stage and storing these as measurements. When set, the Makefile includes additional steps to configure TF-A and OP-TEE with measured boot options, ensuring that each stage of the boot process is verified, contributing to a secure boot chain.
- **DEBUG=1**: this flag enables debug mode, which includes debugging symbols and detailed logging to facilitate troubleshooting. This is particularly useful during the development and testing phases for identifying issues within the secure and non-secure components.

2. **Parallel Build Process**: '-j $(nproc)' tells the 'make' command to run multiple jobs concurrently, with '$(nproc)' setting the number of parallel jobs based on the number of cores available on your system. This speeds up the build significantly by compiling multiple components at once.

3. **Platform Selection**: PLATFORM=zynqmp-zcu104 specifies the target platform as the Zynq UltraScale+ MPSoC (ZCU104), which informs the Makefile to apply specific configuration settings and device trees unique to this hardware.

4. **All Image Target**: the 'all' target compiles all components, while 'image' generates the final boot image files. These include binaries for TF-A, U-Boot, and OP-TEE, packaged for deployment to the ZCU104.

The Makefile itself is organised to facilitate these processes, with specific sections for each component and functionality:

1. **ARM Trusted Firmware (TF-A)**: the Makefile includes settings to build TF-A, specifically configured to support OP-TEE's secure world functions on the ZynqMP. Additional flags, such as 'SPD=opteed', integrate OP-TEE as the secure payload. When the measured boot is enabled, additional parameters are included to log events and use a trusted certificate chain for TF-A (in this setup, certain options related to measured boot become redundant since the measured boot process is implemented in the FSBL rather than in the TF-A).

2. **OP-TEE OS**: OP-TEE is configured and built in this section, with parameters enabling TPM event logging *CFG_CORE_TPM_EVENT_LOG=y* when the measured boot is active. This ensures that OP-TEE can interact with the boot measurements to support secure services in the secure world.

3. **U-Boot**: U-Boot is configured to load the appropriate device trees and system parameters for ZynqMP and then compiled as the board's secondary bootloader. The Makefile provides the 'U-BOOT_DTS' variable to select the device tree suited for the ZCU104, ensuring correct hardware setup.

4. **Device Trees (dtbo)**: the device tree overlays are configured and built for ZynqMP, with 'dtbo' compiling the device tree binaries used by OP-TEE and Linux for initialising hardware peripherals on the ZCU104.

5. **Linux Kernel**: the Linux kernel is configured with specific definitions for the ZynqMP architecture, supporting both standard and debug configurations, and is then compiled.

6. **Buildroot Configuration**: Buildroot is configured for creating the root filesystem, providing minimal packages that support OP-TEE on the ZCU104.

7. **Image Generation**: the 'bootimage' and 'fitimage' targets generate the bootable images (BOOT.bin and fitImage) that consolidate all compiled components, TF-A, OP-TEE OS, U-Boot, and the kernel, into the final image structure needed for deployment on the ZCU104.

The make command provided above executes this entire workflow within the Makefile, building a comprehensive and secure image for running OP-TEE with measured boot functionality on the ZCU104. Each section contributes to the robust setup of the ZynqMP platform, aligning

hardware configuration, bootloader settings, and OP-TEE integration to enable a secure operating environment. The 'zynqmp.mk' Makefile orchestrates the compilation and assembly of all necessary components for the ZCU104 platform, ensuring a streamlined build process for OP-TEE and its associated elements.

Note: the prebuilt firmware from Xilinx can be found at the following link [23]. This is the repository referenced by the 'zynqmp.mk' Makefile for the ZCU104 platform. However, in our case, we are using custom FSBL and PMUFW instead of the precompiled versions, as we have made specific modifications to these components for our project.

Step 4: Verify the Build Output

Check the output: after the build completes, verify that the following files have been generated in the /optee/build/zynqmp directory:

- BOOT.bin
- zynqmp-zcu104.ub

These files are necessary for booting the ZCU104 board with OP-TEE.

Step 5: Prepare SD Card for ZCU104

Before copying the required files to the SD card, we need to format it and set up a partition.

Format and partition the SD Card following this steps:

1. Open a terminal on your Linux machine and run the following command to start the partitioning tool:

   ```
   sudo fdisk /dev/sdb
   ```

   Here, /dev/sdb represents the SD card. Make sure you replace /dev/sdb with the correct device if your SD card is identified differently (e.g., /dev/sdc).

2. Delete any existing partitions on the SD card: press 'd' to delete an existing partition. If there are multiple partitions, repeat the command until all are deleted.

3. Create a new partition: press 'n' to create a new partition. Follow the prompts to create a new partition using the default values, which will use all the available space on the SD card.

4. Set the partition type: press 't' to change the partition type. Enter c to set the partition type to FAT32 (this is required for the ZCU104 bootloader).

5. Write the changes: press 'w' to write the changes to the disk and exit fdisk.

At this point, the SD card is partitioned with a single FAT32 partition, ready to be formatted.

Mount the partition following this steps:

1. After partitioning, we need to format the partition as FAT32. run the following command to format the partition:

   ```
   sudo mkfs.vfat /dev/sdb1
   ```

   Replace /dev/sdb1 with the correct partition if it's different.

2. Now, we need to mount the SD card's partition so that we can copy the necessary files to it. Create a mount point (directory) to mount the SD card:

   ```
   sudo mkdir -p /mnt/images/
   ```

3. Mount the SD card partition to the mount point:

   ```
   sudo mount /dev/sdb1 /mnt/images/
   ```

4. Now that the SD card is mounted, we can copy the required boot files onto it. Copy the necessary files (e.g., BOOT.bin, zynqmp-zcu104.ub) to the SD card:

   ```
   sudo cp sd-images/BOOT.bin sd-images/zynqmp-zcu104.ub /mnt/images/
   ```

   Make sure the source paths (sd-images/BOOT.bin and sd-images/zynqmp-zcu104.ub) point to the correct files.

5. Once the files are copied, it's important to safely unmount the SD card to avoid any corruption. Unmount the SD card:

   ```
   sudo umount /mnt/images
   ```

   At this point, the SD card is ready to be used with your ZCU104 board.

Ensure the boot mode on the ZCU104 board is set to boot from the SD card. Set the switches on the board according to the manufacturer's instructions (usually something like SW6 set to 1-ON, 2-OFF, 3-OFF, and 4-OFF).

This guide walks you through the entire process, from partitioning and formatting the SD card to copying the necessary boot files. After completing these steps, you should be able to boot the ZCU104 board using the SD card.

Step 6: Boot the ZCU104 Board using PuTTY

After preparing the SD card, you can now boot the ZCU104 board and access it via a serial connection using PuTTY.

Follow these steps to correctly boot the board:

1. **Connect the ZCU104 to Your PC**: use a USB-to-serial adapter or the board's on-board USB-serial interface to connect the ZCU104 to your PC. Ensure the USB cable is securely connected between the ZCU104 and your computer.

2. **Find the Serial Port**: open a terminal on your PC and check which serial port is assigned to the ZCU104. You can typically find this by running:

   ```
   ls /dev/tty*
   ```

   Look for the entry that corresponds to the connected device, which may appear as something like /dev/ttyUSB0 or /dev/ttyACM0.

3. **Open PuTTY**: open PuTTY on your computer. If you don't have PuTTY installed, you can use the following command:

   ```
   sudo apt update
   sudo apt install putty
   ```

   This will install PuTTY and its related components on your system.

4. **Configure PuTTY for Serial Communication**: in the PuTTY configuration window A.1:

   - Select Serial as the connection type.
   - In the Serial line field, enter the correct serial port (e.g., /dev/ttyUSB1 in my case).
   - Set the Speed (baud rate) to 115200.
   - Ensure the Data bits is set to 8, Stop bits is set to 1, and Parity is set to None.
   - Set Flow control to None.

5. **Start the Serial Connection**: click "Open" to start the serial session. This will open a terminal window where you should see the boot messages from the ZCU104 board.

6. **Stop the Boot Process**: as soon as the board starts booting, you will see the message:

   ```
   Hit any key to stop autoboot: 2
   ```

   Press any key (e.g., space bar) within the 2-second window to stop the autoboot process and gain control over the boot loader.

7. **Load the Boot Image Manually**: once you have stopped the autoboot, you can manually load the boot image from the SD card by entering the following commands at the ZynqMP prompt:

   ```
   ZynqMP> fatload mmc 0 0x30000000 zynqmp-zcu104.ub
   ZynqMP> bootm 0x30000000
   ```

   These commands will load the zynqmp-zcu104.ub file from the SD card into memory and then boot the system.

8. **Login to the System**: once the system has booted, you should see a login prompt on the terminal window. Login using the default credentials (for example, root as both the username and password).

At this point, the ZCU104 board should be successfully booted from the SD card, and you can begin using the system or testing your demo.

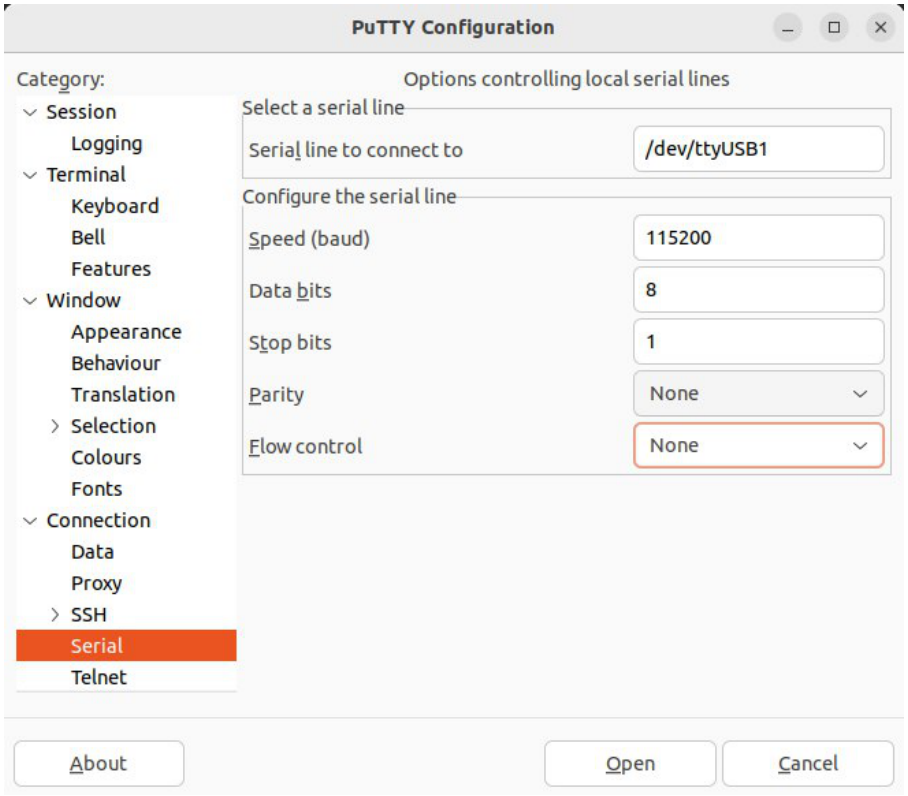

Figure A.1. PuTTY configuration.

Step 7: Run OP-TEE Example

After logging in, you can test OP-TEE by running an example. For example:

```
optee_example_hello_world
```

This will confirm that OP-TEE is running correctly on the ZCU104 board. The optee_example_hello_world command runs a simple example application that demonstrates basic functionality of OP-TEE on supported hardware like the ZCU104 board. Specifically, it shows how a trusted application (TA) in the Secure World communicates with a client application in the Normal World.

A.5 How to run TPM_LOG_TEST

To set up and test the retrieval of the event log in the OP-TEE environment on the ZCU104 platform, I configured a specific test using 'xtest 1024' and made some critical adjustments to both secure memory allocation and test invocation. Here's an organised breakdown of the steps and changes.

Setting Up Secure Memory for the Event Log

In 'optee_os/core/kernel/tpm.c', I defined two essential configuration constants to manage the storage and size of the TPM event log:

```
#define CFG_TPM_MAX_LOG_SIZE 1024
#define CFG_TPM_LOG_BASE_ADDR 0X70000000
```

- 'CFG_TPM_MAX_LOG_SIZE' specifies the maximum allowable size for the TPM event log, set here to 1024 bytes.
- 'CFG_TPM_LOG_BASE_ADDR' is the base address where the event log is stored. This address is located within the secure memory region at '0x70000000', which provides protection and limits access to the log, ensuring that only secure applications in OP-TEE can read it.

Using secure memory helps protect the log against unauthorised access, maintaining the integrity and confidentiality of the stored events, especially critical for measured boot and other sensitive operations.

Creating the Test in 'xtest' Framework

To invoke and test the event log retrieval functionality, I used a specific test case using 'xtest', labeled 'xtest 1024'. The test code is located in 'optee_test/xtest/regression_1000.c' and is activated only if 'CFG_CORE_TPM_EVENT_LOG' is defined. Here's the relevant section of the code that defines 'xtest_tee_test_1024':

```
#ifdef CFG_CORE_TPM_EVENT_LOG
static void xtest_tee_test_1024(ADBG_Case_t *c)
{
    TEEC_Session session = {};
    uint32_t ret_orig = 0;

    xtest_teec_open_session(&session, &tpm_log_test_ta_uuid, NULL, &ret_orig);

    // Test retrieving the TPM event log
    Do_ADBG_BeginSubCase(c, "TPM test service invocation");
    ADBG_EXPECT_TEEC_SUCCESS(c, TEEC_InvokeCommand(&session,
                                TA_TPM_TEST_GET_LOG,
                                NULL, &ret_orig));
    Do_ADBG_EndSubCase(c, "TPM test service invocation");
```

```
    // Test handling a short buffer scenario
    Do_ADBG_BeginSubCase(c, "TPM test passing short buffer");
    ADBG_EXPECT_TEEC_SUCCESS(c, TEEC_InvokeCommand(&session,
                                TA_TPM_TEST_SHORT_BUF,
                                NULL, &ret_orig));
    Do_ADBG_EndSubCase(c, "TPM test passing short buffer");

    TEEC_CloseSession(&session);
}
ADBG_CASE_DEFINE(regression, 1024, xtest_tee_test_1024, "Test
    PTA_SYSTEM_GET_TPM_EVENT_LOG Service");
#endif /* CFG_CORE_TPM_EVENT_LOG */
```

- **Session Initialization**: we open a session using 'xtest_teec_open_session' with the 'tpm_log_test_ta_uuid' (the UUID identifying the Trusted Application for this test).
- **Log Retrieval**: we invoke 'TA_TPM_TEST_GET_LOG', which calls the function in the Trusted Application that retrieves the TPM event log.
- **Short Buffer Test**: we also test 'TA_TPM_TEST_SHORT_BUF', which intentionally uses a short buffer to test the function's behaviour with insufficient space for the event log.

TA Implementation

The TA side handles the actual commands to retrieve and interact with the event log. This code is found in 'optee_test/ta/tpm_log_test/ta_entry.c', specifically in the function 'TA_InvokeCommandEntryPoint', which directs the commands to appropriate handlers:

```
/* Called when a command is invoked */
TEE_Result TA_InvokeCommandEntryPoint(void *pSessionContext __unused,
                                      uint32_t nCommandID,
                                      uint32_t nParamTypes __unused,
                                      TEE_Param pParams[4] __unused)
{
    DMSG("Entering TA_InvokeCommandEntryPoint");
    switch (nCommandID) {
    case TA_TPM_TEST_GET_LOG:
        return test_with_right_parameters();

    case TA_TPM_TEST_SHORT_BUF:
        return test_short_buffer();

    default:
        return TEE_ERROR_BAD_PARAMETERS;
    }
}
```

In this code:

- 'TA_TPM_TEST_GET_LOG' invokes the 'test_with_right_parameters()' function, which calls 'invoke_system_pta' to retrieve the TPM event log using 'PTA_SYSTEM_GET_TPM_EVENT_LOG'.
- 'TA_TPM_TEST_SHORT_BUF' invokes 'test_short_buffer()', which tests the behaviour when the buffer is too short.

Retrieving the TPM Event Log

The function 'test_with_right_parameters()' is responsible for retrieving the TPM event log. This function includes a call to 'invoke_system_pta' with 'PTA_SYSTEM_GET_TPM_EVENT_LOG', as shown below:

```
if (invoke_system_pta(PTA_SYSTEM_GET_TPM_EVENT_LOG, param_types, params) ==
    TEE_SUCCESS) {
    DMSG("Received %i bytes of event log", params[0].memref.size);
```

- 'invoke_system_pta': this function interacts with the OP-TEE System PTA (Pseudo-Trusted Application) to retrieve the event log from secure memory.
- Event Log Retrieval Confirmation: upon successful retrieval, the log size is printed to confirm the number of bytes received, helping verify that the event log is being accessed as expected.

In summary, the test 'xtest 1024' validates the ability to open a session, retrieve the event log, and handle scenarios where the buffer is too short. By executing this test, I can confirm the secure memory configuration and ensure that the Trusted Application is correctly fetching and displaying the event log.

Appendix B

Developers' Manual

B.1 Enabling Measured Boot in FSBL

In this section, we describe the steps to enable measured boot functionality within the FSBL code. Measured boot involves securely recording and verifying the integrity of the boot process by capturing digests of bootloader components and storing them in an event log. This helps ensure that only trusted software is executed on the platform and provides a mechanism for validating the boot chain during system startup. To enable measured boot, specific modifications need to be made in the FSBL configuration file. These changes activate the necessary features for logging hash values (e.g., SHA3-384 digests) and capturing event data during the boot process. This data will be stored in a dedicated event log for later validation.

B.1.1 Added Files: "fsbl_measured_boot.c" and "fsbl_measured_boot.h"

fsbl_measured_boot.c

It contains subroutines to measure the CSU ROM, FSBL, and all non-PL partitions, establishing the foundational measurements for the initial boot stages and system integrity verification.

1. **Tpm_Measure_Rom Function**: this function measures the ROM digest and records it in the event log. The digest is read, reordered to little-endian format, and then logged with metadata. It initializes the event log and writes the ROM digest as an event.

```
int Tpm_Measure_Rom () {
   uint8_t data[48];
   uint32_t temp;
   uint8_t *temp_ptr = (uint8_t *) &temp;
   int i, j;
   int Status = XFSBL_FAILURE;

   for(i=0;i<48;i+=4) {
          temp = Xil_In32((UINTPTR) (CSU_ROM_DIGEST_ADDR+44-i));
          for(j=0;j<4;j++) {
                 data[i + j] = temp_ptr[j];
          }
   }

   XFsbl_PrintArray(DEBUG_INFO, data, XFSBL_HASH_TYPE_SHA3, "ROM
       Digest:");

   event_log_init(EVENT_LOG_START, EVENT_LOG_END);
```

```
    log_ptr2 = event_log_write_header(log_ptr2);

    event_log_metadata_t metadata;
    metadata.id = 0;
    metadata.name = "ROM";
    metadata.pcr = 0;

    log_ptr2 = event_log_record(data, EV_POST_CODE, &metadata, log_ptr2);
    return XFSBL_SUCCESS;
}
```

2. **Tpm_Measure_Fsbl Function**: this function measures the FSBL digest and records it in the event log. It calculates the digest of the FSBL, logs it, and prints the SHA-3 digest for debugging purposes.

```
int Tpm_Measure_Fsbl (XFsblPs_ImageHeader * ImageHeader) {
    int Status;
    u8 PartitionDigest[XFSBL_HASH_TYPE_SHA3] __attribute__ ((aligned
        (4))) = {0};
    XFsblPs_PartitionHeader * PartitionHeader =
        &ImageHeader->PartitionHeader[0];
    PTRSIZE LoadAddress =
        (PTRSIZE)PartitionHeader->DestinationLoadAddress;
    uint32_t SizeInBytes = PartitionHeader->UnEncryptedDataWordLength <<
        2;

    XFsbl_ShaStart((void * )NULL, XFSBL_HASH_TYPE_SHA3);
    XFsbl_ShaDigest((uint8_t *) LoadAddress, SizeInBytes,
        PartitionDigest, XFSBL_HASH_TYPE_SHA3);
    XFsbl_PrintArray(DEBUG_DETAILED, PartitionDigest,
        XFSBL_HASH_TYPE_SHA3, "FSBL SHA3-384 DIGEST");

    event_log_metadata_t metadata;
    metadata.id = 1;
    metadata.name = "FSBL";
    metadata.pcr = 0;

    log_ptr2 = event_log_record(PartitionDigest, EV_POST_CODE,
        &metadata, log_ptr2);

    return XFSBL_SUCCESS;
}
```

3. **Tpm_Measure_Partition Function**: this function measures and records the digest for each partition, such as the ATF, U-Boot, and OP-TEE. Each partition's digest is calculated using SHA-3, and after the final partition, the event log is printed for debugging purposes.

```
int Tpm_Measure_Partition(XFsblPs * FsblInstancePtr, uint32_t
    PartitionNum, PTRSIZE LoadAddress) {
    int Status = XFSBL_SUCCESS;
    u8 PartitionDigest[XFSBL_HASH_TYPE_SHA3] __attribute__ ((aligned
        (4))) = {0};
    XFsblPs_PartitionHeader *PartitionHeader =
        &FsblInstancePtr->ImageHeader.PartitionHeader[PartitionNum];
    uint32_t SizeInBytes = PartitionHeader->UnEncryptedDataWordLength <<
        2;

    XFsbl_ShaStart((void * )NULL, XFSBL_HASH_TYPE_SHA3);
```

```
        XFsbl_ShaDigest((uint8_t *) LoadAddress, SizeInBytes,
            PartitionDigest, XFSBL_HASH_TYPE_SHA3);

        XFsbl_PrintArray(DEBUG_DETAILED, PartitionDigest,
            XFSBL_HASH_TYPE_SHA3, "PARTITION SHA3-384 DIGEST");

        event_log_metadata_t metadata;
        if(partition == 1) {
            metadata.id = 2;
            metadata.name = "Partition 1 - bl31";
            metadata.pcr = 0;
            log_ptr2 = event_log_record(PartitionDigest, EV_POST_CODE,
                &metadata, log_ptr2);
            partition += 1;
        } else if(partition == 2) {
            metadata.id = 3;
            metadata.name = "Partition 2 - U-BOOT";
            metadata.pcr = 0;
            log_ptr2 = event_log_record(PartitionDigest, EV_POST_CODE,
                &metadata, log_ptr2);
            partition += 1;
        } else if(partition == 3) {
            metadata.id = 4;
            metadata.name = "Partition 3 - OP-TEE";
            metadata.pcr = 0;
            log_ptr2 = event_log_record(PartitionDigest, EV_POST_CODE,
                &metadata, log_ptr2);
            partition += 1;

            // Print event log
            uint8_t *start = EVENT_LOG_START;
            log_ptr_end = log_ptr2;
            size_t size = log_ptr_end - start;

            XFsbl_Printf(DEBUG_INFO, "Printing Event Log... starting at
                address %p\r\n", start);
            id_event_print(&start, &size);

            while (size != 0U) {
                event2_print(&start, &size);
            }
        }
        return XFSBL_SUCCESS;
    }
```

fsbl_measured_boot.h

It is the header file for "fsbl_measured_boot.c", defining necessary structures and function prototypes for conducting measurements on non-PL components.

B.1.2 Added Files: "fsbl_measured_pl.c" and "fsbl_measured_pl.h"

fsbl_measured_pl.c

This file provides the subroutines for measuring PL partitions, enabling integrity checks for any programmable logic (PL) data loaded during boot. We do not have a bitstream to load into the

PL; therefore, there is no PL partition. The key points for performing measurements of a PL partition are listed here:

1. **Defining the TPM Measurement Function**: this main function, 'Tpm_Measure_Pl', handles different PL measurement scenarios:

   - Unauthenticated and unencrypted PL data in DDR and DDR-less systems
   - Unauthenticated and encrypted PL data
   - Authenticated data (to be implemented in the future)

   The function accepts the FSBL instance, partition number, AES context, destination, and source as parameters.

```
s32 Tpm_Measure_Pl(const XFsblPs *FsblInstancePtr, uint32_t PartitionNum,
       XSecure_Aes *AesPtr, u8 * Destination, u8 *Source) {
    s32 Status = XST_SUCCESS;
    uint8_t PartitionDigest[XFSBL_HASH_TYPE_SHA3] __attribute__
        ((aligned (4))) = {0};
    const XFsblPs_PartitionHeader *PartitionHeader =
        &FsblInstancePtr->ImageHeader.PartitionHeader[PartitionNum];
    uint32_t SizeInWords = PartitionHeader->UnEncryptedDataWordLength;
    uint32_t SizeInBytes = PartitionHeader->UnEncryptedDataWordLength <<
        2;

    // Check if the bitstream is unencrypted
    if (XFsbl_IsEncrypted(PartitionHeader) != XIH_PH_ATTRB_ENCRYPTION) {
#ifdef XFSBL_PS_DDR
        // Handle unencrypted bitstream for DDR system
        Status = Pl_Measure_Unencrypted(SizeInWords, Source,
            PartitionDigest);
        if (Status != XFSBL_SUCCESS) { goto END; }
#else
        // Handle unencrypted bitstream for DDR-less system
        Status = Pl_Measure_Chunked(FsblInstancePtr, PartitionNum,
            PartitionDigest);
        if (Status != XFSBL_SUCCESS) { goto END; }
#endif
    } else {
        // Handle encrypted bitstream
        Status = Pl_Measure_Encrypted(AesPtr, Destination, Source,
            SizeInBytes, PartitionDigest);
        if (Status != XFSBL_SUCCESS) { goto END; }
    }
END:
    return XFSBL_SUCCESS;
}
```

2. **Measuring Unencrypted PL (DDR System)**: this function, 'Pl_Measure_Unencrypted', measures unencrypted PL data for DDR systems by using the CSU DMA to transfer the bitstream and computing a SHA3 hash.

```
u32 Pl_Measure_Unencrypted(uint32_t SizeInWords, uint8_t *Source,
    uint8_t *PartitionDigest) {
    uint32_t Status;

    // Initialize SHA3 engine
    XSecure_Sha3Initialize(&SecureSha3, &CsuDma);
    XSecure_Sha3Start(&SecureSha3);
```

```
    // Write data to PCAP and SHA3
    Status = Pl_Measure_WriteToPcap_with_SHA3(SizeInWords, Source);

    // Complete SHA3 hashing
    XSecure_Sha3Finish(&SecureSha3, PartitionDigest);

    return Status;
}
```

3. **Measuring Unencrypted PL (DDR-less System)**: 'Pl_Measure_Chunked' handles measuring unencrypted PL in DDR-less systems by chunking the bitstream and transferring it in sections while computing the SHA3 hash.

```
#ifndef XFSBL_PS_DDR
u32 Pl_Measure_Chunked(const XFsblPs *FsblInstancePtr, uint32_t
    PartitionNum, uint8_t *PartitionDigest) {
    uint32_t Status = XFSBL_SUCCESS;
    const XFsblPs_PartitionHeader *PartitionHeader =
        &FsblInstancePtr->ImageHeader.PartitionHeader[PartitionNum];
    uint32_t ChunkSize = 0U, RemainingBytes = 0U, BitStreamSizeWord =
        PartitionHeader->UnEncryptedDataWordLength;
    uint32_t BitStreamSizeByte = BitStreamSizeWord * 4, ImageOffset =
        FsblInstancePtr->ImageOffsetAddress;
    uint32_t StartAddrByte = ImageOffset + 4 *
        (PartitionHeader->DataWordOffset);

    XFsbl_Printf(DEBUG_GENERAL, "Starting chunked transfer...\r\n");

    // Initialize SHA3 engine
    XSecure_Sha3Initialize(&SecureSha3, &CsuDma);
    XSecure_Sha3Start(&SecureSha3);

    while (RemainingBytes > 0) {
        ChunkSize = (RemainingBytes >= READ_BUFFER_SIZE) ?
            READ_BUFFER_SIZE : RemainingBytes;
        Status = FsblInstancePtr->DeviceOps.DeviceCopy(StartAddrByte,
            (PTRSIZE)ReadBuffer, ChunkSize);
        if (XFSBL_SUCCESS != Status) { goto END; }

        Status = Pl_Measure_WriteToPcap_with_SHA3(ChunkSize/4,
            &ReadBuffer[0]);
        if (XFSBL_SUCCESS != Status) { goto END; }

        StartAddrByte += ChunkSize;
        RemainingBytes -= ChunkSize;
    }
END:
    XSecure_Sha3Finish(&SecureSha3, PartitionDigest);
    return Status;
}
#endif
```

4. **Measuring Encrypted PL**: 'Pl_Measure_Encrypted' handles measuring encrypted PL data. This function configures the AES engine for decryption, sets up the CSU DMA for transferring data, and calculates the SHA3 hash of the decrypted data.

```
u32 Pl_Measure_Encrypted(XSecure_Aes *AesPtr, uint8_t *Destination,
    uint8_t *Source, uint32_t Size, uint8_t *PartitionDigest) {
```

```
    uint32_t SssCfg = 0x0U;
    volatile s32 Status = XST_SUCCESS;
    uint32_t CurrentImgLen = 0x0U, NextBlkLen = 0x0U, PrevBlkLen = 0x0U;
    uint8_t *DestAddr = Destination, *SrcAddr = Source;
    uint8_t *GcmTagAddr = SrcAddr + XSECURE_SECURE_HDR_SIZE;

    // Set up SSS and initialize AES
    XSecure_SssSetup(SssCfg);
    XSecure_Sha3Initialize(&SecureSha3, &CsuDma);
    XSecure_Sha3Start(&SecureSha3);

    // Configure AES for decryption
    XSecure_WriteReg(AesPtr->BaseAddress, XSECURE_CSU_AES_CFG_OFFSET,
        XSECURE_CSU_AES_CFG_DEC);
    XSecure_AesReset(AesPtr);

    do {
        // Decrypt block and update image length
        Status = XSecure_AesDecryptBlk(AesPtr, DestAddr, SrcAddr,
            GcmTagAddr, NextBlkLen, BlockCnt);
        if (Status != XST_SUCCESS) { goto ENDF; }

        NextBlkLen = Xil_Htonl(XSecure_ReadReg(AesPtr->BaseAddress,
            XSECURE_CSU_AES_IV_3_OFFSET)) * 4;
        CurrentImgLen += NextBlkLen;

        SrcAddr = (GcmTagAddr + XSECURE_SECURE_GCM_TAG_SIZE);
    } while (NextBlkLen != 0 && CurrentImgLen <= Size);

ENDF:
    XSecure_Sha3Finish(&SecureSha3, PartitionDigest);
    return Status;
}
```

Each of these functions serves a different case for measuring and validating PL partitions. Whether the bitstream is encrypted, unencrypted, in a DDR system, or DDR-less system, the appropriate function is called to compute the integrity of the PL data.

fsbl_measured_pl.h

The header file for "fsbl_measured_pl.c", contains declarations and structures essential for performing PL partition measurements.

B.1.3 Added Files: "fsbl_measured_utils.c" and "fsbl_measured_utils.h"

fsbl_measured_utils.c

Contains essential low-level subroutines for performing measurements, including functions like Tpm_ReadPcr, Tpm_Event, and SHA3 cryptographic routines. We are not using this file in our implementation because we do not have a TPM; instead, we save the measurements in an event log.

fsbl_measured_utils.h

The header file for "fsbl_measured_utils.c", defines function prototypes and structures needed for low-level cryptographic and measurement operations.

B.1.4 Changes in "xfsbl_config.h"

The following code snippet from the "xfsbl_config.h" file enables detailed debugging for measured boot and configures the event log functionality:

Line 82: MEASURED_BOOT macro defines the support for measured boot. When this option is enabled, it ensures that the code responsible for logging the boot events and measurements is included in the FSBL. This code will record hash values of the bootloader and other relevant components, storing them in an event log for later analysis. FSBL_DEBUG_DETAILED_VAL is set to 1U to enable detailed debugging, which will print all logged events and event log values to the console. This is essential for validating the measured boot process, ensuring that all the logged measurements and events are captured and available for review. The other debug flags are set to 0U to suppress unnecessary debug information and focus on the detailed logs related to the event recording.

```
#define MEASURED_BOOT
#ifdef MEASURED_BOOT
    #define FSBL_PRINT_VAL (0U)
    #define FSBL_DEBUG_VAL (0U)
    #define FSBL_DEBUG_INFO_VAL (0U)
    #define FSBL_DEBUG_DETAILED_VAL (1U)
#else
    #define FSBL_PRINT_VAL (1U)
    #define FSBL_DEBUG_VAL (0U)
    #define FSBL_DEBUG_INFO_VAL (0U)
    #define FSBL_DEBUG_DETAILED_VAL (0U)
#endif
```

B.1.5 Changes in "xfsbl_inizialization.c"

The following code snippets are part of the modifications made to enable measured boot functionality in the FSBL. These changes involve conditional compilation blocks that ensure the inclusion of measured boot-related code only when the 'MEASURED_BOOT' macro is defined. Here is a brief overview of the modifications:

1. **Line 83**: this modification ensures that the necessary header file for measured boot "fsbl_measured_boot.h" is included when the 'MEASURED_BOOT' macro is defined. It allows for the inclusion of specific functionality related to measured boot.

   ```
   #ifdef MEASURED_BOOT
       #include "fsbl_measured_boot.h"
   #endif
   ```

2. **Line 166**: this change renames the global variable used for initialization vectors 'Iv' to 'Global_FsblIv' when 'MEASURED_BOOT' is enabled. This avoids any conflicts with similarly named variables in the Xilsecure library.

   ```
   #ifdef MEASURED_BOOT
       extern u32 Global_FsblIv[XIH_BH_IV_LENGTH / 4U];
   #else
       extern u32 Iv[XIH_BH_IV_LENGTH / 4U];
   #endif
   ```

3. **Line 447**: when 'MEASURED_BOOT' is defined, this block ensures the CSU ROM is measured.

   ```
   #ifdef MEASURED_BOOT
       Tpm_Measure_Rom();
   #endif
   ```

4. **Line 1406**: this modification measures the FSBL itself, ensuring that the FSBL code, up to the point where the image header table is loaded, is included in the measurement process. This guarantees that the FSBL code remains deterministic, which is a requirement for measured boot.

```
#ifdef MEASURED_BOOT
    Tpm_Measure_Fsbl(&FsblInstancePtr->ImageHeader);
#endif
```

These changes collectively enable the functionality needed for measured boot, ensuring that various components are measured and included in the overall boot process, providing a foundation for secure boot procedures.

B.1.6 Changes in "xfsbl_partition_load.c"

The following code snippets describe additional modifications to "xfsbl_partition_load.c" for implementing the measured boot functionality. Each change ensures that specific measured boot features are activated only when the MEASURED_BOOT macro is defined.

1. **Line 75**: this block includes necessary header files for measured boot. When MEASURED_BOOT is defined, both "fsbl_measured_boot.h" and "fsbl_measured_pl.h" are included to enable access to measured boot functions.

```
#ifdef MEASURED_BOOT
    #include "fsbl_measured_boot.h"
    #include "fsbl_measured_pl.h"
#endif
```

2. **Line 150**: here, the global variable *Iv* is renamed to *Global_FsblIv* under MEASURED_BOOT to prevent conflicts with the Xilsecure library.

```
#ifdef MEASURED_BOOT
    u32 Global_FsblIv[XIH_BH_IV_LENGTH / 4U] = { 0 };
#else
    u32 Iv[XIH_BH_IV_LENGTH / 4U] = { 0 };
#endif
```

3. **Line 1186**: for measured boot, the variable *BitstreamWordSize* is disabled when certain conditions are met, as it is not used in this context.

```
#if defined(XFSBL_BS) && defined(XFSBL_PS_DDR) && !defined(MEASURED_BOOT)
    u32 BitstreamWordSize;
#endif
```

4. **Line 1210**: this update ensures the correct global variable name *Global_FsblIv* for MEASURED_BOOT is used in memory copy operations.

```
#ifdef MEASURED_BOOT
    XFsbl_MemCpy(FsblIv, Global_FsblIv, XIH_BH_IV_LENGTH);
#else
    XFsbl_MemCpy(FsblIv, Iv, XIH_BH_IV_LENGTH);
#endif
```

5. **Line 1455**: this modification adds *PartitionNum* to the PL parameters in the measured boot context, enabling partition tracking in the measured boot process.

```
#ifdef MEASURED_BOOT
    PlParams.PartitionNum = PartitionNum;
#endif
```

6. **Line 1574** and **Line 1641**: these lines invoke "Tpm_Measure_Pl", a function that measures the programmable logic (PL) portion during loading.

```
#ifdef MEASURED_BOOT
    Status = (u32)Tpm_Measure_Pl(FsblInstancePtr, PartitionNum,
        &SecureAes,
            (u8 *) XFSBL_DESTINATION_PCAP_ADDR, (u8 *) LoadAddress);
#endif
```

7. **Line 1765**: this section measures authenticated and decrypted partitions that are not programmable logic (PL) partitions, adding additional verification during the boot process.

```
#ifdef MEASURED_BOOT
    if (DestinationDevice != XIH_PH_ATTRB_DEST_DEVICE_PL) {
        Status = Tpm_Measure_Partition(FsblInstancePtr, PartitionNum,
            LoadAddress);
        if (Status != XFSBL_SUCCESS) {
            Status = XFSBL_SLB9670_ERROR;
            goto END;
        }
    }
#endif
```

These modifications collectively integrate measured boot functionality into the FSBL, allowing it to track and verify different partitions and log significant events, contributing to a more secure boot process.

B.1.7 Changes in "xfsbl_plpartition_valid.c"

This file includes measurement functionality for authenticated PL partitions, recording their integrity for later verification.

1. **Line 67**: includes "xfsbl_config.h" to define the 'MEASURED_BOOT' macro and conditionally includes additional headers for the measured boot code.

```
#include "xfsbl_config.h"
#ifdef MEASURED_BOOT
    #include "fsbl_measured_boot.h"
    #include "fsbl_measured_utils.h"
#endif
```

2. **Line 597**: prints the SHA3-384 digest for the PCR Event, providing detailed debugging output when 'MEASURED_BOOT' is enabled.

```
#ifdef MEASURED_BOOT
    XFsbl_PrintArray (DEBUG_DETAILED, PartitionHash,
        XFSBL_HASH_TYPE_SHA3,
            "PL SHA3-384 DIGEST");
#endif
```

These modifications add necessary inclusions and detailed debugging for SHA3-384 digest calculations and PCR events, strengthening boot integrity monitoring within the FSBL.

B.1.8 Changes in "xfsbl_plpartition_valid.h"

Line 97: adds the partition number to the PL parameter data structure, ensuring that each PL partition's measurement is accurately associated with its specific partition.

```
typedef struct {
       u8 IsAuthenticated; /**< Authentication flag */
       u8 IsEncrypted;  /**< Encryption flag */
       u64 StartAddress; /** Start address of the partition */
       u32 UnEncryptLen; /**< un encrypted length of bitstream */
       u32 TotalLen;  /**< Total partition length */
       u32 ChunkSize;  /**< Chunk size */
       u8 *ChunkBuffer; /**< Buffer for storing chunk of data */
       XCsuDma *CsuDmaPtr; /**< Initialized CSUDMA driver's instance */
       u32 (*DeviceCopy) (u32 SrcAddress, UINTPTR DestAddress, u32 Length);
                          /**< Device copy for DDR less system */
       XFsblPs_PlEncryption PlEncrypt; /**< Encryption parameters */
       XFsblPs_PlAuthentication PlAuth;/**< Authentication parameters */
       u8 SecureHdr[XSECURE_SECURE_HDR_SIZE + XSECURE_SECURE_GCM_TAG_SIZE];
       u8 Hdr;
       XSecure_Sss SssInstance;
       u8 *Hash; /**< Pointer to store calculated hash */
   /* For MEASURED BOOT, add the PartitionNum to this structure */
#ifdef MEASURED_BOOT
       u32 PartitionNum;
#endif
} XFsblPs_PlPartition;
```

B.2 Enabling Event Logging in FSBL

B.2.1 Added files

event_log.c

This code provides functions for initializing and recording events in a Trusted Computing Group (TCG)-compliant event log, specifically for use with TPM measurements. It is designed to initialize a buffer for storing log events, define specification ID and locality events, and record measurements. The code utilizes the TPM 2.0 specification to format events, with support for the SHA-384 hashing algorithm.

Key functionalities include:

1. Setting up the event log memory buffer.
2. Writing initial events that conform to TCG standards.
3. Recording individual measurement events, which are used to ensure integrity verification in secure boot processes.

The main constants and structures involved are:

- 'TPM_ALG_ID', defining the hashing algorithm as SHA-384.
- 'id_event_header' and 'locality_event_header', which represent standardized TCG event structures.
- Functions like 'event_log_record', 'event_log_write_specid_event', and 'event_log_write_header', which handle event entry and ensure the data format aligns with TCG and TPM specifications.

1. **Event Log Record Function**: this function records a measurement event to the event log buffer. It takes a hash, event_type, metadata_ptr (event metadata), and log_ptr2 (current log pointer). Writes the event information, including the digest and event data, into the event log in a structured format. It also handles the length of the event name.

```
/* Record a measurement as a TCG_PCR_EVENT2 event */
uint8_t *event_log_record(const uint8_t *hash, uint32_t event_type,
         const event_log_metadata_t *metadata_ptr, uint8_t *log_ptr2)
{
    void *ptr = log_ptr2;
    uint32_t name_len = 0U;

    assert(hash != NULL);
    assert(metadata_ptr != NULL);

    if (metadata_ptr->name != NULL) {
        name_len = (uint32_t)strlen(metadata_ptr->name) + 1U;
    }

    ((event2_header_t *)ptr)->pcr_index = metadata_ptr->pcr;
    ((event2_header_t *)ptr)->event_type = event_type;
    ptr = (uint8_t *)ptr + offsetof(event2_header_t, digests);
    ((tpml_digest_values *)ptr)->count = HASH_ALG_COUNT;
    ptr = (uint8_t *)((uintptr_t)ptr +
           offsetof(tpml_digest_values, digests));
    ((tpmt_ha *)ptr)->algorithm_id = TPM_ALG_ID;
    ptr = (uint8_t *)((uintptr_t)ptr + offsetof(tpmt_ha, digest));
    (void)memcpy(ptr, (const void *)hash, TCG_DIGEST_SIZE);
    ptr = (uint8_t *)((uintptr_t)ptr + TCG_DIGEST_SIZE);
    ((event2_data_t *)ptr)->event_size = name_len;
    if (metadata_ptr->name != NULL) {
        (void)memcpy((void *)(((event2_data_t *)ptr)->event),
               (const void *)metadata_ptr->name, name_len);
    }
    log_ptr2 = (uint8_t *)((uintptr_t)ptr +
        offsetof(event2_data_t, event) + name_len);
    return log_ptr2;
}
```

2. **Event Log Buffer Initialization**: this function initializes the event log buffer, taking the start and finish addresses of the buffer as parameters. It ensures the finish address is greater than the start address and updates the log_end pointer.

3. **Event Log Initialization Function**: this function is a wrapper around the event_log_buf_init function to initialize the event log buffer with start and finish addresses.

4. **Write SpecID Event to the Log**: this function writes the SpecID event header to the event log. It copies the id_event_header into the log and updates specific fields like the algorithm ID and digest size.

```
uint8_t *event_log_write_specid_event(uint8_t *log_ptr2)
{
    void *ptr = log_ptr2;
    (void)memcpy(ptr, (const void *)&id_event_header,
           sizeof(id_event_header));
    ptr = (uint8_t *)((uintptr_t)ptr + sizeof(id_event_header));
    ((id_event_algorithm_size_t *)ptr)->algorithm_id = TPM_ALG_ID;
    ((id_event_algorithm_size_t *)ptr)->digest_size = TCG_DIGEST_SIZE;
    ptr = (uint8_t *)((uintptr_t)ptr +
        sizeof(id_event_algorithm_size_t));
```

```
    ((id_event_struct_data_t *)ptr)->vendor_info_size = 0;
    log_ptr2 = (uint8_t *)((uintptr_t)ptr +
           offsetof(id_event_struct_data_t, vendor_info));
    return log_ptr2;
}
```

5. **Write Header to the Event Log**: this function writes the event log header, including both the SpecID event and locality event headers. It first writes the SpecID event, then writes a locality event with a predefined signature.

```
uint8_t *event_log_write_header(uint8_t *log_ptr2)
{
    const char locality_signature[] = TCG_STARTUP_LOCALITY_SIGNATURE;
    void *ptr;

    log_ptr2 = event_log_write_specid_event(log_ptr2);
    ptr = log_ptr2;

    (void)memcpy(ptr, (const void *)&locality_event_header,
           sizeof(locality_event_header));
    ptr = (uint8_t *)((uintptr_t)ptr + sizeof(locality_event_header));
    ((tpmt_ha *)ptr)->algorithm_id = TPM_ALG_ID;
    (void)memset(&((tpmt_ha *)ptr)->digest, 0, TCG_DIGEST_SIZE);
    ptr = (uint8_t *)((uintptr_t)ptr +
           offsetof(tpmt_ha, digest) + TCG_DIGEST_SIZE);
    ((event2_data_t *)ptr)->event_size =
        (uint32_t)sizeof(startup_locality_event_t);
    ptr = (uint8_t *)((uintptr_t)ptr + offsetof(event2_data_t, event));
    (void)memcpy(ptr, (const void *)locality_signature,
        sizeof(TCG_STARTUP_LOCALITY_SIGNATURE));
    ((startup_locality_event_t *)ptr)->startup_locality = 0U;
    log_ptr2 = (uint8_t *)((uintptr_t)ptr +
        sizeof(startup_locality_event_t));
    return log_ptr2;
}
```

event_log.h

Serving as the header file for "event_log.c", this file declares the functions and data structures used to manage and manipulate the Event Log, providing the necessary interface for other system parts to interact with the log.

tcg.h

This file defines macros and data structures for the TPM event log, based on the specifications outlined by the TCG. It is a foundation for logging events in a format compatible with TPM systems.

event_print.c

The code includes two functions, id_event_print and event2_print, that parse and print the contents of two specific event log structures: TCG_EfiSpecIDEvent and TCG_PCR_EVENT2, respectively. These structures are used in TCG standards for event logging, primarily in measured boot processes. Each function extracts and formats the data in the structures for debugging or verification.

1. **id_event_print**: this function parses and prints the details of the TCG_EfiSpecIDEventStruct from the event log. This structure is the first event in the log, providing metadata about the log format, supported hash algorithms, and vendor information. Key components include:
   - Header Fields: contains the PCR index, event type, digest, and event size.
   - Digest Algorithm Details: lists supported hash algorithms and their digest sizes.
   - Vendor Info: additional information provided by the vendor.
2. **event2_print**: this function processes and prints the details of TCG_PCR_EVENT2, which represents a specific measurement event in the log. It includes:
   - Header Fields: PCR index, event type, and the count of digests.
   - Digest Details: lists the digest values for each supported hash algorithm.
   - Event Data: the actual event data, which may include startup locality information or other relevant event details.